SLA Calculus

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Sebastian Vastag

Dortmund

2014

Sebastian Vastag
Lehrstuhl IV – Modellierung und Simulation
Fakultät für Informatik
Technische Universität Dortmund
Martin-Schmeißer-Weg 18
44227 Dortmund

Tag der mündlichen Prüfung: 20.11.2014

Dekan	**Prof. Dr.-Ing. Gernot A. Fink**
Gutachter	**Prof. Dr. Peter Buchholz** (TU Dortmund, Fakultät für Informatik) **Prof. Dr.-Ing. Reinhard German** (Universität Erlangen-Nürnberg, Department Informatik)

Abstract

For modeling Service-Oriented Architectures (SOAs) and validating worst-case performance guarantees a deterministic modeling method with efficient analysis is presented. Upper and lower bounds for delay and workload in systems are used to describe performance contracts. The SLA Calculus allows one to combine model descriptions for single systems and to derive bounds for reaction time and capacity of composed systems with analytic means.

The intended, but not exclusive modeling domain for SLA Calculus are distributed software systems with reaction time constraints. SOAs are a system design paradigm that encapsulate software functions in service applications. Due to their standardized interfaces and accessibility via networks, large systems can be composed from smaller services and presented as services again. A well-known implementation of the service paradigm are Web Services that allow applications with components connected by the Internet. Own services and those rented from providers can be transparently combined by users.

Performance guarantees for SOAs gain importance with more complex systems and applications in business environments When a service is rented by a customer the provider agrees upon a Service Level Agreement (SLA) with conditions concerning interface, pricing and performance. Service reaction time in form of delay is an important part in many SLAs and subject to performance models discussed in this work. With SLAs providers implicate a maximum delay for their products when the customer limits the workload to their systems. Hence customers expect the contracted service provider to deliver the performance figures unless the workload exceeds the SLA. Since contract penalties could apply, providers have a natural interest in dimensioning their service in regard to the SLA. Even for maximum workloads specified in the contracts the worst-case delay has to hold. Moreover, due to the compositional nature of Web Services, customers become providers themselves when they offer their service compositions to others. Again, worst-case performance bounds are of major interest here.

Analyzing models of SOAs is an option to plan, dimension and validate service performance. For system modeling and analysis many methods exist. Queueing Systems and simulation are two well-known approaches in computer science. They provide average and thus long-term performance numbers quite easily using, probabilistic workload and service process descriptions. Deriving system behavior in worst-case situations for performance guarantees is elaborative and can be impossible for more complex systems. Receiving delay bounds usable in SLAs for SOAs by model analysis is still a research issue.

A promising candidate to model SOA with SLAs is Network Calculus, an analytical method to derive performance bounds for network components. Given deterministic descriptions for arrival to and service in a network node hard bounds for network delay and the required buffer memory in routers are computed. A fine-granular separation between short- and long-term goals is possible. Network Calculus models also feature composition of elements and fast analytical analysis. When applied to SOAs with SLAs the problem arises that SLAs are not suitable as a system description and information source for Network Calculus models. Especially the internal service capacity is not exposed by SLAs, since providers consider them as a business secret. Without service process descriptions Network Calculus models cannot be analyzed.

The SLA Calculus is presented as a solution to this problem. As a novel contribution for deterministic model analysis for SOAs, SLA Calculus is an extension to Network Calculus. Instead of service process descriptions, it uses information on latency to characterize a system. Delay of services is not a scalar analysis result anymore, it becomes a process over time that is bound with Network Calculus-style curves, the delay curves. Together with arrival curves the performance contracts in SLAs are formalized by so-called SLA Delay Properties (SDPs) as a description for the service performance in worst-case. Service composition can be modeled by serial and parallel combination of SDPs. The necessary theorems for the resulting worst-case bounds are given and proved. We will present a method to transfer these performance figures to the missing service process description again. Apart from basic theory we will also consider solutions for practical modeling situations. An algorithm to extract arrival and delay curves from measurements, enables the modeler to include already existing systems without given SLAs as model elements. Finally, we will sketch a selection method in form of an optimization problem for services to support the dynamic service selection in SOAs with a Service Broker.

SLA Calculus model analysis will deliver deterministic upper and lower bounds for workload capacities and response times. For upper bounds the worst-case is assumed, thus bounds are pessimistic. The advantage of SLA Calculus is the ability to compute these bounds very fast and to give system modelers a quick overview on system characteristics considering extreme situations. In other modeling methods a lengthy transient analysis would be required.

The strict perspective towards worst-case brought up another analysis target: Until now, relatively little attention was paid to contract conformance between subsequent services within service compositions. When services offer different workload capacities the arrival rate to the system needs to be adjusted to avoid bottlenecks. Additionally, for service compositions no response time contract can be guaranteed without internal buffering to enforce a common arrival rate. SLA Calculus unveils the necessary buffer delays and is able to bound them.

Acknowledgment

This thesis consumed more time and energy than ever expected. To finish against all odds was only possible by the support of many people I would like to express my gratitude.

I express my deepest gratitude to my advisor Prof. Dr. Peter Buchholz for his support, ideas and almost infinite patience. A special thanks goes to my co-advisor Prof. Dr. Reinhard German for supervising the last stage of my work. Furthermore, I would like to thank Prof. Dr. Heiko Krumm and Prof. Dr. Heinrich Müller, who were willing to be members of my Doctoral Committee.

I am grateful to Dr. Falko Bause, Dr. Jan Kriege and Iryna Felko for their advise and fruitful scientific discussions. Additionally I also want to thank the other members of the Modeling and Simulation group for their friendship. I also thank Prof. Krumm and Oliver Dohndorf for their support, when the continuation of my work was in severe question.

The text was proof-read by Tincy Sadic, so the readers due gratitude to her.

Finally and most importantly, I would like to thank my parents and my entire family for their love, encouragement and understanding.

Contents

1 Introduction

In the age of the Internet, applications are implemented as distributed software systems. Service-Oriented Architectures (SOAs) are a design paradigm, based on the idea that processing functions of software systems can be separated into independent instances that are offered as services. Each service offers a capability such as processing power, data storage or a computing function. A workflow using those independent services connects them to a system serving a purpose. Service providers can either be local or remote, service calls over networks are transparent for the application. A system may span over the Internet without geographical restrictions. Basic services are combined to complex systems by composition and may form hierarchies [94] that present themselves as a service again. It is substantial to the SOA approach that every service has an exactly defined function and interface [91]. This allows one to exchange services without functional modification of the composed system. A common implementation of SOA are Web Services [94] that can be hosted at different service providers with individual bias in quality, speed and service. The system performance of SOAs is directly dependent on its components, bottlenecks decrease the overall system performance. Not being able to distinguish between local, remote and composed services the resulting service performance is unknown to the user and not measurable unless one is a customer in contract. This can result in unclear situations where users of a service are dependent on performance and availability of service components. As SOAs have an important role in business applications they need to meet requirements in availability and response times.

Performance guarantees of a service component are often committed by the service provider to the customer in form of a Service Level Agreement (SLA) [17, 109]. SLAs are a form of contract between a user and a service provider issued by service providers as an offer or by customers as a requirement. Besides functional service descriptions and monetary aspects, they include guarantees on service performance [17, 93]. Being part of a business contract, service providers and customers have a substantial interest in providing or renting services that are conform to their SLAs. On the one side customers rely on service guarantees since their business may depend on the service. On the other side providers are willing to fulfill SLA contracts to keep and generate customers. Additionally, contract penalties may apply. With financial interests and legal issues, a line has to be drawn between a service that performs conform to a SLA and a service that fails. For this reason SLAs include maximum and minimum values providing limits on service workload and acceptable performance figures. From the opposite view, one can assume that a service will always react conform with its SLA, as long as the described worst-case workload limit is not exceeded.

Models are used for performance estimation and validation of composed systems. Especially in early design phases analytical models help to estimate performance figures for systems. While these models are very abstract compared to real systems they offer fast analysis results for various applications. Service customers can use them to size the SOA for an intended workload and to select service providers based on performance to avoid shortages at runtime. As performance guarantees in SLAs gain importance in SOAs there is the need to integrate them in the modeling process, too. While SLAs of basic services may offer sufficient information, resulting performance of a service composition is unknown in the first place. A modeling method aware of SLAs would be able to use their values next to the system structure as input. Model analysis would provide performance bounds or even a SLA for the complete system.

This advantage is not limited to general performance estimation, it also supports the process of SLA validation for systems. With an appropriate model service providers are able to decide if a (composed) system given as a model is able to fulfill a required SLA. Furthermore, since the model is based on guarantees for single services and the analysis provides worst-case performance bounds the provider itself can pass guarantees to the customers.

1.1 Challenges

Analytic performance models for SOAs require a formalization of performance limits and guarantees found in SLAs. Results given by model analysis should show the maximum (and minimum) results achievable by the system. This is necessary to make a clear distinction between service models that fulfill requirements even in worst-case situations and those which do not. Stochastic modeling methods like Queueing Theory allow computation of bounds for simple models with elaborative analysis. For detailed models average performance values in equilibrium are computable. In case of SOA with SLAs this introduces several problems: System conformance to SLAs is hard to show if the model provides mean values only. There is no indication if SLA limits are still violated in rare cases. This task can be solved when a deterministic modeling method is chosen that is able to give a binary answer as an analysis result. For models of packet switched networks the Network Calculus approach [107] can deliver tight bounds on performance figures. However, mapping SOA models to Network Calculus is restricted by the information given in SLAs.

A second challenge for SOA modeling is that SLAs present their service as some kind of black-box. Performance commitments reflect the service behavior in extreme situations, for example, for a maximum workload the worst-case response times are given in form of upper bounds. In their information set the service performance is not included and thus the processing rate for service requests is unknown. From the customer's perspective and due to the way SOA economy is structured, there is little chance to get this information next to SLAs. So SOA models have to work with performance guarantees, but should not depend on the knowledge of service rates. This conflicts with the usage of Network Calculus models since their analysis depends on service rates. On the contrary, in Queueing Theory the service rate is a factor that can be computed afterwards.

The hard limits on performance and absolute guarantees sketched above do not apply to all SOA models. In real applications short phases of reduced performance or higher load are accepted by providers and customers, as long as such phases have a limited duration. The reason is that SOAs are influenced by many external factors. Services communicating over a network cannot rely on a completely controlled environment and are influenced by external network traffic, load and other factors. From the abstract model perspective the behavior of services or composed SOAs seems to be stochastic. Hard deadlines, as found in embedded systems, are likely to be violated. Hence requirements in SLAs often have to include tolerances. Target values for performance indicators are set, but violations up to a given short period of time, regulated by rules in SLAs are allowed. A modeling method reflecting the structure of SLAs should formalize and make use of this aspect. As we will see, Network Calculus can provide this.

Modeling and validation of SLAs in SOAs differs in several characteristics from other modeling domains. As a consequence, modeling for SOA has to consider the way performance guarantees and workload restrictions are formulated in SLAs. On the one hand, limits for performance figures of system components have to be formulated and on basis of the analysis a decision whether a modeled system itself is compliant to performance requirements should be possible. On the other hand, a way to deal with hidden service rates in SLAs has to be found.

Based on these requirements the following problems are solved:

- How to build analytical models for systems when only SLAs for components are known?
- How to model performance requirements found in SLAs?
- How to determine performance bounds of SOAs composed of services that feature SLAs?
- How to validate if a given system can fulfill an SLA?
- And, for newly deployed services with unknown SLAs: How to estimate their model parameters?

1.2 Contributions of this Work

Previous observations and requirements motivate the work done in this thesis: The development of an algebraic modeling method to describe systems with nonfunctional requirements for workload and delays based on (min,+) algebra. SLA Calculus is a deterministic calculus for performance bounds in SOAs under worst-case assumptions. Model analysis itself will give performance bounds based on the given limits of model components. It is a strictly analytic approach that can be implemented in an efficient way. This work will include theory, system model and application of a deterministic model for SLAs.

The introduced SLA Calculus is based on Network Calculus, a system theory for computer networks. By transferring the ideas of a model for packet processing to the modeling domain of SOA, a framework for SLA validation is created. Network Calculus features expressions for deterministic bounds of system workload and speed. Functions are used to model these performance values with a high degree of flexibility. For the need of SLA modeling the function set is extended with functions that bound the processing duration for service requests with the same flexibility. The so-called delay curves are the main contribution of this thesis. They enable SLA Calculus to model soft deadlines and other quantitative requirements regarding time behavior in SLAs, and to substitute the missing service rate information. With the curve modeling concept the new calculus is able to separate long-term target performance from temporary variations.

A special focus is set on computing guarantees for composed services. For serial and parallel concatenations of single services to networks forming SOAs, the analytic framework to derive arrival and delay bounds is constructed. Additionally, the resulting performance figures describe the SOA as a service usable in hierarchical models again.

As a second contribution the means are provided to compute missing service curves for SLA Calculus models. This can help service providers to size their systems according to contracts closed with customers. In detail, for given arrival and delay curves the lower bounds on service rates are found. Up to now, the feature of service rate computation has been unknown in Network Calculus. On the contrary, in Queueing Theory this is quite simple due to operational laws and Little's theorem. This work transfers this relationship of arrivals, delay and service to SLA Calculus.

Another aspect is the inclusion of components with unknown SLAs in SLA Calculus models. To provide performance bounds for these components, a curve estimation procedure from measured traces is provided. Some curve estimation schemes exist for Network Calculus, but they deliver too detailed results or are limited to packet traces. Therefore, a completely new method has been developed, that addresses the arrival properties of jobs to a service provider, which differ from packet arrivals to a network component in Network Calculus.

1.3 Outline

This thesis proposes SLA Calculus an analytical method to model and validate SLAs in SOAs, describes its mathematical foundation and compares the results to other modeling methods suitable for SOAs. The text consists of eleven chapters and an appendix.

First, SOAs are introduced as the main use case. Although the main results in this work are on a very abstract level, the application area is highlighted in Chapter 2. Some derivatives of SOA like Web Services or Cloud-Computing will be distinguished. BPEL as description language is shown for example to get an insight into the structure of a SOA. The chapter also includes a discussion on the qualitative and quantitative properties of SLAs. For SLAs several description languages exist, WSLA will serve as an example here. Chapter 3 discusses the special aspects of models for SOA with the inclusion of SLAs. The need to build models based on performance guarantees and the requirements on such appropriate modeling language are formulated.

As a second part, modeling and analysis techniques to describe and validate performance aspects of SOAs are summarized. In detail, we focus on discrete event systems. Widely used is Queueing Theory as stochastic modeling method, an introduction of its basics is given in Chapter 4. Several analysis techniques for Queueing Theory and the underlying more general Markov Chains are presented. We will see that queues can be used to model SOAs, but efficient analysis options for performance bounds are limited for detailed models. Simulation as a second analysis technique and its application to SOAs is shown in Chapter 5. In comparison, working with bounds is natural with the basic technique of this work, Network Calculus. For Network Calculus, the main ideas of the (min,+)-algebra are sketched and Network Calculus is presented in detail in Chapter 6 as a deterministic network system theory. Real-Time Calculus as an application of (min,+)-algebra to real-time systems is briefly introduced.

SLA Calculus as the main contribution of this thesis forms the last part. In order to bound it, Chapter 7 points out the approach of describing delay in form of cumulated functions. This is used to form a model description for SOAs with SLAs with quantitative requirements in Chapter 8. SLA Calculus models do not require parameters on internal service performance, Chapter 9 includes a method to compute the missing service curve. Apart from analytical modeling another elementary problem is solved in Chapter 10. It describes a method to estimate the input parameters to SLA Calculus from measurements.

Finally the thesis will be concluded and several issues left open for future research are highlighted.

1.4 Previous Publications

This thesis is based on three publications [117–119] with myself as the sole author and contributor. The articles have been published and presented at the conferences, ValueTools 2011 [117], MMB 2012 [119] and Winter Simulation Conference 2012 [118].

Basic work for the SLA Calculus is included in [117] with the presentation of delay flows and curves as an addition to the Network Calculus, to model quantitative requirements in SLAs. This idea reappears in Chapter 7. The work also includes a first approach on service curve computation included in Chapter 9.

Publication [119] states the sketched ideas in [117] more precisely, hence it also contributes to Chapter 7. The second, analytic approach to service curve computation Chapter 9 was also published in Chapter [119].

Estimation of arrival and delay curves for SOAs is the contribution of [118]. Chapter 10 is founded on these algorithms.

Additionally earlier results from publications with co-authors are used.

- SIMUTools 2008: [15], “Simulating Process Chain Models with *OMNeT++*“. *ProC/B* is a modeling language initially used for logistical process descriptions. Two parallel diploma thesis [53, 128] described and implemented a minimalistic prototype for analysis with the *OMNeT++* simulation environment. The author reviewed the concepts, completed still missing model elements and rewrote significant parts of the prototype. In [15] the project results are presented.

- SIPEW 2008: [16], “A Framework for Simulation Models of Service-Oriented Architectures”. The work is on SOA performance analysis with *ProC/B* using the ported model library from [15]. Additionally, the SOA model includes a second model tier that reflects TCP/IP traffic generated by distributed services. The author designed and implemented new model components connecting both tiers and was responsible for running and analyzing the example model.

- WinterSim 2009 [17], “Simulation Based Validation of Quantitative Requirements in Service Oriented Architectures“. Simulative SOA performance analysis as in [16] was extended with measurements tailored to quantitative requirements in SLAs.

Programming, experiment runs and the concepts how to integrate the measurements into the model was done by the author.

- Simulation Journal 2010: [19], “A Simulation Environment for Hierarchical Process Chains Based on *OMNeT++*”. This is a more detailed description of mapping *ProC/B* to *OMNeT++* initially described in [15]. The *ProC/B* toolkit evolved since the first publication. A documentation of the introduced class hierarchy in the model library and the general message-based programming style is given. The new model is compared to its predecessor. The author contributed significant work in implementation and the statistical comparison.

2 Service-Oriented Architectures

In this chapter SOAs and SLAs are introduced as a research context and main application field for the SLA Calculus. The concept of a service and its quality is introduced. Several implementations of SOAs have been developed, the most important class, Web Services, is explained in more detail. A special focus is on SLAs for such services, the main terms are highlighted and WSLA as an exemplary SLA definition language is presented in syntax and semantics.

The term service is used in various situations. In general a service describes a specific work a person or machine offers to others. In case of computer science, services are offered by computers or more specific, the software running on the system. In this work the term service is used in two circumstances: to describe communication over networks and for networked software components. Telecommunication systems offer a communication service. The term service is used for the functionality of a network to transmit data packets (RFC 2475 [24]). For Internet hosts implementing the TCP/IP stack the service of data transmission is offered by an operating system. Programs can use connections to other systems by socket interfaces that communicate with the local TCP/IP transport layer implementation. In recent years the concept of network service evolved. With the emerging SOA as a design paradigm, service is also used for the higher level concept of data processing.

2.1 An Architecture for Distributed Systems

The generic term SOA describes a software paradigm for networked, distributed systems. Software resources in SOAs are offered as services [91]. A system is made of (loosely) coupled, distributed services. The concept of a service in SOA is application-centric and abstracts from simple data transmission in RFC 2475 [24]. To describe the concept Papazoglou and van den Heuvel [91] give a definition of a service in SOA:

Definition 2.1.1 (Services in SOA)**.** *A service is a pair of interface and implementation representing a (reusable) unit of work. In SOAs a service has the following properties [91]:*

1. *maintains its own state (self-contained)*
2. *platform independence*
3. *dynamic location, invocation and recombination*

Functions realized in a software are packaged in protocol containers that offer a well defined call and return syntax. The published interface of a service is described in a standard definition language [91]. On an abstract level, a service is a pair containing an interface and a private implementation linked to it (Figure 2.1). Service consumers can use services without any knowledge of implementation details [85, 91]. Components in

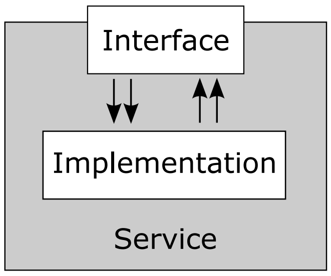


Figure 2.1: A SOA service abstracts an individual implementation with a defined interface

SOAs hide their implementation, only their Application Programming Interface (API) is known and can be discovered, the computing architecture running the service remains hidden. With this implicit encapsulation SOAs are independent of specific hard- or software technology [91]. To stick with the picture of the layered TCP-IP stack, a service in a SOA is a construct located at the application layer.

Example 2.1.1. *Services accessible via Internet can be: user account management and authentication, text translation, credit card payment, data storage (WebDAV), stock ticker, synchronization architecture, a complete Enterprise Resource Planning system, tracking and tracing for parcel services and transaction management. Apart from services providing necessary infrastructure for bigger systems one can also interpret the following examples as services: providing a content management system, hosting a web shop or converting web pages to versions for mobile devices.*

The list in Example 2.1.1 implies the intermediate step towards a SOA. Services are not built for one purpose, they are reusable and can be offered to a private group of users or openly via Internet. Additionally, services can be coupled and composed to more sophisticated services [94]. In this way SOAs are built [91] with services:

Definition 2.1.2 (Service-Oriented Architecture)**.** *A system composed of services is a Service-Oriented Architecture.*

A webshop can be set up from available services in the net by using a database, a payment service, user management and connection to tracking and tracing systems. In an ideal scenario, all functions of composed systems are conducted as services, even the fraction of functionality provided by the system maintainer itself.

2.1.1 SOA Workflows

Commonly SOAs are introduced as a technology to build business applications [1, 91]. One key factor that has drawn the attention to SOAs is the reflection of structured business activities and computing infrastructure in the same meta-model: workflows.

Companies tend to structure their activities into process descriptions. A business process is a set of activities in a given order to meet a business objective [1, Chapter 3.3.2]. These process descriptions can be seen as a model of the company structure and can help in documentation, resolving errors and optimization of time and costs. A business process description that is executable by computers or human participants is a workflow [1, Chapter 3.3.2]. The term workflow comes from office automation projects [1] that existed in companies long before computers were common and paper files were distributed between departments in a structured way.

Activities performed in a company can be ordered and executed conditionally, so workflows can be seen as some kind of scripting language for business objectives. Activities are ordered in a sequence with the semantics, that an activity has to finish before the next one starts, thus a workflow forms a directed graph. Decision nodes can branch the workflow into variants, parallel execution and synchronization of activities can be included as well. There is no standardized model or graphic representation for workflows, but UML activity diagrams [96] or variations cover the limited workflow grammar sufficiently.

SOAs are offered as a solution for tighter business integration without any shared middleware. Each party can offer a subset of their internal activities as a service via network. This allows a more granular control of the information flow and, when done correctly, does not expose business secrets to customers or competitors. Additionally, a service offered via interfaces can be replaced by another service with the same interface. In this way, SOA enables the coupling of relevant parts of business information systems even in short-lived business relationships.

SOAs can help to reuse existing legacy applications by encapsulating them as services and combine them with new applications [9]. Many companies run legacy systems ("do not touch, always repair") that cannot be replaced easily [1]. The strict separation of interface and implementation for a service gives a blueprint on how to integrate these systems. The SOA approach allows one to use old, proprietary software systems as a component in an open and standardized system.

Adopting company structures to SOA was a widely discussed theme in recent years [1, 9, 91]. The topic and technologies became more mature over the time and standards evolved, so that aspects of SOA are found, at least as a design philosophy, in many software products intended for business use today. In the following the key promises and advantages that come along SOAs are examined.

2.2 Web Services

SOAs are a design philosophy, not a specific technology [91]. They are a generic term for loosely coupled systems. Web Services are a specific manifestation of this idea [85, 91, 94], they compare to SOA like C++ to Object-Oriented programming paradigms.

The Web Service paradigm introduced by W3C [121] uses a technology stack built around the XML data format. Interface descriptions are done with Web Services Description Language (WSDL) [40]. A WSDL document includes call and response URLs for a Web Service and their parameter structures and serves the same function as method signatures in programming languages. Call and return format in Web Service are Simple Object Access Protocol (SOAP) messages. In an ideal Web Service scenario (Figure 2.2) services can be discovered from service brokers by implementing a Universal Description, Discovery and Integration (UDDI) registry [89]. The service provider can advertise services by sending WSDL descriptions to the broker. Potential service consumers call the broker and receive

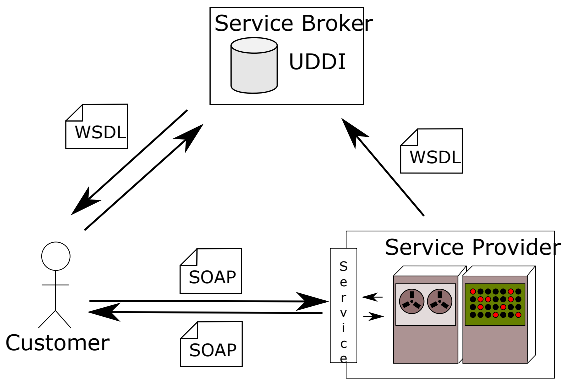


Figure 2.2: A Web Service scenario: A customer queries a service broker for a service. The returned WSDL document is used to establish a SOAP-based communication to the service

answers in form of WSDL documents used to initiate the direct communication between the customer and service provider.

Example 2.2.1. *The company HollowEarth Inc. offers services based on geospatial information systems, their best selling and sophisticated product is a geocoding service. Geocoding is done for digital road networks as found in navigation systems. These networks are large graphs with nodes as road intersections and edges as roads. Given an address in form of text (thus strings) a geocoder identifies the road graph edge or node that represents this address. The result can be used as input into a route planner or a negative geocoding result can help to identify non-existing addresses. The geocoding process is quite laborious as incomplete data, spelling and typing errors in the address have to be considered and corrected. In the past customers sent a file with a set of addresses to geocode to HollowEarth. Several hours later a list of road network IDs was returned. Now HollowEarth has decided to offer geocoding as a service to allow customers to integrate the product into their business workflows. They offer their algorithm as a Web Service to other users by accepting SOAP requests and issuing a WSDL file to a UDDI registry listing geospatial services.*

2.2.1 Service Composition

A single Web Service is intended to perform a well-defined task. If several of those atomic services should be used to perform a more complex task, they can be combined to a new service. A service implementation based on other services is a composite service [1, Chapter 8]. Service composition includes invocations of other services based on underlying business logics [1], thus sequencing and branching is supported. If such a composed service is requested, it behaves like a basic service, hence the composition is transparent to the caller. Similar to procedural programming languages, this allows system modelers to form hierarchical systems including encapsulation and hiding of inner structures. Composition also includes the idea that used services can be located at different companies (or providers), while workflows are limited to one organization.

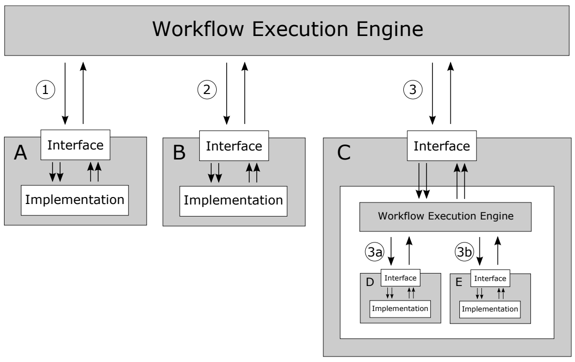


Figure 2.3: Orchestration of Services: A workflow execution engine request services A, B and C in sequence. Service C is a composed service.

Since services in a composed service remain independent, a Workflow Execution Engine (WEE) controls the sequence and conditions of service requests. From a very abstract perspective WEE run scripts defined in service composition languages and provide service interfaces themselves to enable service composition. Figure 2.3 shows how three services (A, B and C) are composed to a sequence of services. The WEE responsible for the composition calls the first service by sending a message to its interface. There is no direct connection between services A and B, service A returns a message to the calling WEE instead. Only then, probably after some internal processing and format conversions, the control flow is delegated to service B by message. Service C is a composition of services D and E itself. A second WEE unites them by executing an own composition script on request, yet the internals stay hidden by the interface. In [111] van der Aalst gives an overview on the supported workflow patterns for several Web Service composition languages. The exemplary language to model process interactions in simple or composed Web Services is (the) Business Process Execution Language (BPEL) [94]. Since WEEs are intended as successors or replacements of normal middleware products, their functionality extends far beyond script execution. They may provide a database access layer, load balancing, security frameworks and other (often vendor-specific) features handy in business environments. Well-known WEE implementations at the time being are WebSphere (IBM), BizTalk Server (Microsoft) or jBPM (jBoss).

Example 2.2.2. *ParcelSink Ltd. offers a door-to-door parcel delivery service. Their payment model is based on the distance between sender and receiver addresses, so ParcelSink considers both locations to determine the transportation fee. Their workflow to process a transportation request is shown in Figure 2.4. The first operation is to determine the start and destination address from the customer, this can be done be telephone or website. Geocoding is a service composition with parallel calls to two services with start and destination address as input. Since geocoding comes along with additional costs and waiting time, a catalog of previously used addresses exists and is used when a customer returns. When both geocoding invocations are done (thus the workflow has to synchronize) their result is used to determine the transportation costs for the invoice. Two invoice copies are sent by email to sender and receiver.*

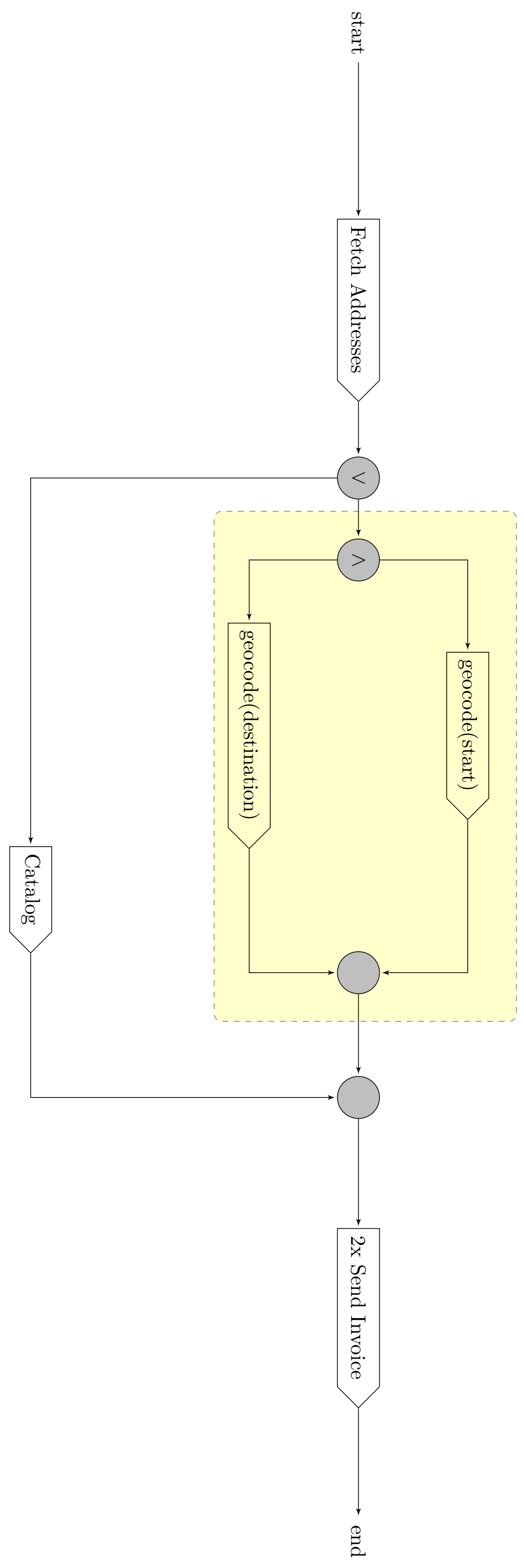


Figure 2.4: The main workflow of ParcelSink Inc.

BPEL

The Business Process Execution Language for web-services, denoted as WS-BPEL or just BPEL, is a standard-based service orchestration technology [57, 79]. BPEL is used to describe interactions of Web Service to reflect business processes [1, 9, 94]. The XML-based language provides means to describe control logics for a Web Service composition [1]. Roughly, BPEL is a high level scripting language triggering service calls to Web Services. In fact, orchestrated web services are considered as executable processes [94] in BPEL. A service composition is based on a WEE, starting the script is again triggered by a Web Service call. The service is only visible by its interface, this allows BPEL to define composed and hierarchical workflows.

BPEL supports Web Service composition but gives no formalism to define an individual Web Service within the language. This gap to describe basic components is filled by using WSDL [1, Section 8.5.1], [9]. Basic service components described in WSDL are considered as an abstract interface description. The network address (URL) of the actual service location is left open. It is expected that BPEL process engines add the appropriate URLs at runtime based on their specific configuration [1, Section 8.5.1]. That provides the opportunity to implement dynamic service composition in the process engine [79, 83]. Service providers, as long as they implement the interface as specified, can be selected based on pricing or performance criteria without changing the workflow itself. Reducing BPEL to a scripting language this would be comparable to changing the environment path to executables.

For service composition and orchestration the scripting part of BPEL provides control structures [1, 57] as found in procedural programming languages:

invoke start a service call,

receive wait for SOAP messages, the script blocks and

reply send a SOAP message to another Web Service.

The expression set also includes control components dedicated to conditional and parallel workflows [87].

sequence A sequence of services that have to be called one after the other

if/then/else conditional statement allowing to choose between several execution paths depending on a variable. It is comparable to conditions in higher programming languages. BPEL 1.1 defines switch statements [1] instead.

while A loop repeating a workflow part. It is comparable to while-loops in higher programming languages.

flow A synchronization element for a set of activities. All activities are started in parallel and the flow element is considered as finished when all started activities are done.

Example 2.2.3. *ParcelSink Inc. has modeled its main business workflow with BPEL, an excerpt from the file with central XML tags is shown in Listing 2.1. The process definition* *for* `transportBookingProcess` *starts with naming the participants in this workflow by enumerating* `<partnerLinks/>`*. Their name attributes relate to the customer itself and WSDL descriptions to used services. The next mandatory section, identified by* `<variables/>` *defines the variables used in this script.*

The workflow description starts in BPEL right after the variable definition using control expressions. Here a sequential process order is chosen. The workflow script execution is triggered by messages received from external sources (Line 19). Start- and destination addresses are saved in two variables and, since service invocations accept a single parameter only, are assigned to complex variable `transportJob` *(Line 26). The first invocation starts the* `fetchAddressService` *to check for completeness of both address entries. Then the workflow decides if the addresses are known or have to be geocoded. If known the catalog service is started and the result overwrites the* `transportJob` *variable (Line 45). Otherwise two synchronized geocoding operations are started in Line 37 using the* `<flow>` *tag. The workflow will block until both addresses are verified and mapped to a road network, the result is stored in* `transportJob` *again. The execution leaves the scope of the if-block and processes the last two commands. Two invoices have to be printed and sent, thus* `printInvoiceService` *is requested two times using a* `<while>` *block (we omit the statements to increment* `i`*). Finally the workflow is finished, if a SOAP message with the transfer papers was sent to the service caller (Line 55).*

The BPEL file in Listing 2.1 is missing some definitions and serves as an example only. For detailed information on BPEL syntax we refer to its language definition in [57].

Listing 2.1: BPEL process description for ParcelSink Ltd.

```
 1 <process name="transportBookingProcess" xmlns="http://docs.oasis-open.
       org/wsbpel/2.0/process/executable">
 2
 3 <partnerLinks>
 4   <partnerLink name="customer" ... />
 5   <partnerLink name="fetchAddressService" ... />
 6   <partnerLink name="startGeocoder" ... />
 7   <partnerLink name="destinationGeocoder" ... />
 8   <partnerLink name="catalogService" ... />
 9   <partnerLink name="printInvoiceService" ... />
10 </partnerLinks>
11
12 <variables>
13   <variable name="StartAddress" ... />
14   <variable name="DestinationAddress" ... />
15   <variable name="transportJob" ... />
16 </variables>
17
18 <sequence>
19   <receive partnerLink="customer" operation="bookTransportStart"
         variable="StartAddress" />
20   <receive partnerLink="customer" operation="bookTransportDestination"
         variable="DestinationAddress" />
21
22   <assign>
23     <copy>
```

```
        <from variable="StartAddress" />
      <from variable="DestinationAddress" />
      <to variable="transportJob" />
    </copy>
  </assign>

  <invoke partnerLink="fetchAddressService" operation="checkAndVerify"
      inputVariable="transportJob" outputVariable="transportJob" />

  <if>
    <condition>
      bpel:isKnown(StartAddress,DestinationAddress) == false
    </condition>
      <flow>
          <invoke partnerLink="startGeocoder" operation="geocode"
              inputVariable="StartAddress" />
      <invoke partnerLink="destinationGeocoder" operation="geocode"
          inputVariable="DestinationAddress" />
    </flow>
    <assign>
      ...
      <to variable="transportJob" />
    </assign>
    <else>
      <invoke partnerLink="catalogService" operation="retrieveAddress"
          outputVariable="transportJob" />
    </else>
  </if>

  <while>
    <condition> i <= 2 </condition>
    <invoke partnerLink="printInvoiceService" operation="printInvoice"
        inputVariable="transportJob" />
    ...
  </while>

  <reply partnerLink="customer" variable="transportJob" />
</sequence>
</process>
```

2.3 Service Level Agreements

With the development of SOAs many systems, even commercial ones, will rely on outsourced services hosted on remote servers. By doing so, the user of the service not only receives processed computing jobs from the service provider, but also uses (partially) several resources at the computing center:

- hardware where the service is executed
- software licenses
- CPU cycles

- storage
- computing infrastructure (building, cooling, etc.)
- energy
- Internet traffic
- know-how of system administrators
- other non-free items like IP addresses, domains, etc.

This list is far from being complete, but serves as justification to discuss an important aspect of services in SOAs: *One has to pay for them.* Business models for SOAs include monthly fees or a pay-per-use charging [9]. Users become customers and system administrators become business contractors. This finally leads to the situation that some kind of quality is expected from services in exchange for money. Expectations and promises for service quality become part of business contracts and finally, they are considered as a requirement for the service. Such requirements like reliability and response times are mandatory to integrate a service with a second or build a composed system. The manifestation of requirements for services are Service Level Agreements [17, 23, 86, 109].

For networked systems RFC 3198 [125] gives a definition for an SLA:

> The documented result of a negotiation between a customer/consumer and a provider of a service, that specifies the levels of availability, serviceability, performance, operation or other attributes of the service [RFC2475]. [...]

The term service in this definition is originally used for the functionality of a network to transmit data packets (RFC 2475 [24]). In the past the concept of network service evolved. Services in a network became a higher level concept of data processing in computer systems reachable over a data network.

RFC 3198 makes a difference between the party that offers the service (provider) and the user of the service (customer). Although for simple systems customer and user can be the same person, we will use this terminology for roles in the remaining work. SLAs can be issued by service providers as an offer or by customers as a requirement. In any case, SLAs are a contract between user and service provider. They can be set up as part of paper contracts in form of human-readable text (and thus unusable for automatic service brokering) or in electronic form (c.f. Section 2.3.3). SLAs include two sets of specifications [17, 23, 86, 93, 109]: a functional service description and requirements for runtime. This thesis will mainly focus on the second SLA part.

2.3.1 Functional Properties

First of all, it has to be defined what a specific service is able to do and how it can be used. Considering Definition 2.1.1, these requirements are mandatory. These are functional or qualitative requirements in an SLA. As a rule of thumb, service properties invariant at runtime are functional properties.

The service syntax or more specifically, its interface description according to Definition 2.1.1 is expressed using description languages. For Web Services WSDL [40] is used. It is XML-based and thus machine readable, a property that helps in automatic generation of source code to access the service. Apart from these technical properties the functional SLA part also includes properties important for business applications: pricing and charging of service usage may be a factor. This can include, for example, payment modalities. Each single job request or a monthly fee may be charged. Also important in business contracts are juridical properties that delegate responsibility between contractors. This can include terms of usage and modes for giving compensation if the rented service is dis-functional.

2.3.2 Nonfunctional Properties

The nonfunctional SLA part includes fractions of the contract that can be measured and quantified, hence its entries are called quantitative requirements [17] or SLA metrics [93]. Barros and Dumas [9] include guarantees, pricing, penalties, delivery modes in nonfunctional service properties. We exclude monetary properties for pricing here and interpret them as a static part of the interface description. However, some SLA specifications may include rules for contract penalties if certain quantitative requirements are not met by the service at runtime [9, 108]. Like the functional part, quantitative requirements are part of the SLA contract between user and provider. The metrics commonly found in SLAs [93] can be divided into two subgroups: restrictions for the customer and obligations for the provider. The following enumeration of SLA metrics used in this work is far from being complete in general. An extensive list is available in [93].

Workload restrictions affect the usage of services by users or customers since they are not free to use the resources of this service vastly. Sending an unlimited number of requests may lead to overloaded computing resources at the (probably unprepared) service provider. In general, service providers can only offer limited resources. They have to be continuously provided and maybe shared, and if one or several customers exceed their quota the service performance will degrade for all. Response times for requests are elongated and buffering occurs. In extreme cases the provider will run out of storage space and deny further request. To ensure system functionality and to achieve customer satisfaction service providers may issue restrictions on the workload sent to their systems. To monitor restrictions and to detect violations the workload has to be quantified [93]. Metrics used to specify SLAs frequently found in system performance models are given in the following list.

Request Rate The number of service calls per time unit. It may be limited to reduce the service load.

Job Size When the size or complexity of a request will use computing resources there may be an upper bound per request or per time interval. Quantification is done in Bytes, data sets or CPU cycles.

With the restrictions for workload, a service provider should be able to dimension and distribute its computing resources. This finally allows him or her to give some guarantees on the service performance.

Response Time Time gap between request and service response. For typical computing applications this is measured in small time scales like milliseconds. In SOA, depending on the service complexity, response times can range from seconds to hours.

Throughput Number of processed requests per time interval.

Utilization Fraction of time the service is not idle

Availability The service shall be reachable and reactive for a certain percentage of time. For obvious reasons this is a minimum requirement. For example, 99% percent availability per year results in up to 4 days of downtime per year.

A more formal definition of request rate, response time, utilization and throughput using average values will be given in Section 4.2 for Queueing Systems.

When giving guarantees and enforcing restrictions one has to decide by some criteria if they are fulfilled or violated. By defining limits for criteria the maximum value a performance metric is allowed to reach is set. For example, maximum request rates can be used to limit workload in SLAs while maximum response times define deadlines for each request. While maximum (and minimum) criteria classify workloads clearly criteria based on statistics are common as well. Most notable are criteria using average values, often in combination with an associated time interval. However, as we will see in the following, defining contracts with statistical values has some drawbacks for SLA validation and modeling. Of course, other statistical figures (median, etc.) are also possible.

In literature different terms for the set of quantitative requirements in SLAs are used. Two terms, Quality of Service [86] and Service Level Objective [125] will be highlighted. They cover the same set of values, but differ in their intention (Figure 2.5).

Quality of Service

When services are used by contractors in a productive environment they expect some predictability for the service performance. As with most systems, performance, availability (reliability) and security are of major interest [85]. A frequently used term for quantitative requirements in SLAs is also Quality of Service (QoS). QoS is related to the nonfunctional properties a service should offer. QoS is about performance metrics [86], response times and throughput are direct results of the QoS (c.f. [85, p. 647]). In [83] QoS is considered as different levels of performance and reliability of a functional identical service. A single SLA file can include a single functional but diverse nonfunctional parts of a service (see Figure 2.5. They often differ in guaranteed service quality and in pricing. The QoS levels can be chosen when the service is selected and further negotiations with the provider begin.

Example 2.3.1. *HollowEarth offers three QoS levels for their geocoding service, offering the same functionality The community edition is for free, but neither response times nor availability are guaranteed. For a fixed rate per request the professional level is available with a guaranteed response time, however the number of requests per time unit is limited.*

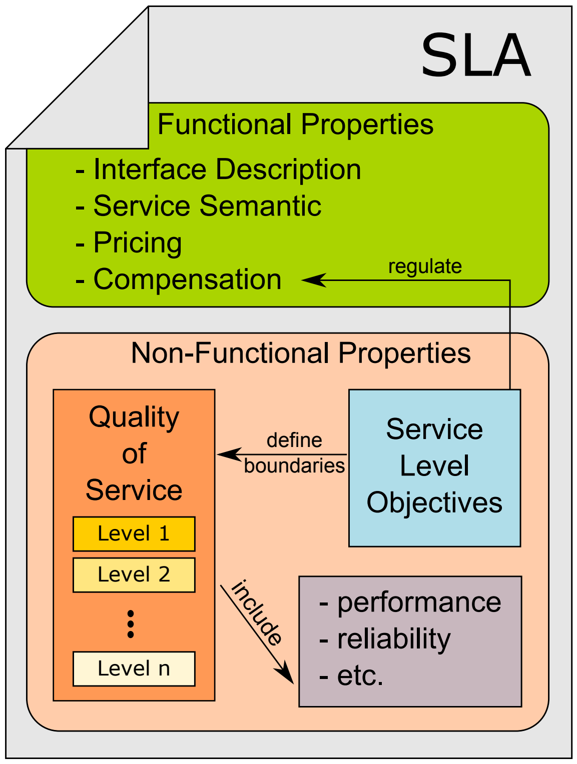


Figure 2.5: Partition of Properties in Service Level Agreements

Multiple QoS levels per SLA are handy during negotiations, though obviously an SLA with n QoS levels can be replaced by n SLAs with one QoS level. In the remainder we will assume one QoS level for all SLAs only.

Service Level Objectives

A guideline to verify if a service meets the intended QoS level are Service Level Objectives (SLOs) [125]. RFC 3198 [125] states on SLOs:

> Partitions an SLA into individual metrics and operational information to enforce and/or monitor the SLA. “Service Level Objectives” may be defined as part of an SLA, [...], or in a separate document. It is a set of parameters and their values. The actions of enforcing and reporting monitored compliance can be implemented as one or more policies.

SLOs define thresholds for performance and reliability properties that help to decide if a service conforms to a QoS level [126]. Thresholds can be realized in form of maximum/minimum values or stochastic expressions based on measurements. [17] shows a blueprint how Service Level Objectives in SLAs can be verified in a simulative environment.

Tolerances for SLOs

Response times in SLAs are formulated in a way that a certain tolerance within a time interval is allowed. The lack of hard deadlines as common in embedded real-time Systems [103] originates from communication infrastructure used for SOAs. Regardless whether a private network or the Internet is used, data communication is not 100% reliable in terms of transmission rates and connections stability. In a network too many factors and events influence the system performance [17]: heavy traffic load, cross-traffic, change of IP packet routes, system downtimes and maintenance, etc. . For the intended application field of business computing for SOA or even Cloud Computing, these communication latency variations do not matter as they are not time critical. The deadlines found in an SLA contract are either requirements of a customer or offers by the provider, but not a central system property as in real-time systems. With the background of unreliable networks, SLAs try to quantify their tolerance on deadlines in SLOs. However, in a complete system performance model these tolerances have to be included.

We will not discuss the specific and in-probable case when the SOA paradigm is used for real-time system design. This would mix two opposed ideas and is a research topic on its own.

2.3.3 SLA Definition Languages

To specify SLAs for Web Services several attempts towards an SLA definition language have been made. Two dialects to be mentioned are WS-Agreement [3] and SLAng [69]. The Web Service Level Agreement (WSLA) language [78] as a third variant gained a big momentum in the industry and is implemented in various software systems [87]. WSLA tries to cover the complete life-cycle of a Web Service. This includes specification, run-time measurements to control the SLA and even a set of actions or penalties that apply when the SLA is not met.

In the following, the language elements found in WSLA will be briefly introduced to give the reader a notion of how SLAs are stated. The language format is XML with the drawback that even small SLA aspects become lengthy files. Hence for the given WSLA example a shortened and simplified WSLA syntax is used.

In principle, WSLA documents include three sections identified by XML tags: The `<Parties>` tag allows one to name all participants that are involved in the agreement, a functional property. `<Obligations/>` holds nonfunctional properties, a mixture of both is in `<ServiceDefinition/>`.

Example 2.3.2. *Parcel service ParcelSink and Geoinformation provider HollowEarth have agreed on an SLA for the geocoding service using a file in WSLA syntax. Listing 2.2 shows the file with the* `<Parties/>` *section for all participants (*`<ServiceDefinition/>` *and* `<Obligations/>` *are shown later). HollowEarth Inc. is declared with* `<ServiceProvider/>` *in Line 3 and related to unique the string* `name="HollowEarth"`*. The service consumer ParcelSink adds itself to the WSLA file with* `<ServiceConsumer name="ParcelSink">`*.*

Listing 2.2: WSLA Example

```
1 <SLA xmlns="http://www.ibm.com/wlsa>
```

```
  <Parties>
    <ServiceProvider name="HollowEarth">
      <Contact>
        <Street>Convex Road 1</Street>
        <City>Atlantis City</City>
      </Contact>
    </ServiceProvider>

    <ServiceConsumer name="ParcelSink">
      <Contact> ... </Contact>
    </ServiceConsumer>

  </Parties>

  <ServiceDefinition name="geocodeService">
    ...
  </ServiceDefinition>

  <Obligations>
    ...
  </Obligations>
</SLA>
```

Service and Measurement Definitions

The `<ServiceDefinition/>` section identifies one or multiple services part of the SLA contract. Each definition can also include several operations that are offered by the service. In particular the `<Operation/>` tag is from major interest as it connects a WSDL service interface to the SLA contract. For each interface performance indicators can be identified with `<SLAParameter/>` tags. Their unit and semantic can be freely defined based on the underlying measurements. An `<SLAParameter/>` tag names a performance value and adds some meta-information and access regulations to it, however the real measurements are defined in `<Metric/>` tags and each `<SLAParameter/>` tag is based on one.

The metrics specify what and how is to be measured. Actual information providers in `<Metric/>` are the `<MeasurementDirective/>` tags. They gather information by giving a reference to performance statistics implemented in WEEs. The WSLA standard does not specify any obligatory statistic functions [78], so implementations may vary. More generic is the reference to an URL that can be requested for the performance value. Functions defined over `<MeasurementDirective/>` tags can be used to build more sophisticated performance indicators. The `<Function/>` tag can apply algebraic operators to values or construct time-series from sequential measurements. Next to basic addition and division, other operators for time series exist, most of them adapted from statistics (mean values, median, counters, change rates, etc.). Metrics carry also a `<Source/>` tag with the name of the party that is responsible for the measurement.

Example 2.3.3. *For the WSLA between HollowEarth and ParcelSink a service and metric definition is shown in Listing 2.3 (this replaces* `<ServiceDefintion/>` *in Listing 2.2). Web Service* `geocodeService` *offers the operation* `geocodeAndVerify` *described by a WSDL file. Two service performance indicators are associated with this operation by* `<SLAParameter/>`

tags: `workload` *(Line 4) and* `responseTime`*(Line 8). Their values will be used later to define the Service Level Objectives based on a implication. For the workload parameter it is stated that the* `workloadMetric` *metric (Line 16)is used for measurement.*

While the definition of `<SLAParameter/>` *tags serves for administrative functions a main part of functionality in a WSLA file is found in* `<Metric/>` *definitions. The* `workloadMetric` *is based on a concatenation of two other metrics,* `callCountTimeSeries` *and* `callCount`*, that shall be explained in reversed order. Metric* `callCount` *(Line 45) will provide a counter of service invocations since the system was started. The actual invocation count (and thus the connection to the running system) is gathered by a* `<MeasurementDirective/>` *tag referring to a server sided method* `InvocationCount`*. Metric* `callCount` *is used by metric* `callCountTimeSeries` *to construct a list of measurements with each element associated to its measurement time (Line 37). In WSLA such lists are named time series and are instantiated by a function* `TSConstructor`*. In this case the list has a limited length of two elements only, when a new value is added the oldest one is dropped. The measurement schedule for this metric is not shown in this example. Typical schedules are measurement cycles of one or five minutes. Finally the time series is used in metric* `workloadMetric` *to determine the number of service invocations between two points of time. Three functions are used here, two select values from* `callCountTimeSeries` *(Lines 19 and 27) while the surrounding one subtracts one value from the other (Line 17). The expression is rather lengthy, but in principle it only calculates*

$$workloadMetric = callCountTimeSeries[0] - callCountTimeSeries[-1] \tag{2.1}$$

if the time series of length n can be indexed by 0 down to $-(n-1)$.

The second SLA parameter `responseTime` *is based on* `responseTimeMetric` *(Line 52), being a composition of* `responseTimeTimeSeries` *and* `responseTimeSource`*. In metric* `responseTimeSource` *(Line 66) a special service interface is used to determine the service delay. It can start the geocoding service without any address data (HollowEarth will add two random addresses on its own) and return the seconds until the service responses. Again a time series is constructed with these measurements in metric* `responseTimeTimeSeries`*, the data structure takes up to five measurements (Line 58). The way the response times are recorded is quite error-prone. Possible error sources are, for example, alien network traffic or a short downtime of the test service. For this reason metric* `responseTimeMetric` *does average (Line 53,* `wsla:Mean`*) the five values in* `responseTimeTimeSeries` *to minimize the effect of temporary measurement errors.*

Listing 2.3: WSLA Example Service Definitions

```
1  <ServiceDefinition name="geocodeService">
2
3   <Operation name="geocodeAndVerify" xsi:type="
       WSDLSOAPOperationDescriptionType">
4     <SLAParameter name="workload">
5       <Metric>workloadMetric</Metric>
6     </SLAParameter>
7
8     <SLAParameter name="responseTime">
9       <Metric>responseTimeMetric</Metric>
```

```
  </SLAParameter>
</Operation>


<!-- workload -->

 <Metric name="workloadMetric" type="float" unit="long">
  <Function xsi:type="Minus" resultType="long">
    <Operand>
      <Function xsi:type="TSSelect" resultType="long">
        <Operand>
          <Metric>callCountTimeSeries</Metric>
        </Operand>
        <Element>0</Element>
      </Function>
    </Operand>
    <Operand>
      <Function xsi:type="TSSelect" resultType="long">
        <Operand>
          <Metric>callCountTimeSeries</Metric>
        </Operand>
        <Element>-1</Element>
      </Function>
    </Operand>
  </Function>
</Metric>

<Metric name="callCountTimeSeries" type="TS">
  <Function xsi:type="TSConstructor" resultType="TS">
    <Schedule>workloadSchedule</Schedule>
    <Metric>callCount</Metric>
    <Window>2</Window>
  </Function>
</Metric>

<Metric name="callCount" type="long" unit="calls">
  <MeasurementDirective xsi:txpe="InvocationCount" resultType="long"
      />
</Metric>


<!-- response time -->

<Metric name="responseTimeMetric" type="double" unit="seconds">
  <Function xsi:type="wsla:Mean" resultType="double">
    <Metric>responseTimeTimeSeries</Metric>
  </Function>
</Metric>

<Metric name="responseTimeTimeSeries" type="TS" unit="seconds">
  <Function xsi:type="TSConstructor" resultType="TS">
    <Schedule>workloadSchedule</Schedule>
    <Metric>responseTimeSource</Metric>
    <Window>5</Window>
  </Function>
  </Metric>

  <Metric name="responseTimeSource" type="long" unit="seconds">
    <MeasurementDirective xsi:type="wsla:responseTime" resultType="
        double" >
      <MeasurementURI>http://services.hollowearth.com/geocodeService/
          testResponseTime</MeasurementURI>
    </MeasurementDirective>
  </Metric>

</ServiceDefinition>
```

Service Level Obligations

Now the metrics are used to formulate service guarantees for the SLA contract. In the section marked with `<Obligations/>` SLOs agreed upon are noted down and also actions taken on SLA violations can be defined in advance. Every obligation includes a logic expression with predicates based on measured values and logic operators (AND, OR, IMPLIES, etc.) combining the predicates. Predicates offer means to compare values found in `<SLAParameter/>` to other variables or user-defined constants. Typical comparison operators like `Less`, `Greater`, etc. are used here. For example a threshold on average response times can be defined in this way.

Very important in SLA contracts between two parties is the `<Implies/>` operator, also known in logic for two values A, B as term $A \Rightarrow B = \neg A \vee B$. When the expression A is true then B has also to be true. Implications can be used to formulate conditions between customer and provider, for instance to implement the service guarantees for providers under the workload restrictions for customers (c.f. Section 2.3.2). Obligations can be limited to be valid within a time period only.

Penalties found in SLAs sometimes are covered by `<ActionGuarantee/>` tags. They can be triggered by SLA violations and notify the parties named in the WSLA document. In [93], these actions are subsumed as “if-then rules”.

Example 2.3.4. *The SLA contract between HollowEarth and ParcelSink includes a single SLO: If the geocoding service is not used more than 480 times in a minute the provider can guarantee a service delay smaller than or equal to 10 seconds. Or, using the WSLA SLA parameters and an implication:*

$$(workload \leq 480/min) \Rightarrow (responseTime \leq 10s)$$

Listing 2.4 shows the WSLA construct for the last part of the contract file. The SLO `SLOWorkloadImpliesResponseTime` *obliges HollowEarth to a response time.*

Listing 2.4: WSLA Example Obligations

```
<Obligations>
  <ServiceLevelObjective name="SLOWorkloadImpliesResponseTime">
    <Obliged>HollowEarth</Obliged>
    <Expression>
```

```
      <Implies>
        <Expression>
          <Predicate xsi:type="Less">
            <SLAParameter>Workload</SLAParameter>
            <Value>480</Value>
          </Predicate>
        </Expression>
        <Expression>
          <Predicate xsi:type="Less">
            <SLAParameter>ResponseTime</SLAParameter>
            <Value>10</Value>
          </Predicate>
        </Expression>
      </Implies>
    </Expression>
  </ServiceLevelObjective>
</Obligations>
```

At this point we finish the introduction on SLA description languages. In case more details are added to the contracts, e.g. tolerances, the syntax becomes more complex and the SLA files more long. For continuation of the ParcelSink example we will use a set of SLOs written down in an informal way:

Example 2.3.5. *Table 2.1 contains the SLOs for all services participating in the workflow. It includes upper limits for the workload, thus the arrival contract, and sets limits for the response time in form of a delay contract. Together both columns form an implication expressed by* `<Implies/>` *tags in WSLA files. Both contract parts are internally split into short and long-term obligations. The long-term contract parts correspond to the level of details on conditions expressed with WSLA constructs as demonstrated in Listings 2.2, 2.3 and 2.4. New are the short-term obligations that formalize the flexibility and tolerance to service variance both contract parties have to accept. This variance is restricted in length and/or size by columns "Length" and "Burst".*

First, from the perspective of a service customer, access to servers hosting some services is restricted. Column "server access" shows if the customer can read performance figures like service rates directly from the machines or if he or she is dependent on information found in SLAs issued by the providers. In this example, the geocoding services are external and thus, not accessible.

For the long-term arrival contracts the maximum number of requests per second is given. For example, for HollowEarth a maximum long-term rate of 8 requests per second is specified. This figure is an upper bound, but to anticipate high-load phases resulting from external influences some variance is allowed by a short-term contract. It specifies a time interval of given seconds (second contract column) with arrival rates exceeding the long-term goal. However, these bursts are limited in time by the first column in the short-term contract. Within 3 seconds the HollowEarth service is prepared to accept requests at a rate of 25 requests/s. Additionally a third figure in column "Burst" allows some arrivals that have no rate limit at all.

Delay contracts are given by maximum response times in seconds for each service. For a limited time this goal may be missed and the maximum short-term delay may apply. This *is combined with a limit on downtimes when zero request are processed.*

With Table 2.1 we present a second geocoding provider responsible for geocoding destination addresses. The provider FlatWorld promises – due to a drastically simplified geodic model – to respond within 5 seconds when there are not more than 15 arrivals per second.

2.3.4 Lower Performance Bounds

While it is common for SLOs to specify upper bounds on arrival processes and delay, lower bounds on service performance can be agreed upon, too. A lower arrival bound defines the minimum workload that has to be sent to a service, this can be done using a minimum request rate or a certain workload size. Since many services do not come for free and contracts may have some minimum duration, lower arrival bounds can help to choose a suitable service provider. Especially if charging includes a basic fee, every offer has to be evaluated by customers if it pays off for the projected workload. Minimum workload levels can be adjusted according to break-even points, a customer immediately knows that the service or contracting scheme does not suit well.

Lower response times limits require a system to process request for a given time period at least. In contrast to maximum delays the need for SOAs working slow requires some justification. First of all, services will have intrinsic delay. This is the shortest response time greater than zero which all requests to a service will experience even when there is no other load. It may be caused by buffers, transmission delays or the implemented service functionality itself. With knowledge of intrinsic delay service customers can conclude that services (or their composition) will never work faster than specified. Second, when a customer agrees on lower SLOs concerning response times, he or she can also express that processing is not required to be faster. This self-imposed restriction may lead to cheaper service offers as excessive reservation of system resources can be avoided by providers. For computing systems such lower bounds can apply to limit the usage of a service, for example, on a low QoS level.

Offering not the fastest service is not necessarily a deficit if the response times are documented well in SLOs. In combination with upper response times the variance of processing speed is limited, service response times become more predicable. Therefore lower bounds on delay are helpful in capacity planning for fork/join systems, for example, when implementing the BPEL `<flow/>` tag. With knowledge of minimum processing times the output buffer size of systems storing jobs waiting to synchronize can be estimated. As a side effect, the customer can also externalize the (virtual) buffers required to implement his or her workflow to the provider.

Equal to the implication found in upper performance contracts a relationship can be constructed for lower SLOs: With a required minimum request rate the provider can ensure that service calls takes at least a specific amount of time since there is always some load on the server.

Example 2.3.6. *For the services used in the ParcelSink workflow SLOs on the lower performance are given in Table 2.2. In a similar way to Table 2.1 the contracts and implications on the arrival process and the resulting response time is given. As long as the customer sends request at the required rate the service will not answer within the* time unit: seconds

		Arrival Contract				Delay Contract			
		Short-term			Long-term	Short-term			Long-term
Service	Server Access	Requests/s	Length	Burst	Requests/s	Max. Delay	Length	Downtime	Max. Delay
ParcelSink	✓	100	3	5	12	-	-	-	5
HollowEarth	×	25	3	10	8	30	8	5	10
Flatworld	×	20	2	-	15	50	2	20	5
Catalog	✓	12	1	10	9	-	-	-	2
Fetch Address	✓	-	-	2	15	-	-	5	10
Print Invoice	✓	-	-	6	25	-	-	3	16

Table 2.1: SLOs for ParcelSink workflow example

time unit: seconds

		Arrival Contract				Delay Contract			
		Short-term			Long-term	Short-term			Long-term
Service	Server Access	Requests/s	Length	Break	Requests/s	Min. Delay	Length	Speed-up Time	Min. Delay
ParcelSink	✓	-	-	-	-	-	-	-	-
HollowEarth	×	-	-	5	3	-	-	3	4
Flatworld	×	1	1	-	2	-	-	-	3
Catalog	✓	-	-	5	1	-	-	-	1
Fetch Address	✓	-	-	-	1	-	-	4	2
Print Invoice	✓	-	-	4	6	-	-	8	0.5

Table 2.2: Lower SLOs for ParcelSink workflow example

time specified by the delay contract. On the long-term a minimum response time is given for all services. Some short-term variation is allowed by a 'speed up' time, during that limited phase services are allowed to answer earlier. In the second line we can read for the HollowEarth service the implication

$$(workload \geq 3/s) \Rightarrow (responseTime \geq 4s)$$

Additionally, the contract should still hold if zero job requests are sent to the service for a maximum of 5 seconds. In return, the customer has to accept an interval of up to 3 seconds with early responses. For the FlatWorld service there are also details available on the minimum arrival rate in short-term. While the long-term contract requires 2 requests per second the customer is allowed to send one request for one second only.

Finally, in other applications different from SOAs and computer science, lower performance bounds have some significant importance. This can be the case in modeling transportation, production processes or chemical reactions. In those modeling domains a process has to run a minimum amount of time until it is considered as completed.

2.4 System Capacity Planning

When SOA workflow descriptions like BPEL files are to be realized service providers offering the required services have to be selected. Abstract process descriptions are related to service endpoints by WEEs. Choosing services and their providers is the process of service selection [79, 83] or capacity planning (provisioning) for Web Services [45, 46]. System capacity planning is about dimensioning the required computing infrastructure [46], it can be a manual process or can be delegated to service brokers implementing more sophisticated selection criteria.

Service brokers in the standard Web Service scenario (c.f. Figure 2.2) are not aware of the performance figures in SLAs of their mediated services and thus cannot consider performance constraints. As a solution QoS-aware extensions to service brokers have been proposed [79, 83, 85]. Service providers register services together with QoS-levels they can guarantee in their SLAs. When customers request a service description via the service broker they add their QoS requirement to their message. The broker selects a service provider that provides a suitable service in terms of semantics and functionality. Special about the QoS-aware service selection is that performance requirements are considered this time and are balanced against monetary cost. This scenario can be even extended by negotiation capabilities between QoS and pricing level [83]. A service selection process includes construction, validation and evaluation [82] of workload, performance and cost models for the SOA.

For optimization tasks the lower bound for response times can help to improve results. By their knowledge optimization targets for systems are not only limited to reduce the upper delay bound. For example, when selecting from a set of systems there might be systems with a small distance between upper and lower bound, while others have more variability in response times. Instead of reducing the maximum response time only, optimization can aim on smaller minimum response times.

Service Selection can run once at the deployment of SOAs. When service providers are selected due to performance requirements it seems reasonable to rerun the service selection when conditions change. Dynamic Service Selection [79, 83, 85] or runtime provisioning [87] is the continuous process of adaption to changing conditions. By switching service providers on the fly performance requirements given by SLAs and other contracts hold under varying workloads. With a QoS-aware service broker [85] BPEL execution engines can change the binding of an abstract interface to a specific service at runtime.

2.5 Cloud Computing

The current state of art in SOA based computing is Cloud Computing. While the industry is quickly adopting, there seems to be no unified definition for the term Cloud Computing [4, 48, 95, 102, 127] in computer science. Different adoption speeds and ideas for computing architecture led to a rift between academia and industry [95]. Armbrust et al. [4] defines it as applications delivered as a service including necessary hardware and system resources via Internet. It maximizes the concept of SOAs by providing every aspect of computing as a service usable on demand without preceding investments. Hayes [51] says that cloud computing is

> “a shift in the geography of computation”.

Functions of software are migrating from personal computers to data centers. As a side effect, hardware can be shared among other customers by cloud data center operators. Due to this “statistical multiplexing” the average data center utilization is higher than in traditional, private centers with positive effects on operating costs [4].

From the perspective of customers the advantage of Clouds is the scalability of computing resources on request. Additional computing power for already running applications is supplied by the service provider on payment. Small companies can start with their products using a minimum on Cloud resources and without investing in own infrastructure. On growing demand or to handle high load phases computational power is rented by the “pay-as-you-go” approach [4]. When demand shrinks or the company goes out of business Cloud resources are let go and no private data centers collecting dust remain.

Hence, Clouds already feature a sophisticated form of dynamic service composition known from Web Services. Moreover, in theory system provisioning and capacity planning for the future is rendered unnecessary for customers, since there is neither lack nor abundance of resources. So Computing Clouds could be the solution to everyone dependent on SOA service performance. When quantitative requirements in SLAs are not met additional resources from the Cloud are simply assigned. Potentially unlimited resources can make you stop thinking of service guarantees. Nevertheless we see Cloud Computing as one of the main application fields for modeling and validation of distributed systems with SLAs. On the one hand, delegating responsibility for performance to the Cloud provider also shifts away the problem of SLA conformance, but does not solve it. Customers still expect performance guarantees for applications in the Cloud, however it merely centralizes the responsibility to the data center operator. On the other hand, customer relying on unlimited resources from the Cloud are ill-advised since their financial resources are likely to be limited. Renting a bad performing Cloud service multiple times just to ensure a minimum QoS may become expensive. Good planning ahead and choosing Cloud providers by their offered SLA is even more important as up to now there is no transparent migration to other providers. While workflows of Web Services are compositions of abstract services with no persistent state, Cloud applications suffer from a phenomena known as "vendor-lock" [4, 48]. Clouds only offer programming interfaces and data structures that are unique to the provider, for example, switching from Amazon EC2 to the Google App Engine is cumbersome.

For Cloud Computing, SLAs are the service contract, and being able to model the system based on SLA information will help customers to use Cloud infrastructures.

3 Modeling Service-Oriented Architectures

In the previous chapter we have seen that SOA and the concept of services are a versatile computing architecture. New or existing program parts can be offered to other internal or external users without losing control on privacy and integrity of the own systems. System builders can pick services to fill functionality of workflows and compose a new system using computing power distributed over the Internet. Moreover, they can start to offer their composed system itself as a service to others. Composition languages offer serial and parallel constructs supported by loops and decision statements.

Next to the functional soundness of a SOA, the performance of such compositions is important. For single services, especially when a contract exists, the performance is guaranteed in SLAs in their SLO section in form of upper and lower bounds. They apply when an implication is met: When the customer limits the workload, the service provider grants an upper limit on the response time. These bounds are often defined for the long-term. Since many factors can influence the performance of a distributed system, short-term variations are accepted as a necessity, too. The variation range is codified in SLAs to identify non-conform services. For business applications this is an important feature since customer relationships and the avoidance of contract penalties depend on the given service guarantees.

With such a focus on performance bounds and the ability to compose services based on others SOA, designers are also interested in SLAs for their own systems. The bounds can be used to estimate the performance for internal use or, when offering composed services to others, to give own service guarantees and to set up contracts. Conversely, if there is a performance bound a composed system has to fulfill, one can select a candidate from a set of similar services based on its SLA and pricing condition as criteria. In this application field decisions have to be made early, ideally before a service is contracted. At that point in time the performance characteristic of a service is known by bounds in its SLA only.

3.1 Performance Models for SOAs

Modeling computing systems is a helpful tool for various tasks like performance prediction, capacity planning, monitoring or optimization. A model is a representation of a system intended to study the system [8, Section 1.8]. Using models has several advantages as there is no necessity that the system already exists in reality, so one can study fictional systems. Based on models, questions related to feasibility and performance can be answered at design time. Often there is the need to test new ideas and modifications for already existing systems, as well. This can be the case when the environment or usage scheme for a system has changed. Testing the modifications at the existing and running system can be too dangerous, too costly or both. Models are a safe and normally cheap alternative compared to experiments with real systems in production. All these benefits apply to the modeling of SOAs. The SOA development cycle [87] includes modeling to support system planning. Performance analysis, prediction of workflow execution times and issuing of service guarantees can be done without interfering with the real or anticipated system.

Service consumers can use performance-aware models to optimize their workflows and to locate bottlenecks. Capacity planning and identification of own requirements can and should be done in early design phases. Constructing SOA performance models can also support the provisioning of services and their providers [79] for a planned SOA. Later on, when the SOA is in production, models can be used to monitor the system by comparison of runtime measurements to the ideal model behavior and enable dynamic service selection. From the providers' perspective, SOA models help to fulfill given service guarantees and to avoid penalties resulting from SLA violations. With performance-aware models and their analysis providers can plan their computing equipment, network connections and other resources in advance. Offering service compositions brings providers into the role of a service consumer themselves. Yet, they still have to issue SLAs for the composed service and customers rely on them. Using a performance model in all design phases can help providers to minimize the number of contract breaches for their virtual product.

Building suitable SOA models and their application for performance analysis are the topics in this work. In preparation for the presentation of modeling methods, some basic terms and definitions are introduced.

3.2 Discrete Event Systems

For system modeling and analysis many concepts, methods and approaches exist. In natural sciences differential equations are used to describe the existing laws of nature [35]. These models are based on the assumption that the state of a system is continuous rather than discrete. A different concept are Discrete-Event Systems (DESs) widely used in computer science [6, 35]. DESs are suitable to describe man-made artifacts like computers, software and the things created by them. In these systems changes are considered, at least at a certain abstraction level, as immediate jumps from one system state to the next.

A system is the entity that modeling tries to describe. Although the term system is fundamental in science, no general definition exists. In [8, 70] a system is defined as a group of objects with interaction for some purpose. For a model of a system it has to be noted that the system definition also holds. This work will settle to a more narrow system definition. We will study artificial systems made from interacting components. Each component has a state described by a set of variables [8]. The combination of all states is the system state. If state changes occur in a discrete set of, not necessarily in even spaced, points in time the system is said to be discrete [8]. Events are instantaneous occurrences from outside or within the system that may lead to system state changes.

3.2.1 System Performance

The meaning of system performance depends on the perspective of a system. When we deal with artificial systems the system has been built for a specific reason. An indicator for system performance is how good this system target is achieved. For applications in computer science system performance is often associated with processing speed. A task is executed with high performance if its computation time is short. However, other aspects of system properties can also be required. Low energy consumption is a common performance indicator for portable embedded systems that rely on batteries. For controlling machines and vehicles real-time systems are applied. Their performance is given by absolutely reliable execution time for tasks. Execution speed, as long as it does not exceed the deadline, is secondary.

In this work, system performance follows the classic wording of fast task execution and thus shorter delays.

3.2.2 Analysis of Discrete Event Systems

Model analysis is the activity of answering the initial questions on the original system by evaluating the model. When the question is about a nonfunctional system property we speak of a quantitative analysis [35, Section 1.2.2].

Two classes of model analysis methods exist: numerical and analytical. Analytical models are based on mathematical descriptions, hence model analysis is performed by reasoning [8, 1.10]. The model is a closed mapping of input to output values. A disadvantage of analytic model analysis is that the descriptiveness of the underlying model itself is limited. Models built with Network Calculus, the base method used in this thesis, belong to the family of analytical models. Many classes of Queueing Systems as presented in the following Chapter 4 are analytical models, too. Other examples for analytic methods are Markov Processes or Petri nets [11].

Numerical model analysis does not work symbolically, it uses actual values. It is applied to models with nonexistent or unknown closed mapping or when it is too difficult to compute. Numerical methods use algorithms to find the model output for single points from the input set. Direct numerical methods give almost exact solutions, their accuracy is only subject to rounding errors. Approximative numerical methods compute a solution that is sufficient for many applications. Although numerical model analysis is commonly performed by computers, it can be still laborious and time-consuming. The advantage of the numerical approach is its applicability to almost all model variants based on Markov Chains.

Since Markovian System theory is widespread, two termini from its analysis are frequently used: transient and steady-state (stationary) analysis differing in their time frame. For Markov Chains, steady-state analysis gives the state probabilities for a system running for an infinite time. Based on this result the average system performance can be concluded. Short variations and exceptions in system performance are neglected. Since this work is not about Markov Chains we will use steady-state analysis in terms of long-term analysis. More details and definitions can be found in [35, Section 5.3.10] and [8, 54]. Transient analysis of a system provides results for a limited time span. In the analysis of Markov Chains, transient analysis gives the probability on having a series of state changes from a specific state A into state B within a limited time interval. Therefore short-term system behavior can be observed, this may include special, unlikely or extreme situations (e.g. transition probability from functional to broken states). A special but frequently analyzed case is given when a specific state A is the initial state of a system, and transient analysis can be used to determine the length of startup phases. For the interested reader, further details and definitions on transient analysis are given in [34, 54] and [35, Section 5.3.9].

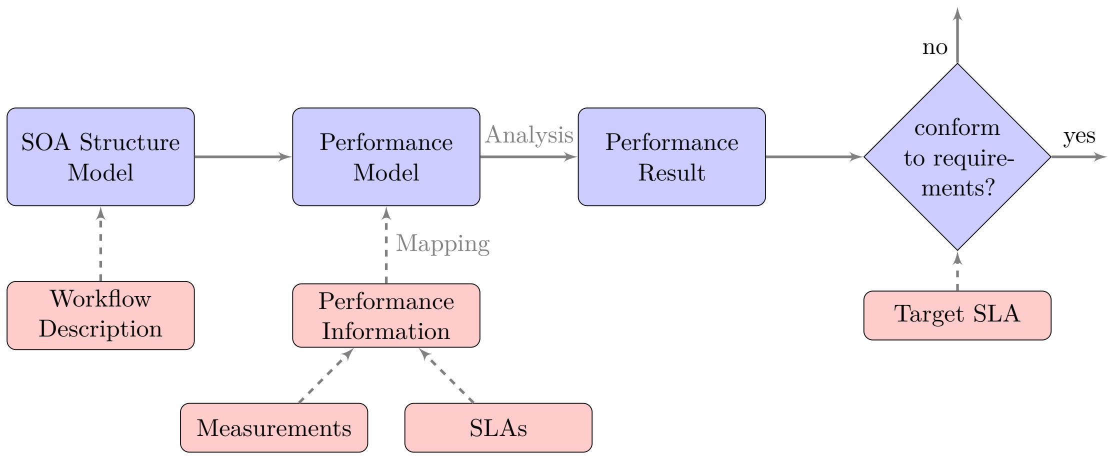


Figure 3.1: SOA performance modeling workflow

3.2.3 A Model-based SLA Validation Workflow

In Figure 3.1 a blueprint for an SLA validation workflow for SOAs is depicted. This is not the final validation workflow for every application, most will add further steps, but it will serve as a prototype for the introduced SOA performance analysis methods. It shows which models need to be created, which information has to be supplied and where the results of this work are employed.

Two models are successively created: the structure model and the performance model. The first holds the SOA workflow structure. It includes basic workflow patterns for the (planned) original system, so usually abstraction steps are involved. BPEL parsing and construction of structure models is not considered in detail in this work. However, when examples are presented, we rely on the brief introduction in BPEL syntax (Section 2.2.1) and assume that tags like `<flow/>`, `<switch/>`, etc. can be mapped to structures of the modeling method. The second model holds the performance of every single service in the workflow. It can be either a new model or, for analytical models, the structure model gets enriched with quantitative figures. These figures are sourced from the information available to the modeler. Taking measurements at system elements is a comfortable option, but often the modeler has to rely on information from second hand. As we will see, SLAs are sources for performance information, too.

After the structure and performance model are set up a quantitative analysis is started. It delivers a result on performance for the whole SOA based on the model. By comparison with the target SLA it can be decided, if the intended system fulfills the requirements. When the SLA validation is part of a service selection process the set of tested services can be accepted or discarded. It can also be integrated into more elaborate schemes.

3.3 SLA Validation and the Worst-Case

The bounds set upon the operational performance of SOAs by SLAs also influence the perspective at SOA models. A service can process job requests either conform to an

SLA or not. Implications in SLAs make this decision dependent on the usage frequency, therefore one has to observe a service's reaction to increasing load. The performance may not drop below the given SLOs. Moreover, this should hold in any situation, even when the guaranteed limits are reached, but not exceeded. This operating point of maximum workload and slow service is described as the worst-case situation for a system. As these bounds are defined by SLOs sections in SLAs one can substitute the worst-case situation with the SLAs and vice versa.

Deciding if a system conforms to an SLA is denoted as SLA Validation. The need for SLA Validation occurs in numerous situations from a provider's and customer's perspective. Service operators are interested in whether they can guarantee a certain QoS requested by customers with their systems. This has to be tested in SLA negotiations [17, 79] when potential customers lay down their requirements in SLAs and ask for an offer. SLA Validation helps to provide this offer by testing the SLA against the model and select an appropriate QoS level. Regarding the perspective of the customer, model-based SLA Validation helps to select service providers by their SLA or QoS, respectively. For a dynamic service composition the selection algorithms can be built upon SLA validation methods. Before a service is replaced at runtime a model modified with the new SLA values is used to test if the target SLA is invalidated or not. Finally, for a running SOA it can be checked if the negotiated SLA is fulfilled.

3.4 Performance Bounds as Analysis Result

In the Web Services economy system with performance guarantees on its products and the risk of SLA contract violations with consequences, model analysis has to provide performance guarantees itself. Moreover, the only way to give guarantees for a model is to derive its performance bounds. These bounds have to be independent on the input to the system as long as it is within an accepted range, even in the worst-case. They also have to be time-independent, as the guarantees should hold in any situation. And finally, they have to be deterministic. Any bound dependent on a stochastic element is not a real limit, it is a weaker statement. This is the case when analysis results depend on statistics.

In this context a model analysis giving average result figures is not suitable for setting up limits. An average performance result cannot be declared as bound since it is very likely to be exceeded (or undercut). Remember, it is the nature of average values that a significant portion of all input is larger or smaller than the average value. We will further explore the problems of average value models in Chapters 4 and 5.

3.5 SLAs as Performance Description in Models

To validate SLAs an adequate model of the architecture and integration of the SLAs in the model is necessary [17]. A SOA modeling method that will give performance limits has to accept bounds and thus SLAs, too. First, without bounds the important worst-case situations for SLA Validations cannot be described. The reason is, equally to the need to give bounds as analysis result, that these extreme situations are not included in models describing the average and expected system operation.

Second, the encapsulation of services in SOA renders SLAs issued by contributing providers to the primary source for model input data. In a general situation for modeling it is assumed that the modeler has to access the original system components. They can be analyzed on their internal structure and system reactions to different workload levels can be tested. In other cases at least historical data is available and allows one to construct performance models. This is not the case for SOAs. In a Web Service environment the provider will not allow customers to access the offered services and their operating environment. As shown, Web Services can be used to offer internal business workflows under controlled conditions to other external users. This offering does not include the allowance for customers to access the computing systems, analyze their structure and do performance measurements for sure. Reasons can be the protection of business secrets, protection of computing infrastructure from manipulations or damage or simply a good sense for privacy. So all information a service provider provides to customers of a service is the interface description (WSDL for example) and the performance guarantee in form of an SLA. Hence, a SOA model has to accept bounds taken from SLAs as the only source of performance figures.

Additionally, SLAs reflect a performance that is guaranteed, but they do not give any information on performance of the underlying computing system. Services are presented by SLAs as a black box system whose reactions on worst-case workloads are known while, according to the SOA paradigm, the inner system structure and the system capacity stay hidden. This leads to the odd situation that SLAs include a manifold of performance indicators but still miss an important value fundamental to system performance modeling: the real processing speed of the computer system hosting the service. Service rates as abstraction are a central element in Queueing Theory and other analytical or simulative modeling methods. The lack of service rates makes system performance modeling and analysis of SOAs cumbersome. A modeling environment for SOAs needs the ability to address this situation.

Using limits as model in- and output enables hierarchical models, too. In a similar way, models of workflows should give performance limits that higher hierarchy level models accept them as input data again.

Apart from system performance specification with SLAs the bounds and service obligations have a very specific property: they include tolerances for the short-term. As presented in Sections 2.3.2 and 2.3.3 the intention of relaxed deadlines is to deal with factors that influence SOAs. Models for SOA with quantitative requirements should also include these tolerances.

3.6 Motivation for a Analytical Modeling Approach

When computing bounds for SLA validation one has to decide on the considered time horizon. Analytical models feature efficient steady-state analysis to show if limits hold for a system during operation in long-term. Since steady-state analysis methods assume the normal, often average operation modes for a system at least two problems are created for SOA analysis: average results and time dependence as the finding might not hold within short time intervals. A transient analysis seems to be more suitable to verify bounds, but is often very time-consuming. The ideal SOA performance analysis methods would unite both time horizons. Network Calculus as a pure analytic approach includes short- and long-term performance aspects in a single analysis, but as we will see, it does not comply with the specifics of SLAs in SOAs.

Still, researching on a SOA modeling method with analytic analysis is desirable. With a fast model analysis offered by analytic methods one gains the ability to estimate the performance of distributed systems at design time and in early development stages. This can reduce the risks and, as a consequence, the costs for SOA projects [85]. Another property of analytic models is their often small set of model parameters and high abstraction grade, they are independent from specific technologies like BPEL or WSLA. While this might be an obstacle to construct almost realistic models, it is an advantage when only some details about the real world system are known. Missing data might hinder the construction of detailed numeric or simulation models because not all mandatory variables can be filled while abstract analytic models still work.

Next to computing performance bounds for a SOA model a fast analysis process is a requirement for many applications. At the same time the ability to give fast and reliable guarantees enables new services. When implemented into service brokers with dynamic service composition they can react to changing environments and workload situations within short time. To predict the optimal composition a model can be constructed and analyzed beforehand. In the following chapters we will see that many analysis approaches need a lot of processing time to give bounds, for example, simulation is often a matter of minutes up to hours. For systems that have to react fast this can be too lengthy. Analytical modeling approaches can give, even based on drafts of a SOA, estimates on performance figures within reasonable time.

In the next sections we will present related work in the research field of SOA modeling and assess it with the following set of requirements. They are based on the specifics of SOA systems, descriptions and implementations of Web Services in Section 2.2 and the way SLAs are contracted.

Hierarchical models For simple atomic models as well as service compositions the same modeling approach can be used. Models for service compositions can be constructed from models of their included services. Additionally, service models can be used transparently as service in other models.

Performance guarantees Target performance indicators in SLAs describe the behavior in worst-case. If worst-case assumptions are model parameters, the results derived from model analysis should be equal or better in reality. This is different from stochastic models like Queueing Theory which gives results in form of distributions and average values. Conversely models should give bounds to support SLA Validation and the issuing of own SLAs.

Tolerances Although upper and lower bounds in SLAs exist they are also formulated in a way that short-term violations are not considered as SLA contract violation. The separation between long-term performance and short-term behavior during disturbances and higher workload have to be considered.

Service rate Real processing or service speed of a computing system running a service is not given in SLAs. They are only known to providers themselves and are sometimes kept as a business secret. Knowledge of such service rates is central for many widely used system models, but a modeling approach for SLAs should do without. In case the service rate is needed for system sizing for example, the modeling approach should be able to derive it from the performance model.

Simplicity and expressive parameters: A model, analytic, simulative or even graphical, should be understandable. We admit that simple is relative and subjective, but it is often preferable if in fact people with no background knowledge at least understand a model partially.

Another obligatory and common entry on this list is the wish for a fast model analysis.

4 Queueing Theory

Queueing Theory is the science of analyzing Queueing Network models and has a strong mathematical foundation that includes stochastics. Several extensive books cover the topic of Queueing Theory [32, 54, 71, 84]. As Network Calculus, Queueing Theory deals with system models for DESs. Systems are abstracted to service centers respective queues and their networks [71, Section 1.2]. They are considered as a class of discrete-event systems, but emphasize stochastic description and behavior [35]. Analytical evaluation is based on the network structure and the parameters of queues [71, Section 1.2]. Model analysis itself is simple for small Queueing Network models. When a limited set of queue variants is used in Queueing Networks, analysis can be performed by mapping to Markov Chains. Since the Markov Chain state space usually grows exponentially with the model size and thus the computation time, we discuss the options for computing results with numerical methods. For a subset of Queueing Networks, product-form networks, analysis can be further simplified. The direct product-form approach and Mean-Value Analysis (MVA) are presented.

Analysis results for product-form networks are primarily average values or probability distributions for performance indicators found in systems running for an infinite time span (and thus steady-state). This in contrast to Network Calculus is used to derive upper and lower bounds on performance figures. Naturally, there are many fields of application where Network Calculus or Queueing Theory can be used and some others where one tool is preferred [55]. However, Queueing Theory is an important approach to system performance analysis [71] and the methods for describing SOA performance developed in the following chapters have to be compared to it. For this reason we will add a model mapping from SOAs to Queueing Networks and present excerpts from the extensive work of Menascé on system capacity planning with Queueing Systems. We will discuss the options on SLA modeling with Queueing Systems based on the already known ParcelSink workflow example from Chapter 2, advantages and disadvantages of average values for SOA and SLA modeling will be pointed out.

4.1 System Model in Queueing Theory

A queue is a very abstract but very powerful system model used in Queueing Theory, it consists of a waiting line and at least one server [11, Chapter 3]. Arriving entities enqueue in the waiting line if the server is occupied. After service, the entity will leave the system without further delay. The original application for Queueing Systems and their networks was to derive the blocking rate of manually operator switched telephone networks [32]. With this historical background the incoming entities are called customers and the active station is the server.

Two processes are connected to a queue: the arrival and the service process. Notation, if not stated otherwise, is adopted from [35, Section 6.2.1]. The arrival process is characterized by interarrival times Y, the service process by service time Z [25]. For several customers both processes form a stochastic sequence, which can be described by a random distribution. Given expectation values $E[Y]$ and $E[Z]$ for the underlying distributions the following properties can be stated [25]:

Arrival Process The arrival rate λ to a system is given by

$$\lambda = \frac{1}{E[Y]} \tag{4.1}$$

On the long-term, λ is the average number of customers arriving per time unit.

Service Process The average service rate has symbol μ and relates to the expected service time by

$$\mu = \frac{1}{E[Z]} \tag{4.2}$$

On the long-term, μ is the average number of served customers if the server is busy.

The distribution types allow the classification of queues into a scheme, in literature the Kendall Notation [8, 32, 35, 54] with six components is common. Well known [8, 32, 54] are queues with exponentially distributed interarrival and service times and a single server, in Kendall notation *M/M/1/∞/∞/FIFO* or shorthand *M/M/1*. The only parameter λ of the exponential distribution determines the mean value $\frac{1}{\lambda}$ and vice versa. It is the only continuous distribution featuring the memoryless property ("*M*arkovian" property) [25]. That is, the expected interarrival time to the next customer's arrival is $\frac{1}{\lambda}$ and the distribution remains exponential, independent from the last arrival time [54]. The history of previous arrivals does not influence the present arrivals, the same holds for service times.

A Poisson process is a counting process modeling a discrete state space in continuous time. The claim to fame is that the times between state changes are exponentially distributed with parameter $\frac{1}{\lambda}$ [70, Section 5.7]. The relationship yields that the number of arrivals to an *M/M/1* queue within a time interval is Poisson distributed. A result in Queueing Theory, also known as Burkes's theorem ([35, Section 6.7.1] and [58, Section 5.5.6]), states a property for departure processes: When the arrival process to a queue is Poisson with rate λ, the service time distribution is Markovian, then the output is also Poisson with rate λ. When Poisson processes are multiplexed or demultiplexed the resulting processes are Poisson again [25], a result that enables a simple analysis of the networks of queues. Further details on Queueing Theory can be found in [8, 11, 32, 35, 54, 71].

4.2 Performance Metrics in Queueing Systems

Performance analysis in Queueing Theory can be founded on analytical methods when a restricted class of queues and networks is used. For *M/M/1* queues the analysis can be done based on arrival rate λ and service rate μ only. This allows extremely fast model analysis with computer programs and the option to solve simpler models by pen and paper. Analysis will yield at least one of the following performance measures:

Response Time S: The total time ("sojourn time") a customer spend in the queue and service section [26]. To speed up the service of a system response times have to be kept as small as possible.

Waiting Time W: Queueing time for a job waiting to be serviced [26], $W = S - Z$ holds.

Population X: The population X of customers in a queue is the customer in the server section and the others in the waiting line.

Utilization ρ: Utilization is the proportion of time the server is busy [35, Section 6.3], thus there is at least one customer in the queue. Let $\pi_n = P[X = n]$ be the probability to have n customers in the system. Then π_0 is the probability for the server to be idle and $1 - \pi_0$ to be busy.

A combined performance measure is traffic intensity, denoted ρ [26, 35]:

$$\text{traffic intensity} = \frac{\text{avg. arrival rate}}{\text{avg. service rate}} \text{ or in symbols } \rho = \frac{\lambda}{\mu} \tag{4.3}$$

When a server operates at rate μ and is $(1 - \pi_0)$ of the time active, it will have a throughput of $\mu(1 - \pi_0)$. For a lossless system in steady state we can assume that the system is stable, in other words input and output are balanced:

$$\lambda = \mu(1 - \pi_0) \tag{4.4}$$

We can replace λ with (4.3) and continue with

$$\begin{aligned} \rho \cdot \mu &= \mu(1 - \pi_0) && \text{Eqn. (4.3)} \\ \rho &= 1 - \pi_0 \end{aligned}$$

and see that for single servers with unlimited queue length traffic intensity is the same as utilization. Since the condition for a stable single server system is $\lambda < \mu$ [8, 26, 35], we also see that $0 \leq \rho < 1$.

Throughput λ: Throughput (rate) is the number of served customers per time interval, therefore it is given by Equation 4.4 [35]. It equals the arrival rate.

4.2.1 Operational Laws

Some simple but useful theorems for single queues and Queueing Networks are in the set of "operational laws" [54, 84]. The relationship between expected values of population, arrival rate and waiting time in Queueing Systems is known as Little's Law.

Theorem 4.1 (Little's Law)**.** *In Queueing Systems with arrival rate λ, average population $E[X]$ and average response time $E[S]$ the following relationship holds:*

$$E[X] = \lambda \cdot E[S] \tag{4.5}$$

Interestingly, the theorem is independent of operating policies used in the waiting line, it is also not influenced by service time distribution. This fundamental relationship in Queueing Systems has been proved by Little [77] in 1961, non-theoretical derivations are found in [11, 35, 54].

If a customer visits a queue v times to finish [80, 82, 84] two theorems apply:

Theorem 4.2 (Service Demand Law)**.** *Let v_i be the number of visits a customer requires to complete queue i. The service demand D_i for queue i is*

$$D_i = v_i \cdot E[Z_i] = v_i \cdot \frac{1}{\mu_i} \tag{4.6}$$

Theorem 4.3 (Forced Flow Law)**.** *For a queue i with average number of visits v_i and arrival rate λ the average throughput is $\lambda_i = v_i \cdot \lambda$.*

4.2.2 Steady-State Analysis

Queue analysis for steady-state requires an expression for $E[X]$, other performance figures are based on it. In Markovian Queueing Systems this can be done via Markov Chain analysis: The number of customers i in queues is mapped to states p_i of a Continuous-Time Markov Chain (CTMC). For single queues the CTMC is known as birth-death chain [35, Section 5.4]). Transition rate from p_k to p_{k+1} (a "birth") is arrival rate λ, from p_k to p_{k-1} (a "death") service rate μ. Deriving $E[X]$ is then equal to finding the steady-state probability $\pi(k)$ of the CTMC being in state p_k. The closed term is

$$E[X] = \frac{\rho}{1-\rho} = \frac{\lambda}{\mu - \lambda} \tag{4.7}$$

A detailed derivation is in [54, Section 31.2] or [35, Section 6.6.1].

With the mean population for a *M/M/1* queue, based on rates λ and μ only, other performance figures can be found easily. When (4.7) is combined with Little's Law in Theorem 4.1 one is able to compute the expected response time:

$$E[S] = \frac{1}{\mu - \lambda} \tag{4.8}$$

Further recasting gives an easy way to determine the average service rate for given arrival rates and response times.

$$\mu = \lambda + \frac{1}{E[S]} \tag{4.9}$$

Example 4.2.1. *HollowEarth has the opportunity to dimension their new system in such a way that the SLO in Table 2.1 is not violated.*

$$\textit{(workload } \leq 8 \textit{ requests per second)} \Rightarrow \textit{(responseTime } \leq 10s\textit{)} \tag{4.10}$$

is relevant for system sizing. To avoid contract breaches of maximum response time the dedicated server is dimensioned for the given worst-case workload.

We model a workload with the arrival rate $\lambda = 8$, for the average response time we use $E[S] = 10s$. Applying these values to Equation 4.9 we compute $\mu = 8 + \frac{1}{10} = 8.1$ as the processing rate for the new server.

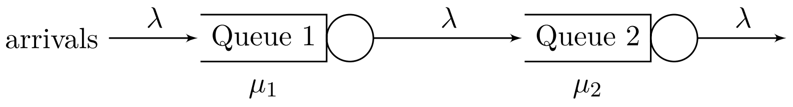


Figure 4.1: Two queues forming a tandem system.

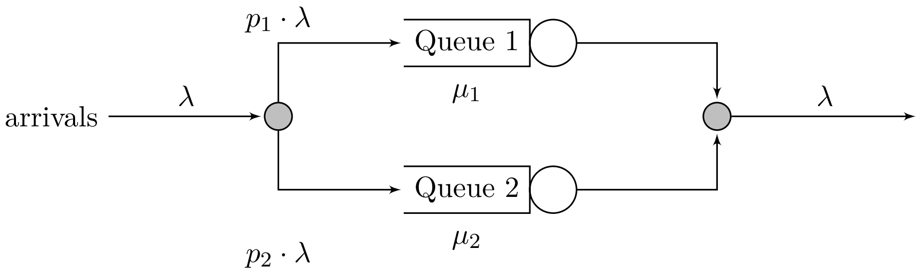


Figure 4.2: Two parallel queues with routing probabilities p_1 and p_2.

4.3 Queueing Networks

Basic queues can be combined to Queueing Networks by providing their output to other queues as an input [35, 71]. This enables the construction of parallel and serial connections [35, Section 1.3.3].

Queues are concatenated to allow customers to traverse the systems sequentially. Let a set of m queues be indexed by $i = 1 \dots m$. They are connected serially if the output of queue i is used as input to queue $i+1$, Figure 4.1 shows a general example. The first queue in the sequence accepts an arrival process with the rate λ and each queue has a service rate of μ_i. According to Burkes's Theorem the outgoing arrival process of each queue is a Poisson process with rate λ, this also holds for the composition.

The other case is the parallel configuration of queues. Parallelization can be realized by distribution (routing) of load. Again let there be a set of m queues with service rate μ_i, $1 \leq i \leq m$ and an arrival process of rate λ. Additionally there is a routing probability $p_i > 0$ for each queue and $\sum_i p_i = 1$ holds. A parallel configuration exists if every queue receives an arrival process with the rate $\lambda_i = p_i \cdot \lambda$. Figure 4.2 shows the basic configuration with a reunion of output processes. The cumulated output process of all m parallel queues has again an average rate λ.

To describe arbitrary Queueing Networks with m queues the idea of routing probabilities can be extended to a routing matrix $P \in \mathbb{R}^m$ with $p_{ij} \in P$ as a routing probability from queue i to j. The distinction between infinite or finite population becomes visible when Queueing Networks are classified as open or closed. In open networks customers enter the system, pass through the queues and leave the system. Closed Queueing Networks keep the customers in a cycle by feeding the output back into the network input. Depending on the network class an analysis method has to be chosen, for example, MVA which works with closed networks. It is also possible to convert an open network to a closed one by connecting in- and output [26, Section 10.1.5]. Within the shortcut a queue is inserted, it's service rate is equal to the arrival rate of the replaced source.

4.3.1 Queueing Network Analysis

In theory, any Queueing Network can be mapped to a CTMC and analyzed with numerical methods. Arbitrary process distributions can be approximated by phase-type distributions [27, 41] that, in turn, can be mapped into embedded Markov Chains [27]. Using such approaches the number of chain states to be considered increases dramatically, a "state space explosion" [27] happens. This results in large CTMCs, when models are detailed and analysis becomes critical in terms of computation and memory requirements, more efficient solutions are needed.

Product-Form Networks

Product-form network analysis extends single queue results to Queueing Networks analyzing each queue in isolation. Performance figures for complete networks are found by result combination. The price to pay for product-form networks and their efficient analysis is that allowed queue variants are limited. Results hold for steady-state only. In literature [25] three important product-form network classes are frequently mentioned: Jackson networks for open networks, Gordon/Newell for closed and Baskett, Chandy, Muntz and Palacios (BCMP) networks as their generalization and extension.

Historically, Jackson networks are the first Queueing Networks with known product-form property [25]. They are open networks of *M/M/1* and $M/M/n$ queues with n servers, all customers in the network are equal. Cycles in networks destroy the Poisson property of arrival processes in general, but analysis of Jackson networks is still possible in this case [35].

The Jackson Theorem [25, 35] can be summarized as follows: Let $i = 1, \ldots, M$ be an index on all queues participating in an open Queueing Network, P describes the structure, arrival process has the rate λ. Service rates are given per server, so μ_i is the service rate of m_i servers in queue i. The arrival rate λ_i to each queue is [35]

$$\lambda_i = r_i + \sum_{j=1}^{M} \lambda_j p_{ji} \text{ for } i = 1, \ldots, M \tag{4.11}$$

Summand r_i stands for arrivals to the queue i from the environment. It is assumed that the network is stable, thus $\lambda_i < \mu_i$ holds. The utilization of queue i is $\rho_i = \frac{\lambda_i}{\mu_i}$ [54, Section 32.2], therefore the expected population is $E[X_i] = \frac{\rho_i}{1-\rho_i}$. Mean network population $E[X_1 + X_2 + \cdots + X_M]$. When the modeling assumptions of Jackson networks are not sufficient for the intended application BCMP [10] networks can be used. Although they still have the product-form, additional combinations of service time distributions and scheduling disciplines with multiple customer classes in open or closed networks are supported [82].

Analysis of closed product-form Queueing Networks involves the computation of a normalization constant, due to the fact that there is no arrival process. Thus, flow balance equations are not completely determined [5]. As a consequence, state probabilities for all possible network states π_i do not sum up to 1. This can be fixed by the normalization constant, but its computation is elaborative [25, 54]. Buzen's convolution algorithm provided the first efficient method to analyze Queueing Networks [25, 82]. It computes the normalization constant in an iterative approach over the number of queues in a network.

Mean-Value Analysis

In many real world applications, response times or queue lengths are of greater interest than state probabilities as an intermediate result. An analysis method giving these figures for closed product-form Queueing Networks is MVA. Instead of solving linear equations for π, MVA is based on the iterative computation of the average queue length in a Queueing Network [84]. When for n customers the average queue length is known MVA gives the result for $n+1$. Starting from $n=0$ results for arbitrary customer numbers are found. As a side effect, computation of normalization constants for closed product-form networks can be avoided [25, 26]. The algorithm is fast and implemented in most software solutions for Queueing Networks analysis [84].

The algorithm presented here is suitable for *M/M/1* queues and is limited to one customer class. For multiple classes, load-dependent servers and other service time distributions extensions exist [84]. Central for MVA is the following, from a customer's perspective encouraging observation one can make at the moment when he or she queues at the end of a waiting line: The number of already waiting customers is independent from his or her arrival or even existence. This is formalized as the Arrival Theorem [26, 71]. As a consequence, the response time for every customer is always the waiting and service time of the other waiting customers plus his or her own service time.

Consider a closed Queueing Network with m queues. The MVA equations are given for queue $i \in 1 \dots m$ and are dependent on the number of customers n in the network. Let $S_i(n)$ be the average response time for a customer and the function $\overline{n_i}(n)$ denotes the waiting line length. The service rate of queue i is μ_i, thus $E[Z_i] = \frac{1}{\mu_i}$.

The first equation is the response time equation based on the components of response time:

$$\begin{aligned} S_i(n) &= E[Z_i] + E[W_i] && (4.12)\\ &= E[Z_i] + \overline{n_i}(n) \cdot E[Z_i] && (4.13)\\ &= E[Z_i] \cdot (1 + \overline{n_i}(n)) && (4.14)\\ &= E[Z_i] \cdot (1 + \overline{n_i}(n-1)) \qquad \text{Arrival Theorem} && (4.15) \end{aligned}$$

By inclusion of visit count v_i and applying the Service Demand Law we get

$$v_i \cdot S_i(n) = D_i \cdot (1 + \overline{n_i}(n-1)) \tag{4.16}$$

For Queueing Networks, routing probabilities $p_{ij} \in P$ can be captured by v_i, too.

The second equation gives the throughput. When all response times are summed up, Little's Law relates the network throughput $\lambda(n)$ to population n:

$$n = \lambda(n) \cdot \sum_{i=1}^{n} v_i \cdot S_i(n) \tag{4.17}$$

and by solving for throughput $\lambda(n)$:

$$\lambda(n) = \frac{n}{\sum_{i=1}^{m} v_i \cdot S_i(n)} \tag{4.18}$$

Using Little's Law once again the waiting line length for n customers at a specific queue is given by the third equation:

$$\overline{n_i}(n) = \lambda_i(n) \cdot S_i(n) \tag{4.19}$$

and applying the Forced Flow Law leads to

$$\overline{n_i}(n) = v_i \cdot \lambda(n) \cdot S_i(n) \tag{4.20}$$

For analysis the algorithm initializes with $\overline{n_i}(0) = 0$ for $i = 1 \dots m$ [25]. Number of visits v_i is the i-th element of vector v with $v = v \cdot P$. Then for $n = 0 \dots K$ customers equations (4.16), (4.18) and (4.20) are iterated for all queues i. The first iteration is trivial since $S_i(1) = \frac{1}{\mu_i}$.

From the algorithm description follows that the computational effort depends on m and K. MVA computes all queue populations in parallel [25], so there is some memory usage but it is a non-issue for realistic models. The only outputs of MVA are the average population and response time of each queue. On the one hand, direct computation of average performance figures is an advantage in terms of speed. On the other hand, state probabilities π_k are not found, so no indication of the maximum queue length is known [84]. When state probabilities for waiting line length are needed modifications of MVA [25] have to be used. Another drawback of average results are the missing options for a transient analysis. MVA cannot determine the time it takes for a Queueing Network to reach steady state, the time to recover from system breakdowns [84] or bounds of response times.

Approximations and Non-Product Form Networks

For many modeling tasks product-form networks are a viable option. Their queues can be combined to networks and several average performance figures can be found by analysis. In return the user has to accept several restrictions and downsides:

- Poisson arrival processes only
- First Come First Serve (FCFS) queueing strategy in Jackson networks, some more, but not always suitable in BCMP networks
- Restrictions on customer classes

An example for a Queueing Networks class that cannot be decomposed for product-form analysis are fork/join networks [84, Section 15.6]. They are used to model parallelism and concurrency that involves synchronization between queues [35]. Figure 4.3 shows the general (sub-) model of a fork/join network with k parallel queues. A request is split at the fork node into k sub-requests, each subnetwork receives one sub-request and processes it independently. The original request is considered to be complete when all sub-requests are completed [84], in Figure 4.3 this is symbolized by the join node on the right.

For analytical fork/join analysis only some results for two *M/M/1* queues are known [88]. In small networks a numeric Markov Chain analysis can be performed. In larger models, fork/join splits can be replaced by routing probabilities again giving product-form, but introducing a semantic error. Otherwise, approximative methods for non-product-form nets can be applied. In [114] Varki presented a MVA extension that allows one to approximate the average response times for closed fork/join Queueing Networks with more than two *M/M/1* queues in synchronization. Networks consist of subsystems i including $K_i \geq 1$ parallel, but equal queues with exponential service time distribution. The response time equation of the modified MVA [84, 114] replacing (4.16) is:

$$v_i \cdot S_i(n) \approx D_i \cdot (H_{K_i} + \overline{n_i}(n-1)) \qquad (4.21)$$

with the harmonic number $H_{K_i} = \sum_{j=1}^{K_i} \frac{1}{j}$ [84] adding the synchronization overhead. The error in approximation has been shown to be smaller than 5% for less than 6 queues in parallel [114]. For further details we refer to [114, 115] as original works. A limitation on Varki's approach is that queues in a subsystem are required to an equal service rate. The method of Duda and Czachórski [44], summarized in [26, Section 10.6.2], allows diverging rates in exchange for higher analysis complexity. Each parallel subsystem is replaced by a single queue with a load-dependent service rate. The parametrization of the queues is done by an approximative method [26] considering the state space. In the end, if the model includes synchronization, approximation errors have to be accepted.

Other reasons for non-product-form networks are [84] non-exponential service time distributions with high variability, queues with blocking [7], sharing of limited resources or priority scheduling. For their analysis, the decomposition approach gives approximative results [25]. Queueing Networks are fragmented in subnets that are analyzed in isolation, the overall solution is deduced from single results. Another approach to be mentioned is the Product-Form Approximation, the model is modified in a way that it approximates the original Queueing Network and is in product-form. For further reading we recommend [26].

Even if the network is in product-form there are still viable reasons to use approximative methods for analysis. When resources are constrained they can save memory and processing time [25]. Solutions are "good enough" for many modeling applications, for example, when design alternatives for early system drafts are explored.

4.4 SOA Models with Queueing Networks

Queueing Systems are widely used for modeling computer systems and also for SOAs. In general, mapping a real world system to queues or a Queueing Network leads to a very abstract model. There are no options to add variables and control structures are unknown. Nevertheless many works on general computer systems and SOA modeling and analysis exist. Exemplary related work on performance modeling in SOA with Queueing Theory are the well-known books of Menascé and Almeida.

In a series of books about Web Technologies [80], E-Commerce [81], Web Service capacity planning [82], a Queueing System-based modeling approach for system performance analysis is adapted to evolving technological approaches. A summary of the three books geared towards QoS in models can be found in [84]. To show the general approach in SOA modeling with Queueing Networks some results from the Web Service book [82] are presented here.

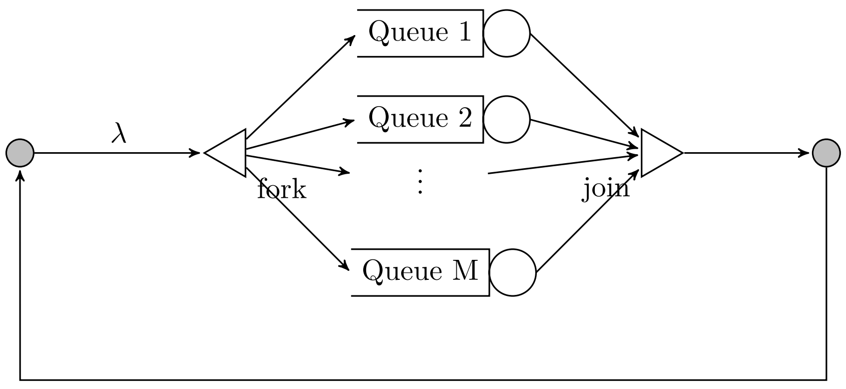


Figure 4.3: A fork/join node with M parallel queues

The Web Service capacity planning approach in [82] involves three kinds of models: a workload, a performance and finally a cost model. In summary, the workload model captures the arrival process by its average rate λ and the service demand each arrival brings to the system. The performance model is the operating speed of the server. Monetary costs are reflected in the cost model, it has a minor role in [80–82] only.

4.4.1 System Level and Component Level Models

In Menascé's books two Queueing Networks performance models are used for systems: the system level and the component level model. A model is applied depending on the perspective and insight to the real system. System level modeling is black-box modeling done by service consumers that do not have access to the system hardware (or detailed statistics). In principle, this model works without a given service rate only by inspection of the input and output process. Since the knowledge of system internals is minimal for customers, the model reduces to a queue. Mapping of Web Service elements is not complex, but differs for infinite (thus open) and finite population (closed) Queueing Networks. Open models are used for workloads sent by an unknown number of client systems to the server, only the arrival rate is of interest. A single service request is mapped to a Queueing System customer, arrivals are considered to be Markovian [82]. For closed models, the number of clients to the server is limited and each client is allowed to send one request to the system. Steady state analysis for simple systems is done with analytical results for closed single server Queueing Networks or MVA. Component level models capture more details and are thus the domain of the service providers. Setting them up requires access to the system. Computer components like CPUs and mass storages are represented by queues and combined to open and closed Queueing Networks. The model mapping for open and closed networks is similar to system level models. Although not explicitly stated, in [82] open networks are analyzed using their product-form property. Performance figures for closed networks are computed with MVA. Complexer models use load-dependent servers and multiple customer classes to represent different workload categories.

Main performance metrics in Web Service models in [82] are response time, throughput, availability and costs. According to [82] a high throughput is favored by the provider. Improving the response time of services is central in Menascé's models as this performance metric is visible to customers. The "end-to-end" response time for Web Services [82] is the sum of network time and service time. Service consumer and provider will have a different perspective on both time intervals. From the consumer's perspective only the complete response time $S_k = Z_k + W_k$ is measurable, the provider has insight into the service process and can measure W_k and Z_k individually. Possible network delays due to router congestions are ignored. For models with a high abstraction level this inaccuracy can be accepted since network transmission times and processing delay differ in magnitudes [16]. As an option, delay stations [58] can model transmission delays.

In [82] Web Service capacity planning does not involve SLAs. Instead an "adequate capacity" for service providers is allocated to deliver a QoS level accepted by customers. The definition of adequate remains open, but is linked to SLA fulfillment.

4.4.2 Mapping of BPEL Structures

For a Web Service workflow a Queueing Networks performance model can be constructed. The structure of the network is equal to the execution graph for BPEL files [85]: For each Web Service named in the BPEL file a system level model, thus a queue, is added to the model. The execution graph structure is captured with routing matrix P.

Sequences in BPEL are mapped straightforwardly to Queueing Networks, the queues are aligned to a serial concatenation maintaining their ordering. If `<switch/>` or other conditional statements in BPEL have to be mapped the lack of variables in Queueing Networks becomes immediately visible. Switch expressions can be mapped to parallel queues under loss of details. Routing based on variables is replaced with probabilities for the outcomes of decisions. Consider a switch statement that branches the workflow W into several subworkflows W_i depending on a expression a. $p_i = P[a \Rightarrow W_i]$ indicates the probability for choosing path W_i based on the value of a. Then p_i is also the routing probabilities to subworkflow W_i. Practical determination of p_i based on a becomes cumbersome, when detailed estimations or even exact results are required.

Synchronization in BPEL is subject to `<flow/>` tags. Mapping to Queueing Networks can be done directly when fork/join network constructs are used. The disadvantage of fork/join nets is their lack of product-form, thus analysis has to be done by approximation (see also Section 4.3.1) or the theoretic option of analysis based on full state space enumeration.

BPEL also allows one to define loops in workflows. Without variables the Queueing Theory is unable to model any counting process, so this BPEL feature can be approximated as a shared resource only. In [82] an average repetition count v_i for a request at a service i is used as a factor to increase the service demand D_i.

Example 4.4.1. *A Queueing Network model for the ParcelSink workflow is shown in Figure 4.4. It includes a network with five queues, one for each service interface, that replicates the basic workflow structure in Figure 2.4. After the address has been fetched the workflow execution does either geocode the input or query the transport papers from the catalog. The decision is modeled by stochastic routing between two parallel subnets. For this example we assume a probability of 40% that the addresses are already in the database, the* *query process is represented by a single queue. Geocoding is subject to two parallel queues in a fork/join construct. After address handling the Queueing Network ends in a single queue for printing the invoice. Since this has to be done two times the repetition count is set to* $v_{printinvoice} = 2$. *It is a system level model since we have no insight to the inner structure of HollowEarth and FlatWorld geocoder services.*

4.4.3 Modeling SLAs with Queueing Networks

While basic workflow patterns can mapped to Markovian Queueing Networks, transfer of SLAs is challenging when an exact or approximative analysis is intended. Process models, as shown in the previous sections, are solely based on rates λ and μ. Hence, input bounds found in SLAs have to be modeled with these parameters. Naturally, their expressiveness is limited. Performance Guarantees, response times for instance, can be mapped to analysis results. As these results are expectation values again, Markovian Queueing Networks are not a perfect modeling tool for SLAs with limits.

When the modeler has full access to the infrastructure of used SOA services, workload and service process can be determined with statistical methods (c.f. [70]). For customer perspective models SLAs are the only source of information, the included SLOs can be used to extract performance figures. Bounds to which the system has to be conform, even in worst-case situations, may be used as average input parameters. Anyway, thanks to Little's Law missing service rates are not an obstacle.

Example 4.4.2. *To map the SLA for the ParcelSink workflow we use the Queueing Network constructed in Example 4.4.1 and assume* M/M/1 *queues.*

For analysis, an arrival rate to the network input has to be specified. According to the long-term contract in Table 2.1 we have $\lambda = 12$. *Further, service rates for all queues are required. Rates* $\mu_{FetchAddress} = 15$, $\mu_{Catalog} = 9$ *and* $\mu_{PrintInvoice} = 25$ *are known due to internal hosting. For the contracted geocoder services we extract the rates from their SLAs. In Example 4.2.1 we derived a service rate of* $\mu_{HollowEarth} = 8.1$ *for the HollowEarth service. Equally, the SLO in the SLA for FlatWorld stating*

$$\textit{(workload } \leq 15 \textit{ requests per second)} \Rightarrow \textit{(response time } \leq 5s\textit{)} \tag{4.22}$$

results in $\lambda_{FlatWorld} = 15$ *and* $\mu_{FlatWorld} = 15.2$. *Figure 4.4 includes the rates, too.*

Only long-term contracts have been considered in the example above. Simultaneous inclusion of short-term arrival bursts is not possible in the Poisson arrival process.

4.4.4 SLA Validation

With a fully parametrized Queueing Network for a SOA workflow, SLA validation reduces to analysis and subsequent comparison of results and target figures. Still, within this simple blueprint, there is enough tolerance to produce errors or to misunderstand results. An analysis method has to be chosen by network type and should consider whether exact or approximative results are suitable. Soon one will see that some kind of model transformation is required first, the likelihood to find an unsupported model feature is high.

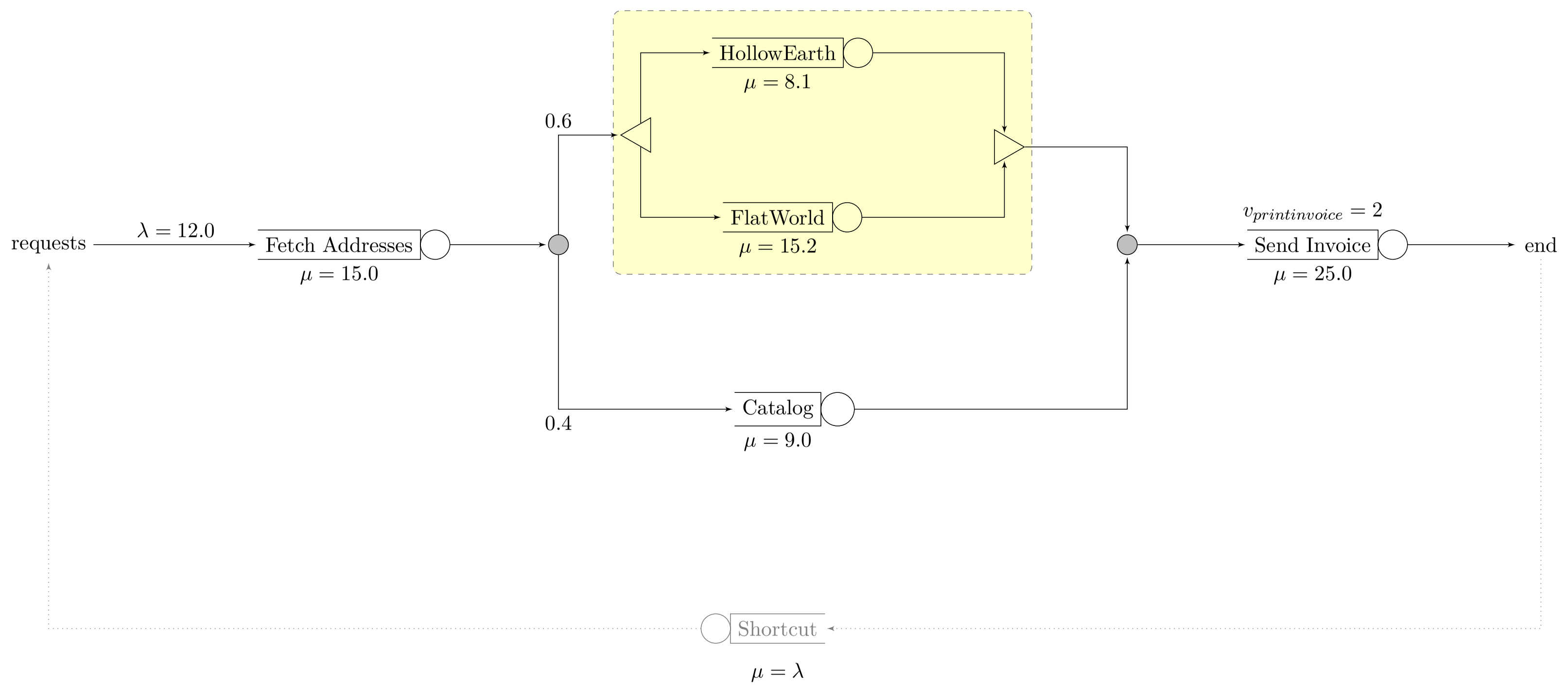


Figure 4.4: ParcelSink workflow modeled with an open Queueing Network. For MVA, it is closed by a shortcut (gray).

Example 4.4.3. *To validate the SLA for the ParcelSink workflow we analyze the Queueing Network for the response time. Intention of our model is to show that SLO*

$$\textit{max. } 12 \textit{ requests per second } \Rightarrow \textit{max. } 5 \textit{ seconds response time}$$
$$\lambda \leq 12 \Rightarrow E[S] \leq 5s$$

holds on long-term.

Analysis includes some compromises due to the fork/join of HollowEarth and FlatWorld.

- *In a first approach we bring the network to product-form by substituting the fork/join with routing probabilities $p = 0.5$ for each service. Solving the Queueing Network with the help of the Jackson Theorem and MATLAB gives an average system delay of $E[S] = 2.5211$.*
- *When the model with routing is transformed to a closed Queueing Network, MVA gives us $E[S] = 2.350691$ excluding the shortcut queue. We use the JMVA component included in the `Java Modelling Tools` [22], on the recommendation of [26, Section 10.1.5] we populate the network with 100 customers.*
- *Varki's MVA extension approximates $E[S] = 3.1617$ excluding the shortcut queue. Since synchronized queues have to be equal we set $\mu_{HollowEarth} = \mu_{FlatWorld} = 8.1$ assuming HollowEarth is the bottleneck. Again, population is 100 customers.*
- *For guidance, simulative analysis paying attention to routing and fork/join constructs gives $E[S] \approx 3.1$ (JSIM component of the `Java Modelling Tools`).*

Compared to the claimed response time of 5 seconds there is still room left and, based on the Queueing Network analysis, the given SLA is valid for the ParcelSink workflow.

With the methods and options of Markovian Queueing Networks the example above gives a positive outcome for the SLA validation. The expected system time is within the bound given by the SLA implication. But comparing expectation or average values to maximum values in guarantees is pointless. It is in the nature of averaged figures that some individuals are below average and some are higher than average. There is no information in the analysis result on how the system response time variates and if or how often the response time bound of 5 seconds is exceeded. In the opposite, bounds of SLOs are used to determine the unknown service rate μ for corresponding services. From worst-case bounds the average behavior is postulated but no information on its variance is included in the model. This is a fundamental error when Markovian Queueing Networks and bounds are combined since there is simply no option to model upper or lower bounds with average values.

Example 4.4.4. *In Figure 4.5 we provided a histogram of response times for the ParcelSink workflow recorded at a simulative analysis as we will present in Chapter 5. We used exponential arrival and service time distributions with the parameters in Example 4.4.3. A majority of service requests finished within the given response time guarantee of 5 seconds, but obviously processing times exists that violated the performance contract. The positive result derived with product-form analysis in Example 4.4.3 is invalidated.*

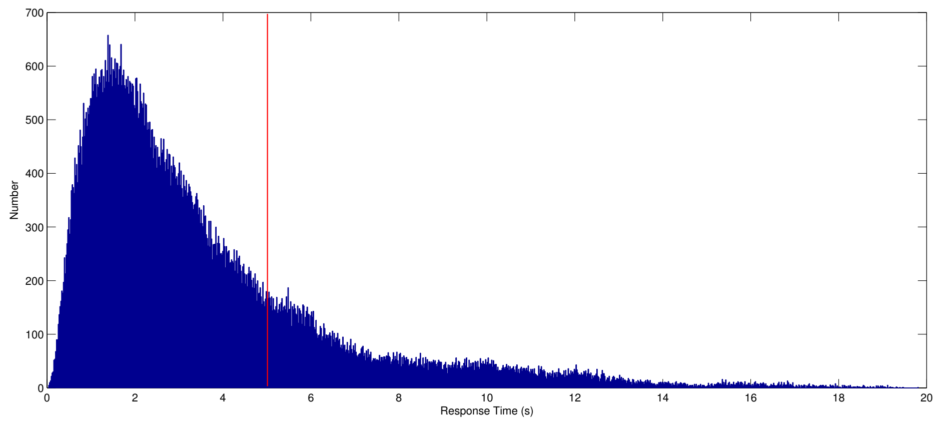


Figure 4.5: Response Time Histogram for ParcelSink Workflow: Many service response times violate the 5s response time contract.

SLAs Conflict with Poisson Assumptions

A fundamental assumption necessary for queues or Queueing Networks with exact solutions are workload and service models with identically and independently distributed (IID) interarrival and service times. This enables the use of Poisson processes with average rates as the only parameter, closed representations for steady-state results for single *M/M/1* and *M/M/m* can be easily derived. According to Burke's theorem the output of Markovian queues is also Poisson, this allows one to construct Queueing Networks that even include (de-) multiplexing of customer flows that are also Poisson. In the end, the classes of product-form networks allow an efficient and exact analysis.

The problem in, but not limited to, performance models for SOA is that there are no IID interarrival and service times in real systems. While in many modeling applications this simplification can be accepted in exchange for product-form property this is a severe limitation when working with SLAs. Request arrivals tend to be grouped into bursts [82] visible as peaks in the short-term arrival rate. They are said to be correlated. Arrival bursts lead to occupied waiting lines and thus in high system population and increasing response times, bounds specified SLAs are likely to be violated in systems under load. Assuming IID arrivals for a system with correlated arrivals leads to an underestimation of required resources. Also a reduced service rate is often not a phenomena limited to a single customer. Service levels degrade for a time period, for instance, when a background maintenance job blocks system resources. The effect on response times is similar to peak arrival rates.

Since bursts and variations in arrival and service processes are the norm in real-world SOAs, the inclusion of peak rates can give better results. In theory, infinitely small interarrival or potentially infinite service times are possible due to the memoryless exponential distribution, hence Poisson processes do not explicitly exclude such batch arrivals or periods of degraded service. But when modeling with bursty arrivals the features to specify or bound peaks are missing for Poisson processes. Moreover, when request bursts or low service phases are accepted in a way specified in SLAs the modeler is interested in including them. As a consequence, a more detailed workload and service model has to drop the memoryless exponential distribution to allow such variations. Other distributions, such as Normal or Erlang can be included in workload and service models resulting in a *G/M/1* or even *G/G/1* system without product-form property. This also conflicts with the application requirement of Varki's MVA extension for fork/join queues, it requires all parallel queues to be Markovian. Hence, including details on processes leads to loss of Poisson property and thus the corresponding Queueing Networks are neither in product-form nor efficient approximation algorithms can be applied. Remaining analysis options are an elaborative state space Markov Chain analysis or fallback to simulation models.

An intuitive option to model bursty workload is to add a safety factor to the arrival process to cover short-term variations without loosing efficient analysis. For instance, one could size service capacity for an arrival rate of $x \cdot \lambda$ with $x > 1$, but give service guarantees to the customer for a rate of λ only. In terms of SLAs compliance we would be on the safe side, but at the cost of running an over-sized system. To avoid over-provisioning with Web servers or SOA services Menascé introduces a burstiness factor [80–82] to capture high but short lived arrival peaks. Similar to the previous proposal it increases service demand D, but the factor is based on statistics of the arrival process. It is the fraction of the measurement interval with an arrival rate that exceeds λ [80]. The burstiness factor helps to adjust workload to reduce missed deadlines, but it has to be determined at an already existing system (or simulation model). Due to estimation and averaging, is not suitable for guarantees.

Errors in Routing Probabilities

In the ParcelSink example we assumed routing probabilities $p_{geocoding} = 0.6$ and $p_{catalog} = 0.4$. This resulted in a workload reduction to the geocoders ($p_{geocoding} \cdot \lambda = 4.8$) and the database service ($p_{catalog} \cdot \lambda = 7.2$). However, these routing probabilities are average values again and are likely to be undercut or excelled during normal system operation for limited time intervals. The model analysis does not give any information on the system performance when these probabilities variate. Keeping the worst-case in mind additional model analysis steps have to be provided with all workload taking one of each possible routes.

Example 4.4.5. *The workload split of 0.6 to 0.4 in the ParcelSink workflow is an average estimation only. To be prepared for short-term fluctuations or a higher percentage of returning customers that will use the catalog more frequently two worst-case scenarios are analyzed. Assuming the worst-case workload to the geocoders (*$p_{geocoding} = 1.0$ *and* $p_{catalog} = 0.0$*) the fork/join subnetwork is exposed to the full workload* $\lambda = 12$*. Since the HollowEarth service limits the performance of the fork/join construct the subnetwork becomes unstable due to* $\lambda > \mu_{HollowEarth} = 8.1$*. As a consequence, the complete Queueing Networks is unstable, service requests will pile up and the response time is* ∞*. An equal effect is visible when the other extreme case of* $p_{geocoding} = 0.0$ *and* $p_{catalog} = 1.0$ *is tested. The catalog service has to process requests with rate* $\lambda > \mu_{catalog}$ *and becomes unstable, too.*

Quality of Approximations Unclear

We add another point of criticism on SOA Queueing Network models including SLAs: Approximative results cannot provide reliable service guarantees. When MVA is used to analyze Queueing Networks, it gives exact results for product-form networks. For workflow patterns using synchronization, MVA extensions give approximations only. Although Varki has demonstrated the approximation error is about 5% for her MVA extension compared to exact results [114] for most models, MVA does not include any information in general on how good an approximative result for a special model is. There are no confidence intervals as estimation quality indicators. When SLAs are based on them the combination with monetary interests might be risky. Other analysis methods that can provide confidence intervals such as simulation require more detailed models and processing time.

5 Simulation

We have seen that modeling and analysis of large SOAs is possible with Queueing Networks. Mapping of a Web Service to a single queue is straightforward and BPEL workflows can be replicated by Queueing Networks. The size of models is potentially not limited, as long as the network has a product-form or can be approximated by a product-form network. Analysis scales linear with the number of queues. The disadvantage of Queueing Networks is that, even in Jackson or BCMP networks, their descriptiveness for modeling is limited. Neither system downtimes nor batches of customer requests in arrival flows as common in SOAs can be modeled. As a consequence, worst-case situations are hidden for analysis, thus SLA validation with Queueing Networks is unreliable. When special attributes are added to the model, for instance synchronization required by BPEL workflows, the product-form is lost and the quality of approximative solution techniques becomes unclear. Detailed workload and service models with non-Markovian distributions further reduce options in approximative analysis. Numerical analysis is, even for small models, highly time- and resource consuming and thus a theoretical option only. Another, only briefly mentioned issue in the last Chapter is the lack of variables in Queueing Networks. Decision operators and loops in workflows have to be replaced with routing probabilities, a method that brings additional inaccuracy into models and results.

Discrete Event Simulation is a form of state based model analysis [8, Section 1.10] and a far more powerful analysis method than Queueing Theory. Its applicability is not limited by model classes and, when random variables are used, they can have any distribution. Moreover, models can potentially include any feature that can be formulated with programming languages. Variables, loops and communication between elements can be used, hence SOA models can be by far more detailed than Queueing Networks. Instead of giving average results simulative analysis takes a different approach by scanning the (unlimited) state space to explore system dynamics. Approximative results can be given with the help of statistics and without the need for full state space evaluation. When average figures are computed their variance and other moments can be provided to show the quality of estimation. Discrete-event simulation is based on a time advance routine [70, Section 1.3.1] that transforms the model state and advances the simulation or model clock. The sequence of values of model state variables at each occurred event in a simulation run is called trajectory. By accessing the trajectory during simulation runs or by inspection of recorded trace files, even rare or extreme events normally hidden beyond average values can be found.

The flexibility of simulative analysis allows customized statistics of model elements. Good software support with user-friendly interfaces extends the user base. Therefore, simulation is a candidate for SOA performance analysis.

5.1 *ProC/B* Modeling Language

Simulative model analysis is a computerized task performed by simulation software. Simulators include well optimized implementations of the time advance loop and the event queue. A graphical user interface, component libraries and statistic functions add further convenience. Several free and commercial software systems exist for discrete event simulation. In this work the *ProC/B* tool set [14–17, 60, 116, 120] in combination with *OMNeT++* [112, 113] will be used.

The process chain paradigm by Kuhn [65] allows to model process chains in the field of logistics. A formalization of process chains was developed in the Collaborative Research Center "Modeling of Large Logistics Networks" (CRC 559) [33]. *ProC/B* [14, 120] is a graphical language [12] to model and analyze elements in logistic networks by analytic and simulative methods. It adds several features of Queueing Systems to process chain descriptions to formalize process chains [14]. This includes the sharing of and competition for limited resources, as well as, convenience modeling elements.

Initially *ProC/B* models were analyzed with the *HIT* simulation environment [21]. Unfortunately, compiler support for the used Simula programming language has declined on modern architectures and operating systems. For this practical reason *ProC/B* was implemented from scratch [15] using the *OMNeT++* simulation environment [112, 113].

5.1.1 Basic *ProC/B* Elements

The modeling language *ProC/B* is based on Process Chain Elements (PCEs) symbolizing activities and Function Units (FUs) that capture resources and structures of modeled systems.

Process Chain Elements

PCEs describe tasks in a process chain, their representation in *ProC/B* is a pointed rectangle. The workflow of a system, called a process chain in *ProC/B*, is modeled by connections of PCEs. The most simple connector is a handover of processes from PCE A to PCE B when A has finished the service. This is symbolized by directed arcs and equals a routing probability of 1 in Queueing Networks. In detail, *ProC/B* includes three subtypes for PCEs:

Delay PCE Processes are delayed for a given deterministic or random time period. For logistical models this can be exploited to model the transportation time of goods.

Code PCE Executes a block of programming code to manipulate variables attached to the process. Programmable PCEs allow one to include new features used to express model behavior by code and not with standard elements. For example, new measurement methods or custom trace writers can be added.

Call PCE Request a service or resource from a FU. Service calls block the process until the service is finished. To perform activities PCEs may require resources with limited availability in the model. If a PCE cannot acquire all necessary resources to process a model object, the object is enqueued in a waiting section.

Loop PCE Repetition of workflows enclosed by two PCEs marking the start and end of a loop.

We will introduce some of these elements with examples, for a detailed description of all language elements we refer to [13, 120].

Process chains are not limited to sequenced activities only. For parallel compositions AND and OR connectors exits that accept two ore more subworkflows. OR connectors select a single subworkflow to continue for processes. They can be configured to act as a probabilistic router similar to Queueing Networks. The boolean operation mode selects the subworkflow based on a boolean expression as known from general programming languages, a substantial advantage to Queueing Networks. The semantics of AND connectors is the synchronization of parallel workflows. Arriving processes are forked at opening connectors by creating copies and sending them to each subworkflow. Closing connectors exist to join processes that finished their sub-workflows. AND constructs are comparable to the fork/join queues in Queueing Networks, but they offer complete freedom on arrival process properties and the inner service structure.

Source and sink are specialized PCEs that mark the beginning and end of a process chain. The source generates entities (customers, containers, trucks, etc.) to be handled by the process chain. In detail, interarrival times for the entities can be specified by several types of random distributions offered by the *ProC/B* implementation. Modelers can choose from constant, exponential, normal and other distributions.

Function Units

Function Units offer services (“functions”) to be used by PCEs. Two types of FUs exist: atomic and composed FUs.

The service of an atomic FU is to offer resources to processes. Server FUs are comparable to servers in Queueing Theory, offering an abstract service to PCEs as their resource. For the service duration a server FU is blocked and other processes have to be enqueued (default: FCFS). In atomic FUs the service process are modeled and one can choose between several service time distributions again. With this semantic the allocation of a server can also be interpreted as binding of a process to a limited resource. Servers can be configured to support preemption or processor sharing, for details we refer to the language specification in [13]. Resources can also be modeled with storage and counter FUs. Storages offer (limited) capacity while counters can be incremented and decremented down to zero.

Composed FUs capture the structure of the model by offering encapsulated process chains as a service. In this way *ProC/B* implements reusability of model parts and hierarchies. To construct more complex models composed FUs can be created by specifying one or multiple service interfaces and by implementing them with a process chain. Call PCEs from higher hierarchy levels can use these interfaces in an identical way to atomic FUs. The only noteworthy difference for process chains implementing the service interface is their lack of an own source.

A distinctive feature of *ProC/B* is the modeling of resource sharing, several PCEs can access the service of a single FU at the same time. Depending on the queueing strategy and service capacity of the FU the performance figures will variate. A classic example for resource sharing within the context of logistic models are truck unloading processes (several PCEs in parallel workflows) that have to share a single forklifter (a single FU).

5.1.2 Analysis and Measurement Streams

If a simulation model includes random variables, two runs with the same input configuration (but different random generator seeds) will result in two different trajectories and output values. In fact the output itself is a random variable [8] likewise the input is random. The trajectories will also differ in the gathered statistical data for the model run. To analyze stochastic models many time-consuming simulation runs of the same model with different seeds are necessary followed by a statistical evaluation.

To get performance figures the user can add statistical operators called measurement streams to *ProC/B* models via Graphical User Interface (GUI). Measurements are recorded in simulation runs at FUs when a process chain offered as function is called or finishes. At any FU, the model itself is also a FU, utilization, population, response times and other figures known from Queueing Theory can be measured. Next to the averages a *ProC/B* analysis gives the standard deviation and the confidence interval for each metric. When built-in statistics are not sufficient for a modeling application the user can access detailed trace files recorded on simulation runs and analyze them with his or her own tools.

However, extracting reliable results even from simple models requires at least two scarce resources: experience and patience.

Example 5.1.1. *In Example 4.2.1 we computed the service rate for the service provider hosting the HollowEarth geocoding service using an* M/M/1 *queue. For an arrival rate of* $\lambda = 8.0$*, a service rate* $\mu = 8.1$ *the expected response time is* $E[S] = 10$ *seconds. The queue is highly utilized with* $\rho = \frac{8.0}{8.1} \approx 0.98$*. We are going to compare these analytic findings to results estimated by a ProC/B model.*

Table D.1 shows simulation results for increasing simulation time. Averaged measurements for long simulation runs (1000000 and 1 Million seconds) conform to analytic findings on expected response times and population. However, short simulation runs show different results: In the first experiment with 10000 seconds model time the average response time is 7.13 seconds while we sized the system for 10 seconds. The experiment with 20000 seconds model time gives a better result. An advantage of simulative analysis is that the approximative quality is visible in the width of confidence intervals. Considering the 90% confidence interval for the first run the average response time of the HollowEarth service is in the interval $[5.578, 8.691]$*. So, according to the first run, the interval is even below the expected result and with 90% confidence the system is faster than 10 seconds for each request in average. However, further experiments show that our assumption of 10 seconds response time does still hold. Longer simulative analysis narrows the interval, but the result is still approximative and subject to random noise.*

The reason for such misleading results is the poor estimation quality of short simulation runs combined with a high system utilization. Queues under heavy load converge slowly towards steady-state, the observed standard deviations are high. Increasing simulation time does indeed reduce the width of confidence intervals, the results become more reliable.

Removing measures taken in the initial, transient model phase improve the results further. A primer on how to identify the transition from a transient phase to steady-state in non-terminating simulations is given in [70, Chapter 9] or [8, Chapter 11.5].

The reader might argue that the example is chosen in a way that the utilization of the system is high and the simulation performs bad. The author has to confess this is true, but, it is based on a simple reason: In the next section we will describe how to map complete SOA workflows to *ProC/B* to analyze systems and to do SLA validation. Since we are interested in performance figures for systems in worst-case situations we will have to observe highly utilized systems to validate given performance guarantees. Slow convergence to steady state values will be the general case and not the exception for simulative SLA Validation in SOA models. Moreover, even for the average system load on a SOA model, some services might be highly utilized. When this is not visible or known beforehand, slowly converging models can be identified by analysis only.

Example 5.1.2. *We reconfigured the* M/M/1 *model from Example 5.1.1 to a lower arrival rate* $\lambda = 5.5$ *resulting in a utilization of* $\rho = \frac{5.5}{8.1} \approx 0.68$. *Except for the arrival rate, nothing else is changed. Using results from Queueing Theory we expect a population of 2.1154 and thus, an approximative response time of 0.384 seconds in steady state. The results of simulative analysis are given in Table D.2. Again the simulation result conforms the analytical findings. Moreover, due to lower utilization, the numbers converge towards the expected result in shorter time. Even for the short run with 10000 seconds model time the numbers are usable as an approximative result. Confidence intervals are tighter from the beginning.*

Simulative analysis for a given model is a task performed best by computers, so it is convenient to increase model time for better results by adjusting the experiment parameters once. Unfortunately this does also increase the runtime of the simulation program itself. The waiting time for reliable results can be significant and hence renders simulation a bad choice when fast analysis is required.

Example 5.1.3. *We added the approximative CPU Time for each simulation experiment to Tables D.1 and D.2 when the model is analyzed using OMNeT++ and the author's Personal Computer (i5 quad-core running at 3.3 GHz, 8 GB RAM). Even for rough approximative analysis of simple* M/M/1 *queues several seconds up to minutes of CPU time have to be spent. More confidence is given by even longer analysis, but the gain is not linear to CPU time.*

Since we emphasized functional correctness for the *ProC/B* implementation instead of performance, these numbers should not be understood as a general statement on the duration of simulative analysis. Nevertheless a trend is visible when the estimation quality is improved and thus, the confidence intervals are narrowed: simulative analysis is time-consuming.

5.2 SOA Models with *ProC/B*

The *ProC/B* modeling language can be used to model SOAs and analyze them using simulation. When workflows are compared to logistic process chains at an abstract level, the ideas are quite similar, hence many features in the *ProC/B* language simplify the creation of detailed SOA models. Structure and behavior of workflows can be analyzed for performance and dependability [16]. We present a mapping from BPEL workflows to *ProC/B* models here. Service usage is represented by PCEs and a Web Service finds its counterpart in functionalities offered by FUs. For details on creating SOA models with *ProC/B* we refer to articles [16, 17, 116].

The mapping, as other mappings for other modeling methods presented in this work, does not consider transmission times of SOA requests. If such a level of detail is required, additional PCEs and FUs can be added to cover the network structure. On the practical side, such a model would be very detailed and does not separate network structure from the workflow. For *OMNeT++* model libraries including TCP/IP packet transmissions and network hardware exists, for example the INET framework (an overview on frameworks is given in [113]). It allows modeling of host-to-host communication in networks. In [16] the INET framework and the *ProC/B* model world was brought together in two tier models. High request rates lead to network congestions in the INET model visible as an increase of response times in the *ProC/B* world [116].

5.2.1 Model Mapping

To use *ProC/B* for SOA modeling, a mapping of workflow structures for process chains has to be done. We summarize the mapping of [16] here. Beginning and end of a SOA workflow are marked by *ProC/B* sources and sinks. The source models the service request arrival process, thus for Web Services each generated entity symbolizes a SOAP message triggering the workflow. A request is considered as finished when the entity arrives at the sink.

The workflow structure and sequence of service calls is modeled with PCEs. When a Web Service is described with BPEL, for each abstract service interface a Call-PCE is added to the model. Since a *ProC/B* process entity waits at Call-PCEs until a reply is received, the synchronous `<receive/>` and `<reply/>` mechanism in BPEL is simulated. Sequencing Web Service calls with `<sequence/>` is a simple concatenation of PCEs to a process chain in the given call sequence. For other BPEL statements *ProC/B* connectors can be used:

<switch/>: OR-connectors can reroute a process entity depending on *ProC/B* variables or random.

<while/>: Loops can be realized in *ProC/B* with Loop PCEs.

<flow/>: Parallel execution and synchronization is the domain of AND-connectors.

Infrastructure of service providers, as well as, composed services are captured with FUs. Web Service workflows can be offered to other users as a Web Service again, so if the internal structure is known, it can be modeled as a process chain. With composed FUs their functionalities can be included in other process chains as a sub-workflow. When it comes to an atomic service provider or when no further details on a service are known server-FUs are used.

Some advice on the multiplicity of FUs when connected to PCEs should be considered. For the modeling of service providers with FUs at least two variants exist from which can be chosen. Both differ in model semantics and it is up to the decision of the modeler which variant suits best. In the first variant, all PCEs for workflow elements that send a request to the provider are connected to the FU in a $n : 1$ manner. This model's resource sharing and analysis will show the performance degradation of the provider on multiple customer access. We recommend this variant, if the infrastructure of the data center providing the service is known in detail and all service requests are processed on the same machine. The second variant is to replicate the service providers FU and to connect PCEs in a $1 : 1$ manner to the set of providers. As an alternative, the capacity factor in server FUs can be adjusted. In this way resource sharing is not included in SOA models but the aspect of service guarantees per contract is taken into account. Of course, both variants can be mixed in a single model.

Example 5.2.1. *Figure 5.1 includes a ProC/B model for the ParcelSink Ltd. workflow defined in Listing 2.1. Model `ParcelSink_Main` specifies the structure of the workflow `transport` with six Call-PCEs, five servers and two kinds of connectors. The workflow is intended to be called by customers, so no source specifies the arrival process in this ProC/B model. Instead a virtual source (and sink) acting as some kind of function interface is used. Addresses for a packet delivery are collected by PCE `fetch_addresses`, it uses the resources of the server-FU `Webserver`. In the next step the workflow instance hands the request over to a probabilistic routing connector. With a probability of 60% the addresses have to be geocoded, otherwise the catalog service is used. In the geocoding section the BPEL file requires a synchronized geocoding of start (at `HollowEarth` server) and destination (at `FlatWorld` server) addresses, so a ProC/B AND-connector forks and joins the request. As a final task the bill is send by PCE `print_invoice`, in the BPEL file the service is called two times by a single request. In this model we have chosen to use two CallPCEs connected to the same `PDF_generator` server instead of Loop-PCEs. The effect due to resource usage at the server is the same, but the loop variant comes with a syntactic overhead (two PCEs, additional process variables). When done the successful service request is sent back.*

5.2.2 Inclusion of Quantitative Requirements from SLAs

In the BPEL mapping to *ProC/B* we omitted the arrival and service process since necessary values have to be extracted from the SLAs for each service. When SLAs are to be validated the sources generating arrival process can be parametrized with the user constraints found in included SLOs. It is best practice in *ProC/B* models to separate the workload model from the remaining workflow encapsulated in a composed FU, since measurements can be taken at FUs only.

Example 5.2.2. *Service requests are send to the ParcelSink workflow by external customers. In the ProC/B model this is expressed by an additional model layer enclosing the ParcelSink model. Figure 5.2 shows the top level of our ProC/B model. The source generates a request flow (exponential distributed, $\lambda = 12$), the PCE `commission_parcel_service` sends the requests to FU `ParcelSink` offering service `transport`. The internal structure of the FU is given by the workflow shown in Figure 5.1.*

Service processes are modeled with distributions and parameters in the PCEs representing service usage as depicted in Figure 5.1.

Example 5.2.3. *The ParcelSink workflow model is used to validate the SLA implicating a response time of 5 seconds when the load is limited to 12 requests per second. Parameters and distributions are equal to the Queueing Network analyzed in Example 4.4.3, Table D.3 shows the found average figures as a result. Based on Queueing Theory results we expect a turnaround time of approximately* 3.1617 *seconds. Simulative analysis approximates similar performance figures for the workflow. The* 10^6 *seconds experiment gives an average response time of 3.1722 seconds and* $[3.1164, 3.2280]$ *as a confidence interval. However, we had to wait for this result for about 53 minutes. Shorter simulation runs give weaker results: for the 10000 seconds run the observed response times average to 3.3683 seconds, the 90% confidence interval extends to* $[2.9009, 3.8357]$.

Simulative analysis can be, as shown in this example, very time-consuming to get reliable average results comparable to known analytic findings.

Detailed Process Models

While in Queueing Networks we restricted models to Poisson arrival and service processes to enable a fast analysis, simulation offers the freedom to choose distributions without any effect on analysis options. While Exponential distributions work well for drafted systems, Gamma, Erlang and Weibull distributions may lead to more realistic simulation results. Of course, chosen distributions will affect analysis results, so the modeler has to select carefully. In case of the SLAs performance modeling scenario no information on possible distributions is available, so assumptions have to be made. This adds another source of errors to simulation models. For further reading on random distributions and their application in simulation models we refer to books [8, 70].

More detailed distributions often require more than one parameter which is, with limited information, difficult to find. Extensive knowledge on the real process is required. When no further distribution parameters are known they can be fitted to traces that can be captured only from already existing systems. For SOA models we again face the problem that access measurement to systems providing services is limited. SLAs cannot be used as a detailed source for these figures, since SLAs and their description languages have no options to define such distribution characteristics.

For network traffic even more advanced models exist. Contrary to previous examples, packet and job arrivals are not IID but correlated in realistic systems [80, 82]. In [62, 64] Kriege presented a Markovian arrival process model that involves moment fitting. For fitting input models to trace data a software toolkit exists [20]. The traffic generator has been implemented for *OMNeT++* [63] and, in combination with the *ProC/B* library, it can also drive process chain models.

5.2.3 SLA Validation

Mapping SOA workflows to *ProC/B* is straightforward, since the global system ideas are quite similar, but adding quantitative requirements for model-based SLA validation takes more effort. SLA validation in *ProC/B* accepts contract violations as a result of the stochastic behavior of complex SOAs. Moreover, occurrences of SLA violations are considered as random variables whose expected value can be computed by simulative analysis [17]. An SLA is validated by showing that performance figures of interest are below a given bound, for example, an accepted failure rate. Since steady-state is assumed, but simulation times are naturally limited this is done with confidence interval for a significance level. The interval has not to overlap the given bound for successful validation.

Either one- or two-sided confidence intervals are used [17]: When SLA violation can be decided per request and the SLA defines a maximal failure probability two-sided confidence intervals are used. SLOs given by average values are validated with one-sided confidence intervals.

While computing average results and their confidence intervals is a basic functionality in *ProC/B*, the binary decision on a SLA violation for single requests needs to be added. In [17] a method blueprint was presented how to measure SLA violations in *ProC/B* for aspects of workload, response time and reliability. The SLA model has three groups of quantitative requirements, each with an average value, as well as, constraints defined by sliding windows covering the processes that finished last:

sla.load captures the workload sent to a service. Restrictions can be implied on minimum interarrival times or the number of arrivals within an interval.

sla.perf captures the response times. Similar to the arrival process limits can be imposed on maximum delay or the fraction of processes that finish in time.

sla.avail is availability. A certain percentage of service calls have to be successful.

To support the necessary measurements a modeling pattern is used: Each service under measurement is encapsulated in a FU and a second FU layer embeds the service FU and acts as a measurement proxy. SLA violations are decided right in the proxy FUs using CODE-PCEs and *ProC/B*'s own scripting language. For details on used *ProC/B* language options we refer to the original paper [17]. No direct support in *ProC/B*'s SLA model is given for implications found in WSLA files. Contract breaches on arrival rate and delays are visible, but no correlation between both. However, since *ProC/B* can be extended with programming languages or by additional code in the *OMNeT++* library, such functionality can be added manually.

Example 5.2.4. *We checked the ParcelSink simulation model for response time violations by evaluating SLA expression* $sla.perf := (t_{max} = 5s)$*. In this example we require a quite low conformance level of at least 80%. The results are appended to Table D.3, a histogram plot for the short first run was already given in Figure 4.5. The probability of exceeding the response time is about 0.1793 with a confidence interval (90%) of* $[0.1789, 0.1797]$ *for the* 10^6 *seconds experiment. Based on that figures the workflow is consistent with the requirement of less than 20% timed out requests.*

Supplemental to the use of confidence intervals in SLA validation Example 5.2.4 and 5.2.3 give us two more insights: Relying on average performance figures only as we did for Queueing Networks leads to wrong assumptions on service conformance. Although the average response time of about 3.3 seconds has some comfortable gap to the upper bound of 5 seconds there are many outliers that violate the SLA. Simulation gives us these information on variance and thus, on the confidence intervals allowing the classifying of the results. Here, with a failure rate of 20%, it is unlikely that customers would accept this service. Second, with short simulation runs the user cannot assume that he receives results for steady state. For the shorter runs our requirement on 20% conformance was not met and confidence intervals also showed a significant distance.

The problem of long simulation runs also occurs in situations when the probability of rare events has to be determined with high precision. In SLA validation this can be the task of to show that very long response times are very unlikely. When workload and processing speeds are driven by random processes, there will be many requests that completed in normal service conditions and only a few in or nearly none in worst-case situations. As a consequence, most data produced by simulation runs are of minor interest for SLA validation. Or, with a change of perspective, when a “forbidden” system state does not occur during extensive simulations, it is still no proof that it might occur in even longer runs.

Example 5.2.5. *A rare event in the ParcelSink workflow model is a response time of 20 seconds or longer. As seen in previous Example 5.2.4 we checked the model for* $sla.perf := (t_{max} = 15s)$*. The results show a very low probability of 0.0006346. However, computation of narrow confidence intervals again took several minutes. Depending on the required level of precision the user has to wait 53 minutes or longer for a result. In the short 10000s and 20000s runs the rare event did not occur at all. Given the case that the short run had been the only experiment, the user could assume that such long response times do not exist at all.*

Preemptive Timeouts

Although simulation of SOA can take a significant amount of time there is a unique advantage: It allows specific measurements for SLAs and, due to step-by-step model execution, to highlight events and react on them. A semantic extension for SOA service call timeouts [16] was added to *ProC/B* in order to include deadlines in models and to avoid process blocks by lost, not returning calls to PCEs. Adding such custom behavior to Queueing Networks would prevent efficient analysis. The timeout semantics is as follows: When a process entity “arrives” at a call-PCE, and thus triggers a function offered by a FU, a timer starts to tick. When the FU finishes within time the process continuous as usual. If the timer runs down to zero before the process returns the following semantics apply [16]:

- A copy of the lost process instance with a variable set prior to the function call continues on the process chain.
- The late original process is terminated and removed from its actual queue
- Reserved process variable `in_time` is set to false.

The timeout-flag can be used in combinations with OR-connectors to model system reactions on missed deadlines and to implement own measurements registering these events, for example, the percentage of timed out requests.

Example 5.2.6. *We added a timeout to the customer's view on the ParcelSink workflow to identify requests with response times exceeding 5 seconds. The modifications are visible in Figure 5.3: the Call-PCE for the ParcelSink service marks timed out processes. A OR-connector allows processes that finished in time to pass to the sink, others are piped through a Code-PCE first. The Code-PCE includes a statement that increments a counter for timeouts, its outcome and other results are enlisted in Table D.4.*

Obviously, with the timeout extension a fraction of about 10% of all requests per experiment violated the performance contract only. Timed out requests are forced to complete within 5 seconds and thus the waiting line lengths are manipulated. This helps waiting, but not timed out requests to be processed earlier and leads to turnaround times that are off the expected values for a system without timeouts. Further applications for timeouts and integration into the *ProC/B* GUI have been investigated in Johann Kaufmann's diploma thesis [59] (a summary is given in [116]).

Errors in Simulation Models

Simulation models and simulation software are complex software constructs with manifold options to include errors. Some of them can be considered as user errors due to limited modeling practice, but others are less obvious and woven into model structures. Faulty analytical models are identified by being not computable or giving implausible results on user errors, infinite or negative response times for example. The disadvantage of simulation is that errors are not always immediately visible or occur in rare situations only.

Since simulation libraries offer programming language constructs or require the coding of behavior simulation models are a class of software. The general problem with software is the existence of programming errors that cannot be avoided or proven to be non-existent for non-trivial programs. Errors in simulation models can lead to an abort of the experiment run and the necessity to debug the model. Although this can be annoying breaking simulation runs are a less critical error. More severe are errors in model parts causing behavior not intended by the modeler. Model elements can simply react wrong on input due to programming errors, that, when not central to the model, can remain unnoticed and distort results.

The *ProC/B* editor includes some efficient strategies for model validation based on a transformation of *ProC/B* structures to Petri Nets [18]. In detail, the program is able to identify rare errors like partial deadlocks. They occur when two model elements wait for each other, but are also dependent on each other. Often these errors exist in models with high abstraction level since the human factor of the real system is lost. Second, to ensure that model input and output are in a fixed relation, thus no customers or request get lost by accident, the number of tokens in the Petri Net is ensured to be limited (Boundedness property [18]). The third error type is the non-ergodic behavior of model elements that lead to the absence of a steady-state distribution. Furthermore, syntactic errors in variable definitions and *ProC/B* statements are found. There is no sanity check for errors in input data [18].

Choosing a too low level of abstraction can also be considered as an error. The manifold modeling options of simulation models entice modelers to include a extensive level of detail in their constructs. When several persons participate in the modeling process everyone wants to include his or her domain knowledge into the model. Due to increased complexity simulation becomes processing intensive in real time and it takes longer in model time to reach steady-state. For example, in a higher abstraction grade complete model parts can be replaced by a single element with a fitted service process distribution.

5.2.4 Applicability of Simulation to SOA

Discrete event simulation offers many options to model SOAs. There is no need to use abstract queues, and statistics can highlight every aspect of a system. The show stoppers for SLA validation are the long analysis times without the guarantee that important worst-case situations are included in the runs. Other obstacles are the need for detailed information to replicate a system performance of a real system. Next to process rates themselves their distributions are required in simulative models. If a distribution is unknown assumptions have to be made. This, together with analysis, renders simulation to a inferior choice to analytical methods when early system designs need to be evaluated.

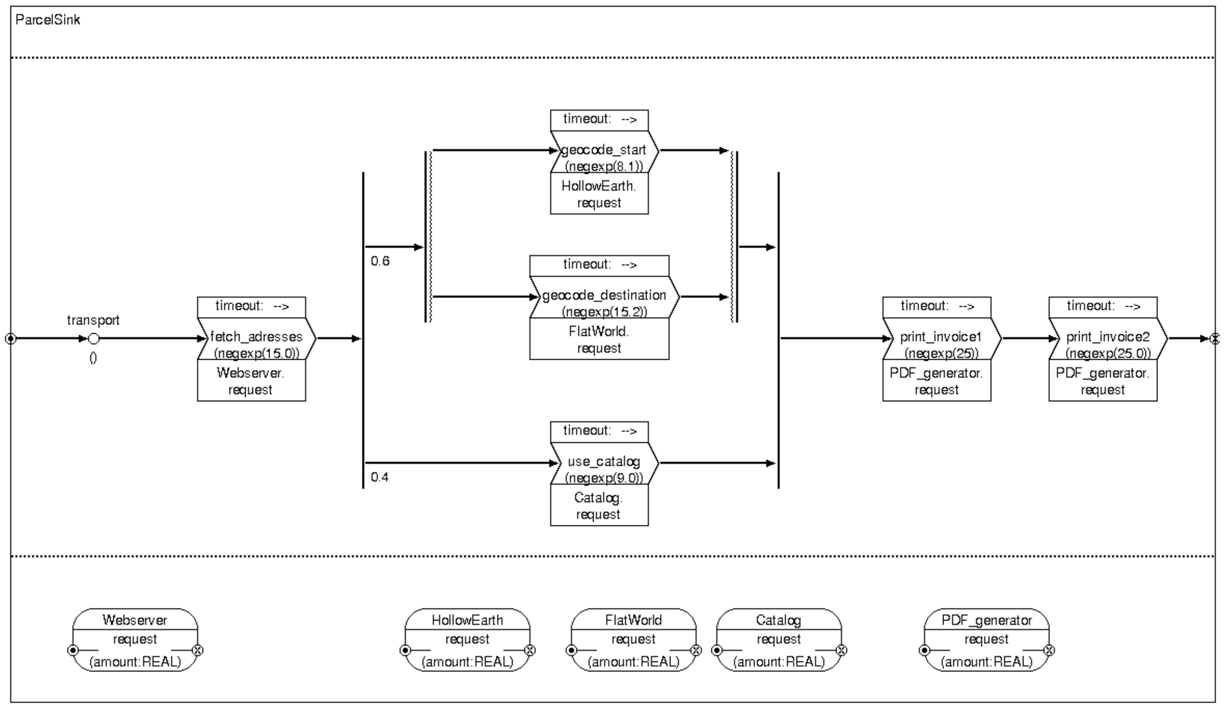


Figure 5.1: Screenshot of ParcelSink *ProC/B* model in *ProC/B* editor. The model includes two synchronized calls to both geocoding services.

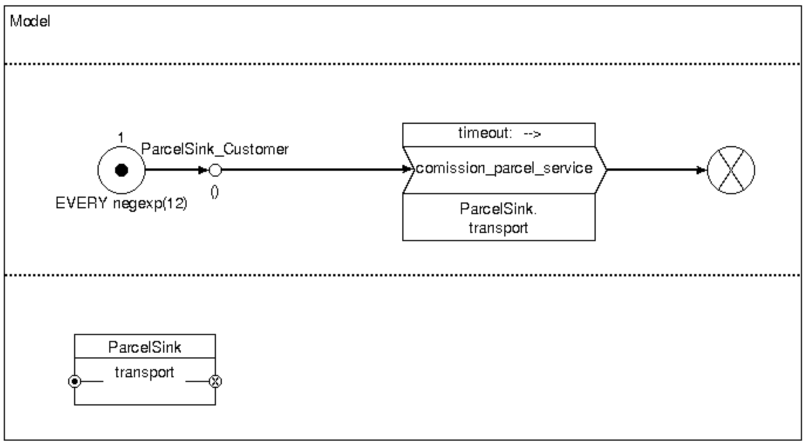


Figure 5.2: Screenshot of ParcelSink top-level model in *ProC/B* editor modeling the arrival process. The source generates requests arrivals ($\lambda = 12$) and sends them to the ParcelSink model on the lower left.

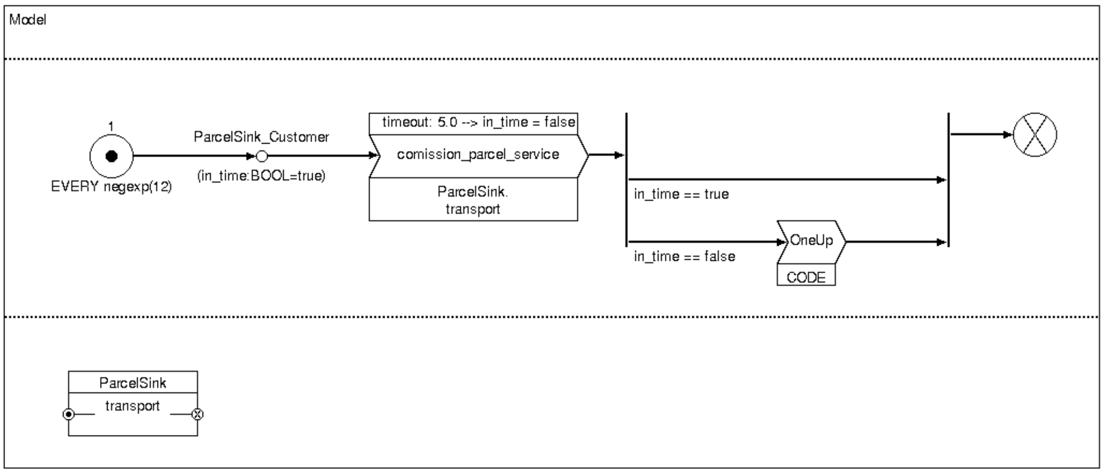


Figure 5.3: Screenshot of the ParcelSink *ProC/B* model with a timeout constraint. Requests that do not finish within 5 seconds are cancelled immediately and counted.

6 Network Calculus

For general system modeling and performance analysis Queueing Theory and Simulation have been presented. They can be used to model SOAs and to estimate their performance. When it comes to provide definitive performance guarantees as found in SLAs both methods have limited applicability. Getting figures for worst-case or best-case situations requires a lengthy computation or is not supported. Analysis becomes faster when approximative or average value-based analysis methods are used, but they can only provide long-term average values. If performance guarantees for SLAs are required, these results are almost useless. Neither peaks in the job arrival rate nor short service breakdowns are visible. Simulation computes a sample of the possible trajectories and provides results based on statistical evaluation. Especially results which result from rare events have to be computed with high precision and require long simulation runs. This also comes in expense of long processing times that can be too slow for online reconfiguration and simple draft models. An additional disadvantage of simulations is the need of detailed input data for detailed models.

Now we will move to a system performance model that has a completely different view on systems and their specification. It uses bounds for the description of arrival and service processes and, depending on the input quality, gives reliable upper bounds for delay and buffer sizes. The Network Calculus system theory for deterministic Queueing Systems is an application of the (min,+)-algebra. The (min,+)-linear system theory (for an introduction see Baccelli et. al. [6]) allows one to describe a limited class of discrete-event systems as linear time-invariant systems. Network Calculus uses this property to model elements of packet data networks and analyze their performance [2, 38, 107]. For example, it can be used to derive and prove worst-case bounds on packet delay and buffer sizes. Network Calculus has its origins in the books of Chang [38] and Le Boudec and Thiran [107]. They in turn refer to the work of Cruz [42, 43] on analyzing network delay with (min,+) algebra. Thiele et. al. used (min,+)-algebra in his Real-Time Calculus to derive execution bounds in embedded systems [103–105]. Network Calculus and Real-Time Calculus are strongly related and use the same basic theory for different applications.

There are several reasons to choose the Network Calculus system theory instead of other analysis methods. In its original incarnation, Network Calculus does not use any stochastic elements, every model element has a deterministic behavior. Anyway, for arrival and service processes short-term variations and long-term limits can be modeled. This allows one to give guarantees and bounds on performance values. Model analysis is purely analytic and results can be computed extremely fast compared to other analysis methods, especially simulation. Downsides on the Network Calculus approach are a high abstraction level of models and the focus on worst-case performance values. This work will propose some ideas to overcome these limitations for SLA validation.

The contribution in this work towards a calculus for Service Level Agreements is based on the Network Calculus and Real-Time Calculus approaches, therefore the key elements are introduced. The reference books of Chang [38], as well as, Le Boudec and Thiran [107] use an algebraic approach. A later paper, [30] gives a short but brilliant introduction to the topic focusing on service curves (c.f. Section 6.2.8). In this work Network Calculus is described from the unusual perspective of cumulated arrivals and service resources over time forming functions. Functions are later abstracted to curves that can be manipulated using (min,+) algebra. This form of introduction is chosen to help the reader in understanding the ideas in SLA Calculus as a natural extension of Network Calculus and Real-Time Calculus principles.

6.1 (min,+)-Algebra Basics

Network Calculus and its derivatives are based on the so-called Min-plus or (min,+) algebra. It is different from the well-known elementary algebra taught at school, used for common calculations and for solving equations. Basically, elementary algebra is created with two mappings, addition and multiplication, and the set $\mathbb{R}$ of real numbers. We write this $(\mathbb{R}, +, \cdot)$ or shorthand $(+, \cdot)$. Neutral Element of addition is 0 and 1 for multiplication.

(min,+) algebra uses the minimum of two numbers instead of addition and addition instead of multiplication. In reference to elementary algebra the minimum is still said to be the additive operation and the addition becomes the multiplicative operation (notation (min,+)). Like their original counterparts the operator min() and + form a dioid [107] with $+\infty$ as neutral element of addition and 0 as neutral element of multiplication.

Example 6.1.1. *The term* $(5 + 0) \cdot (2 + 4)$ *in elementary algebra is evaluated to a value of 40. With (min,+) it is expressed as* $\min(5, +\infty) + \min(2, 4)$ *and gives a value of 7.*

The (min,+) Dioid

For the (min,+) algebra some properties can be stated, we recap results from [38, 107] here and refer to these books for further details. Let $\mathbb{R} \cup +\infty = \mathbb{R}_{+\infty}$. The additive min() operator forms a commutative and idempotent semi-group $(\mathbb{R}_{+\infty}, \min())$:

Zero element $\forall a \in \mathbb{R}_{+\infty}\ \exists z \in \mathbb{R}_{+\infty} : \min(a, z) = a$. In this case $z = +\infty$.

Associativity $\forall a, b, c \in \mathbb{R}_{+\infty} : \min(\min(a, b), c) = \min(a, \min(b, c))$

Commutativity $\forall a, b \in \mathbb{R}_{+\infty} : \min(a, b) = \min(b, a)$

Idempotency $\forall a \in \mathbb{R}_{+\infty} : \min(a, a) = a$

Also the multiplicative + operator forms a commutative semi-group $(\mathbb{R}_{+\infty}, +)$:

Neutral Element $\forall a \in \mathbb{R}_{+\infty} \exists n \in \mathbb{R}_{+\infty} : a + n = a$. In this case $n = 0$.

Associativity $\forall a, b, c \in \mathbb{R}_{+\infty} : (a + b) + c = a + (b + c)$

Commutativity $\forall a, b \in \mathbb{R}_{+\infty} : a + b = b + a$

Together $(\mathbb{R}_{+\infty}, \min())$ and $(\mathbb{R}_{+\infty}, +)$ form a commutative dioid:

Zero element $z = +\infty$ **for** $\min()$ **is absorbing for** $+$ $\forall a \in \mathbb{R}_{+\infty} : a + z = z$

Distributivity of $+$ **with respect to** $\min()$ $\forall a, b, c \in \mathbb{R}_{+\infty} : \min(a, b)+c = \min(a+c, b+c)$

While $(\mathbb{R}_{+\infty}, \min(), +)$ is a commutative dioid there are some differences to the common commutative field $(\mathbb{R}, +, \cdot)$. In $(\mathbb{R}, +, \cdot)$ we are used to $a + (-a) = 0$. This cancellation property is exploited to solve equations. The $(\mathbb{R}_{+\infty}, \min())$ semi-group is missing a cancellation $\min(a, -a) = +\infty$, as a result solutions for some equalities are not unique [90].

Wide-sense Increasing Functions

In Network Calculus wide-sense increasing functions are fundamental. Notation and definitions in this thesis will be similar to [107].

Definition 6.1.1 (Wide-sense increasing function). *A function is wide-sense increasing if and only if $f(a) \leq f(b)$ for all $a \leq b$.*

Definition 6.1.2 (Wide-sense decreasing function). *A function is wide-sense decreasing if and only if $f(a) \geq f(b)$ for all $a \leq b$.*

Definition 6.1.3 (Casual Function). *A function f is casual if $f(t) = 0$ for $t < 0$. [74]*

Definition 6.1.4 (Sets of wide-sense increasing functions). *$\mathcal{G}$ is the set of wide-sense increasing functions with $f(t) \geq 0\, \forall t, f \in \mathcal{G}$. $\mathcal{F}$ is the subset of $\mathcal{G}$ with casual functions. $\mathcal{F}_0$ is the subset of $\mathcal{F}$ with $f(0) = 0$ for all $f \in \mathcal{F}_0$.*

The notation of the (min,+) algebra is extended to wide-sense increasing functions [107]. Let $f, g \in \mathcal{G}$. The point-wise minimum $min(f(t), g(t)) = min(f, g)(t)$ is written shorthand $f \wedge g$ for all t. Addition can be written as $f(t) + g(t) = (f + g)(t)$ or shorthand $f + g$ for all t.

Equations are not point-wise. It is $f = g$ if $f(t) = g(t)$ $\forall t$ holds. The same holds for inequalities ($\leq$,$\geq$).

Convention of Left-Continuous Functions

Network Calculus models data flows in networks with wide-sense increasing functions. At packet level these data flows are sequences of discrete events. For this reason the Network Calculus traffic arrival model uses step functions that are non-continuous at the packet arrival times. Discontinuity makes calculations hard. However, at least a one-sided continuity can be assumed.

We will recap the Weierstrass definition for continuity here because the difference between continuity and one-sided continuity is directly observable. A function $f : S \to \mathbb{R}$ is continuous in point $c \in S$, if for any number $\epsilon > 0$ there is a $\delta > 0$ such that $\forall s \in S$ within range $c - \delta < s < c + \delta$ the following holds:

$$f(c) - \epsilon < f(s) < f(c) + \epsilon \tag{6.1}$$

or equivalently $|s - c| < \delta$ follows $|f(s) - f(c)| < \epsilon$. ϵ is used to define a neighborhood around $f(c)$. The function values will stay within this neighborhood when the input range

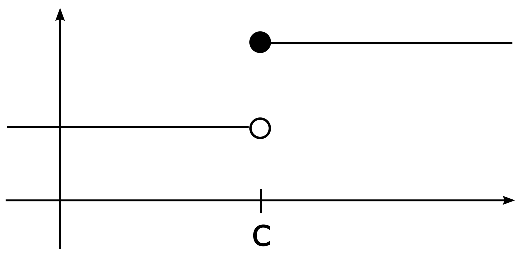


Figure 6.1: Left-continuous in point c.

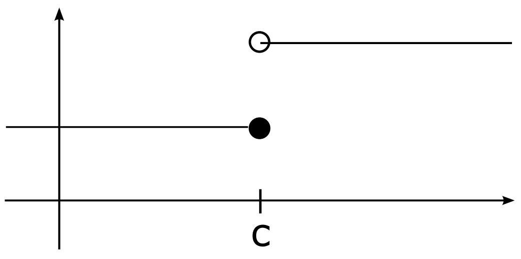


Figure 6.2: Right-continuous in point c.

spanned around c by δ is just small enough. When f is continuous in c this works for any neighborhood around $f(c)$ no matter how small. When f is not continuous in c there are neighborhoods around $f(c)$ given by some ϵ not covered by the function input range for some δ.

For one-sided continuity the input range is only spanned around c in one direction.

Definition 6.1.5 (Left and Right-continuous Functions)**.** *A function $f : S \to \mathbb{R}$ is left-continuous in point $c \in S$ when for any number $\epsilon > 0$ there is a $\delta > 0$ such that $\forall s \in S$ within range $c - \delta < s < c$ the inequality $|f(s) - f(c)| < \epsilon$ holds.*

It is right-continuous in point c if $c < s < c + \delta$ and $|f(s) - f(c)| < \epsilon$.

Example 6.1.2. *Function f_c is left-continuous in c.*

$$f_c(x) = \begin{cases} 1 & \text{if } x \le c \\ 2 & \text{else} \end{cases} \tag{6.2}$$

It is neither right- nor generally continuous in c.

Per convention [107, Section 3.1.3] functions $f \in \mathcal{G}$ are left-continuous. Figures 6.1 and 6.2 show left- and right-continuous functions.

6.1.1 Infimum and Supremum

In Network Calculus frequently the infimum is used instead of the minimum to denote the lower bound of a set. The minimum of a set $S \in \mathbb{R}$ is defined by

$$\min(S) = \{a \mid a \in S \text{ and } a \le b \ \forall b \in S\} \tag{6.3}$$

Using the smallest element of a set as lower boundary is quite intuitive, for the closed set $[s, t] \in \mathbb{R}, s < t$ we can expect the lower bound to be s.

For open set (s, t) the lower bound is not s because s is not in the set. To get the lower bound the infimum is used.

$$\inf(S) = \{\max(a) \mid b \ge a \ \forall b \in S\} \tag{6.4}$$

It is safe to replace minimum with infimum in general when working with open, partially open and closed sets because if a minimum of a set exists it is equal to the infimum. Let $[a, b]$ be a closed set, then the following is true:

$$\min([a, b]) = \inf([a, b]) = a$$

Upper Bounds of an open set are, instead of a maximum, expressed by the supremum.

$$\sup(S) = \{\min(a) \mid b \leq a \ \forall b \in S\} \tag{6.5}$$

6.1.2 Function Curvature

The function curvature is an important property in Network Calculus. A function f is convex in $\mathbb{R}$ if for all $x, y \in \mathbb{R}$ and $0 \leq z \leq 1$ the inequality $f(zx+(1-z)y) \leq z \cdot f(x)+(1-z) \cdot f(y)$ holds. It is concave if $f(zx + (1 - z)y) \geq z \cdot f(x) + (1 - z) \cdot f(y)$.

A function f is star-shaped if $\frac{f(t)}{t}$ is wide-sense decreasing $\forall\ t > 0$ [107, Definition 3.1.9]. For two values s, t with $0 < s \leq t$ this gives

$$\frac{f(s)}{s} \geq \frac{f(t)}{t} \tag{6.6}$$

In star-shaped functions the rate of growth decreases over time, thus concave functions are star-shaped [107, Theorem 3.1.4].

6.1.3 Convolution

Convolution is an operation between two functions. The result is a new function forming the overlay of both functions. It plays an important role in mathematics, natural sciences and technical applications. In elementary algebraic systems convolution is an integral over a product of two real-valued functions f and g:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(t-s) \cdot g(s) \, \mathrm{d}s \tag{6.7}$$

The effect of convolution can be explained as follows: Although convolution is commutative, let us call function f the input signal, g the filter and h the output signal. Filter g is used to scale the value of input f at a specific point in time t to form the output. Without convolution this is expressed as $h(t) = f(t)g(t)$. The distinctive feature of convolution is that the output is not only dependent on a single point in time. It also includes previous values of f averaged by filter g. To address previous points in time variable s is used to shift time t backwards. The value of the input function is thus found by $f(t - s)$, it is weighted by the filter g evaluated in s. As an intermediate step we get

$$f(t-s) \cdot g(s) \tag{6.8}$$

Finally we sum up all weighted values by integrating over s. By moving s in interval $(-\infty, \infty)$ all possible function images of f weighted by g are considered.

6.1.4 Linear Time-Invariant Systems

A system described by operator S reacts on input $x(t)$ with output $S[x(t)]$. x is a function ("signal") of time index t. The system is linear if for inputs $x_1(t), x_2(t)$ and factor c the following holds:

$$S[x_1(t) + x_2(t)] = S[x_1(t)] + S[x_2(t)] \qquad \text{(Superposition)} \tag{6.9}$$

$$S[c \cdot x_1(t)] = c \cdot S[x_1(t)] \qquad \text{(Scaling)} \tag{6.10}$$

So factors applied to the input are reflected at the output and signals can be combined. A system is time invariant if the reaction to an input is always the same regardless of the time the input arrived:

$$y(t) = S[x_1(t)] \Rightarrow y(t+\Delta) = S[x_1(t+\Delta)] \quad \forall\Delta \tag{6.11}$$

The concept of a linear system has an outstanding importance in system theory, communication technology and many other disciplines [39, 49]. Assuming linearity simplifies a wide range of equations and models. However, in reality there is no system that reacts strictly linear. An arbitrary factor applied to inputs will not always be reflected at the output. The theory can still be applied since many systems are linear within a range of inputs and only behave non-linear for inputs that exceed their specification.

Impulse Response

In system theory a linear system is completely characterized by its impulse response function. In theory this response can be measured if a Dirac impulse $\delta(t)$ is sent to the system. The Dirac function is given by

$$\delta(t) = \begin{cases} +\infty & t = 0 \\ 0 & t \neq 0 \end{cases} \tag{6.12}$$

with the assumption that $\int_{-\infty}^{+\infty} \delta(t)\,\mathrm{d}t = 1$. The Dirac impulse is an ideal signal, all other signals to the system can be represented as combinations of it. Since the system is linear it can be scaled by multiplication with a factor. It is also time-invariant so the impulse can also be shifted along the time axis. When a linear system's reaction to one impulse is known the reaction to a series of impulses is known as well.

6.1.5 (min,+) Convolution

For Network Calculus convolution is a frequently used operation. Since the calculus is based on (min,+) algebra the convolution has to be defined with respect to the changed operators. The additive operators are exchanged for the minimum function, hence the integral is replaced by minimum operation. In particular, to allow non-continuous input functions, the infimum is used [107].

Definition 6.1.6 ((min,+) Convolution)**.** *Let f and g be two functions or sequences in $\mathcal{F}$. The (min,+) convolution of f and g (notation $f \underline{\otimes} g$) is the function*

$$(f \underline{\otimes} g)(t) = \inf_{0 \le s \le t} \{f(t-s) + g(s)\} \tag{6.13}$$

If $t < 0 : (f \underline{\otimes} g)(t) = 0$.

Convolution has the following properties ([107, Theorem 3.1.5] and [107, Theorem 3.1.6]). Let functions f, g, $h \in \mathcal{F}$.

Closure of $\underline{\otimes}$: $(f \underline{\otimes} g) \in \mathcal{F}$. Convolution is closed in $\mathcal{F}$.

Associativity of $\underline{\otimes}$: $(f \underline{\otimes} g) \underline{\otimes} h = f \underline{\otimes} (g \underline{\otimes} h)$. The order of operator application does not matter.

Zero element for $\min()$ **is absorbing for $\underline{\otimes}$:** The zero element for $\min()$ is the function $\epsilon \in \mathcal{F}$ defined as

$$\epsilon(t) = \begin{cases} +\infty & t \geq 0 \\ 0 & t < 0 \end{cases}$$

One has $f \underline{\otimes} \epsilon = \epsilon$.

Neutral element for $\underline{\otimes}$: $f \underline{\otimes} \delta_0 = f$. $\delta_0(t) = \infty$ for $t > 0$ and $\delta_0(t) = 0$ for $t \leq 0$ is the Dirac function equivalent for Network Calculus, see also Section 6.2.4.

Commutativity of $\underline{\otimes}$: $f \underline{\otimes} g = g \underline{\otimes} f$.

Distributivity of $\underline{\otimes}$ with respect to $\min()$**:** $\min(f, g) \underline{\otimes} h = \min(f \underline{\otimes} h, g \underline{\otimes} h)$.

Addition of a constant: $\forall K \in R^+ : (f + K) \underline{\otimes} g = (f \underline{\otimes} g) + K$

Functions passing through the origin: If $f(0) = g(0) = 0$ then $f \underline{\otimes} g \leq \min(f, g)$. When f and g are star-shaped then $f \underline{\otimes} g = \min(f, g)$ holds.

Convex functions: If f and g are convex then $f \underline{\otimes} g$ is convex.

Isotonicity: If $f \leq g$ and $f' \leq g'$ then $f \underline{\otimes} f' \leq g \underline{\otimes} g'$. The operator does not change the ordering.

(min,+)-convolution is associative, commutative, distributive in respect to min() and closed in $\mathcal{F}_0$. Again $(\mathcal{F}, \min, \underline{\otimes})$ is a diod [38, 47].

6.1.6 (min,+) Deconvolution

The dual operation to $\underline{\otimes}$ in (min,+) is deconvolution.

Definition 6.1.7 ((min,+) Deconvolution)**.** *Let $f, g \in \mathcal{F}$. The (min,+) deconvolution of f by g (notation $f \underline{\oslash} g$) is the function*

$$(f \underline{\oslash} g)(t) = \sup_{s \geq 0} \{f(t + s) - g(s)\}$$

Deconvolution has the following properties [107, Theorem 3.1.12]: Let functions f, g, $h \in \mathcal{F}$.

Isotonicity of $\underline{\oslash}$: If $f \leq g$ then $f \underline{\oslash} h \leq g \underline{\oslash} h$ and $h \underline{\oslash} f \geq h \underline{\oslash} g$.

Composition of $\underline{\oslash}$: $(f \underline{\oslash} g) \underline{\oslash} h = f \underline{\oslash} (g \underline{\otimes} h)$.

Composition of $\underline{\oslash}$ and $\underline{\otimes}$: $(f \underline{\otimes} g) \underline{\oslash} g \leq f \underline{\otimes} (g \underline{\oslash} g)$.

Duality between $\underline{\oslash}$ and $\underline{\otimes}$: $f \underline{\oslash} g \leq h$ iff $f \leq g \underline{\otimes} h$.

Self-deconvolution: $(f \underline{\oslash} f)$ is a sub-additive function of $\mathcal{F}$ such that $(f \underline{\oslash} f)(0) = 0$.

6.1.7 The (max,+)-Algebra

For the definition of lower curve contracts and their combinations we will use the (max,+)-algebra. Its definition equals the (min,+)-algebra when the minimum-operator is replaced by a maximum-operator. For this reason the algebra is only briefly introduced, more details can be found in [6], [107, Section 3.2] and [38, Section 6.1].

The algebraic structure $(\mathbb{R} \cup \{-\infty\}, \max(), +)$ is a dioid [107, Section 3.2]. Additive operation max() is associative, commutative and has $-\infty$ as zero element. The multiplicative operation + also features associativity, commutativity and 0 as neutral element. Also distributivity of + with respect to max() is given.

The point-wise maximum $max(f(t), g(t)) = max(f, g)(t)$ for all t is written shorthand $f \vee g$.

(max,+) Convolution

With the operation from the (max,+) dioid a (max,+) variant of convolution can be defined [107, Definition 3.2.1]:

Definition 6.1.8 ((max,+) Convolution)**.** *Let f and g be two functions or sequences in $\mathcal{F}$. The (max,+) convolution of f and g (notation $f \overline{\otimes} g$) is the function*

$$(f \overline{\otimes} g)(t) = \sup_{0 \le s \le t} \{f(t-s) + g(s)\} \tag{6.14}$$

If $t < 0 : (f \overline{\otimes} g)(t) = 0$.

6.1.8 Sub-additive and Super-additive Functions

Definition 6.1.9 (Sub-additive Function)**.** *A function f is said to be sub-additive if [107, Def. 3.1.11]*

$$f(s) + f(t-s) \ge f(t) \text{ for } s \le t \tag{6.15}$$

Corollary 6.1 (Sub-additivity of concave functions)**.** *Any concave function f with $f(0) = 0$ is sub-additive.*

In the same sense super-additive functions exist.

Definition 6.1.10 (Super-additive Function)**.** *A function f is said to be super-additive if*

$$f(s) + f(t-s) \le f(t) \text{ for } s \le t \tag{6.16}$$

Corollary 6.2 (Super-additivity of convex functions)**.** *Any convex function f with $f(0) = 0$ is super-additive.*

Both proofs are given in the Appendix.

6.1.9 Sub-additive Closure

Definition 6.1.11 (Sub-additive and Super-additive Closure)**.** *For function $f \in \mathcal{F}$ the sub-additive closure $\overline{f}$ is recursively defined [38]*

$$\overline{f}(0) = 0 \tag{6.17}$$

$$\overline{f}(t) = \min(f(t), \min_{0<s<t}(\overline{f}(s) + \overline{f}(t-s))),\ t > 0 \tag{6.18}$$

The super-additive closure $\underline{f}$ is defined when min() *operators are replaced with* max()*.*

In combination with convolution sub-additive functions have an interesting property described in [107, Theorem 3.1.6] and [38, Lemma 2.1.5 (xi)]. We adopt the notation of [38] and write $f^\circ = f \wedge \epsilon$ with ϵ being the neutral element for min().

Lemma 6.3. *When two functions f, g are sub-additive then*

$$\overline{(\,f \wedge g\,)} = \overline{((\,f \wedge g\,)^\circ)} = \overline{f^\circ \wedge g^\circ} = \overline{f} \underline{\otimes} \overline{g} \tag{6.19}$$

holds [38, Lemma 2.1.5].

6.2 System Model

The basic system model for Network Calculus is quite similar to the model used in Queueing Theory for Queueing Systems (Section 4.1) or linear system theory (Section 6.1.4). The systems to model are computer networks transmitting data, the main idea is to describe them as systems that are linear time-invariant under the (min,+) algebra. Signals to these systems are arrivals of network packets over time. The impulse response describes how service resources are used and allows one to derive the output signal in form of departing packets. The noteworthiness for Network Calculus is that signals are not modeled with functions giving their value at a certain point of time. Instead, cumulative functions are used for input, output, as well as, the impulse response. System states are unknown to Network Calculus, they are hidden in bounds and limits on the signals.

Computer networks are hierarchical systems built of subnets, switches, routers, single network hosts and their connections. The Network Calculus model can be applied to an atomic component or combinations forming a bigger system. Network Calculus provides the tools to compute performance values for composed systems from the results of basic systems.

Regardless of their specific purpose, network components can be abstracted to a simple system that performs the service of transmitting data. This type of system is the basic Network Calculus system model, in this work it will be called a network element. Figure 6.3 contains the basic system model. As common in the queueing notation the arrivals enter the system from the left and, after the network service has been applied, leave to the right. To process arrivals a network element needs resources in form of “service”. The service process is described by a flow of resources arriving from the top. Resources might be limited and not always sufficient to process all arrivals. In the case of a resource shortage, comparable to Queueing Systems, processing stations have to enqueue arrivals until there

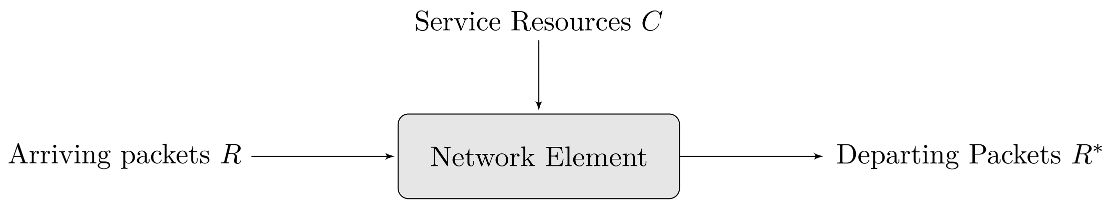


Figure 6.3: Network Calculus system model: A network element transmits data (packets). This task requires service resources.

are enough resources at hand. Unused resources leave the network element immediately and cannot be kept for later usage. So the service resources in Figure 6.3 are the available, but not actually used service capacity at this point of time.

Data packets or low level bitstreams are the initially intended usage domain for Network Calculus. Of course, the modeling method can also be applied to other areas, one could also specify computation tasks, customers or packets in logistics as arrivals. The term service could be replaced with energy, CPU time or load capacity of trucks. Examples for other areas of application are sensor networks [98], optimization of network routes [29] and even energy withdrawal from batteries [73].

6.2.1 Arrival and Departure Flows

Let $r(t)$ be a function describing the arrival process with the number of arrivals to a network element at time t. Depending on the model $r(t)$ is measured in number of bytes or other quantities. When the arrivals are packets $r(t)$ gives the size. Figure 6.4 shows an instance of $r(t)$ with discrete, equally sized packet arrivals.

Function $r(t)$ is a sequence of discrete arrivals with detailed information on every packet. For high level modeling a simplified description without discrete arrivals is preferable. The approach in Queueing Theory is to find a distribution for packet interarrival times in $r(t)$. The approach in Network Calculus is different. Information on arrival processes is abstracted by summing up the number of arrivals to network elements within time interval $[0, t]$. The so-called arrival flow can be easily determined by integrating $r(t)$.

Definition 6.2.1 (Arrival Flow)**.** *Let $r(t)$ be the number of arrivals of an arrival process at time t. The cumulative sum*

$$R(t) = \int_0^t r(x)\,dx \tag{6.20}$$

is the arrival flow of the process in interval $[0, t]$.

$R(t)$ is wide-sense increasing and $R(t) = 0$ for $t \leq 0$, thus $R(t) \in \mathcal{F}$. For $r(t)$ with discrete arrivals $R(t)$ becomes a non-continuous step function. In this case $R(t)$ is a discrete model for the arrival process. Figure 6.4 shows the arrival flow $R(t)$, too. $R(t)$ has been constructed bottom-up by integrating given function $r(t)$.

A fluid model for the arrival process exists when $R(t)$ has a derivate $\frac{\mathrm{d}R}{\mathrm{d}t}$, hence $R(t)$ has to be continuous. In this case the rate of arrivals at time t can be expressed by the slope of a function $R(t)$. For this reason function $r(t)$ is also referred as the arrival rate function

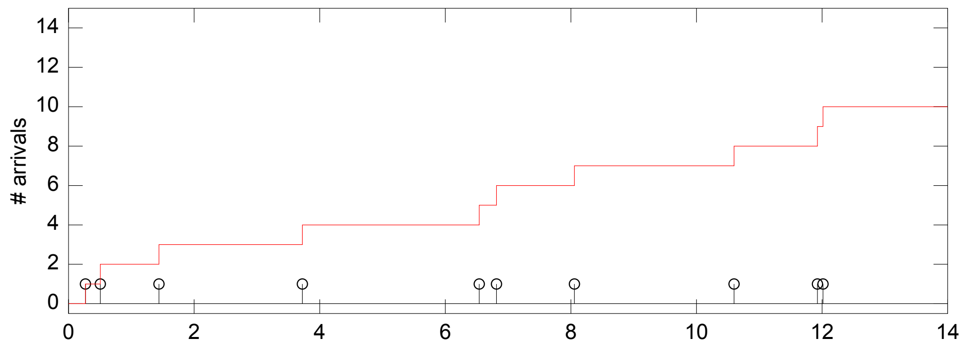


Figure 6.4: Arrival function $r(t)$ with discrete arrivals (the spikes). It is used to form the arrival flow $R(t)$ (step function).

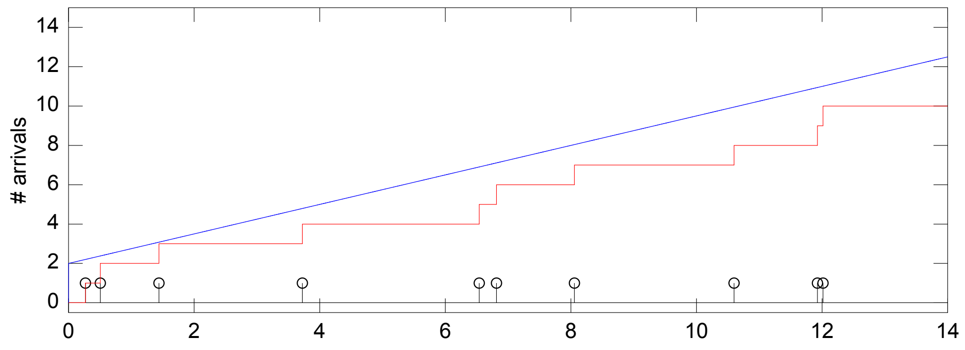


Figure 6.5: Arrival flow $R(t)$ is bound from above by an arrival curve α^U

in Network Calculus [107, Section 1.1]. In a fluid model, $r(t)$ is also continuous and cannot capture individual packet arrivals. So $r(t)$ can only be constructed top-down from a known arrival flow $R(t)$ by derivation.

However, choosing between fluid models with continuous flows or discrete models built from arrival processes is seldom necessary in Network Calculus. Arrival flows are further abstracted by bounding functions, so fluidity of flows can be assumed as long as the resolution of a discrete flow is not required.

The basic system model in Network Calculus also includes the packet departure process. The outgoing flow in interval $[0, t]$ is denoted as $R^*(t)$ following the same principle of abstraction by cumulation as the arrival process. Thus, $R^*(t)$ has the same mathematical properties as $R(t)$. If two network elements A and B are concatenated, the output of A is fed as input into B. Hence R^* is also often referred to as the outgoing arrival flow [107].

6.2.2 Backlog and Delay

Processing or transmission of packets by network elements introduces delays (response times) to the packet flow. When several packets arrive at a busy element also queueing in

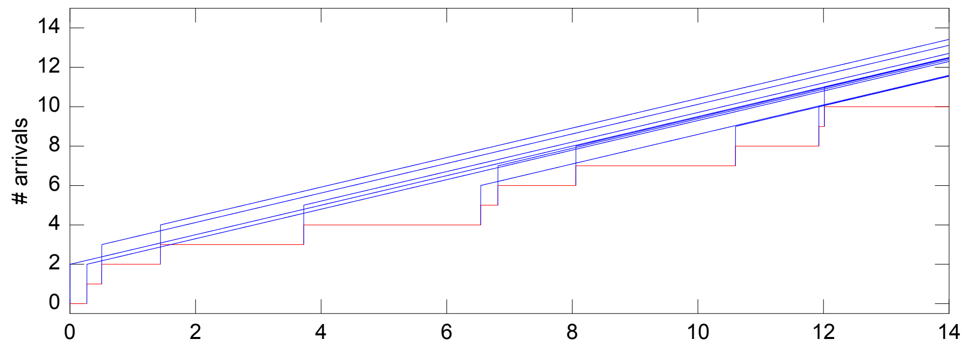


Figure 6.6: Illustration of $R(t)$ being constrained by α^U in every time interval ($R \leq R \underline{\otimes} \alpha^U$)

a buffer may occur. The amount of bytes or packets held back inside a network element is called backlog [107, Section 1.1.2] and is comparable to queue population in Queueing Systems (Section 4.2). Delay and backlog for a system can be derived by the distances of corresponding flows R and R^*.

The vertical deviation between two functions $f(t), g(t)$ is the difference at the same time t [107, Definition 3.1.15]:

Definition 6.2.2 (Vertical Deviation)**.** *Let $f, g \in \mathcal{F}$ be two functions. The vertical deviation $v(f,g)$ between both functions is*

$$v(f,g)(t) = |g(t) - f(t)| \tag{6.21}$$

Horizontal deviation between two functions is the difference in input to get the same output. The definition is also based on [107, Definition 3.1.15].

Definition 6.2.3 (Horizontal Deviation)**.** *Let $f, g \in \mathcal{F}$ be two functions. The horizontal deviation $h(f,g)$ between both functions is defined as*

$$h(f,g)(t) = \inf\{d \geq 0 \textit{ such that } f(t) \leq g(t+d)\} \tag{6.22}$$

Figuratively $h(f,g)(t)$ is the additional time $g(t)$ needs to reach the level $f(t)$. Figure 6.7 illustrates this.

Backlog Bounds

System backlog can be computed from vertical deviation between arrival and departure flows [107, Def. 1.1.1]:

Definition 6.2.4 (Backlog)**.** *Let R and R^* be corresponding arrival and departure flows for a system. The amount of backlogged arrivals in the system at time t is $v(R,R^*)(t)$.*

A system is said to be in a backlog interval $[a,b]$ if $v(R,R^*)(t) > 0\ \forall t \in [a,b]$. When backlogging in a system occurs, arrivals have to be stored in a memory whose capacity is naturally limited. Necessary buffer sizes for the system can be dimensioned adequately

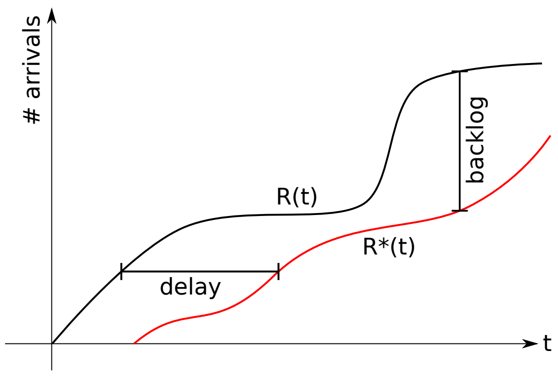


Figure 6.7: Horizontal and vertical deviations between $R(t)$ and $R^*(t)$

by computing the maximum backlog. The maximum buffer occupation in a system with arrival flow R and departure flow R^* is $b_{\max}(R, R^*)$ with

$$b_{\max}(a,b) = \sup_{t\geq 0} \{v(a,b)(t)\} \tag{6.23}$$

Delay Bounds

The horizontal deviation is used in Network Calculus to obtain a bound on the system delay [38, 107]. Le Boudec and Thiran define this latency as virtual delay [107, Section 1.1.2]:

Definition 6.2.5 (Virtual Delay)**.** *Let R and R^* be corresponding arrival and departure flows for a system. The virtual delay for arrivals to the system at time t is*

$$h(R, R^*)(t) \tag{6.24}$$

Similar to maximum backlog we can also state the maximum virtual delay.

Definition 6.2.6 (Maximum Virtual Delay)**.** *The maximum virtual delay in a system with arrival flow R and departure flow R^* is $h_{\max}(R, R^*)$ with*

$$h_{\max}(a,b) = \sup_{t\geq 0} \{h(a,b)(t)\} \tag{6.25}$$

6.2.3 Curves as Flow Abstraction

Using cumulated functions to describe arrival and departure processes at a network introduces a connection to classical system theory. The flows are the signals sent to and emitted by the network element. It has not been specified in the previous sections how arrival flows R and R^* are derived. These functions can be seen as one of many samples that can be measured at a system. One could opt to extract the average arrival and departure rates for the network. This is the approach taken for Queueing Networks, a very condensed process model enabling efficient analysis.

The approach chosen in Network Calculus is to use deterministic upper and lower bounds for processes. Individual flows are replaced with functions. A flow is a wide-sense increasing function and its increase within a time interval can be bounded by another wide-sense increasing function.

Arrival Curves

To characterize arrival flows and to set bounds on arrival rates, Network Calculus abstracts individual arrival flows with functions called arrival curves conforming to the arrival curve property [103]. An upper arrival curve is a function that serves as a time-invariant upper bound for an arrival flow $R(t)$ or departure flow $R^*(t)$.

Definition 6.2.7 (Upper Arrival Curve)**.** *A function $\alpha^U(t)$ is a upper arrival curve for arrival function $R(t)$ if for all intervals $[s,t]$*

$$R(t) - R(s) \leq \alpha^U(t-s) \quad \forall\, 0 \leq s \leq t \tag{6.26}$$

If the relation holds R is said to conform to upper arrival curve α^U.

In Figure 6.5 $R(t)$ is bound from above by an arrival curve: The flow never exceeds the curve constraint defined by α^U. In this case, we have $R(t) \leq \alpha^U(t)\ \forall\, t$. However, this is not enough to fulfill the arrival curve property. Definition 6.2.7 requires that the relation holds for all time intervals. Figure 6.6 illustrates the idea by applying the bounding curve to several points of $R(t)$. For this example of a discrete model the points are chosen immediately before the arrival of another packet. Notable is the application of α^U at $t = 0.3$ and $t = 6.5$. With the application of α^U at $t = 0.3$ the original bound on R set by the arrival curve prototype at $t = 0$ in Figure 6.5 is replaced with a lower one. It is the upper limit until the arrival at $t = 6.5$. Starting from this point, $R(t)$ will stay below the new boundary.

Up to this point, flows and curves were defined with elementary algebra and (min,+) algebra was not used at all. It has been shown in [107, Lemma 1.2.3] and [38, Lemma 2.2.2] that the idea of overlapping intervals is equivalent to (min,+) convolution of arrival flow and curve.

Theorem 6.4 ((min,+)-Algebra Constraint)**.** *Arrival function R is constrained by α^U iff*

$$R(t) \leq (R \underline{\otimes} \alpha^U)(t)\ \forall t \tag{6.27}$$

Proof. Using the definition of (min,+) convolution we can rewrite the right side of $R(t) \leq (R \underline{\otimes} \alpha^U)(t)$ as $R(t) \leq \inf_{0 \leq s \leq t}(R(t-s) + \alpha^U(s))$. R and α^U are only defined for $s \geq 0$, so the case for $s < 0$ is not considered. The infimum operator gives the minimal solutions of $R(t-s) + \alpha^U(s)$ for all combinations of s and t with $0 \leq s \leq t$. All of them are greater or equal to the left-side $R(t)$. The inequality stays true if the infimum operator is removed:

$$R(t) \leq R(t-s) + \alpha^U(s)\ \forall 0 \leq s \leq t$$

and α^U is an arrival curve. □

Lower Arrival Curves

In a similar approach arrival flows can be enforced in the model not to drop below a certain bound within a time interval.

Definition 6.2.8 (Lower Arrival Curve)**.** *A lower arrival curve α^L satisfies the relation*

$$\alpha^L(t-s) \leq R(t) - R(s) \quad \forall\, 0 \leq s \leq t \tag{6.28}$$

Lower arrival curves are not included in the work of Le Boudec and Thiran in [107], they are a concept of Real-Time Calculus [104, 106]. The relation can be mapped to an expression in (max,+)-algebra.

Theorem 6.5 (Lower Arrival Curve Constraints)**.** *Let R by an arrival flow and $\alpha^L \in \mathcal{F}_0$ a lower arrival curve bounding R from below. Then $R \geq R \overline{\otimes} \alpha^L$ holds.*

Proof. Using the Definition of $\overline{\otimes}$ we can rewrite the right side

$$R(t) \geq \sup_{0 \leq s \leq t} \Big(R(s) + \alpha^L(t-s) \Big) \tag{6.29}$$

The supremum operator gives the maximum solution of $R(s) + \alpha^L(t-s)$. All solutions, even the maximum one, are smaller or equal to the left side. So inequality stays true when the supremum operator is removed:

$$R \geq R(s) + \alpha^L(t-s) \tag{6.30}$$

and by reordering

$$R(t) - R(s) \geq \alpha^L(t-s) \tag{6.31}$$

□

6.2.4 Arrival Curve Instances

Arrival curves are a sub-model for arrival flows in Network Calculus. Every $f \in \mathcal{F}$ can serve as arrival curve. To simplify successive computations only a small set of basic functions is used. Most of them are linear or grow stepwise. A catalog of common functions can be found in [107, Section 3.1.3]. On the one hand, the usage of simple functions with one or two parameters offers limited precision for flow bounds. On the other hand, simple curve models have prominent advantages: The semantics of curve parameters are known and can be associated to real world systems. With arrival curves discrete arrivals can be abstracted to a continuous, even derivable input model. Additionally, such basic arrival curves can be combined to more complex functions that are still arrival curves. In the following we will also show how those combinations can be exploited to partition an arrival curve into zones that separate short-term and long-term behavior of a flow. Curve composition also allows one to approximate arbitrary arrival curves to improve precision.

Linear Function or peak rate function: $\lambda_R(t) = R \cdot t$. When used as upper arrival curve, λ_R limits the arrival rate to a maximum of R. λ_R is convex and concave.

Affine Function is a vertically shifted linear function (c.f. Figure A.1):

$$\gamma_{r,b} = \begin{cases} r \cdot t + b & \text{for } t > 0 \\ 0 & \text{for } t = 0 \end{cases} \tag{6.32}$$

$\gamma_{r,b}$ is concave.

Rate-Latency Function is a horizontally shifted linear function (c.f. Figure A.2):

$$\beta_{R,T} = R \cdot max(0, t - T) \tag{6.33}$$

$\beta_{R,T}$ is convex.

Burst-Delay Function unit impulse function for Network Calculus systems (c.f. Figure A.3):

$$\delta_T(t) = \begin{cases} 0 & \text{for } t \leq T \\ \infty & t \text{ else} \end{cases} \tag{6.34}$$

Step Function (c.f. Figure A.4)

$$u_T(t) = 1_{t>T} = \begin{cases} 0 & \text{for } t \leq T \\ 1 & \text{else} \end{cases} \tag{6.35}$$

6.2.5 Interpretation of Curve Parameters

The rate of the bounded arrival flow is of major importance in curve definition. For curves based on linear function λ_R this is straightforward. They allow a maximum arrival rate of $\frac{\mathrm{d}\lambda_R}{\mathrm{d}t} = R$ to the bounded arrival flows. When λ_R is used for a lower bound, arrival flows are expected to have a rate of R at least.

Affine functions $\gamma_{r,b}$ are a very expressive function class for arrival curves. For upper bounds, equal to linear curves, they allow a peak arrival rate of r. Burst size parameter b gives some additional capacity for system arrivals that are not conform to rate r. To explain this in detail a result from [107] will be used: An arrival flow R that is bound by $\gamma_{r,b}$ is also compliant to a leaky-bucket system with leak rate r and bucket capacity b. This kind of imaginary device is sketched in Figure 6.8. It is based on the assumption that processing of R in the gray box requires fluid (e.g. water). The fluid is stored in the bucket with capacity b on top of the processor. When the system starts the bucket is completely filled. A pipe refills it with a constant rate r, when the bucket is filled additional fluid is lost (thus the bucket overflows without any further consequences). The processor removes fluid from the basin on demand, equal sized arrivals use up the same volume. If the bucket is depleted the processing has to stop until there is enough fluid available again.

With this physical model the parameter in $\gamma_{r,b}$ can be explained as follows: When the arrival flow has a rate smaller or equal to r the processor will never stop due to fluid shortages, so the arrival flow can pass by. For moments with arrival rates greater than r the processor can use the fluid reserve in the bucket. However, the arrivals that come in with higher rate may not require more than b fluid in sum. Such an arrival burst of size b

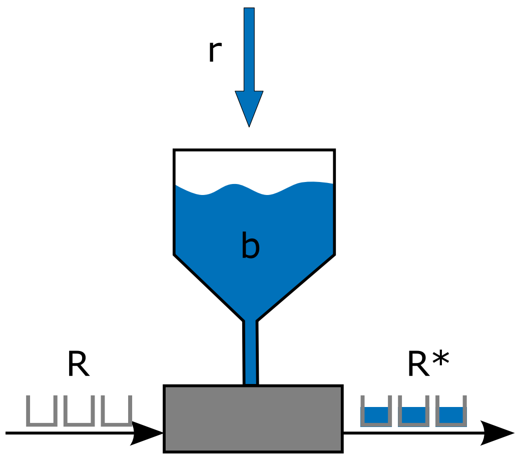


Figure 6.8: Leaky Bucket with bucket capacity b and refill rate r.

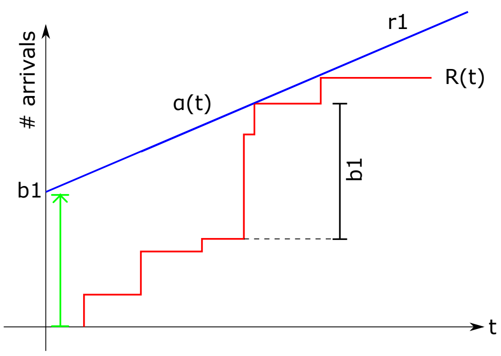


Figure 6.9: An upper arrival curve γ_{r_1,b_1} allows arrival bursts up to size b_1.

may happen only once or the resources get depleted, hence the arrival flow is allowed to have a burst of size b. It is also possible to distribute capacity b to several smaller bursts b_i with $\sum_i b_i \leq b$. It should be also said that the fluid level in the bucket can be restored within phases when the arrival rate is smaller than r, but independent of the phase length the reserve can only be filled up to capacity b.

Burst Control with Curve Compositions

Using one affine curve only can be sufficient in some models to bound arrival flows. However, a single burst limit may result in loose bounds. Within burst size b any variant of arrival process behavior is possible: an arrival of a single packet of size b or the instantaneous arrival of several small packets that sum up to burst size b. Theoretically these packets could have a infinitely high arrival rate. The work of Le Boudec and Thiran [107] associates the term burstiness with this behavior. Figure 6.9 shows an example of this situation. In the middle of the arrival flow plot a packet burst arrives, the packet arrival rate within the burst duration is very high. This rate is also referred as burst rate. So parameter b allows the modeler to set an upper bound on the burstiness, but gives no additional means to control the arrival rate within bursts.

An option to bound the burst rate in a more strict way is to use a second affine function in the curve. Let this curve be $\gamma_{r_2,b_2}(t) = r_2 \cdot t + b_2$. For the first affine function we also introduce $r = r_1$ and $b = b_1$ to unify the notation. New function γ_{r_2,b_2} is intended to regulate arrival rates of packets that arrive faster than r_1, hence it makes sense to require $r_2 > r_1$. Also the burst size should be limited, so we should have $b_1 > b_2$. Figure 6.11 shows the modified arrival curve as a combination of two simple functions. Except for the intersection point all t are double covered by γ_{r_1,b_1} and γ_{r_2,b_2}. We define our combined upper arrival curve α^U more precisely with the point-wise minimum of both functions:

$$\alpha^U(t) = \min\left(\gamma_{r_1,b_1}(t), \gamma_{r_2,b_2}(t)\right) \tag{6.36}$$

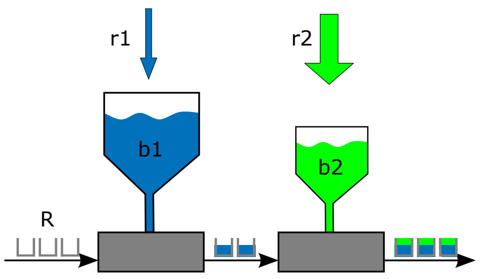


Figure 6.10: A tandem of leaky buckets equal to $\gamma_{r_1,b_1} \wedge \gamma_{r_2,b_2}$.

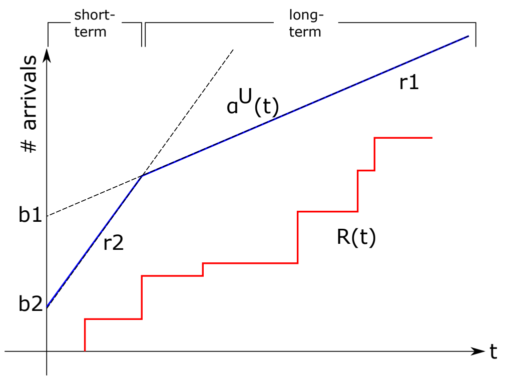


Figure 6.11: Short-term bursts are controlled with γ_{r_2,b_2}, the long-term behavior with γ_{r_1,b_1}.

Now with the second function arrival flow bursts are still possible, but they have to comply to rate r_2. In terms of arrival process modeling this can also be interpreted as setting bounds for different time intervals. Function γ_{r_2,b_2} bounds arrivals in the short-term while γ_{r_1,b_1} sets a bound for average arrivals in the long-term.

Again an analogy to leaky buckets for arrival curves with two affine functions can be constructed. An arrival flow bound by composition $\gamma_{r_1,b_1} \wedge \gamma_{r_2,b_2}$ can be processed by a tandem of leaky buckets with capacity b_1 and b_2 and leak rates r_1 and r_2. Figure 6.10 shows the construct. The tandem stops if one bucket runs out of fluid, so the first station does not queue for the second. As $b_2 < b_1$ and $r_1 < r_2$ the additional bucket can control the characteristics of bursts that have taken place outside the control domain of the first. This is independent of the ordering of the buckets [107].

To generalize arrival curves based on piecewise-linear functions we use the following notation. For upper arrival curves a set of functions γ_{r_i,b_i}, $i \in 1 \dots n$

$$\alpha^U = \min_i \left(\gamma_{r_i,b_i}\right) \tag{6.37}$$

Using concave piecewise-linear functions as upper arrival curves adds clarity to the arrival process modeling, allows approximating continuous functions and simplifies computations. In a visual interpretation, each affine function adds a line segment to the arrival curve with semantics of rate and burst size. The modeler can interpret the piecewise-linear function as a combination of leaky buckets. Additionally, the curve as a process model stays simple. Modeling the worst-case is supported by concave upper bounds as well. Using the construction method in Equation (6.37), the concave arrival curve includes arrival rates in decreasing order. The rates and sizes of arrival bursts are visible to the modeler.

Apart from user advantages, concave functions have desirable properties for (min,+)-related operations. In [107, Definiton 1.2.4] and [107, Corollary 3.1.1] Le Boudec and Thiran define a set of “good” functions with the following equivalent properties for a function $f \in \mathcal{F}$:

1. f is sub-additive and $f(0) = 0$

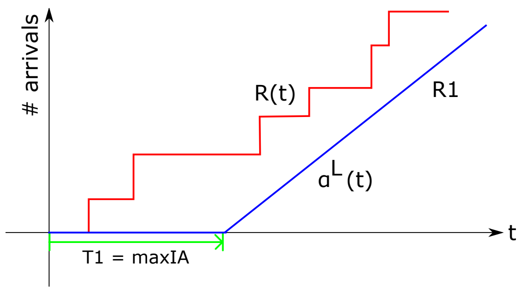


Figure 6.12: A lower arrival curve β_{R_1,T_1} allows interarrival times up to size T_1.

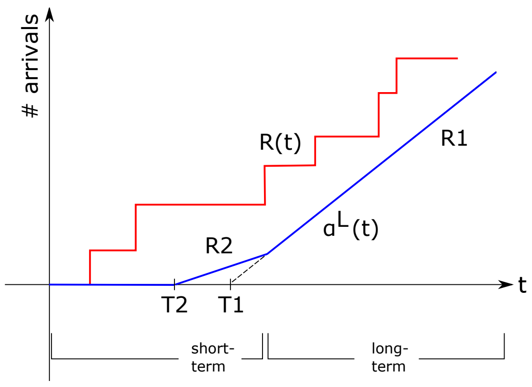


Figure 6.13: Short interarrival times and minimum arrival rate are controlled with β_{R_2,T_2}, the long-term behavior with β_{R_1,T_1}.

2. $f = f \underline{\otimes} f$

3. $f = \overline{f}$

Property 3 follows immediately from property 2 and the definition of the sub-additive closure. Arrival curves constructed as shown in (6.37) are members of the set of "good" functions. Due to the definition of affine functions we have $\gamma_{r,b}(0) = 0$ and $\gamma_{r,b}$ is concave, thus piecewise-linear arrival curves are sub-additive and property 1 is fulfilled.

By property 3 concave piecewise-linear functions are equal to their sub-additive closure. Using them as arrival flow envelope implies bounds that include all necessary, but no redundant information. Additionally for "good" functions f, g Lemma 6.3 applies, thus $f \underline{\otimes} g = f \wedge g$. This result can be used to simplify and accelerate implementations of (min,+)-related operations.

Inter-Arrival Time Control with Lower Bounds

As for upper arrival curves, the arrival rate is also central for lower envelopes. With a lower arrival curve defined by a linear function λ_R the arrival rate can be bounded from below to be at least rate R or higher.

The convex equivalent for affine curves are rate-latency functions $\beta_{R,T}$ (Figure A.2). They define a minimum arrival rate of R and include a translation T that, if positive, shifts the linear function part to the right. For lower envelopes arrival flow T has the semantics of the maximum interarrival time. Figure 6.12 shows this interpretation. Offset T marks the longest theoretical time interval with no arrivals and thus an arrival rate of 0. Within this interval the flow rate may decrease to values smaller than rate R or completely stall. For rate-latency curves used as lower envelopes is no figurative model similar to leaky buckets. In Chapter 7 an alternative description based on "debt" towards the arrival flow is developed.

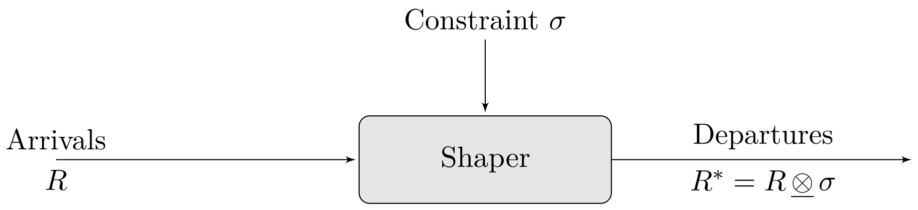


Figure 6.14: A shaper enforcing an arrival curve σ on arrival flow R.

For more detailed lower arrival envelopes two rate-latency curves β_{R_1,T_1} and β_{R_2,T_2} with $R_1 > R_2$ and $T_1 > T_2$ can be combined using operator max() as shown in Figure 6.13. Analogously, the upper case β_{R_1,T_1} controls the arrival flow behavior in the long-term, requires a rate of R_1 and allows a time interval of length T_1 with a lower rate. However, the throttling of arrival flow in short-term can be controlled with the second function β_{R_2,T_2}. Constant T_2 defines the maximum time interval with a flow rate of 0 and R_2 is the minimum arrival rate while the flow drops below long-term rate R_1. In general, a convex version for lower arrival curves is formed with the maximum operator:

$$\alpha^L = \max_i (\beta_{R_i,T_i}) \tag{6.38}$$

6.2.6 Shaper Elements

A shaper is a Network Calculus model element that enforces an arrival curve constraint to an arrival flow (traffic regulation). So for an arbitrary arrival flow R passing through a shaper implementing curve σ the output flow is [107]

$$R^* = R \underline{\otimes} \sigma \tag{6.39}$$

Equivalently one can see σ as the impulse response of the shaper [38]. In [38] the construct is called a maximal f-regulator with curve f. Figure 6.14 shows the shaper with input and output relationships.

Shapers can be classified by the way they handle the additional traffic part that is not compliant to σ. A policer [107, Section 1.5] just tags packet arrivals as 'conform' or 'non-conform' and forwards all traffic, thus the part of departing packets marked as conform forms R^*. A greedy shaper [107] is equipped with a cache or buffer to store non-conformant arrivals (e.g. an Ethernet card). In Figure 6.15 an arrival flow to a shaper is plotted together with the shaping curve. Since some arrivals are not conform to the constraint σ (marked as gray areas) they are buffered. As soon as the arrival flow drops below the arrival curve constraint σ the difference is emitted from the buffer. Figure 6.16 shows the resulting departure flow.

The buffer occupation $x(t)$ of a greedy shaper enforcing arrival constraint σ is given by

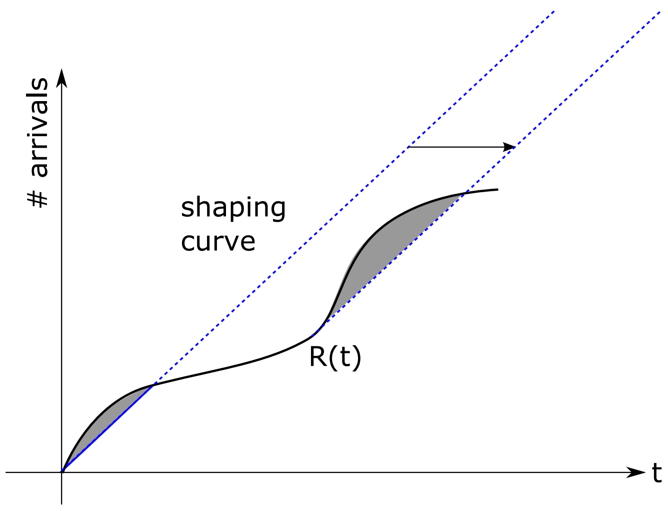


Figure 6.15: Input flow to a shaper, its linear shaping curve is the dotted blue line. Non-conform traffic is marked gray.

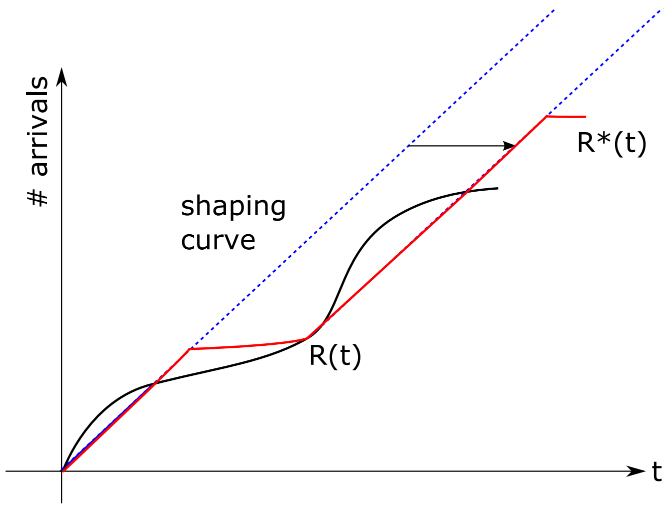


Figure 6.16: The shaper buffers the output of non-conform traffic. Outgoing traffic is always conform.

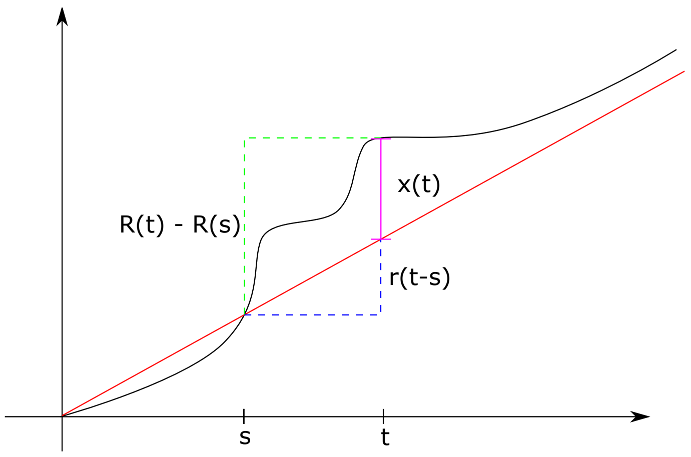


Figure 6.17: Buffer size requirement in burst phases: maximum difference between flow R and linear shaping curve $\sigma(t) = rt$ in all intervals $[s, t]$.

[107, Corollary 1.5.2]:

$$\begin{aligned}
x(t) &= v\left(R, R^*\right) && (6.40)\\
&= R(t) - R^*(t) && (6.41)\\
&= R(t) - (R \underline{\otimes} \sigma)(t) && (6.42)\\
&= R(t) - \inf_{0\leq s\leq t} \{R(s) + \sigma(t-s)\} && (6.43)\\
&= \sup_{0\leq s\leq t} \{-R(s) - \sigma(t-s)\} && (6.44)\\
&= \sup_{0\leq s\leq t} \{R(t) - R(s) - \sigma(t-s)\} && (6.45)
\end{aligned}$$

This is the maximum difference between arrivals in the interval and the output allowed by the shaping curve in interval $t - s$. Figure 6.17 shows the application of used terms.

Lemma 6.6 (Arrival Constraints with Shapers)**.** *A flow R constrained by σ passing through a shaper implementing σ does not require buffering.*

Proof. Based on backlog $x(t) = v\left(R, R^*\right)$ we can write

$$\begin{aligned}
x(t) &= R(t) - R^*(t) && && (6.46)\\
&\leq R \underline{\otimes} \sigma - R^*(t) && \text{Theorem 6.4} && (6.47)\\
&= R \underline{\otimes} \sigma - R \underline{\otimes} \sigma && \text{Definition Shaper} && (6.48)\\
&= 0 && && (6.49)
\end{aligned}$$

□

As a consequence, saying a flow is constrained by σ is equivalent to saying a flow is passing a shaper with curve σ and no buffering/backlog occurs.

Definition 6.2.9 (Shaper Conformity)**.** *A flow is σ-conform to a shaper implementing limit σ when no buffering is required while the flow passes the shaper.*

6.2.7 Service Flows

Whenever an arrival to a network element has to be processed, routed or delivered the component has to invest service capacity to the arrival. The delivered service towards the arrival flow can be measured in resource units. In Queueing Systems one would describe the service (or resource consumption) process by its service time distribution. Network Calculus uses a resource flow expressing the delivered work units at any point in time.

In the original work of Le Boudec and Thiran [107] the concept of a resource flow is not used, instead the service curve abstracting this flow is introduced as the missing link between R and R^*. Here the resource flow $C(t)$ is adopted from Real-Time Calculus [105] as it seems to be more intuitive for the reader. In the Network Calculus system model (Figure 6.3) $C(t)$ expresses the service resources available to the network element.

The amount of service resources available to a system at time t is expressed with the resource function $c(t)$. It has to be pointed out that $c(t)$ does not necessarily describe the consumed resources at t, it includes the service the system *could* deliver. As for arrivals and departures a cumulated flow of a potential service in interval $[0, t]$ can be formed.

Definition 6.2.10 (Resource Flow)**.** *Let $c(t)$ be the number of service resources available to a service process at time t. Then the integral*

$$C(t) = \int_0^t c(x)\,dx \tag{6.50}$$

is the resource flow of the service process in interval $[0, t]$*.*

The definition assumes a fluid model for $C(t)$ as there must be a derivative $c(t)$ (c.f. Section 6.2.1). Resource Flow $C(t) \in \mathcal{F}$ is the amount of service capacity applicable to arrivals by a system in interval $[0, t]$.

6.2.8 Service Curves

The idea of generalizing cumulated arrival functions with arrival curves has proven to be useful. When the resources available to a system are expressed with $C(t)$, a similar abstraction can be made. Service curves reflect scheduling, delays and service rates of a system. Arrivals can be buffered or queued at an element that offers a service curve. In contrast to Queueing Systems, Network Calculus does not specify how this buffering happens. The queueing discipline (FIFO, LIFO, etc.) is left open and implicitly encoded in the service curve.

Again upper and lower bounds for service flows can be given. In [107] the lower bound of resource flows is introduced as service curve and later [107, Definition 1.6.1] extended to a maximum service curve indicating the upper boundary. As already done with resource flow C we adopt the notation of Real-Time Calculus [103–105].

Definition 6.2.11 (Lower Service Curve)**.** *Let $C(t)$ be a resource flow. Its lower service curve β^L satisfies:*

$$\beta^L(t-s) \le C(t) - C(s) \quad \forall\, 0 \le s \le t \tag{6.51}$$

The lower service curve is a bound from below on the service a system is able to deliver to the arrival flow in any time interval.

As the service capacity of a system is likely to be limited an upper bound makes sense also. Upper service curve β^U sets a bound from above on the resource flow: the sum of performed work units up to time t will never exceed $\beta^U(t)$. The upper service curve is also the impulse response of a network element (c.f. Section 6.1.4).

Definition 6.2.12 (Upper Service Curve)**.** *Let $C(t)$ be a resource flow. Its upper service curve β^U satisfies:*

$$\beta^U(t-s) \geq C(t) - C(s) \quad \forall\, 0 \leq s \leq t \tag{6.52}$$

A detailed work on different service curve classes can be found in [30].

Service Curve Instances

A service curve β is required to be wide-sense increasing and $\beta(0) = 0$, thus $\beta \in \mathcal{F}$. Again the function set used in practical modeling is rather limited to keep the parameter set small and simplify computations. Members in the catalog of wide-sense increasing functions for arrival curves in Section 6.2.4 can also be chosen for service curves. Piece-wise linear functions like λ_R and $\beta_{R,T}$ express a single service rate. Function compositions can be used to approximate more detailed service curves.

For $\lambda_R(t) = R \cdot t$ parameter R can be interpreted as a fixed service rate. A server with $\beta^U = \beta^L = \lambda_R(t)$ can draw $\frac{\mathrm{d}\lambda_R}{\mathrm{d}t} = R$ resources per time unit from the resource flow. Linear service curves are not suitable to model interruptions and slowdowns. The only option would be to average reduced and normal service rates by choosing a lower value for R, but this would repeat mistakes of Queueing Systems. An advantage of the service curve concept is the ability to separate latencies from long-term rates. For this reason the piece-wise linear rate-latency function $\beta_{R,T}(t)$ is common for service curve definitions. Parameter R is the sustainable service rate, it can be continuously delivered. Delays are modeled with T as system latency. By definition of rate-latency curves T shifts the previously linear function to the right. As a result, $\beta_{R,T}(t) = 0$ for $t \in [0, T]$ indicates no service at all in this interval.

Example 6.2.1. *The manufacturer of a hard disk type claims his device can read at least 100 MB/s. Due to the nature of the rotating discs one has to wait until the surface with requested data moves under the heads. The mean access time is 60 ms, thus a full rotation gives 120 ms latency in the worst-case. The service curve offered by this drive is $\beta^L(t) = \beta_{0.12,100}(t) = max(0, 100(t - 0.12))$.*

To model service processes in more detail, service curves can be composed from basic functions. For convex curves an approach equal to lower arrival curves in Section 6.2.5 can be applied.

6.2.9 Server Elements

As in Queueing Systems the output process of a system depends on the input and service processes. With arrival and service curves, there are tools at hand to give upper and

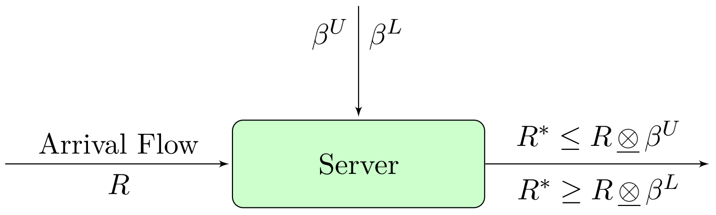

β^U
β^L
Arrival Flow
R
Server
$R^* \leq R \underline{\otimes} \beta^U$
$R^* \geq R \underline{\otimes} \beta^L$

Figure 6.18: A server providing at least a service of β^L and a maximum service of β^U to arrival flow R.

lower bounds on the last two processes. Using them in Network Calculus allows one to calculate deterministic envelopes for the output process. Again (min,+)-convolution is a key operator in computations.

The second Network Calculus element next to shaper is the server representing a service guarantee. While in [107, Definition 1.3.1] servers are implicitly introduced with service curves, [38, Definition 2.3.1] defines a so-called f-server with service curve f.

Definition 6.2.13 (Server Element)**.** *A network element that guarantees a service curve $\beta^L \in \mathcal{F}$ to an arrival flow R such that the departure flow is given by*

$$R^* \geq \beta^L \underline{\otimes} R \tag{6.53}$$

is called a server element.

In [107, Definition 1.6.1] the server is extended to include upper service curves $\beta^U \in \mathcal{F}$ by satisfying $R^* \leq \beta^U \underline{\otimes} R$. Figure 6.18 shows a single server with upper and lower service bounds. Figure 6.19 displays the relationship between an arrival flow R, lower service curve β^L and the lower output bound given by $R^* \geq R \underline{\otimes} \beta^L$. The convolution is applied to the convex curve β^L at every point of R, the curve is moved along the arrival flow. Resulting $R^*(t)$ is shown as a dashed line, it is the lower envelope of the departure flow when the system delivers the lowest service. For upper envelopes the case is similar to the application of arrival curves to flows, Figure 6.6 with $\alpha^U = \beta^U$ would be a valid example.

Corollary 6.7 (Lower Departure Bound)**.** *For a server processing an arrival flow bounded by α^L from below with lower service curve β^L the output flow is bound from below by curve*

$$\alpha^{L\prime} \geq \alpha^L \underline{\otimes} \beta^L \tag{6.54}$$

Proof. Replace R with α^L in Definition 6.2.13. ◻

A special case of servers is the delay element. It is presented in [42] or in the examples of [107, Section 1.3.1]. Delay elements introduce a maximum delay T between arrival and departure flows by using the burst-delay function δ_T as service function. Except for latency the flow passes the element without further modifications, so $h(R, R^*)(t) = T$ for all t holds.

Example 6.2.2. *A delay element implementing arrival curve constraint δ_0 has no effect on the arrival flow. δ_0 is the neutral element for (min,+) convolution (c.f. Section 6.1.5).*

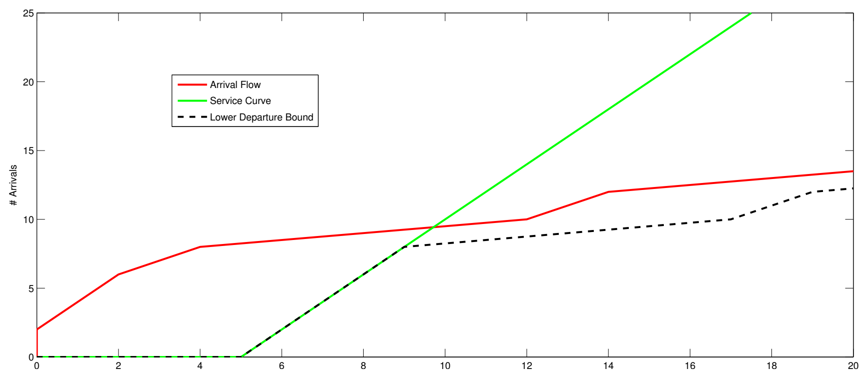


Figure 6.19: Effect of a service curve β^L convolved with arrival flow R: The lower departure flow bound is $R^* = R \underline{\otimes} \beta^L$.

Increased Departure Flow Burstiness

Definition 6.2.13 gives the lower output flow bound for servers by shifting arrival bounds to the right: Processing time is added according to service curves. For upper departure flow bounds backlogs during processing have to be considered. Whenever a flow is processed by a server with non-zero delay arrivals have to be buffered in the element. In worst-case the buffered arrivals can be emitted to the departure flow at the rate specified by the service curve. This behavior adds some variance to the departure flow, or, in wording of Network Calculus, results in an increase of burstiness. Its limit can be computed by [107, Theorem 1.4.3]:

Theorem 6.8 (Upper Departure Bound (Server)). *For a server processing an arrival flow conform to α^U with lower service curve β^L the output flow is bound from above by curve*

$$\alpha^{U\prime} = \alpha^U \underline{\oslash} \beta^L \tag{6.55}$$

For a stable system $\alpha^U \leq \alpha^{U\prime}$ holds. Figure 6.20 shows the effect to an arrival curve. For all time t the maximum backlog that might be observed in future is added by shifting the curve vertically. For example, the reader may consider the maximum backlog respective of the vertical deviation at $t = 5$ that is reflected in $\alpha^{U\prime}(0)$.

6.3 Worst-Case Bound Analysis

Network Calculus is an excellent tool to get deterministic performance bounds for systems under worst-case conditions. To achieve this the modeler has to omit flows in general and start to work with the abstracting curve concept only.

Any instance of an arrival flow R can be bounded from above by α^U. To reenact the worst-case we set $R = \alpha^U$ to get the maximum number of arrivals within any time interval. For network elements that feature a resource flow C the worst-case is given by a shortage of resources. We can assume that $C = \beta^L$ is the minimum service capacity for a server. Thus, arrival and service curves are not only a characterization of system processes, they also provide the worst-case situations for their processes.

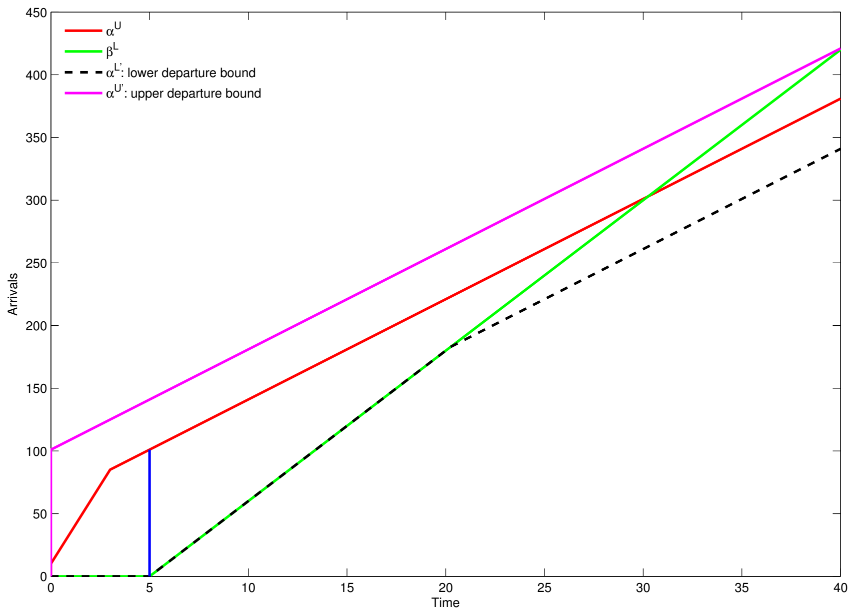


Figure 6.20: Increased Departure Flow Burstiness for Servers due to full backlog processing in best-case (worst-case for departure flows).

6.3.1 Backlog Bound

By assuming the worst-case situation described above, the maximum backlog is given by

$$b_{\max}\left(\alpha^U,\beta^L\right)=\max_{t\geq 0}\left\{v\left(\alpha^U,\beta^L\right)\right\} \tag{6.56}$$

using the notation of Definition 6.2.2 and the result from ([107, Theorem 1.4.1]).

For example, $b_{\max}\left(\alpha^U,\beta^L\right)$ can be used to dimension the buffer size in a router distributing network packets with a minimum service β^L. When the arrival flow is enforced to stay below curve α^U the computed buffer size will never be exceeded in any situation.

Maximum backlog $b_{\max}\left(\alpha^U,\beta^L\right)$ does not exist for all combinations of α^U and β^L. This is the case when β^L is not sufficient to process the arrival flow. The existence of $b_{\max}\left(\alpha^U,\beta^L\right)$ can be exploited to formulate a stability criterion for a system (implicit in [107, Theorem 1.4.1]): A system with input bound α^U and service envelope β^U is stable if $b_{\max}\left(\alpha^U,\beta^L\right)<\infty$. In Queueing Systems this corresponds to the requirement $\rho<1$ on utilization.

6.3.2 Maximum System Latency

To find the maximum system latency a system is also investigated under worst-case conditions for the same arguments as for the backlog: When R is replaced with α^U the worst-case for arrivals is used. R^* is redundant data as its lower envelope can be computed

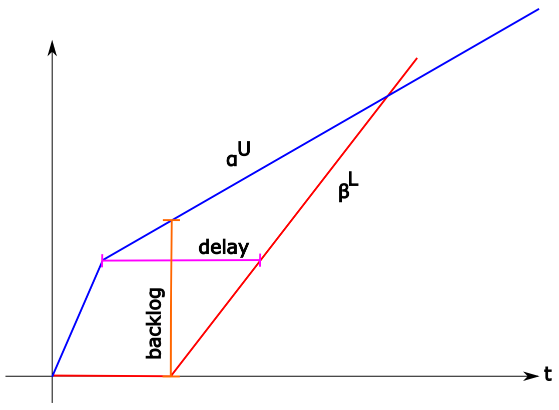


Figure 6.21: Delay and backlog for the worst-case with an arrival flow equal to α^U and service equal to β^L.

by $R \underline{\otimes} \beta^L$. Assuming the worst-case, we can also write $\alpha^U \underline{\otimes} \beta^L$ and using Definition 6.2.5 we can conclude

$$h_{\max}\left(\alpha^U, \alpha^U \underline{\otimes} \beta^L\right) \tag{6.57}$$

to be the maximum system latency that is guaranteed in any situation for a system with bounds α^U and β^L. In [107, Theorem 1.4.2] it has been shown that the convolution in Equation 6.57 can be avoided. Using the notation above we get

$$h\left(R, R^*\right)(t) \le h_{\max}\left(\alpha^U, \beta^L\right) \text{ for all } t \tag{6.58}$$

Figure 6.21 illustrates the principle of backlog and maximum delay when applied to bounding curves.

6.4 Networks of Network Calculus Elements

In this section we will consider serial and parallel constructs of Network Calculus elements. Elements are combined and the new arrival, service and departure bounds are formed.

An interesting fact about Network Calculus has to be stated in the beginning: Parallelization of elements in Network Calculus can seldom be found in literature, this holds especially for the deterministic Network Calculus presented here. This results from the bias of Network Calculus towards worst-case results and the specifics of the underlying (min,+)-algebra. For shapers Network Calculus gives the same algebraic results for serial and parallel constructs, therefore there is no need to include them explicitly.

6.4.1 Shaper Concatenation

Shaper can be concatenated to serial networks. For a tandem of two shapers the concatenation has a shaping curve given by the (min,+)-convolution of both original curves. The theorem and proof are based on [38, Theorem 2.2.7] and its proof.

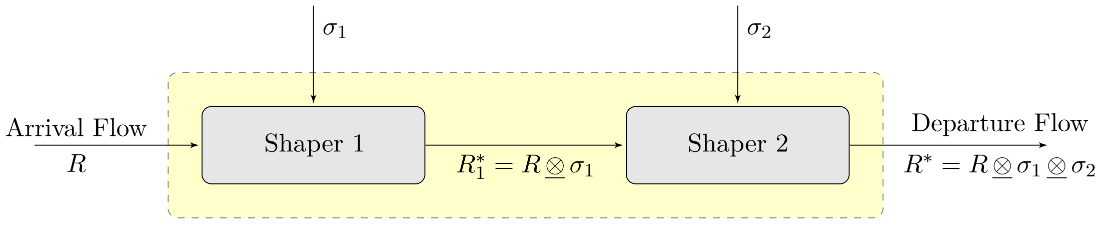

σ_1
σ_2
Arrival Flow
R
Shaper 1
$R_1^* = R \underline{\otimes} \sigma_1$
Shaper 2
Departure Flow
$R^* = R \underline{\otimes} \sigma_1 \underline{\otimes} \sigma_2$

Figure 6.22: Two shapers connected to a tandem limit an arrival flow to $\sigma_1 \underline{\otimes} \sigma_2$.

Theorem 6.9 (Tandem Shaper)**.** *Let σ_1 be the shaping curve of the first shaper and σ_2 of the second. A concatenation of both shapers has shaping curve σ_c given by*

$$\sigma_c = \sigma_1 \underline{\otimes} \sigma_2 \tag{6.59}$$

The result is independent of the shaper arrangement.

Proof. Let R be an arrival flow entering the first shaper. Then the output R_1^* is given by

$$R_1^* = R \underline{\otimes} \sigma_1 \qquad \text{by Eqn. (6.39)} \tag{6.60}$$

When R_1^* is fed into the second shaper the flow is limited to

$$R^* = R_1^* \underline{\otimes} \sigma_2 \qquad \text{by Eqn. (6.39)} \tag{6.61}$$

Combining (6.60) and (6.61) we get

$$\begin{aligned} R^* &= (R \underline{\otimes} \sigma_1) \underline{\otimes} \sigma_2 && (6.62)\\ &= R \underline{\otimes} (\sigma_1 \underline{\otimes} \sigma_2) && \text{using associativity of } \underline{\otimes} \quad (6.63) \end{aligned}$$

Independence of the shaper ordering follows from the associativity of (min,+)-convolution (Section 6.1.5). □

Serial Composition is not limited to tandem systems. Repeating the tandem concatenation step by step for n shapers with shaping curves $\sigma_1 \ldots \sigma_n$ gives

$$R^* = R \underline{\otimes} \bigotimes_{i=1}^{n} \sigma_i \tag{6.64}$$

6.4.2 Shaper Parallelization

In the book of Chang [38, Theorem 2.2.9] the parallel ordering of two shapers (“maximal f-regulators”) is described as ”filter bank summation“. Figure 6.23 shows the parallel construct, it applies the shaping curve of each shaper simultaneous to the arrival flow. Hence semantics of the left distribution point marked with ”&“ are the duplication of flow R to flows R_1 and R_2 with $R = R_1 = R_2$. For the right junction point semantics are the minimum service of both servers to the original flow R.

Theorem 6.10 (Filter Bank Summation)**.** *For two shapers with shaping curves σ_1 and σ_2 the parallel construct is a shaper with*

$$\sigma(t) = \sigma_1 \wedge \sigma_2 \tag{6.65}$$

if $\overline{\sigma_1} \wedge \overline{\sigma_2}$ is sub-additive.

The proof is also from [38]:

Proof. The theorem can be rewritten as

$$\begin{aligned}
R^* &= R_1^* \wedge R_2^* && (6.66)\\
&= (R \underline{\otimes} \overline{\sigma_1}) \wedge (R \underline{\otimes} \overline{\sigma_2}) && (6.67)\\
&= R \underline{\otimes} (\overline{\sigma_1} \wedge \overline{\sigma_2}) \qquad \text{distributivity of } \underline{\otimes} && (6.68)
\end{aligned}$$

As $\overline{f_1} \wedge \overline{f_2}$ is sub-additive and $\overline{f_{1,2}}(0) = 0$ one can replace the term with its sub-additive closure

$$\begin{aligned}
R^* &= R \underline{\otimes} \overline{(\overline{\sigma_1} \wedge \overline{\sigma_2})} && (6.69)\\
&= R \underline{\otimes} (\overline{\overline{\sigma_1}} \underline{\otimes} \overline{\overline{\sigma_2}}) \qquad \text{by [38, Lemma 2.1.5 (xi)]} && (6.70)\\
&= R \underline{\otimes} (\overline{\sigma_1} \underline{\otimes} \overline{\sigma_2}) && (6.71)\\
&= R \underline{\otimes} \overline{(\sigma_1 \wedge \sigma_2)} && (6.72)
\end{aligned}$$

□

In combination with Lemma 6.3 we get the interesting result that shapers in parallel yield the same bounds as serial concatenated shapers.

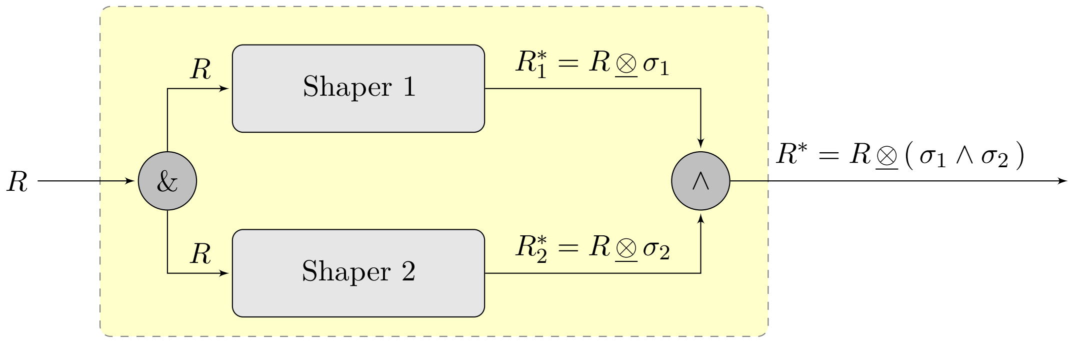

R
R
Shaper 1
$R_1^* = R \underline{\otimes} \sigma_1$
&
$\wedge$
$R^* = R \underline{\otimes} (\sigma_1 \wedge \sigma_2)$
R
Shaper 2
$R_2^* = R \underline{\otimes} \sigma_2$

Figure 6.23: Two shapers in parallel (Filter Bank Summation [38])

6.4.3 Server Concatenation

Combining servers is equal to the combination of their service curves, therefore their impulse responses can be concatenated. This allows one to combine several model elements to a single model description. Figure 6.24 shows two servers in a tandem configuration. The offered service curve is given by the following theorem [107, Theorem 1.4.3]:

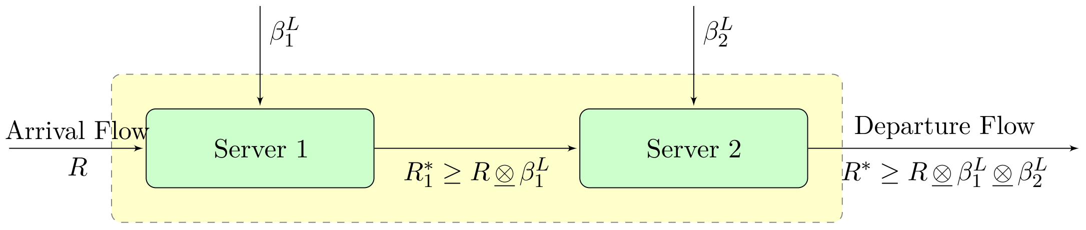

β_1^L
β_2^L
Arrival Flow
R
Server 1
$R_1^* \geq R \underline{\otimes} \beta_1^L$
Server 2
Departure Flow
$R^* \geq R \underline{\otimes} \beta_1^L \underline{\otimes} \beta_2^L$

Figure 6.24: Two servers connected to a tandem offer service curve $\beta_1^L \underline{\otimes} \beta_2^L$ to an arrival flow.

Theorem 6.11 (Server Concatenation)**.** *Two servers in sequential concatenation with individual service curves β_1^L and β_2^L are traversed by an arrival flow. The combined system offers a service curve of $\beta_1 \underline{\otimes} \beta_2$.*

We repeat the proof from [107, Theorem 1.4.3].

Proof. Assume two servers S_1 and S_2 with service curves β_1^L and β_2^L. Both servers are in a tandem configuration (Figure 6.24). Arrival flow R is processed in S_1 first, the output is

$$R_1^* \geq R \underline{\otimes} \beta_1^L \tag{6.73}$$

R_1^* is fed into S_2 and finally emitted as

$$\begin{aligned} R_2^* &\geq R_1^* \underline{\otimes} \beta_2^L & (6.74)\\ &\geq (R \underline{\otimes} \beta_1^L) \underline{\otimes} \beta_2^L & (6.75)\\ &= R \underline{\otimes} (\beta_1^L \underline{\otimes} \beta_2^L) & (6.76) \end{aligned}$$

So the combined service curve of the tandem system is $\beta_1 \underline{\otimes} \beta_2$. □

Shapers and Tandem Servers

The combined service curve of tandem servers can of course be used to compute backlog and delay bounds. For arrival curve α^U and service curves β_1^L, β_2^L we have a maximum backlog of $b_{\max}\left(\alpha^U, \beta_1^L \underline{\otimes} \beta_2^L\right)$ and a maximum delay of $h_{\max}\left(\alpha^U, \beta_1^L \underline{\otimes} \beta_2^L\right)$. For both bounds the arrival curve to the first server is of importance, while the limits for the flow from server 1 to server 2 are neglected. The reason is the commutative property of $\underline{\otimes}$, it does not matter if server 1 or 2 processes the arrivals first. A comparable assumption is found in product-form Queueing Networks: A Poisson arrival process with rate λ processed by a queue keeps its rate at departure (if the system is stable). As a result, all queues in a concatenation have the same arrival rate and can be rearranged, populations for single queues can be combined with the commutative sum for global analysis.

For upper limits on backlog and delay in server tandems the increase of burstiness by Theorem 6.8 has no impact, too. Even more, in a quite astonishing result presented in [107, Theorem 1.5.2] Le Boudec and Thiran show that adding a shaper to manipulate

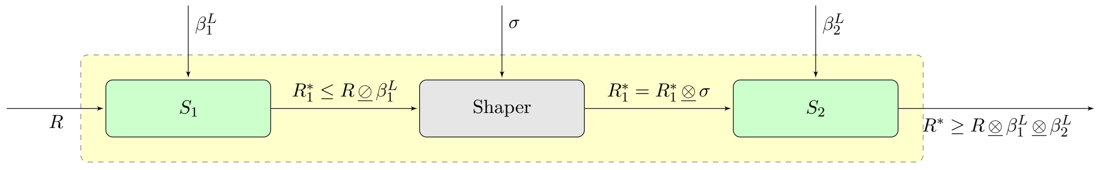

β_1^L
σ
β_2^L
R
S_1
$R_1^* \le R \,\underline{\oslash}\, \beta_1^L$
Shaper
$R_1^* = R_1^* \,\underline{\otimes}\, \sigma$
S_2
$R^* \ge R \,\underline{\otimes}\, \beta_1^L \,\underline{\otimes}\, \beta_2^L$

Figure 6.25: In a server tandem a shaper is used to prepare the departure flow of the first server for the second.

the arrival flow does not influence the results. Consider the system in Figure 6.25. Two servers S_1 and S_2 from a tandem system are serving an arrival flow α^U. Each server offers a service limited by β_1^L respectively β_2^L. The output flow of S_1 suffers from a burstiness increase due to Theorem 6.8, it is limited from above by $\alpha^U \,\underline{\oslash}\, \beta_1^L$. To smooth out those bursts towards S_2 a greedy shaper is placed in between. It implements a shaping curve σ with $\sigma(t) \ge \alpha^U(t)$ for all t. For the described system [107, Theorem 1.5.2] states that the shaper can be ignored from the perspective of maximum backlog and delay analysis.

The proof is based on commutativity of (min,+)-convolution, due to its relevance for SLA Calculus we repeat a variation using maximum delay instead of backlog here. Since the shaping curve is also an service curve ([107, Corollary 1.5.1]) offered by the shaper to the arrival flow the delay bound of the system is:

$$h(\alpha, \beta_1^L \,\underline{\otimes}\, \sigma \,\underline{\otimes}\, \beta_2^L) = h(\alpha, \sigma \,\underline{\otimes}\, \beta_1^L \,\underline{\otimes}\, \beta_2^L) \tag{6.77}$$

As one can see, the delay bound is not affected by the position of the shaper in the system. Furthermore, since $\sigma \ge \alpha$, it is true that

$$h(\alpha^U, \sigma \,\underline{\otimes}\, \beta_1 \,\underline{\otimes}\, \beta_2^L) = h(\alpha^U, \beta_1^L \,\underline{\otimes}\, \beta_2) \tag{6.78}$$

by [107, Lemma 1.5.2], a shaper at the system entrance will not modify the flow at all. Thus, the worst-case delay and, in a similar argumentation, the backlog are not influenced by adding a greedy shaper.

We again emphasize that this result of transparent shapers holds only for worst-case delay bounds and not for average delay.

6.4.4 Server Parallelization

Similar to parallel shapers Chang in [38, Theorem 2.3.5] gives a result for the lower service curve of parallel servers. “Filter bank summation” for servers handles arrival flows equal to parallel shapers (see Figure 6.26). An arrival flow R is forked to flows R_1 and R_2. After passing the servers departure flows R_1^* and R_2^* are joined to a single flow R^* again. Here it is important that parallel servers in Network Calculus (and thus the connectors & and $\wedge$ in Figure 6.26) combine the semantics of a (de-) multiplexer for routing and fork/join. In case of fork/join $R = R_1 = R_2$ is given by duplication for synchronization. For the routing case $R = R_1 + R_2$ semantics are based on the worst-case assumption of Network Calculus: The worst theoretic router sends all arrivals to the bottleneck component, thus the system performance is determined by the weakest server.

Hence the combined system capacity is given by the following theorem [38]:

Theorem 6.12 (Service of Parallel Servers)**.** *Let S_1 and S_2 be two servers with service curves β_1^L and β_2^L. Both server in parallel offer a service curve of $\beta_1^L \wedge \beta_2^L$ to the arrival flow.*

Proof. The proof equals proof 6.4.2 with an inequality instead of an equality [38]. □

The consequence of this result is that in parallel systems the slowest element determines the output bounds of the departure flow and thus the response time. According to the author's knowledge, literature to Network Calculus does not extend beyond this result of Chang on parallelization of servers. In terms of modeling it does not pay off to include parallel elements in Network Calculus as they can be replaced with their serial counterpart. As a consequence, demultiplexing for packet flows is a "neglected research field" [97].

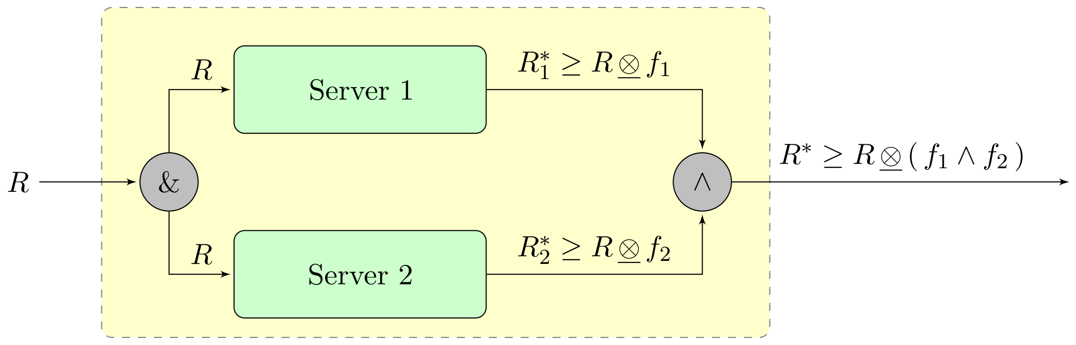

R
R
Server 1
$R_1^* \geq R \underline{\otimes} f_1$
&
$\wedge$
$R^* \geq R \underline{\otimes} (f_1 \wedge f_2)$
R
Server 2
$R_2^* \geq R \underline{\otimes} f_2$

Figure 6.26: Two servers in parallel

6.5 Real-Time Calculus

With Network Calculus a prominent representative of (min,+)-based calculi has been introduced. The descriptive power of curve contracts can be extended to other modeling domains apart from packet networks. Real-Time Calculus is a variant of Network Calculus for computing upper and lower performance bounds in models for real-time systems [124]. Real-time Systems are characterized by the ability to finish computation tasks within a given deadline. Typically such kind of performance guarantees are required in embedded systems controlling a machine, vehicle or consumer electronics. It is a good example how ideas from Network Calculus initially can be transferred to other application domains.

There is no single work in literature serving as authoritative definition of Real-Time Calculus in the same way as the works of Le Boudec and Thiran [107] or Chang [38] do for Network Calculus. Real-Time Calculus is the union of of results presented in a set of papers. Beginning from the initial work [103] Real-Time Calculus evolved in several papers [36, 37, 104–106, 124] to a modeling framework. The PhD thesis of Wandeler [122] includes a extensive summary on Real-Time Calculus. A detailed and critical comparison of Network Calculus and Real-Time Calculus from the perspective of service curves is provided by Bouillard et. al. [30]. Due to the different application domains we cannot provide one single system model for Real-Time Calculus. However, there is a core of common ideas that are summarized in the following.

6.5.1 System Model

The basic model elements in Real-Time Calculus are resources that process arriving entities. Such a resource can be a processor, a router or a specific subsystem in an (embedded) system component. As in Queueing Theory and Network Calculus the granularity of the model is subject to the level of detail chosen by the modeler.

In Real-Time Calculus a more general arrival process model than in Network Calculus is used [36, 37]. Instead of packets event flows are modeled. An incoming event can be, for example, a processing job in an embedded system. Event flow R has, except for the modeled quantities, the same properties as a Network Calculus arrival flow (Section 6.2.1). It is bounded with arrival curves from above (α^U) and below (α^L) [36, 37].

The number of processing units (e.g. CPU cycles) available to an element in a time interval is denoted with resource flow $C(t)$, its upper and lower envelopes are given by service curves β^L and β^U [36]. Real-Time Calculus requires all model elements to conform to a strict service curve [122, 124, Section 2.3.1]. We repeat the definition of [30], an equal definition is provided by [107]:

Definition 6.5.1 (Strict Service Curve)**.** *A system offers a strict service curve* β^L *if, during any backlogged period* $]s,t]$ *,* $R^*(t) - R^*(s) \geq \beta^L(t-s)$ *holds.*

In practice, a system implementing a strict service curve has a guaranteed output that does not drop below β^L.

Figure 6.27 shows a single Real-Time Calculus resource that accepts an incoming event flow R. Processed events leave the resource in form of departing arrival flow, notation R' instead of R^* is used. The resource also accepts a flow C of processing capabilities that are consumed when R is processed. Their availability in a time interval is guaranteed and limited by envelopes β^U and β^L.

Given the situation that not all processing capabilities available to a resource are consumed, a flow of remaining processing capabilities $C'(t) = C(t) - R'(t)$ leaves the system.

6.5.2 System Analysis

Primary goals in Real-Time Calculus model analysis are to determine the worst-case response time and backlog. For real-time systems these results help to keep deadlines and to size buffers in the embedded hardware. Intermediate steps towards these values are the computation of upper and lower bounds for departure flows and the remaining resource flows.

The outgoing arrival curves and remaining service capacity bounds are computed as

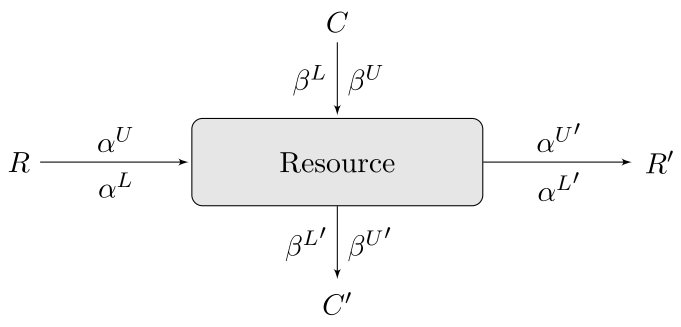


Figure 6.27: A Real-Time Calculus resource with associated flows and their bounds (in style of [104])

follows [104, 122]:

$$\alpha^{L'} = \min\left\{(\alpha^L \underline{\oslash} \beta^U) \underline{\otimes} \beta^L, \beta^L\right\} \tag{6.79}$$

$$\alpha^{U'} = \min\left\{(\alpha^U \underline{\otimes} \beta^U) \underline{\oslash} \beta^L, \beta^U\right\} \tag{6.80}$$

$$\beta^{L'}(t) = \max_{0\le s\le t}\left\{\beta^L(s) - \alpha^U(s)\right\} \tag{6.81}$$

$$\beta^{U'}(t) = \max_{0\le s\le t}\left\{\beta^U(s) - \alpha^L(s)\right\} \tag{6.82}$$

Figure 6.27 includes the bounds for outgoing flows, too. Real-Time Calculus also uses delay and backlog bounds from Definitions 6.2.5 and 6.2.4. For modeling more complex systems, resources are combined to networks. Routing of flows is not limited to event arrival flows only, it also extends to processing and remaining processing capability flows. This allows one to model systems with priorities for different event types and to determine the overall performance bounds for each priority class [104].

When the processing unit works without priorities a Generalized Processor Sharing (GPS) element is given. The processing resources are proportionally distributed on each arrival flow. While [92] gives results for linear service curves only, Thiele et. al. [105] extends GPS in Real-Time Calculus to general service curves. A similar result is given in [101].

Each flow R_i, $i = 1 \dots n$ is associated to a weight ϕ_i with $\sum_{i=1}^n \phi_i = 1$. During operation each flow receives the share

$$\frac{\phi_i}{\sum_{j\in B(t)} \phi_j} \tag{6.83}$$

of the available resource flow. $B(t)$ is the set of backlogged (thus active) flows at time t [107, Section 2.1.2]. A flow can allocate at least ϕ_i processing capacity, if it is the only active flow it can even take advantage of the full system capacity. Then the resource flow available for each arrival flow R_i is bounded by [105] $\beta_i^L = \phi_i \beta^L$ and $\beta_i^U = \beta^U$.

The GPS model in Real-Time Calculus can also be used to describe loop constructs with a fixed iteration count. When each single job in an event flow has to be executed n times in a processing unit, each repetition $i = 1 \dots n$ has an equal share ϕ_i of processing resources available. Since an event processed in a loop is finished, that is when it has completed the last iteration, a processing unit with resource flow bounds β^L and β^U offers bounds

$$\frac{1}{n}\beta^L \text{ and } \beta^U \tag{6.84}$$

if an event is processed n times.

6.6 Software Support

To analyze Network Calculus or Real-Time Calculus models software suites exist. Tailored to models of data networks is the DISCO network calculator [50, 99]. The stand-alone software is implemented in Java, network models are formulated using the GraphML language [50]. Another tool for Network Calculus is NC-Maude [31].

Resulting from the modeling domain of Real-Time Calculus the *RTC Toolbox* [123] was presented in the PhD thesis of Wandeler [122]. It is a Java library for (min,+)-operations using the *MATLAB* computer algebra system as a front end. The software is under constant development since 2006 at the ETH Zürich. In this work the *RTC Toolbox* is the primary tool to analyze the following exemplary Network Calculus models.

In [28] further proposals for implementations of (min,+)-operators, including a discussion of an earlier version of the *RTC Toolbox*, are given.

6.7 SOAs Models with Network Calculus

Network Calculus offers extensive methods to model discrete-event systems that have to conform to QoS. The network-oriented model can also be applied to other domains as Real-Time Calculus does. Of course this also extends to the modeling domain of SOAs with SLAs.

A work on QoS modeling and analysis with Network Calculus is the PhD-thesis of Krishna Pandit [90]. The intention is to give a unified model for QoS performance analysis for packet-switched networks. The modeled networks are reconfigurable systems. Pandit also uses a combined modeling approach with queues and Network Calculus elements in one model. Arrival curves are used to regulate the arrival process towards a queue. For the combination simulative results are given.

For the modeling domain of SOAs with SLAs Network Calculus is rarely used. In Eckert et al. [45] Web Service workflows are analyzed for their worst-case behavior. Their goal is to support capacity planning by avoidance of SLA violations due to performance bottlenecks. Sequential Web Service workflows are used as blueprints for a business process. In accordance with the assumption that a service can be replaced by a semantically equal one, the model is used to support service selection by the workflow controller.

6.7.1 Mapping of BPEL Structures to Network Calculus

The mapping of Web Services with BPEL to Network Calculus is similar to Queueing Networks. Each Web Service participating in a workflow is modeled by a Network Calculus server element instead of a queue. In the approach of [45] only sequences of services are considered. Here we extend the application of Network Calculus to Web Services by adding parallel (synchronized) constructs and loops.

Semantics of the Network Calculus workload model are substantially changed. Instead of single data packets the entities passing through the network are service requests (c.f. [45]). Therefore, the abstraction level in models changes from packet to application layer.

Sequences of service interfaces in BPEL become modeled with serially concatenated servers using Theorem 6.11. For a coherent ordering of server elements the expression sequence of the BPEL file can be used, but is not important due to associativity of (min,+) convolution.

Routing decisions using `<switch/>` tags are abstracted to parallel server constructs according to Theorem 6.12. Since Network Calculus has no descriptive means for variables or decisions all routing rules are lost in abstraction. Instead, the worst-case for routing decisions is implicitly assumed by exposing all possible routes to the full arrival flow. Synchronization of services in BPEL with `<flow/>` tags is expressed with parallel servers using Theorem 6.12, too.

Loops resulting from `<while/>` tags can be emulated by using results for GPS in Real-Time Calculus under the condition that the maximum repetition count is known.

Example 6.7.1. *The network of server elements describing the ParcelSink workflow is shown in Figure 6.31. The BPEL file names five service interfaces, each one is represented by a Network Calculus server element here. Both geocoder services, instantiated by HollowEarth and FlatWorld, are subject to synchronized execution and thus modeled as parallel servers. In the BPEL file a* `<switch/>` *statement routes requests to the pair of geocoders or to the internal catalog depending on the familiarity of the addresses. Since Network Calculus has no notion of variables and thus decisions, this system aspect is also abstracted to parallel routing. It is known for the invoice printing service that it is always called two times, so it annotated in the network with repetition count $n = 2$. The triple of address fetching service, printing service and the combined geocoder/catalog section is in sequence, thus its server elements are concatenated.*

6.7.2 Modeling SLAs with Network Calculus

For including SLAs into Network Calculus models the two-sided contract between customer and provider on maximum workload and guaranteed response time is split. While the workload is part of the model input, analysis will provide the maximum delay. The flow of request arrivals is bounded with upper arrival curves. In [45] the arrival contract is modeled with leaky buckets. When general piecewise linear functions are used, the interpretation guidelines in Section 6.2.5 can be transferred to flows of job requests.

Since Network Calculus server elements are used to describe the workflow, the knowledge of service curves in this type of model is mandatory. In [45] rate-latency curves model the sustainable service rates and the service startup-times. As discussed in Section 3.5 the service capacity itself is not part of SLAs issued by providers, for SOA modeling with Network Calculus it has to be provided in addition. Access to computing systems in order to create performance models and thus service curves is a privilege of service providers. Given that condition, workflow models using Network Calculus are suitable to service providers and apply less to service customers. To provide an example based on the ParcelSink workflow we change our role and perspective to that of a service provider with full access and detailed knowledge on service capacities. The presented service curves are the results of a service curve computation in the upcoming Chapter 9.

Example 6.7.2. *To limit the arrival process an upper arrival curve*

$$\alpha^U_{ParcelSink} = \min(\gamma_{100,5}, \gamma_{12,269}) \tag{6.85}$$

is arranged using the ParcelSink SLO in Table 2.1. Bursts are limited to 3 time units at a rate up to 100 requests per second. It limits the arrival flow to a rate of 12 requests per time unit on the long run ($269 = 3 \cdot 100 + 5 - 12 \cdot 3$*).*

The service process of the first geocoder is bound from below by

$$\beta^L_{hollowearth} = \begin{cases} \max(\beta_{8,10}, \beta_{25,13.4}) & t < 18 \\ \beta_{8,2.375} & t \geq 18 \end{cases} \tag{6.86}$$

and for the second a bound of

$$\beta^L_{flatworld} = \begin{cases} \beta_{20,5} & t < 7 \\ \beta_{15,4.33} & t \geq 7 \end{cases} \tag{6.87}$$

is known.

For the remaining services the lower service curves are given by

$$\beta^L_{catalog} = \begin{cases} 0 & t < 2 \\ \beta_{12,1.1667} & 2 \leq t < 3 \\ \beta_{9,0.556} & t \geq 3 \end{cases} \tag{6.88}$$

$$\beta^L_{fetchaddress} = \beta_{15,9.8667} \tag{6.89}$$

$$\beta^L_{printinvoice} = \beta_{25,15.76} \tag{6.90}$$

Figure 6.28 includes plots for $\alpha^U_{ParcelSink}$ *and the basic service curves.*

6.7.3 SLA Validation

After modeling Web Service workflows with Network Calculus, the end-to-end delay is computed in [45] and considered as the global workflow response time in worst-case. By comparison of delay limits required by customers a workflow configuration is either accepted or discarded. Hence, the first step in Network Calculus SLA validation is the computation of the global service curve β^L for all services participating in a workflow by Theorem 6.11. In a second step the maximum horizontal distance (Equation (6.3.2)) between arrival contract α^U and β^L gives the worst-case system latency. As a second performance figure the throughput of a workflow is read from the resulting sustainable service rate in β^L [45].

For all computations in the following example the *RTC Toolbox* [123] introduced in Section 6.6 is used.

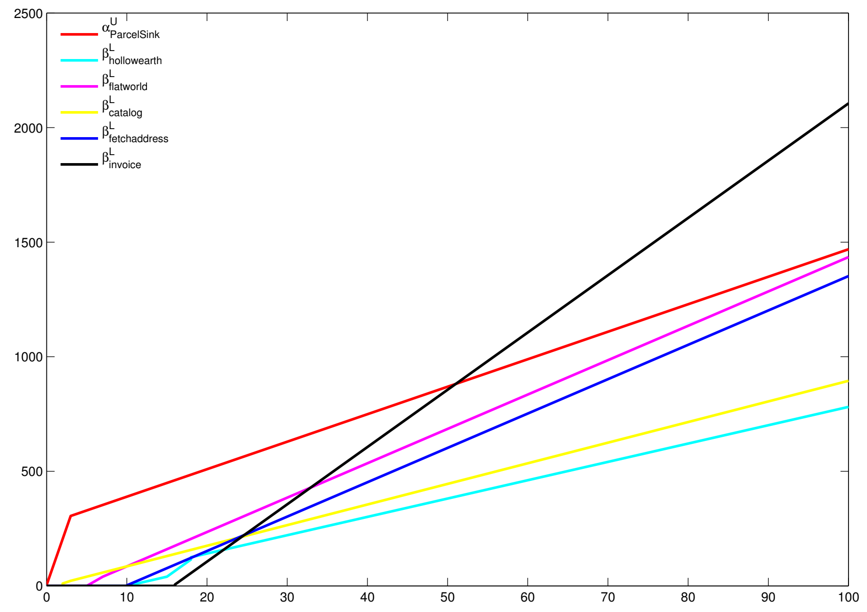


Figure 6.28: Arrival and Service Curves for ParcelSink Services.

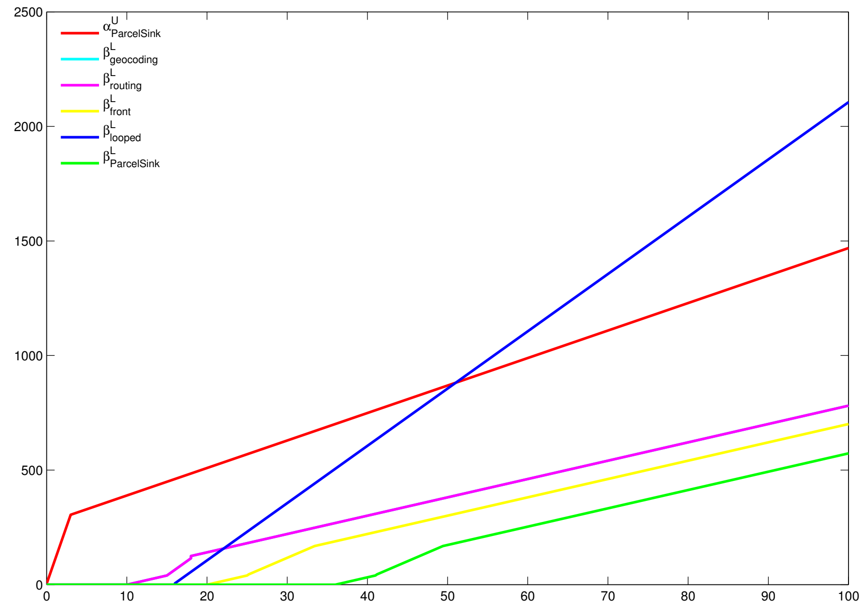


Figure 6.29: Intermediate and global service curves for ParcelSink workflow.

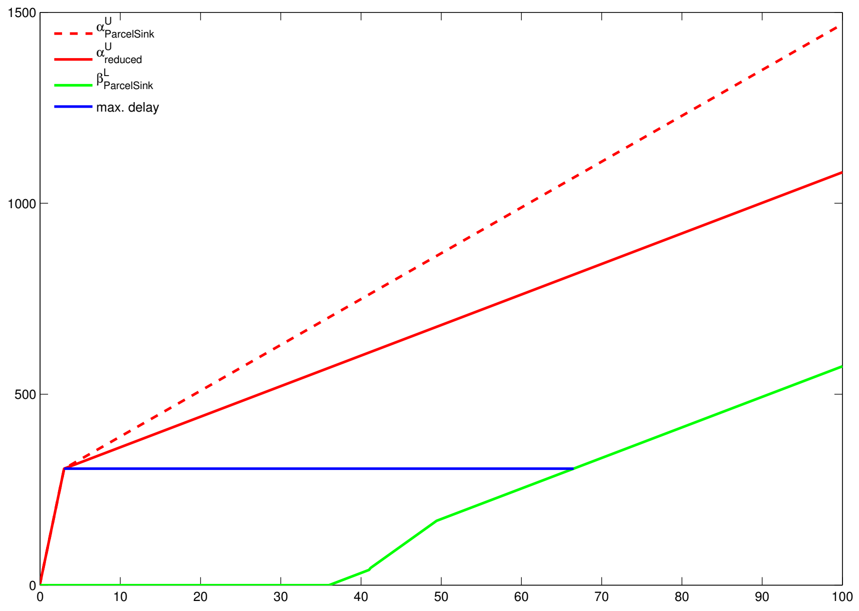


Figure 6.30: Maximum response time of ParcelSink workflow with reduced arrival curve.

Example 6.7.3. *Given the arrival curve contract α^U and the service curves from Example 6.7.2 the response time guarantee of 5 seconds is to be validated. Based on the ParcelSink workflow structure the global service curve $\beta^L_{ParcelSink}$ is computed step by step (Figure 6.29 shows the intermediate results). The workflow uses two geocoders in synchronization (yellow box in Figure 6.31), so Theorem 6.12 gives*

$$\beta^L_{geocoding} = \beta^L_{hollowearth} \wedge \beta^L_{flatworld} = \begin{cases} \max(\beta_{8,10}, \beta_{25,13.4}) & t < 18 \\ \max(\beta_{8,2.375}) & t \geq 18 \end{cases} \tag{6.91}$$

Obviously, the better lower service bound of FlatWorld cannot contribute to the combined worst-case performance when it is synchronized with HollowEarth. The service curve describing the worst-case routing between the geocoders and the internal address database is found using Theorem 6.12 again.

$$\beta^L_{route} = \beta^L_{geocoding} \wedge \beta^L_{catalog} = \beta^L_{geocoding} \tag{6.92}$$

The geocoder section is the bottleneck here. Before the service curve for the complete workflow can be derived the loop construct for `print_invoice` *has to be considered. Using Equation 6.84 the offered service bound is*

$$\begin{aligned} \beta^L_{looped} &= 0.5 \cdot \beta^L_{printinvoice} && (6.93)\\ &= 0.5 \cdot \beta_{25,15.76} && (6.94)\\ &= 0.5 \cdot (25 \cdot \max(0, t - 15.76)) && (6.95)\\ &= \beta_{12.5,15.76} && (6.96) \end{aligned}$$

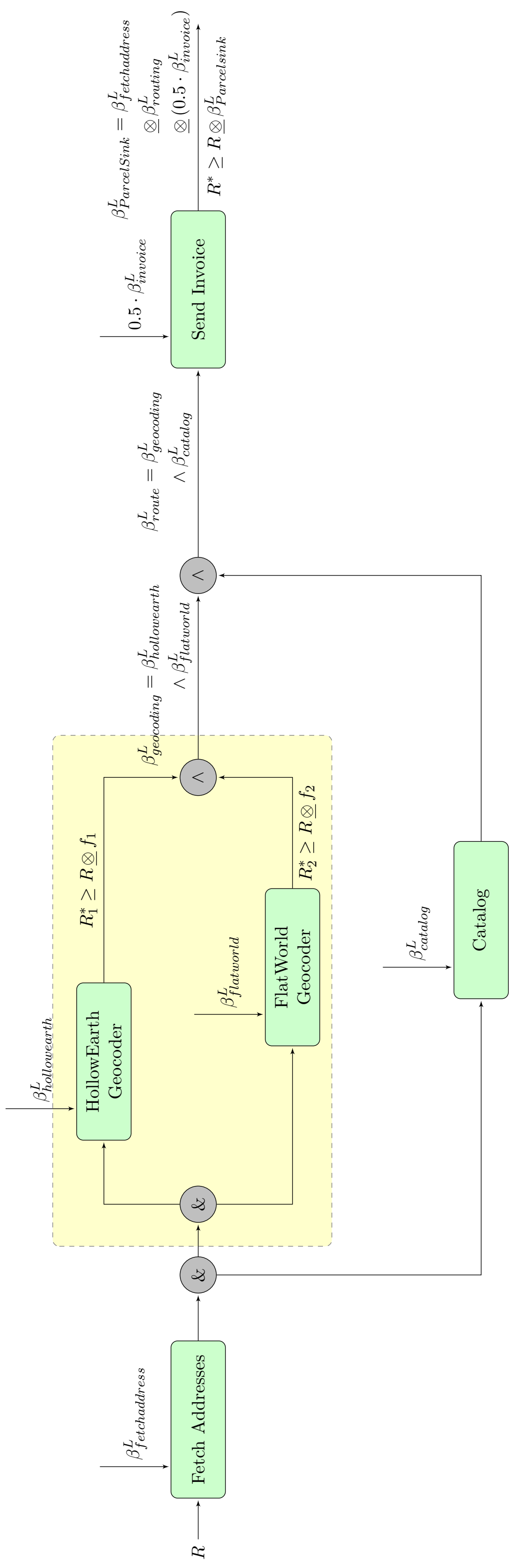

R
$\beta^L_{fetchaddress}$
Fetch Addresses
&
&
$\beta^L_{hollowearth}$
HollowEarth Geocoder
$R_1^* \geq R \underline{\otimes} f_1$
$\beta^L_{flatworld}$
FlatWorld Geocoder
$R_2^* \geq R \otimes f_2$
$\wedge$
$\beta^L_{geocoding} = \beta^L_{hollowearth} \wedge \beta^L_{flatworld}$
$\beta^L_{catalog}$
Catalog
$\wedge$
$\beta^L_{route} = \beta^L_{geocoding} \wedge \beta^L_{catalog}$
$0.5 \cdot \beta^L_{invoice}$
Send Invoice
$\beta^L_{ParcelSink} = \beta^L_{fetchaddress} \underline{\otimes} \beta^L_{routing} \underline{\otimes} (0.5 \cdot \beta^L_{invoice})$
$R^* \geq R \underline{\otimes} \beta^L_{Parcelsink}$

Figure 6.31: ParcelSink example modeled with Network Calculus

The global service curve is given by serial concatenation (Theorem 6.11) of the remaining services and β^L_{route}:

$$\beta^L_{ParcelSink} = \beta^L_{fetchaddress} \underline{\otimes} \beta^L_{route} \underline{\otimes} \beta^L_{looped} \tag{6.97}$$

$$= \begin{cases} \beta_{8,36} & t < 41 \\ \beta_{25,39.4} & 41 \leq t < 41.2 \\ \beta_{15,38.2} & 41.2 \leq t < 49.4286 \\ \beta_{8,28.375} & t \geq 49.4286 \end{cases} \tag{6.98}$$

In view of that lower bound for the service, the processing of request in the ParcelSink workflow might completely stop for 36 time units. Otherwise the minimum long-term service rate is 8. Furthermore, the service curve is not convex due to the rate increase in interval $[41, 49.4286)$*. The curve is still super-additive, which can be easily verified by computing* $\beta^L_{ParcelSink} \overline{\otimes} \beta^L_{ParcelSink} = \beta^L_{ParcelSink}$*.*

Now, by knowing the service characteristic, we can use the given arrival bound to compute the maximum delay by Equation (6.25)*.*

$$h_{\max}\left(\alpha^U, \beta^L_{ParcelSink}\right) = \sup_{t \geq 0} h\left(\alpha^U, \beta^L_{ParcelSink}\right)(t) \tag{6.99}$$

$$= \infty \tag{6.100}$$

According to Network Calculus analysis the workflow is not a stable system under the load specified by α^U. Or, in terms of SLA validation, the system is not conform to the given customer workload contract. The reason for the negative result is the gap between long-term arrival rate (12 per time unit) and long-term service rate (8 per time unit) resulting from HollowEarth. When we analyzed the workflow with Queueing Theory, the bottleneck was concealed in average rates and the stochastic routing was sending 60% of requests to the geocoders only. Here, with Network Calculus, the system overload in worst-case is exposed.

Example 6.7.4. *One can create a stable system by reducing the workload send to the workflow. We match long-term arrival and service rates by redefining the arrival curve to:*

$$\alpha^U_{stable} = \min(\gamma_{100,5}, \gamma_{8,281}) \tag{6.101}$$

With this arrival bound we recompute the delay:

$$h_{\max}\left(\alpha^U_{stable}, \beta^L_{ParcelSink}\right) = \sup_{t \geq 0} h\left(\alpha^U, \beta^L_{ParcelSink}\right)(t) \tag{6.102}$$

$$= 63.5 \tag{6.103}$$

As a result, the provider running the ParcelSink workflow can guarantee a maximum system latency of 63.5 seconds for the reduced system load. The delay bound of 5s required by the SLA is still not fulfilled.

Analysis gave us two results based on the maximum delay bound. The first states that the system is not sufficient enough for the initial workload contract, so the SLA validation fails due to the unbounded delay. The second result is found after adjustments to the arrival contract have been made to get a stable system. A figure for the maximum delay was identified, still not sufficient, but it can serve as a starting point to negotiate a new SLA.

Discussion of Results

Basically, we modeled and analyzed the SOA of ParcelSink in a way common for data networks and received a single, deterministic performance guarantee on delay. Although the information on maximum delay is valuable, several questions on SOA performance are left open. Tolerances for workload have been included in the arrival curve and the service curve separates short- and long-term processing. The monolithic analysis result on maximum delay indicates short-term system performance in a theoretic worst-case situation with a low probability for real world systems. Network Calculus analysis is lacking separation of short- and long-term results, zero information is given for long-term delays more important in SLAs. When SLAs with tolerances are to be validated the approach will not provide sufficient information.

We also have to remind the reader that for the exemplary model analysis the knowledge of service curves, starting in Example 6.7.2, was explicitly assumed. As discussed in Chapter 3 in a SOA scenario SLAs provide delay guarantees, but leave the processing rate of obliged systems open. For the service selection scenario in [45] this gap in knowledge is closed by the introduction of a service broker, aware of the service descriptions of each service. Moreover, service curves can be negotiated between customer and provider. For the model proposed in [45] this is a sound solution, but such negotiation mechanisms are not known to existing Web Service protocols. With unknown service curves analysis is impossible and, from a SOA customer's perspective, Network Calculus is unsuitable for SLA validation.

Still Network Calculus has some advantages for SOA modeling and SLA validation. The Calculus is deterministic, under the assumption of correct model parameters the results will also be deterministic upper bounds. This is different from other modeling methods like Queueing Theory or Simulation that yield average values instead of bounds. As seen in our example, from the deterministic perspective the outcome of a validation may completely differ.

Even within the deterministic bounds there is the option of including tolerances in model parameters. Upper and lower curves give a high degree of freedom to describe workload and performance variations. By adjusting the shape of concave (and convex) curves the flow can be characterized in short-term behavior and long-term target performance. When we limited curves to piece-wise linear functions we gained an intuitive way to model tolerances. The analysis process itself is efficient and results can be computed fast, with the hardware of Example 5.1.3 computations took less than a second.

In summary, Network Calculus is not the answer to the problem, but a good starting point for achieving a solution.

7 An Approach to Delay Modeling

In the previous chapters three modeling methods for modeling and performance analysis of SOAs were presented and applied to SOAs, that have to be compliant to SLAs with quantitative requirements. With Queueing Networks the lack of deterministic bounds is a major problem when service guarantees are to be validated. More detailed models or non-average results break their efficient analysis option. Simulation does give more insight into performance behavior of SOAs, but requires lengthy computations and detailed models. The results still give no performance guarantees, as worst-case situations may not be reached during simulative analysis.

Network Calculus and Real-Time Calculus can provide deterministic worst-case bounds for systems, they have algebraic foundations and offer fast computable results. By knowledge of bounding curves for arrivals and service/resource availability in a system, performance numbers like output bounds, backlog and system delays can be derived. But in case of SOA models their service process descriptions only require information that cannot be extracted from SLOs in SLAs. Additionally, concentration on worst-case scenarios limits the use of Network Calculus for modeling domains with less strict time constraints. This also limits the applicability to models with SLA performance descriptions.

Up to now, one specification in SLOs did not receive enough consideration in SOA modeling: response time limits with their short-term tolerances. In the following chapters SLA Calculus is proposed as a performance modeling method for SOAs with SLAs, that combines the strengths on Queueing Theory and Network Calculus. SLA Calculus is based on the ideas of Network Calculus to describe processes by their deterministic bounds. In exchange for service curves, SLA Calculus uses curves to bound system delay. This allows service customers to build performance models based on the information and service promises given by service providers. Thus, SLA Calculus is suitable for models from a provider and customer perspective.

7.1 Curves for Delay Contracts

Development of SLA Calculus is based on the idea that processing of an arrival emits a second event with the processing time, namely the delay as seen by the processed event. The flow of these delay events corresponds to the distance between arrival and departure flows. The delay flow within a time interval can variate, but the rate is regulated by means of Network Calculus. By imposing bounds on the delay flow the underlying system performance is characterized.

The regulating element in the SLA Calculus system model will be arrival functions setting bounds on the delay flow within time intervals. Depending on the configuration of the so-called delay curve one will be able to include classical hard deadlines, as well as, less restrictive requirements as found in SLAs. By interpretation, delay times can be seen as a liquid (discrete: packets) with a flow regulated by one or several leaky buckets. One can define a sustainable rate of delays, as well as, the acceptance of delay burst by giving the leaky bucket an appropriate capacity. Instead of hard deadlines there is “additional” delay available to compensate short-term fluctuations.

It is the primary goal of this thesis to describe the idea of delay flows and their bounding with methods of Network Calculus. First it is formalized for general systems, then it will be specialized to SOA models in the following sections to utilize Network Calculus for SLA descriptions.

7.1.1 Delay Process Modeling

The delay process for a system modeled with SLA Calculus is generated by a function over arrival and departure flows. Thus, delay functions for SLA Calculus are second level expressions defined over elementary functions in Network Calculus. To the best of our knowledge this concept remains unused in the Network Calculus literature.

The delay function in SLA Calculus uses inverses of both input flows or curves whose difference provides the desired vertical distance as delay value. However, only strictly (non-) increasing functions can be inverted, but arrival flows are monotonically increasing step functions. For functions $f \in \mathcal{F}_0$ we solve this issue by usage of a pseudo-inverse $(f)^{-1}$ introduced in upcoming Definition 7.3.1.

Definition 7.1.1 (SLA Calculus Delay Function). *Let $R(t)$ be the arrival flow to a system and $R^*(t)$ the corresponding departure flow. The delay at time t is the vertical deviation (Definition 6.2.2) between both inverted flows:*

$$d\left(R, R^*\right)(t) = (R^*)^{-1}(R(t)) - (R)^{-1}(R(t)) \tag{7.1}$$

The subtrahend cannot be reduced to t due to the characteristic of the pseudoinverse. Delay is a quantity measured in a unit of time. For simplicity, the unit of the used time basis for t, e.g. milliseconds, seconds or minutes, should be used.

Figure 7.1 shows how a delay function between an arrival and departure flow is constituted. The blue horizontal arrows mark the delay for discrete arrivals that entered the system until they left. With every arrival leaving the system after processing such an arrow can be drawn between input and output flows. It can be considered as a discrete delay event occurring whenever an entity is entering the system for processing.

Definition 7.1.2 (Delay Event). *A delay event triggered by arrival i is given by $l_i = (w_i, t_i)$ with $w_i = h\left(R, R^*\right)(t_i)$ as event weight specified by the delay experienced by i and t_i as arrival time of i.*

To simplify the notation following operators are introduced: $t(e_i) = t_i$ and $w(e_i) = w_i$.

7.1.2 Delay Flow

To construct delay contracts with $d\left(R, R^*\right)(t)$ its value is interpreted as an arrival flow of delay events. This allows one to construct a delay flow in form of a cumulated delay function.

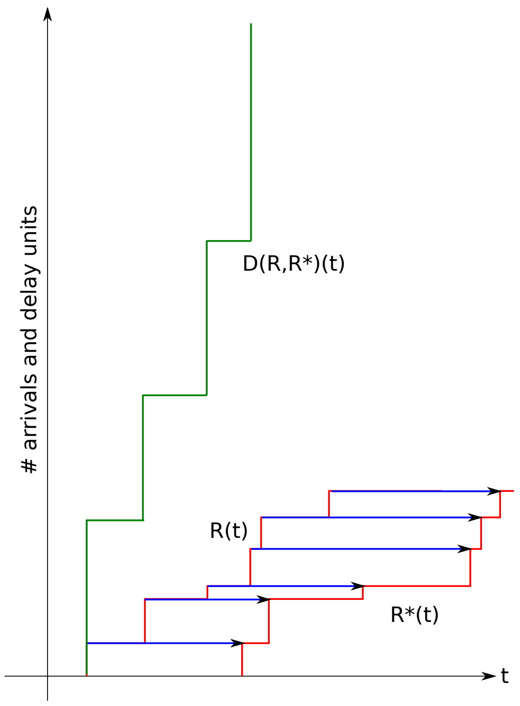


Figure 7.1: Delay Function $d\left(R, R^{*}\right)(t)$ is given by the length of the blue arrows (delay events). Delay flow $D\left(R, R^{*}\right)(t)$ is the continuous sum over $d\left(R, R^{*}\right)(t)$.

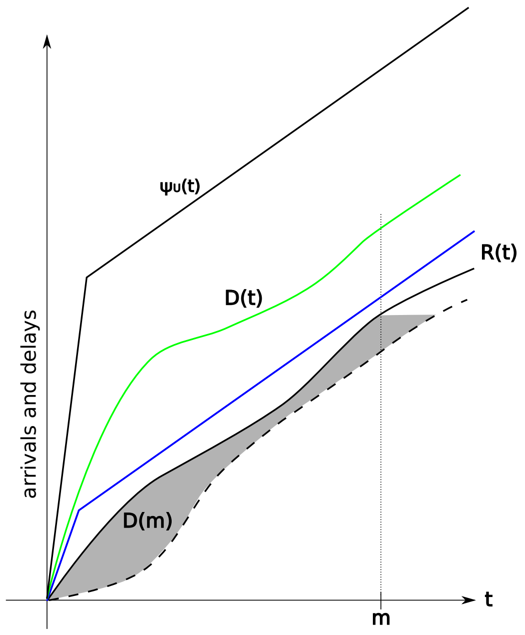


Figure 7.2: Continuous model: arrivals $R(t)$ with upper bound, departure flow $R^{*}(t)$ and resulting delay $D(m)$ bounded by Ψ^{U}.

Definition 7.1.3 (SLA Calculus Delay Flow)**.** *Let $d\left(R, R^{*}\right)(t)$ be a delay function for a system with arrival flow R and departure flow R^{*}. Then Delay Flow*

$$D\left(R, R^{*}\right)(t)=\int_{0}^{t} d\left(R, R^{*}\right)(x) \,\mathrm{d} x \tag{7.2}$$

is the cumulative sum of delays in time interval $[0, t]$.

If the context allows an abbreviation, notation $D(f, g)(t)=D(t)$ is used. $D(t) \in \mathcal{F}$ and $D(t)=0$ for $t \leq 0$ and $D(t)$ is wide-sense increasing. Thus, $D(t)$ features the same properties as arrival flows and can be described with similar algebraic methods.

Figuratively speaking about discrete delay events, each event adds the processing time of its corresponding arrival to the delay flow (Figure 7.1). As a result, $D(t)$ is a step function. Equal to arrival flows the delay model is continuous when $\frac{\mathrm{d}}{\mathrm{d} x} D(t)=d(t)$ exists. The rate of delay emitted by a system in t is then given by the slope of $D(t)$. Delay function $d(t)$ is also continuous without delay events. Figure 7.2 includes continuous arrival, departure and delay flows. It shows also a noteworthiness for the area between flows R and R^{*} in interval $[0, m]$ found by the integrative function $D(m)$: Due to the way it $h\left(R, R^{*}\right)(t)$ defined the integrative sum also includes an (almost triangular) area right of m.

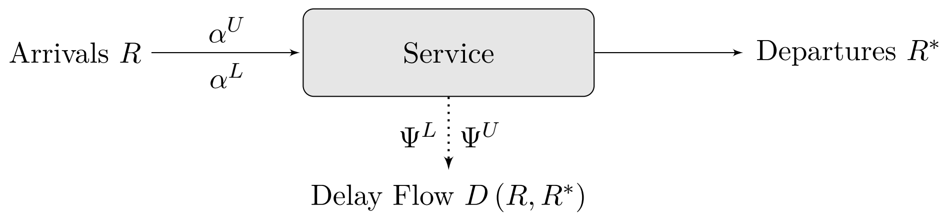


Figure 7.3: Model element with delay: A service processes arrivals (requests). Their waiting times are cumulated to a delay flow $D(R, R^*)$.

With the introduction of delay flows in modeling the basic Network Calculus system model is altered. Figure 7.3 shows a model element focusing on arrival and delay flows, resource flow C is not part of the model anymore.

7.1.3 Delay Curves

The central concept in this work is to describe limits on delays within a time interval by the same tools as deployed on arrival processes. Delay curves bound delay flows emitted by systems. Similar to arrival curves one can construct upper and lower bounds on the cumulated delay in a time interval.

Definition 7.1.4 (Upper Delay Curve)**.** *An upper delay curve Ψ^U for delay function $D(t)$ satisfies the relation*

$$D(t) - D(s) \leq \Psi^U(t - s) \quad \forall\, 0 \leq s \leq t \tag{7.3}$$

Ψ^U is the upper bound for the sum of delay events in time interval $[0, t]$. Some trivia on symbol Ψ: In contrast to lower-case notation used in Network Calculus for curves, Ψ is a Greek upper-case character. By definition of delay flows the expressed delay quantity is based on integration. Integration functions are denoted with uppercase letters and for delay curves this scheme is continued.

Using (min,+)-convolution, Equation 7.3 can be written as

$$D \leq D \underline{\otimes} \Psi^U \tag{7.4}$$

This is shown by Theorem 6.4 when R is replaced by D. Figure 7.2 shows a (continuous) delay flow D given as the area between R and R^*. The delay flow itself is bounded by Ψ^U from above.

Definition 7.1.5 (Lower Delay Curve)**.** *A lower delay curve Ψ^L for delay function $D(t)$ satisfies the relation*

$$D(t) - D(s) \geq \Psi^L(t - s) \quad \forall\, 0 \leq s \leq t \tag{7.5}$$

$\Psi^L(t)$ is the lower bound for the sum of delay events in time interval $[0, t]$. With the help of (max,+)-convolution, Equation 7.5 can be written as

$$D \geq D \overline{\otimes} \Psi^L \tag{7.6}$$

Again the proof is straightforward using Theorem 6.5 and by setting $R = D$.

Upper and lower delay curves provide the tools to model worst-case and best-case system performance with delay contracts. In the following we will give details on the interpretation of upper and lower curves in context of performance models, as well as, semantics of function types used to instantiate delay contracts.

7.1.4 Delay Curve Interpretation

For system performance models in computer science the upper bounds on delay are of major interest, since response times within these bounds are considered as well-performing and thus desirable. Hence, in combination with given arrival process bounds, upper delay curves indirectly set constraints on the minimum processing rates in systems. Further we make the following realistic assumption:

Assumption 7.1 (Relationship of Load and Delay)**.** *For systems modeled with SLA Calculus we postulate that delay grows with the load level. Precisely, delay is a monotonic increasing function Δ of the load. Thus, for two load levels l_1 and l_2 with $l_1 < l_2$ and a function*

$$\Delta : load \rightarrow response\ time \tag{7.7}$$

we have $\Delta(l_1) \leq \Delta(l_2)$.

Arrival and delay curves are algebraically identical. Therefore they can be defined with the same function classes as listed in Section 6.2.4 for arrival curves. Delay flows are non-continuous functions. The underlying delay function itself changes its value in non-continuous steps, whenever an arrival to the system was processed. As for arrival processes the use of piecewise linear functions to construct delay curves enables one to abstract from discrete delay events to a fluid model that will be used in this work. Semantics for curve primitives in order to set bounds on delays in system models are defined as described in the following paragraphs.

Linear Delay Curves for Deadlines

Maximum system delays are modeled with linear delay curves of the type $\lambda_r = r \cdot t$.

Theorem 7.2 (Maximum Delay from Linear Delay Curve)**.** *In a system with arrival flow R constrained by α^U from above and a delay flow bound by λ_r from above the maximum system delay is r.*

Proof. Consider a system with fixed delay T and infinite service rate. The service curve can be expressed as a burst-delay curve $\delta_T(t)$ (Equation 6.34). With delay being a function of load (Proposition 7.1) we consider the maximum, thus worst-case arrival flow $R = \alpha^U$. Retaining the worst-case the lower departure envelope is computed by

$$\alpha^{L'} = (\alpha^U \underline{\otimes} \delta_T)(t) = \alpha^U(t - T) \tag{7.8}$$

The lower output flow bound is the arrival curve shifted by T, the horizontal deviation between arrivals and departures is $d\left(\alpha^U, \alpha^{L'}\right)(t) = T\ \forall t$. By Definition 7.1.3 this results in delay flow

$$D\left(\alpha^U, \alpha^{L'}\right)(t) = \int_0^t (\alpha^{L'})^{-1}(\alpha^U(x)) - (\alpha^U)^{-1}(\alpha^U(x))\,\mathrm{d}x \tag{7.9}$$

$$= \int_0^t (\alpha^U)^{-1}(\alpha^U(x)) + T - (\alpha^U)^{-1}(\alpha^U(x))\,\mathrm{d}x \qquad \textit{Lemma } 7.3 \tag{7.10}$$

$$= \int_0^t T\,\mathrm{d}x = Tt \tag{7.11}$$

The delay flow has thus a slope of $\frac{\mathrm{d}}{\mathrm{d}t}Tt = T$ and can be bounded from above with a delay curve $\lambda_T = Tt$. □

Until now, the addition of delay curves did not add anything to the expressiveness of a Network Calculus model. However, delay curves are not limited to linear functions.

Affine Delay Curves

If affine functions of type $\gamma_{r,b}$ are used to define delay curves, bursts in the delay flow can be bounded. The control of delay bursts extends expressiveness of delay models beyond the definition of hard deadlines. Bursts in the delay flow of a system can occur in several situations:

- The modeled system may stop for a limited time interval. Processing of arrivals is delayed for this interval, the burst is immediately visible in the delay flow.
- From the user's point of view the system slows down processing of arrivals for a time interval. Reasons for this can be heavy workload induced by other system users or a phase of system self-maintenance.
- In non-fluid models b marks the maximum size of a single delay event.

Using affine delay curves of type $\gamma_{r,b}$ allows one to define a maximum execution time of r time units in the same way as for linear curves. Additionally, burst size b can be used to relax strict deadlines for a short period until delay events reach a sum of b. To stay in the picture of leaky buckets, delay flows are controlled with bucket capacity b allowing delay arrivals not conform to rate r.

Composed Delay Curves

Using a single affine function $\gamma_{r,b}$, respectively, a leaky bucket to control the delay process introduces more flexibility in delay modeling than simple deadlines. To model the short-term behavior during bursts of delay events occurring on stalled processing, curves have to include more details. In Network Calculus, short-term bursts in arrival flows are controlled with compositions of affine functions (c.f. Section 6.2.5). The same principle can be exploited to control delay flows in short and long-term. Composing delay curves from two or more affine curves allows us to control the delay process by setting delay rates for different time intervals.

Let γ_{r_1,b_1} and γ_{r_2,b_2} be two upper delay functions. In the long-term we require the delay flow to be limited by rate r_1, for the short-term a higher rate can be accepted, thus $r_2 > r_1$. Equally, delay burst capacity is smaller for the short-term delay model, thus $b_1 > b_2$. Offset b_1 can be considered as long-term tolerance towards system slowdowns. In a discrete model b_2 is the maximum delay event size. As done for upper arrival curves the min() operator is used to form a concave envelope:

$$\Psi^U(t) = (\gamma_{r_1,b_1} \underline{\otimes} \gamma_{r_2,b_2})(t) = \gamma_{r_1,b_1} \wedge \gamma_{r_2,b_2} \tag{7.12}$$

Using more than two affine functions allows us to define more detailed constraints on delay. Furthermore, arbitrary delay curves can be approximated with piecewise linear functions.

7.1.5 Lower Delay Curve Interpretation

Lower delay curves are the lower envelope for the delay flow in a system used to model the minimum response time, thus one can ensure processing takes at least a specific amount of time. Or, for given arrival process bounds, lower delay curves indirectly set the maximum processing rate of a system.

While Queueing Theory combines upper and lower bounds in average performance values, (min,+)-based calculi can specify their range. This adds information on the variance of processing speed. To construct lower delay envelopes we use the convex function set which also provides descriptions of lower arrival and service curves. If no information on response times is available, delay curve $\Psi^L(t) = 0$ is the lowest valid bound that can be used as substitute.

Linear Lower Delay Curve

With a linear delay curve $\Psi^L = \lambda_r$ one can model the minimum response time for a system. The rate of the delay flow bounded from below is at least r, thus each arrival has to endure a delay greater or equal.

Rate-Latency Lower Delay Curves

As for the upper bounds, adding burst capacities to the lower envelope extends modeling flexibility. Rate-Latency curves $\beta_{R,T}$ (c.f. Section 6.2.4) are used to form lower delay envelopes, including some tolerance towards the minimum response times. While in arrival curves the relocated x-axis intersection is the maximum interarrival time of network packets or customers (compare Section 6.2.5) the interpretation for delay curves changes, too. Translation variable T can be interpreted as the maximum interarrival time between two delay events in delay flows. Within a time interval of length T in a continuous model the system is able to work in a high processing rate. Zero or less delay than rate R demands has to be emitted into the delay flow. It is the lag on an ideal delay curve λ_r.

Composed Lower Delay Curves

Lower envelopes for delay flows are formed by compositions of convex curves. To minimize the parameter set and to establish common semantics with lower arrival curves, we will use rate-latency functions of type $\beta_{R,T}$ to compose piecewise linear functions as lower delay curves. A lower delay curve with i segments is given by

$$\Psi^L(t) = \max_i(\beta_{R_i,T_i}(t), 0) \text{ with } R_{i+1} < R_i \text{ and } T_{i+1} > T_i \geq 0 \tag{7.13}$$

In case of two rate-latency functions β_{R_1,T_1} and β_{R_2,T_2} the parameter set can be interpreted as follows: R_1 is the minimum delay rate for the system, R_2 an even lower rate caused by a short-term system speed up. T_1 gives the long-term tolerance towards higher processing rates. In a discrete model T_2 is the maximum interarrival time between delay events. Additional line segments can add more detailed lower limits to the delay process and thus indirectly to the maximum processing rate.

7.2 Delay Contract Conformity

The reader might ask for the practical use of delay curves and how to utilize a system to process arrivals in such a speed that is according to the delay model. In this thesis, no method to enforce a system's behavior with delay curves will be presented. Instead, delay curves are considered as a model for a delay contract that is either met or not.

A system operates conform to a delay curve if the following definition applies:

Definition 7.2.1 (Conformance to Upper Delay Curve Contracts)**.** *Let Ψ^U be an upper delay curve for a system and D the delay flow emitted by the system. The system is conform to the delay contract modeled with Ψ^U if*

$$D(t) \leq (\Psi^U \underline{\otimes} D)(t) \tag{7.14}$$

holds for all t.

Equally, curve contracts for lower bounds are met when following definition applies:

Definition 7.2.2 (Conformance to lower delay curve contract)**.** *Let Ψ^L be a lower delay curve for a system and D the delay flow emitted by the system. The system is conform to the delay contract modeled with Ψ^L if*

$$D(t) \geq (D \overline{\otimes} \Psi^L)(t) \tag{7.15}$$

holds for all t.

The delay contract is guaranteed to the arrivals (job requests) entering the system.

7.3 Horizontal Integration Revisited

Definition 7.1.3 for delay flows includes the integral over the horizontal distance between arrival and departure flow. While the concept of horizontal integration is complementary to the commonly used vertical integration, it cannot be used straightforward in many applications. To compute a delay flow $D\left(f, g\right)(t)$ for two functions $f, g \in \mathcal{F}$ the horizontal deviation between f and g and its integral have to be included as well. This step turns out to be problematic in applications due to the handling of integration variables. For instance, to evaluate expression

$$h\left(f,g\right)(x)\,\mathrm{d}x = \inf\left\{\tau \geq 0 : f(x) \leq g(x+\tau)\right\}\,\mathrm{d}x \tag{7.16}$$

two values have to be computed: $f(x)$ and $g(y)$ with $y = \tau + x > x$. This renders the use of horizontal integration in computer algebra systems like *MATLAB* cumbersome. However, the integral over the horizontal deviation can be replaced with a more common vertical one [117]. In the following paragraphs the transformation using inverse functions is described.

7.3.1 Pseudoinverse

The basic idea to compute delay flow $D\left(f,g\right)(t)$ is to use a special inversion for monotonically increasing f and g in underlying function $d\left(f,g\right)(t)$. In [107, Proposition 3.1.1] it is demonstrated how the maximum horizontal distance between two wide-sense increasing functions is found using the so-called pseudoinverse for one function. For every strictly increasing function a left-inversion [107, Sec. 3.1.4] exists:

$$\forall\, t_1 < t_2, f(t_1) < f(t_2)\ \exists\, f^{-1} \in \mathcal{F} : f^{-1}(f(t)) = t\ \forall t \tag{7.17}$$

However, when $f \in \mathcal{F}$ is wide-sense increasing for some inputs a, b and $a < b$ plateaus exist with $f(a) = y = f(b)$. In this case f is not left-invertible due to $f^{-1}(y) = a$ and $f^{-1}(y) = b$, $a \neq b$. As a consequence, one cannot find a left-inverse for the catalog of functions in Section 6.2.4 used to instantiate curve contracts. Instead pseudoinverse [107, Def. 3.1.7] functions are used.

Definition 7.3.1 (Pseudoinverse)**.** *The pseudoinverse $(f)^{-1}$ of function $f \in \mathcal{F}$ is given by:*

$$(f)^{-1}(x) = \inf\left\{t : f(t) \geq x\right\} = \sup\left\{t : f(t) < x\right\} \tag{7.18}$$

Also, $(f)^{-1} \in \mathcal{F}$ if $f \in \mathcal{F}$ and $(f)^{-1}(0) = 0$ [107, Theorem 3.1.2]

Example 7.3.1. *The graphs in Figure 7.4 show a simple example. Function $f(x) \in \mathcal{F}$ in (a) is composed from linear segments with $f(x) = 1$ for $x \in [1,3]$. Its pseudoinverse function $(f)^{-1}$ in (b) is not continuous at $x = 1$. Due to the infimum operator in Definition* (7.3.1) *we have $(f)^{-1}(1) = 1$ instead of $(f)^{-1}(1) = 3$.*

For curve estimation the pseudoinverse of rate-latency function $\beta_{R,T}$ is required.

$$\beta_{R,T}^{-1}(t) = \begin{cases} 0 & \text{for } t = 0 \\ \frac{t}{R} + T & \text{for } t > 0 \end{cases} \tag{7.19}$$

The following lemma shows an effect when one function is shifted along the time axis.

Lemma 7.3 (Pseudoinversion and Horizontal Shift)**.** *Given are two functions $f, g \in \mathcal{F}_0$ with $g(t) = f(t-c)$. For their pseudo-inverses the following can be stated:*

$$(g)^{-1}(f(t)) = (f)^{-1}(f(t)) + c \tag{7.20}$$

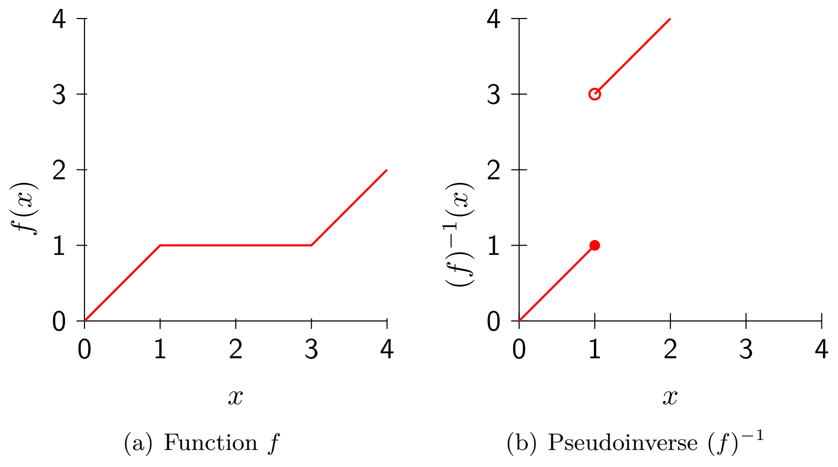


Figure 7.4: Function f and its pseudoinverse $(f)^{-1}$

Proof.

$$
\begin{aligned}
(g)^{-1}(f(t)) &= \inf\{x : g(x) \geq f(t)\} && \textit{Definition } 7.3.1 && (7.21)\\
&= \inf\{x : f(x-c) \geq f(t)\} && && (7.22)\\
&= \inf\{\Delta + c : f(\Delta) \geq f(t)\} && \Delta = x - c && (7.23)\\
&= \inf\{\Delta : f(\Delta) \geq f(t)\} + c && && (7.24)\\
&= (f)^{-1}(f(t)) + c && \textit{Definition } 7.3.1 && (7.25)
\end{aligned}
$$

□

7.3.2 Horizontal Distance by Inversion

By application of the pseudoinverse the horizontal deviation function can be replaced.

Theorem 7.4 (Transformation of Horizontal Deviation)**.** *Let $f, g \in \mathcal{F}$.*

$$h\,(f,g)\,(t) = (g)^{-1}(f(t)) - t \tag{7.26}$$

Proof. We start from Definition 6.2.3 of horizontal deviation and apply Definition 7.3.1 of the pseudoinverse.

$$
\begin{aligned}
h\,(f,g)\,(t) &= \inf\{d \geq 0 : f(t) \leq g(t+d)\} && && (7.27)\\
&= \inf\{d \geq 0 : g(t+d) \geq f(t)\} - t + t && && (7.28)\\
&= \inf\{d + t \geq 0 : g(t+d) \geq f(t)\} - t && && (7.29)\\
&= \inf\{\Delta \geq 0 : g(\Delta) \geq f(t)\} - t && d + t = \Delta && (7.30)\\
&= (g)^{-1}(f(t)) - t && \textit{Definition } 7.3.1 && (7.31)
\end{aligned}
$$

□

As a result, delay flows can be computed by

$$D\left(f,g\right)(t) = \int_0^t (g)^{-1}(f(x)) - x \,\mathrm{d}x \tag{7.32}$$

In [107, Section 3.1.4] a small but useful list for the functions used in curve definitions is given:

$$(\lambda_R)^{-1} = \lambda_{\frac{1}{R}} \tag{7.33}$$

$$(\delta_T)^{-1} = \delta_0 \wedge T \tag{7.34}$$

$$(\beta_{R,T})^{-1} = \gamma_{\frac{1}{R},T} \tag{7.35}$$

$$(\gamma_{r,b})^{-1} = \beta_{\frac{1}{r},b} \tag{7.36}$$

7.4 Backlog Curves

While we used the horizontal deviation between incoming and outgoing flows as system delay and imposed limits with delay curves, similar constructs for the backlog given by vertical deviation are also possible. Indeed, backlog flows and even curves can be formulated in a similar manner as delay curves. However, we will see that backlog curves are quite redundant for modeling needs with SLAs.

Definition 7.4.1 (SLA Calculus Backlog Function)**.** *The backlog $b(t)$ for job request flow R and departure flow R^* at time t is the vertical deviation (Def. 6.2.2) between both functions*

$$v\left(R,R^*\right)(t) = R^*(t) - R(t) \tag{7.37}$$

As known for arrival and delay, a backlog flow in form of an arrival flow can be formulated:

Definition 7.4.2 (Backlog Flow)**.** *Let $v\left(R,R^*\right)(t)$ be the backlog for an arrival curve and a departure flow. Backlog Flow $B(t)$ is the cumulative sum of backlogged arrivals in interval $[0,t]$.*

$$B(R,R^*)(t) = \int_0^t v\left(R,R^*\right)(x)\,\mathrm{d}x \tag{7.38}$$

For abbreviation, notation $B(t) = B(R,R^*)(t)$ is used.

Finally, also upper and lower bounds in form of backlog curves are possible:

Definition 7.4.3 (Backlog Curves)**.** *Upper backlog curve Υ^U and lower backlog curve Υ^L for backlog flow $B(t)$ satisfy the relation*

$$\Upsilon^L \,\overline{\otimes}\, B \leq B \leq B \,\underline{\otimes}\, \Upsilon^U \tag{7.39}$$

Backlog curves will not be used for SLA modeling in this thesis, here, delay curves are preferred. SLA Calculus is intended for black box modeling of SOAs considering the customer perspective. When sending job requests to a service, the response time is of more interest to the customer than the behavior of internal buffer processes. Without proof we assume that SLAs do not put any limitations on the internal buffering of a service.

Nevertheless, an application in other modeling domains than SOA can be imagined. Backlog is a synonym for storage space. In computer applications buffer occupation, as long as it stays below its bound, is negligible in cost and system performance impact. For models of transport and logistic systems one might opt to model the storage occupation with backlog curves. It is of interest because storage space for material goods has to be rented and paid.

8 SLA Calculus Modeling

With delay curves a new quantity can be used in (min,+)-based calculi. The delay "generated" by a system can be controlled in short- and long-term. The available capacity for acceptance of delay can also be used to impose requirements on the unknown service process in an indirect way. By the third curve class Network Calculus is enabled to model quantitative requirements in SLAs.

This chapter introduces an abstract model for performance reasoning and validation for SOAs. SLA Calculus reflects the case when performance guarantees and SLAs are used to construct a model and shall be given by the model. For this reason SLA Calculus focuses on deterministic bounds - input and output values are curve contracts imposing upper and lower bounds on performance figures. Network elements are redefined to service representatives and their arrivals to requests to a system or service. To model contracts on workloads, arrival curves limit request arrival processes generated by service customers. The previously introduced delay curves are used to substitute service rates not defined in SLAs. The calculus presented in this thesis works on a very high abstraction level. We will use a discrete model of requests with continuous time domain and abstract jobs to continuous request flows. Again, as in other presented SOA models, small network latencies are ignored to simplify the model. This allows us to use an unified service model for different technologies like Web Services, Cloud Computing or other implementations following the SOA paradigm. Our model intents to satisfy the requirements for a SOA model presented in Section 3.6.

8.1 Basic Service Model

The basic model element for SLA Calculus completes the delay flow node in Figure 7.3. A service node accepts and processes workload in form of job requests sent by a WEE. After processing a job leaves the service and returns to the WEE to be forwarded into subsequent services. The input/output model is similar to other discrete-event models. Additionally, for each processed job a delay event is generated by the service node. Figure 8.1 shows the system model for a service with input, output and delay process. The input flow and the delay flow are guarded by shaper elements to implement curve contracts, the server element symbolizes the request processing.

8.1.1 Request Model

The request arrival process to a service is the workload in form of a job request sequence. For the modeling domain of SOAs, it is assumed that these job requests are discrete computation tasks that return a result or confirmation.

Definition 8.1.1 (Request Arrival Event). *Arrival event $e_i = (w_i, t_i)$ for the ith request is a tuple with arrival time t_i and event weight w_i. Default is $w(e_i) = 1$.*

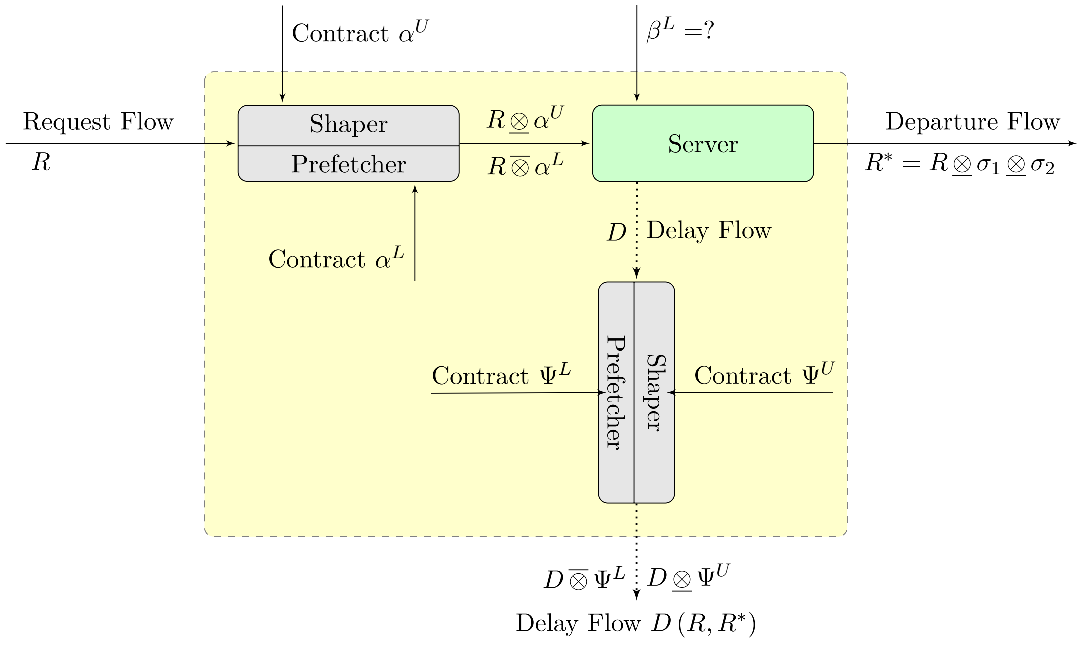

Contract α^U
$\beta^L = ?$
Request Flow
R
Shaper
Prefetcher
$R \underline{\otimes} \alpha^U$
$R \overline{\otimes} \alpha^L$
Server
Departure Flow
$R^* = R \underline{\otimes} \sigma_1 \underline{\otimes} \sigma_2$
Contract α^L
D Delay Flow
Contract Ψ^L
Prefetcher
Shaper
Contract Ψ^U
$D \overline{\otimes} \Psi^L$
$D \underline{\otimes} \Psi^U$
Delay Flow $D(R, R^*)$

Figure 8.1: Basic service model element: A shaper marks non-conform traffic to a server implementing the service.

Depending on the system to model, weight can express the request size in bytes or the processing complexity (c.f. [17]), for example in CPU cycles. The idea is that workload for a service depends on the task to execute and not on the input size. To model the request arrival process, arrival function $r(t)$ is used.

Definition 8.1.2 (SLA Calculus Request Arrival Function)**.** *Let e_i, $i \in \mathbb{N}$ be a sequence of request arrivals events to a service. Then arrival function $r(t)$ to the service is given by*

$$r(t) = \sum_i w(e_i) \text{ such that } t(e_i) = t \tag{8.1}$$

This equals the arrival modeling in Network Calculus with the exception that simultaneous arrivals are possible. Data packets arrive serially as they are transmitted, in SOA jobs can arrive in batches and thus, from an abstract perspective, in parallel. Based on the modified arrival function arrival flow R can be set up.

Definition 8.1.3 (SLA Calculus Arrival Flow)**.** *Let $r(t)$ be an arrival function to a service. Then arrival flow $R(t)$ is given by*

$$R(t) = \int_0^t r(x)\,\mathrm{d}x \tag{8.2}$$

Function $R(t)$ is the cumulated number of service request arrivals in the time interval $[0, t]$.

8.1.2 Processing and Response Model

Request flows are processed by Network Calculus servers. The resource flow C consumed by the server component for processing exists, but is left unspecified, as well as, its limits β^L and β^U. We require FCFS scheduling for the server. A request is processed when it is visible in the server's departure flow.

A SOA job request is considered as done if the processing service sends a message or signal that marks the finishing of processing (c.f. Section 2.2.1). Depending on the system architecture this message can only be a status report or may contain result data. In a Web Services architecture such messages are sent back to the BPEL workflow execution engine, in the SLA Calculus model this coincidences with finished processing in the internal server component.

To abstract response messages introduce departure events.

Definition 8.1.4 (Request Departure Event)**.** *Departure event $m_i = (w_i, t_i)$ for the ith processed request is a tuple with $w(m_i) = w(e_i)$ for the corresponding arrival event e_i and $t(m_i)$ as departure time.*

Definition 8.1.5 (SLA Calculus Departure Function)**.** *Let m_i, $i \in \mathbb{N}$ be messages send by a SOA service when original job requests e_i are processed. Then departure function $r^*(t)$ is given by*

$$r^*(t) = \sum_i w(m_i) \text{ such that } t(m_i) = t \tag{8.3}$$

Similar to arrival functions departure functions form a departure flow R^*:

Definition 8.1.6 (SLA Calculus Request Departure Flow)**.** *Let $r^*(t)$ be the departure function of a service. Then departure flow $R^*(t)$ is given by*

$$R^*(t) = \int_0^t r^*(x)\,\mathrm{d}x \tag{8.4}$$

Figure 8.1 shows R^* as a departing arrival flow leaving the service to the right.

8.1.3 Response Time Model

Based on the request and response model the SLA Calculus response time model for a SOA service is set up. The response time is measured beginning at the point of time the service receives a request. This event is captured by the request arrival model and thus R. Processing of a job is considered as finished when the service sends the corresponding SOA message in the request departure model, namely R^*.

The response time model equals the delay flow $D(R, R^*)$ from Definition 7.1.3 using request arrival flow R and request departure flow R^*. Furthermore, delay is a monotonic increasing function of load, thus Proposition 7.1 still applies.

In Figure 8.1 the delay flow is leaving the basic system model to the bottom.

8.2 Composition Model

Although the composition model is abstract an instance for workflow control is required. For Web Services or similar SOA variants the existence of WEEs is assumed. As discussed in Chapter 2 the overhead of the WEE and network transmission times are negligible and thus are not considered. The modeler is free to add delay nodes in front of services to include those factors. Still there is an exception for the impact of the WEE on system delay. When a service has finished processing and the subsequent part of the workflow is to be called requests are buffered beforehand. As we will see this is necessary to comply with arrival contracts of subsequent services.

For SLA Calculus models we define a set of service compositions in style of the basic workflow compositions in Section 2.2.1. The combinations are geared to BPEL tags to allow an easy mapping of SOAs to models.

Definition 8.2.1 (Serial Composition)**.** *Let there be a set of $n \geq 2$ services and a common request flow R towards these services. Services are in serial composition if the departure flow R_i^* of service i is the arrival flow R_{i+1} to service $i+1$ for all $i \in 1 \ldots n-1$.*

A tandem service is the serial composition of two services.

If a service call employs a service several times we say that it is in a loop.

Definition 8.2.2 (Loop Composition)**.** *A service is in a loop with n iterations when the departure flow of the ith iteration is the input of iteration $i+1$ for all $i \in 1 \ldots n-1$.*

For parallelism we distinguish between routing for parallel workflow segments and synchronized request processing.

Definition 8.2.3 (Routing Composition)**.** *Let there be a set of n services and a common request flow R towards these services. They are in a routing composition, if arrival flow R is split into n subflows R_i, $i \in 1 \ldots n$ with $\sum_i R_i = R$ and each R_i is processed by the respective service i. The demultiplexing scheme is arbitrary. At the composition output the flows R_i^* departing from every service are joined to a common departure flow R^* (Multiplexing). A request $e_j \in R_i$ is finished when $m_j \in R_i^*$.*

In diagrams notation $||$ is used to symbolize demultiplexing, $\times$ joins the flows again.

Definition 8.2.4 (Fork/Join Composition)**.** *Let there be a set of n services and a common request flow R towards these services. They are in a fork/join composition if*

- *arrival flow R is replicated n times to flows R_i, $i \in 1 \ldots n$, hence arrival event $e_j \in R$ creates events e_{ij}.*
- *each R_i is processed by the respective service i, it departs as flow R_i^*.*
- *request $e_{ij} \in R_i$ is finished when departure event $m_{ij} \in R_i^*$ for all i.*

At the composition output the flows R_i^* are joined to an common departure flow R^*, hence events $m_{ij} \in R_i^*$ become event m_j.

In diagrams notation & is used to symbolize the fork, $\times$ denotes the join.

To support the upcoming composition theorems a small but nonrestrictive precondition on workflow models is set:

Definition 8.2.5 (Workflow)**.** *A workflow is a serial composition of one or more services. Each of these services is either a basic service or a composition based on Definitions 8.2.1, 8.2.2, 8.2.3 or 8.2.4.*

8.3 SLA Model

SLAs and guarantees found in their SLO sections are modeled in SLA Calculus with curve contracts. The time-invariant bounds are used to express commitments on request arrival processes and guarantees on response times. They allow one to generalize from instances of the request and response time models to SLAs.

SLAs give indirect information on the performance of the described service. For example, the SLO sections in a WSLA file as described in Section 2.3.3 can define a maximum workload and, by implication, associate a response time guarantee. For delay curves a similar implication has to be made, delay curve contracts are only valid for the arrival process to a system which is bounded (c.f Section 7.2). This dependency found in SLAs and general delay contracts are formalized in SLA Delay Properties (SDPs).

To enable SDPs for modeling there is still one model element missing in our toolkit.

8.3.1 Prefetcher

A prefetcher is the complementary model element to a shaper enforcing lower arrival envelopes α^L to arrival flows. It has the ability to output arrivals into the departure flow before their arrival by creating debt towards the arrival flow.

Definition 8.3.1 (Prefetcher Element)**.** *A prefetcher network element enforces a lower output bound α^L on departure flow R^*. Whenever the output drops below α^L events in R are send prior to their arrival to the prefetcher, thus a debt towards the arrival flow is build up. The prefetcher stops prefetching and decreases debt towards the arrival flow as soon as possible.*

In terms of Network Calculus this translates into the following input-output relationship:

Theorem 8.1 (Input-Output Characterization of Prefetchers)**.** *Consider a prefetcher implementing a lower envelop α^L for arrival flow R. At $t = 0$ there is no debt towards R and $R(t) > 0$ for $t \geq 0$ holds such that prefetching is possible. The output flow is given by*

$$R^*(t) = (R \overline{\otimes} \alpha^L)(t) \tag{8.5}$$

The proof is a (max,+)-version of the proof for [107, Theorem 1.5.1] adapted to the discrete time version in [38, Section 6.2.1]. It is shown in the Appendix.

Prefetchers are a continuous time variant of another network element. Chang describes in [38, Section 6.2.1] the minimum g-regulator. A packet stream modeled by a marked point process $\Psi = (\tau, l)$ passing this element is conform to lower service guarantee $g(t)$. For sequence $n = 0, 1, 2, \ldots$ the process elements are given by functions $\tau = \{\tau(n)\}$ and $l = \{l(n)\}$. $\tau(n)$ marks the arrival time of the $n+1$th packet and $l(n)$ is the packet length, $L(n)$ is the cumulated packet length and thus the service requirement of the network element. Traffic is said to be g-regular when $\tau(n) - \tau(m) \geq g(L(n) - L(m))$ holds for all $m \leq n$.

The difference between g-regulators and prefetchers is about how packet lengths are taken into account. With a g-regulator a flow is conform when each packet, and thus, its payload has arrived completely in such a way that the lower bound holds. Packet transmission in a network takes little but considerable time, with the arrival of the first bits of the packet header the required payload is still not delivered. In contrast, arrivals to the prefetcher are singular events with no transmission time. This simplification is acceptable in the modeling domain of SOAs, transmission times of service messages have no high impact on the overall system performance in comparison to job processing times.

Definition 8.3.2 (Prefetcher Earliness)**.** *Let R be the input flow or curve into a prefetcher implementing σ. Then the earliness of R in t is*

$$h_{early}\left(R, \sigma\right)(t) = \inf\left\{d \geq 0 \text{ such that } R(t) \geq \sigma(t-d)\right\} \tag{8.6}$$

Definition 8.3.3 (Prefetcher Debt)**.** *The debt $b_{early}\left(R, \sigma\right)(t)$ required by a prefetcher implementing σ towards the arrival flow R is given by*

$$b_{early}\left(R, \sigma\right)(t) = \min\left\{0, -((R \overline{\otimes} \sigma)(t) - R(t))\right\} \tag{8.7}$$

Lemma 8.2 (Lower Arrival Envelops with Prefetchers)**.** *A flow R with lower envelope σ passing through a prefetcher implementing σ does not require to build up debt.*

Proof.

$$\begin{aligned}
b_{early}\left(R, \sigma\right)(t) &= -\max\left\{0, R^*(t) - R(t)\right\} && \text{Theorem 8.1} && (8.8)\\
&\leq -\max\left\{0, R^*(t) - (R \underline{\otimes} \sigma)(t)\right\} && && (8.9)\\
&= -\max\left\{0, (R \underline{\otimes} \sigma)(t) - (R \underline{\otimes} \sigma)(t)\right\} && && (8.10)\\
&= 0 && && (8.11)
\end{aligned}$$

□

In a similar way to shapers in Definition 6.2.9 the conformance to a prefetcher can be defined.

Definition 8.3.4 (Prefetcher Conformance)**.** *A flow is σ-conform to a prefetcher implementing a lower envelope σ when no debt is build up while the flow passes the prefetcher.*

With the shaper-equivalent for lower bounds SDPs can be formulated.

8.3.2 SLA Delay Properties

SDPs symbolize the SLA performance contract for a service and implicitly the service itself. Since this contract applies to the service provider, as well as the customer, SDPs include boundaries for both sides.

Definition 8.3.5 (SLA Delay Property)**.** *An SLA Delay Property (SDP) is a set of arrival (α) and delay (Ψ) curve contracts*

$$\{\alpha^L, \alpha^U, \Psi^L, \Psi^U\} \tag{8.12}$$

with α^L,α^U, Ψ^U and $\Psi^L \in \mathcal{F}_0$. α^U and Ψ^U are sub-additive, α^L and Ψ^L are super-additive. Conditions $\alpha^L \leq \alpha^U$, $\Psi^L \leq \Psi^U$ apply and $\alpha^U(t), \Psi^U(t) > 0$ for all $t > 0$.

Arrival curves α^U and α^L address the service customers, they describe the allowed service workload. Delay curves Ψ^U and Ψ^L are the obligations for the providers.

SLA Delay Properties and the Basic Service Model

A basic service model is instantiated for each service with the curve contracts given by the characterizing SDP. The four curves are assigned to shaper and prefetcher subcomponents shown in Figure 8.1.

In this context, the input shaper receives α^U as a shaping curve. Whenever arrivals in R have to be backlogged non-conform load to the system is recognized. R also checked for lower envelope α^L by the lower bound mode of the shaper, the prefetcher component. After passing the shaper and prefetcher the request flow is processed in the server element. It will emit a delay flow D bound by Ψ^L and Ψ^U implemented by the delay shaper and prefetcher. For the time being, service curve β^L remains unknown.

The existence of bounds allows one to define conformance of the request arrival process:

Definition 8.3.6 (Service Customer Conformance)**.** *Let $SLA = \left\{\alpha^U, \alpha^L, \Psi^U, \Psi^L\right\}$ be an SDP for a service and R the request flow sent by the customer towards the service. A basic service model as shown in Figure 8.1 is parametrized with SLA. The customer acts conform to*

- *the upper arrival contract in SLA, if R is conform to the input shaper implementing α^U by Definition 6.2.9*
- *the lower arrival contract in SLA, if R is conform to the input prefetcher implementing α^L by Definition 8.3.4*
- *the arrival contract in SLA, if both previous conditions hold.*

A lower bound α^L for R is not always known or required by an application or service. In those cases the envelope can be set to $\alpha^L = 0$ as default.

The input shaper in the basic model does not add additional delay as long as the arrival contract holds and can be neglected:

Theorem 8.3 (No Reshaping Delay for Conformant Flows)**.** *An arrival flow R with upper envelope α^U is conform to an SDP $\{\alpha_i^U, \Psi_i^U, \alpha_i^L, \Psi_i^L\}$ if*

$$h_{\max}\left(\alpha^U, \alpha_i^U\right) = 0 \tag{8.13}$$

holds.

Proof. Let R be the flow towards and R^* be the flow departing from the shaper implementing α_i^U.

$$\begin{aligned}
h_{\max}(R, R^*) &= h_{\max}\left(R \underline{\otimes} \alpha^U, R \underline{\otimes} \alpha_i^U\right) && (8.14)\\
&= \max_t \left\{\inf\left\{d \geq 0 : (R \underline{\otimes} \alpha^U)(t) \leq (R \underline{\otimes} \alpha_i^U)(t+d)\right\}\right\} && (8.15)\\
&= \max_t \left\{\inf\left\{d \geq 0 : (R \underline{\otimes} \alpha^U)(t) \leq (R \underline{\otimes} \alpha_i^U)(t) + (R \underline{\otimes} \alpha_i^U)(d)\right\}\right\} && (8.16)\\
&= \max_t \left\{\inf\left\{d \geq 0 : (R \underline{\otimes} \alpha^U)(t) - (R \underline{\otimes} \alpha_i^U)(t) \leq (R \underline{\otimes} \alpha_i^U)(d)\right\}\right\} && (8.17)\\
&= \max_t \left\{\inf\left\{d \geq 0 : 0 \leq (R \underline{\otimes} \alpha_i^U)(d)\right\}\right\} && (8.18)\\
&= 0 && (8.19)
\end{aligned}$$

since $R \underline{\otimes} \alpha_i^U \in \mathcal{F}_0$ and wide-sense increasing (Line 8.16 follows from the sub-additivity of $R \underline{\otimes} \alpha_i^U$ and Lemma 6.6 leads to Line 8.18). □

A server in Network Calculus is stable when a maximum delay (or backlog) bound exists between the arrival and service curve. This definition of system stability in Network Calculus is replaced by arrival conformance in SLA Calculus. Service capabilities on workloads are determined by upper arrival contracts and not by service curves. Workload exceeding the arrival curves leads to contract breaches, not necessarily to buffer overflows as in Network Calculus. The arrival contract definition gives us an easy method to check conformance:

Corollary 8.4. *A flow bounded by α^U from above is conform to arrival contract α_i^U when*

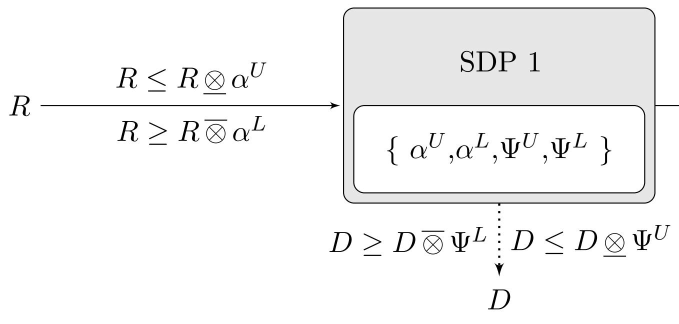
$$b_{\max}\left(\alpha^U, \alpha_i^U\right) = 0 \text{ or } h_{\max}\left(\alpha^U, \alpha_i^U\right) = 0 \tag{8.20}$$

Proof. Application of Lemma 6.6 (backlog) and Theorem 8.3 (delay). □

Delay curves in an SDP address the service providers with response time guarantees. Their conformance is defined by a combination of Definitions 7.2.1 and 7.2.2

Definition 8.3.7 (Service Provider Conformance)**.** *Let $SLA = \left\{\alpha^U, \alpha^L, \Psi^U, \Psi^L\right\}$ be an SDP for a service and D the delay flow emitted by the service. The service provider acts conform to*

- *the upper delay contract in SLA, if D is conform to the delay shaper implementing Ψ^U by Definition 6.2.9*



R
$R \leq R \underline{\otimes} \alpha^U$
$R \geq R \overline{\otimes} \alpha^L$
SDP 1
$\{ \alpha^U, \alpha^L, \Psi^U, \Psi^L \}$
R^*
$D \geq D \overline{\otimes} \Psi^L$
$D \leq D \underline{\otimes} \Psi^U$
D

Figure 8.2: Service modeled by an SDP

- *the lower delay contract in SLA, if D is conform to the delay prefetcher implementing Ψ^L by Definition 8.3.4*
- *the delay contract in SLA, if both previous conditions hold.*

Again, a lower bound Ψ^L for the response time is not always required, in this case one can set $\Psi^L = 0$. When $SLA = \{0, \alpha^U, 0, \Psi^U\}$ the notation is abbreviated to $SLA = \{\alpha^U, \Psi^U\}$.

An SLA modeled by an SDP is valid when both contracting parties act according to the SDP. Since the customer is the active contractor generating the workload the provider has to process the following implication has to hold:

Definition 8.3.8 (SDP Fulfillment). *Let SLA be an SDP. The upper/lower contract in SLA is fulfilled if*

$$\begin{aligned}&\textit{Service customer acts conform to upper/lower contract in SLA}\\&\Rightarrow\\&\textit{Service provider acts conform to upper/lower delay contract in SLA}\end{aligned} \tag{8.21}$$

An SDP A is fulfilled if

$$\textit{Service customer is conform to SLA} \Rightarrow \textit{Service provider is conform to SLA} \tag{8.22}$$

holds.

Whenever the customer acts non-conformant to the SDP the provider is not obliged to keep his response time guarantee. The consequence for a SOA performance model is that it is only valid, if all contracting parties are active within the specified bounds. In any other case, the model simply does not apply. An assignment of an SDP to a service is depicted in Figure 8.2, it abstracts the detailed basic service model in Figure 8.1 to the requirements for conformant arrival and delay flows.

The implication in SDPs has also a different effect for systems operating in normal, not worst-case conditions.

Lemma 8.5 (Delay Curve for Non Worst-Case Loads). *A service guarantees SDP $SLA = \{\alpha^U, \alpha^L, \Psi^U, \Psi^L\}$. The input R to the system is further restrained with upper arrival curve α^U_v using factor v, $0 < v \leq 1$ such that $\alpha^U_v = v \cdot \alpha^U$. Regardless on the changed input flow the delay contract is not altered.*

Proof. When R is conform to α_v^U, and since $\alpha_v^U \leq \alpha^U$, R is also conform to α^U. Given that, the customer acts conform to the SDP. The service provider also has to react conform to the SDP, thus delay contract Ψ^U (and Ψ^L) applies, but is not obliged to scale the reaction time by v. $\square$

In other words, reducing the load to the service does not improve the delay curve contract initially intended for the worst-case load. The curve contracts are artifacts representing guarantees and not actual performance figures. Of course, in a real system the average response time would improve, but that does not interfere with worst-case bounds.

8.4 Contracts for Composed Services

A single SDP represents the contracted performance bounds for a SOA service. Services can be composed to larger systems as described in Section 2.2.1, therefore a performance model has to reflect this. By combining SDPs one can construct a performance bound model that merges all SLAs of participating services into a single SDP. It can either serve as a performance description for a workflow or as a component model for higher service hierarchies.

To form networks of services described by SDPs results for concatenation, parallelization and synchronization are required. The combination of two SDPs is performed by combination of their corresponding curve contracts. For upper arrival curves serial and parallel combinations have been presented previously in Sections 6.4.1 and 6.4.2 in the context of Network Calculus shaper elements. For lower arrival and delay curves as well as upper delay curves results for such constructs are still missing and will be derived in the following.

8.4.1 Prefetcher Concatenation

The serial concatenation of prefetchers is given by the following theorem.

Theorem 8.6 (Prefetcher Concatenation)**.** *Let R be an arrival flow passing through a tandem of prefetchers with lower arrival envelopes α_1^L and α_2^L. The departure flow is given by*

$$R^* = R \overline{\otimes} (\alpha_1^L \overline{\otimes} \alpha_2^L) \tag{8.23}$$

Proof. R_1^* is the departure flow of the first prefetcher given by

$$R_1^* = R \overline{\otimes} \alpha_1^L \tag{8.24}$$

When R_1^* is fed into the second prefetcher we have

$$\begin{aligned} R_2^* &= R_1^* \overline{\otimes} \alpha_2^L && (8.25)\\ &= (R \overline{\otimes} \alpha_1^L) \overline{\otimes} \alpha_2^L && (8.26)\\ &= R \overline{\otimes} (\alpha_1^L \overline{\otimes} \alpha_2^L) \qquad \text{associativity of } \overline{\otimes} && (8.27)\end{aligned}$$

$\square$

8.4.2 Prefetcher Parallelization

Results for parallel shapers and thus for upper arrival bounds are given for Network Calculus in Section 6.4.2. They are based on Lemma 6.3 that allows us to replace (min,+)-convolution by the pointwise minimum if the functions are sub-additive. To derive a SDP for parallel patterns, results for lower arrival bounds are also needed, and a similar result has to be shown for the (max,+)-convolution.

Theorem 8.7 (Convolution of Super-Additive Functions)**.** *When two functions f and g are super-additive then their convolution can be simplified to the point-wise maximum.*

$$\underline{f} \,\overline{\otimes}\, \underline{g} = \underline{f \vee g} \tag{8.28}$$

The proof is based on the proof of Lemma 2.1.5 (xi) in [38] and is given in the Appendix. Now filter bank summation for prefetchers can be formulated.

Theorem 8.8 (Prefetcher Filter Bank Summation)**.** *For two prefetchers with curves g_1 and g_2 the parallel construct is a prefetcher with curve*

$$f = g_1 \vee g_2 \tag{8.29}$$

if $\underline{g_1} \vee \underline{g_2}$ is super-additive.

Proof. Let R_1^* and R_2^* be the departure flow of the two prefetchers and R^* the output of the combined system. From the definition of a prefetcher one can write

$$\begin{aligned} R^* &= R_1^* \vee R_2^* && && (8.30)\\ &= R \,\overline{\otimes}\, \underline{g_1} \vee R \,\overline{\otimes}\, \underline{g_2} && \text{Theorem 8.1} && (8.31)\\ &= R \,\overline{\otimes}\, (\, \underline{g_1} \vee \underline{g_2} \,) && \text{distributivity of } \overline{\otimes} && (8.32) \end{aligned}$$

Term $\underline{g_1} \vee \underline{g_2}$ is super-additive and $\underline{g_1}(0) = \underline{g_2}(0) = 0$, hence one can replace the term with its super-additive closure.

$$\begin{aligned} R^* &= R \,\overline{\otimes}\, \underline{\underline{g_1} \vee \underline{g_2}} && && (8.33)\\ &= R \,\overline{\otimes}\, (\underline{\underline{g_1}} \,\overline{\otimes}\, \underline{\underline{g_2}}) && \text{Theorem 8.7} && (8.34)\\ &= R \,\overline{\otimes}\, (\underline{g_1} \,\overline{\otimes}\, \underline{g_2}) && && (8.35)\\ &= R \,\overline{\otimes}\, \underline{(\, g_1 \vee g_2 \,)} && && (8.36) \end{aligned}$$

□

8.4.3 Arrival Contract Conformance in Tandem Services

Compared to Network Calculus imposing arrival contracts on services requires additional arrangements in SLA Calculus when . Contract conformance of simple serial compositions of two services are already dependent on workload restrictions and their calling sequence.

Stability by Conformance

For concatenated servers in a Network Calculus system, stability is determined using the combined service curve (Section 6.3.1). Serial compositions of services may become “unstable” in SLA Calculus analysis, too. This is the case when the arrival flow towards the composition is conform to the first, but not to one of the subsequent services being a bottleneck. Hence, we need an approach to determine the workload bound acceptable for all services in a serial composition. In the basic service model arrival contracts are guarded with shapers. Service compositions are, in terms of arrival contracts, the composition of their arrival flow shapers.

Corollary 8.9 (Upper Arrival Contract for Tandem Services)**.** *Given the situation of two services $i \in \{1,2\}$ with SDPs $SLA_i = \{\alpha_i^U, \Psi_i^U\}$ in a service tandem by Definition 8.2.1. The common input bound accepted by the composition is*

$$\alpha_{common}^U = \alpha_1^U \underline{\otimes} \alpha_2^U \tag{8.37}$$

Proof. Let R be the request arrival flow towards the first service. Its arrival contract is, given the basic service model, modeled by α_1^U-shaper conformance (Definition 6.2.9). It limits a conform arrival flow to $R_1^* = R \underline{\otimes} \alpha_1^U$. Appending the second service is equal to the concatenation of both shapers. When R_1^* is fed into the next shaper implementing α_2^U the flow is limited to

$$\begin{aligned}
R_2^* &= R_1^* \underline{\otimes} \alpha_2^U && && (8.38)\\
&= (R \underline{\otimes} \alpha_1^U) \underline{\otimes} \alpha_2^U && && (8.39)\\
&= R \underline{\otimes} (\alpha_1^U \underline{\otimes} \alpha_2^U) && \text{associativity of } \underline{\otimes} && (8.40)
\end{aligned}$$

□

With this result, a service composition can be considered as stable, when the arrival flow is conform to the composition of input shapers.

Output Burstiness Breaks Arrival Contracts

Limiting the arrival flow towards a composition is done according to the shaper with the weakest arrival contract. Still there is a second kind of Network Calculus node in our basic service model: The effect of servers on arrival contracts has to be considered, too.

Theorem 6.11 and the commutativity of (min,+)-convolution allows Network Calculus servers to be concatenated in an arbitrary sequence without changing the service of the composition. Furthermore, in Section 6.4.3 we saw that even shapers can be inserted into those server chains to enforce arrival envelopes, without influencing results on maximum delay or backlog. When joining basic service model elements (Figure 8.1) chains of shapers and servers are connected. This time, the effect of shapers cannot be ignored. In the SLA Calculus model world the statement “greedy shapers come for free” [107, Section 1.5.3] does not hold. Fixed arrival contracts to services and possible output burstiness increases render the situation more complicated.

Given the situation of two services S_1, S_2 aligned to a tandem with identical SDPs $SLA = \{\alpha^U, \Psi^U\}$. A request flow conform to $\alpha^U = \alpha^U \underline{\otimes} \alpha^U$ arrives at the system. Furthermore, service curve β_1^L is known for S_1. The departure flow for S_1 is limited from above by $\alpha_1^{U'} = \alpha^U \underline{\oslash} \beta_1^L$ by Theorem 6.8. Now consider the situation with $\alpha_1^{U'}(t) > \alpha^U(t)$ for some t, thus the burstiness increase induced by S_1 renders the arrival flow non-conform to α^U required by S_2. As a consequence, S_2 is not obliged to delay contract Ψ^U anymore, and no upper delay limit for the tandem system can be computed.

Three options are available to recreate a conformant arrival flow to subsequent services:

1. Limit the input flow bound to the serial composition as such that $\alpha_i^U \underline{\oslash} \beta_i^L \leq \alpha_{i+1}^U$ for all services $i \in \{1, \ldots, n-1\}$. However, this would lead to large unused capacities especially when a large set of diverse services participates in the service chain.

2. Require all service providers to shape departure flows as such that $\alpha_i^{U'} = \alpha_i^U$, and to include the resulting delay in their response time guarantees. At a first glance this option seems favorable for simplicity of SOA performance modeling, since no burstiness increases have to be considered. At a second glance the requirement cannot be fulfilled by many service providers and is seldom found in SLAs.

3. Apply shaping to every departure flow as such that the arrival contract to the subsequent service always holds ("Reshaping"). This time, in contrast to the situation described in Section 6.4.3, the costs of shaping cannot be shifted or removed. Shaping requires backlogging, and backlogging produces additional delay that has to be included in the response time model.

A possible implementation of the third option using WEEs realizing service workflows is shown in Figure 8.3. Two services with given SDPs are in a serial composition, the request flow towards service 1 is limited by $\alpha_{common}^U = \alpha_1^U \underline{\otimes} \alpha_2^U$. The departure flow leaving service 1 suffers from burstiness increase described by $\alpha_{common}^U \underline{\oslash} \beta_1^L$. It is up to the WEE to prepare the request flow using a shaper with shaping curve α_2^U to comply to the composed arrival contract again.

In the following we will give bounds for the third option's additional delay and we will include them as reshaping delay φ in workflow composition patterns. If one of the first options is used, the results for workflow composition patterns do still hold with $\varphi = 0$.

Delay due to Reshaping

Bounding delay due to reshaping is to find the maximum virtual delays of arrivals passing the shapers in WEEs. Since their input is the output of the server in our basic service model, it involves the computation of output flow bounds $\alpha^{U'} = \alpha^U \underline{\oslash} \beta^L$ by Theorem 6.8, too. Thus, the maximum delay for a shaper taking $\alpha^{U'}$ as input and implementing a shaping curve σ is

$$d_{reshape} = \max_t \left\{ d\left(\alpha^{U'}, \sigma\right)(t) \right\} \tag{8.41}$$

using Definition 6.2.5.

Although the reshaping delay can be easily computed in Network Calculus, this approach has a severe drawback for SLA Calculus models since β^L is unknown. However, using

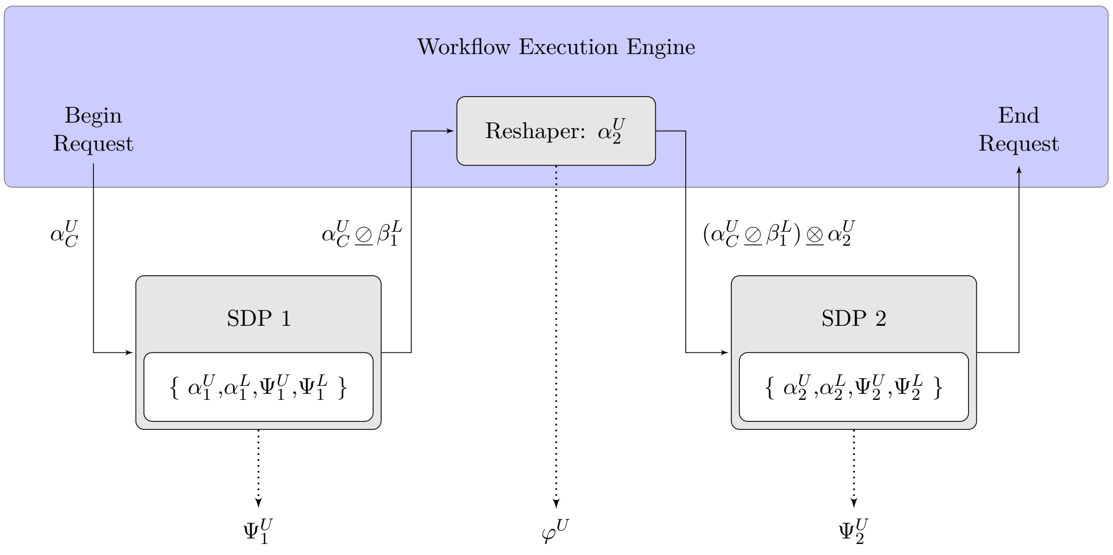

Workflow Execution Engine
Begin Request
Reshaper: α_2^U
End Request
α_C^U
$\alpha_C^U \underline{\oslash} \beta_1^L$
$(\alpha_C^U \underline{\oslash} \beta_1^L) \underline{\otimes} \alpha_2^U$
SDP 1
$\{ \alpha_1^U, \alpha_1^L, \Psi_1^U, \Psi_1^L \}$
SDP 2
$\{ \alpha_2^U, \alpha_2^L, \Psi_2^U, \Psi_2^L \}$
Ψ_1^U
φ^U
Ψ_2^U

Figure 8.3: Service call and reshaping by WEE for two subsequent services. Curves identify upper flow envelopes.

service curve computation as presented in the upcoming Chapter 9, one is able to find a sufficient replacement for β^L just by the knowledge of an SDP.

Theorem 8.10 (Lower Service Curve from Delay Bounds)**.** *Let α^U be the upper arrival curve and Ψ^U be the upper delay curve for a system. A lower service curve β^L sufficient to process a flow limited by α^U is given by*

$$\beta^L(t) = \alpha^U \left(t - \frac{\mathrm{d}}{\mathrm{d}t} \Psi^U(t) \right) \underline{\oslash} \, \alpha^U(t) \tag{8.42}$$

The proof is given later in Chapter 9, too. In Theorem 8.10 input α^U is shifted in such a way that it represents the worst-case output flow that is allowed by Ψ^U. The service curve is reconstructed by deconvolving the input from the output.

Service curve computation applied to the SDP service model enables us to compute the reshaping delay for request departure flows. The departure flows in turn are dependent on the service input. For serial concatenations the output can be computed by successive application of Theorem 6.8.

Definition 8.4.1 (Upper Service Departure Bounds)**.** *Let $SLA_i = \{\alpha_i^U, \alpha_i^L, \Psi_i^U, \Psi_i^L\}$ be the ith service in a serial composition and β_i^L the provided service curve. Then the departure flow $\alpha_i^{U\prime}$ of SLA_i is given by recursion*

$$\alpha_i^{U\prime} = \alpha_{i-1}^{U*} \underline{\oslash} \, \beta_i^L \tag{8.43}$$

$$\alpha_i^{U*} = \alpha_i^{U\prime} \underline{\otimes} \, \alpha_{i+1}^U \tag{8.44}$$

The recursion stops if there is no further input bound, we set $\alpha_0^{U*} = \alpha_{common}^U$ computed by Corollary 8.9.

Using a common upper arrival bound ensures that the arrival flow is conform to each service of the composition in its long-term rate. Burstiness increases exist, but can be removed by shapers for the cost of additional delay.

Corollary 8.11 (Maximum Reshaping Delay in Service Tandems)**.** *Let two services be described by SDP $SLA_1 = \{\alpha_1^U, \alpha_1^L, \Psi_1^U, \Psi_1^L\}$ and SDP $SLA_2 = \{\alpha_2^U, \alpha_2^L, \Psi_2^U, \Psi_2^L\}$ be in a serial composition. The arrival flow towards the composition has an upper envelope of α_C^U and is arrival conform for each service. Then the maximum delay due to reshaping of the departure flow of service 1 to be consistent with arrival contract α_2^U is*

$$h_{\max}\left(\alpha_C^U \underline{\oslash} \beta_1^L, \alpha_2^U\right) \tag{8.45}$$

with β_1^L as a service curve provided by the first system.

Proof. Application of Theorem 6.8 gives us the upper departure flow bound for SLA_1. To conform to the second arrival contract the departure flow of SLA_1 is shaped to envelope α_C^U again. Definition 6.2.6 is used to compute the virtual shaper delay. Since the shaping curve is the service curve of a shaper [107, Corollary 1.5.1] one can replace the shaper output with shaping curve α_C^U. □

For the upcoming theorems the reshaping delay has to be included in the descriptions of delay flow bounds. To reduce the number of used symbols we define a function that gives us an upper delay curve limiting the additional delay flow.

Definition 8.4.2 (Reshaping Delay Curve)**.** *Let two services be in a serial composition, they are described by SDPs $\{\alpha_{i-1}^U, \alpha_{i-1}^L, \Psi_{i-1}^U, \Psi_{i-1}^L\}$ and $\{\alpha_i^U, \alpha_i^L, \Psi_i^U, \Psi_i^L\}$. The additional delay flow due to reshaping between the first and second service is bound from above by a linear delay curve (Theorem 7.2):*

$$\varphi_i^U(t) = h_{\max}\left(\alpha_{i-1}^{U'}, \alpha_i^U\right) \cdot t \tag{8.46}$$

Earliness and Lower Output Bounds

A phenomenon inversely to burstiness increase at server outputs can be observed for lower output bounds. The output of a service might drop below the lower input bound of the subsequent service, as a consequence the guaranteed lower delay contract becomes invalid.

In the basic SLA Calculus service model (Figure 8.1) a prefetcher implementing α^L acts as a guard for the lower bound. Again a common lower arrival bound, similar to Corollary 8.9, can be stated giving contract conformance for a workflow by the combination of prefetchers.

Theorem 8.12 (Lower Arrival Contract for Tandem Services)**.** *Given the situation of two services $i \in \{1, 2\}$ with SDPs $SLA_i = \{\alpha_i^U, \Psi_i^U\}$ in a service tandem by Definition 8.2.1. The common lower input bound accepted by the composition is*

$$\alpha_{common}^L = \alpha_1^U \overline{\otimes} \alpha_2^U \tag{8.47}$$

Proof. Lower arrival bound α^L_{common} is equal to α^L_{common} prefetcher conformance (Definition 8.3.4). Hence, it follows from the concatenation of prefetchers by Theorem 8.6. □

For a serial concatenation of prefetchers this lower bound would give a "stable" system. However, this stability is compromised by the server elements in the basic service model. As their processing requests do depart later, the output bound might be unacceptable for the next service.

The lower output envelope $\alpha^{L'}_1$ of a server component is given by Corollary 6.7: $\alpha^{L'}_1 \geq \alpha^L_1 \underline{\otimes} \beta^L_1$. By definition of the (min,+)-convolution we can say that $\alpha^{L'}_1(t) \leq \alpha^L_1(t)$ for all t, figuratively lower bounds get even lower when passing servers. For a second service in serial composition this might result in a situation with $h_{early}\left(\alpha^{L'}_1, \alpha^U_2\right)(t) > 0$ for some t. The implication in the lower delay contract of service 2 will evaluate to false.

Inversely to Definition 8.4.1 the lower service departure bounds in a serial composition can be computed using recursion.

Definition 8.4.3 (Lower Service Departure Bounds). *Let $SLA_i = \{\alpha^U_i, \alpha^L_i, \Psi^U_i, \Psi^L_i\}$ be the ith service in a serial composition, β^L_i the provided service curve. Then the lower envelope of departure flow $\alpha^{U'}_i$ of SLA_i is given by recursion*

$$\alpha^{L'}_i = \alpha^{L'}_{i-1} \underline{\otimes} \beta^L_i \tag{8.48}$$

The recursion stops if there is no further input bound. We set $\alpha^{L'}_0 = \alpha^L_{common}$ computed by Theorem 8.12.

For service i a too low, non-conform flow arriving from service $i-1$ is identified when

$$\max_t \left\{h_{early}\left(\alpha^{L'}_{i-1}, \alpha^U_i\right)(t)\right\} > 0 \text{ (Definition 8.3.4)} \tag{8.49}$$

Contrary to the upper bounds, this knowledge on earliness cannot be used to add a correction term to lower delay curves in a sound way. While a greedy shaper can delay early arrivals, there is no realistic concept to force a prefetcher to deliver requests which did not arrive yet, therefore we did not include it in Definition 8.4.3. One could inject missing requests into the flow to keep subsequent services busy, but this approach cannot be justified for SOA. For this reasons lower arrival bounds and the implication on the lower delay contract get a different interpretation than upper bounds.

- Equation (8.49) is used to identify possible lower contract breaches in a workflow. The implication is marked as unsatisfiable.

- Regardless on lower arrival contract breaches we continue to compute lower delay bounds. While SLA validation including lower bounds is out of reach the information on delay bounds is still valuable. It provides the modeler with information on the intrinsic delay and thus, the minimum response time for requests for any workload.

Now results for combinations of single SDPs to compositions are derived. Next to curve combinations, according to workflow patterns, the reshaping delay is added.

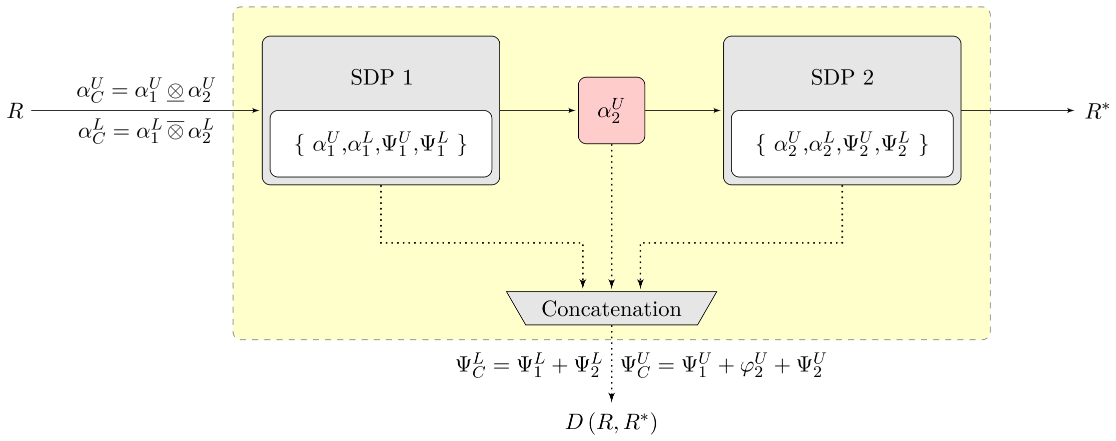

R
$\alpha_C^U = \alpha_1^U \underline{\otimes}\, \alpha_2^U$
$\alpha_C^L = \alpha_1^L \overline{\otimes}\, \alpha_2^L$
SDP 1
$\{ \alpha_1^U, \alpha_1^L, \Psi_1^U, \Psi_1^L \}$
α_2^U
SDP 2
$\{ \alpha_2^U, \alpha_2^L, \Psi_2^U, \Psi_2^L \}$
R^*
Concatenation
$\Psi_C^L = \Psi_1^L + \Psi_2^L$
$\Psi_C^U = \Psi_1^U + \varphi_2^U + \Psi_2^U$
$D(R, R^*)$

Figure 8.4: Concatenation of two services described by SDPs

8.4.4 Serial SLA Delay Property Compositions

Sequences of services as described in Definition 8.2.1 are mapped to feed-forward networks [38] of service providers. A similar model is proposed in [45]. Figure 8.4 shows a construction blueprint including the shaper provided by WEEs to handle output burstiness. For limits on delay flows in tandem systems the following can be stated:

Theorem 8.13 (Upper Delay Curve Summation)**.** *Let D_1 and D_2 be two delay flows generated by a tandem system with upper delay curves Ψ_1^U and Ψ_2^U. Then the combined delay $D_C = D_1 + D_2$ is upper-constrained by $\Psi_C^U = \Psi_1^U + \Psi_2^U$.*

Proof. Using Definition 7.1.4 we get $D_1 \leq D_1 \underline{\otimes}\, \Psi_1^U$, $D_2 \leq D_2 \underline{\otimes}\, \Psi_2^U$ and $D_C \leq D_C \underline{\otimes}\, \Psi_C^U$.

$$
\begin{aligned}
D_C(t) &\leq (D_1 \underline{\otimes}\, \Psi_1^U) + (D_2 \underline{\otimes}\, \Psi_2^U) \\
&= \inf_{0 \leq s \leq t} \left\{ D_1(t-s) + \Psi_1^U(s) \right\} + \inf_{0 \leq v \leq t} \left\{ D_2(t-v) + \Psi_2^U(v) \right\} && \text{Def. 6.1.6} \\
&= \inf_{0 \leq s \leq t} \left\{ \inf_{0 \leq v \leq t} \left\{ D_1(t-s) + \Psi_1^U(s) + D_2(t-v) + \Psi_2^U(v) \right\} \right\} \\
&= \inf_{0 \leq s \leq t} \left\{ \inf_{0 \leq v \leq t} \left\{ D_1(t-s) + D_2(t-v) + \Psi_1^U(s) + \Psi_2^U(v) \right\} \right\} \\
&\leq \inf_{0 \leq s \leq t} \left\{ D_1(t-s) + D_2(t-s) + \Psi_1^U(s) + \Psi_2^U(s) \right\} && s = v \\
&= \inf_{0 \leq s \leq t} \left\{ D_C(t-s) + \Psi_C^U(s) \right\} \\
&= (D_C \underline{\otimes}\, \Psi_C^U)(t) && \text{Def. 6.1.6}
\end{aligned}
$$

□

For lower delay bounds a similar result can be shown using (max,+)-algebra.

Theorem 8.14 (Lower Delay Curve Summation)**.** *Let D_1 and D_2 be two delay flows generated by a tandem system with lower delay curves Ψ_1^L and Ψ_2^L. Then the combined delay $D_C = D_1 + D_2$ is lower-constrained by $\Psi_C^L = \Psi_1^L + \Psi_2^L$.*

Proof. Using Definition 7.1.5 we get $D_1 \geq D_1 \overline{\otimes} \Psi_1^L$, $D_2 \geq D_2 \underline{\otimes} \Psi_2^L$ and $D_C \geq D_C \overline{\otimes} \Psi_C^L$.

$$\begin{aligned}
D_C(t) &\geq (D_1 \overline{\otimes} \Psi_1^L) + (D_2 \overline{\otimes} \Psi_2^L) \\
&= \sup_{0 \leq s \leq t} \left\{ D_1(t-s) + \Psi_1^L(s) \right\} + \sup_{0 \leq v \leq t} \left\{ D_2(t-v) + \Psi_2^L(v) \right\} && \text{Def. 6.1.8} \\
&= \sup_{0 \leq s \leq t} \left\{ \sup_{0 \leq v \leq t} \left\{ D_1(t-s) + D_2(t-v) + \Psi_1^L(s) + \Psi_2^L(v) \right\} \right\} \\
&\geq \sup_{0 \leq s \leq t} \left\{ D_1(t-s) + D_2(t-s) + \Psi_1^L(s) + \Psi_2^L(s) \right\} && s = v \\
&= (D_C \overline{\otimes} \Psi_C^L)(t) && \text{Def. 6.1.8}
\end{aligned}$$

□

With the previous results the following theorem on SDP concatenation can be stated:

Theorem 8.15 (SDP Concatenation)**.** *Given is a set of SDPs $SLA_i = \{\alpha_i^U, \alpha_i^L, \Psi_i^U, \Psi_i^L\}$, $i \in \{1, \ldots, n\}$ and β_i^L is computed by Theorem 8.10. Then the serial concatenation to SDP $SLA_C = \{\alpha_C^U, \alpha_C^L, \Psi_C^U, \Psi_C^L\}$ is given by*

$$\alpha_C^U = \underline{\bigotimes}_{i=1}^{n} \alpha_i^U \tag{8.50}$$

$$\alpha_C^L = \overline{\bigotimes}_{i=1}^{n} \alpha_i^L \tag{8.51}$$

$$\Psi_C^U = \sum_{i=1}^{n} \left(\Psi_i^U + \varphi_i^U \right) \tag{8.52}$$

$$\Psi_C^L = \sum_{i=1}^{n} \Psi_i^L \qquad \textit{if } \max_t \left\{ h_{early}\left(\alpha_{i-1}^{L'}, \alpha_i^L \right)(t) \right\} = 0 \ \forall i = 1 \ldots n \tag{8.53}$$

If the condition for Ψ_C^L does not hold, $\sum_{i=1}^{n} \Psi_i^L$ computes the intrinsic delay.

Proof. Let R be the request arrival flow processed by the serial concatenation of services.

1. The concatenation of n services follows from Corollary 8.9. Repeating the concatenation for $\alpha_i^U, \ldots, \alpha_n^U$ leads to
$$R^* = R \underline{\otimes} \underline{\bigotimes}_{i=1}^{n} \alpha_i^U \tag{8.54}$$
2. The concatenation of n services follows from Theorem 8.6 and the associativity of $\overline{\otimes}$. Lower arrival bound α_C^L is equal to α_C^L prefetcher conformance (Definition 8.3.4).
3. The first summand of upper delay bound Ψ_C^U follows from $D = \Psi^U$ and Theorem 8.13. The second summand is the cumulated reshaping delay necessary between services by Corollary 8.11.
4. Lower delay bound Ψ_C^L also follows from $D = \Psi^L$ and Theorem 8.14. It holds as long as the input to subsequent services does not violate the lower arrival contract, violations are detected by Equation (8.49).

□

8.4.5 Parallel SLA Delay Property Compositions

For parallel compositions of services results for limits of arrival flows are given by the filter bank summation of shapers (Theorem 6.10) and prefetchers (Theorem 8.8). To combine delay curves in a parallel setup, a method to combine delay flows is derived at first.

Theorem 8.16 (Upper Bound for Parallel Delay Flows). *Two systems S_1 and S_2 are combined to a parallel system by Definition 8.2.3 or Definition 8.2.4. S_1 emits a delay flow bound by Ψ_1^U, the equivalent case holds for S_2. The arrival flow R is bounded by α_1^U and α_2^U Then the upper delay bound for both systems processing in parallel is $\Psi_P^U = \Psi_1^U \vee \Psi_2^U$.*

Proof. The proof considers both possible cases of a fork/join synchronization or a routing scheme.

The routing case (Definition 8.2.3) assumes arrival flow R is demultiplexed to R_1 and R_2 in an arbitrary scheme. S_1 and S_2 work and finish requests independent of each other, they have departure flows R_1^* and R_2^*. Recalling Definition 7.1.1 and 7.1.3 the delay flow D_P is thus given by

$$D_P(t) = \int_0^t (R^*)^{-1}(R(x)) - (R)^{-1}(R(x))\,\mathrm{d}x \tag{8.55}$$

Due to the routing flow R is split into subflows R_1 and R_2 with $R = R_1 + R_2$, after processing the departure flows R_1^* and R_2^* are multiplexed to $R^* = R_1^* + R_2^*$. For the delay we then observe

$$D_P(t) = \int_0^t (R_1^* + R_2^*)^{-1}((R_1 + R_2)(x)) - (R_1 + R_2)^{-1}((R_1 + R_2)(x))\,\mathrm{d}x \tag{8.56}$$

Since S_1 and S_2 process their arrivals independent of each other, one of both will finish jobs arrived in t later. To enable an independent determination of delay the integral is split up. The maximum of both delays is the crucial factor on the departure process.

$$\begin{aligned}
D_P(t) \leq \int_0^t & [(R_1^*)^{-1}(R_1(x)) - (R_1)^{-1}(R_1(x)) \\
& \text{and } (R_2^*)^{-1}(R_2(x)) - (R_2)^{-1}(R_2(x))]\,\mathrm{d}x && (8.57)\\
= \int_0^t & \max\{(R_1^*)^{-1}(R_1(x)) - (R_1)^{-1}(R_1(x)), \\
& (R_2^*)^{-1}(R_2x)) - (R_2)^{-1}(R_2(x))\}\,\mathrm{d}x && (8.58)\\
= \max & \left\{\int_0^t d\left(R_1, R_1^*\right)(x)\,\mathrm{d}x, \int_0^t d\left(R_2, R_2^*\right)(x)\,\mathrm{d}x\right\} \text{ Def. 7.1.1} && (8.59)\\
= \max & \left\{D_1(t), D_2(t)\right\} \qquad \text{Def. 7.1.3} && (8.60)
\end{aligned}$$

Delay flow D_1 is bound by Ψ_1^U and D_2 by Ψ_2^U from above, thus

$$D_P \leq D_P \underline{\otimes} \left(\Psi_1^U \vee \Psi_2^U\right) \tag{8.61}$$

Synchronization (Definition 8.2.4) is a special case of the results above with $R_1 = R_2 = R$. □

Theorem 8.17 (Lower Bound for Parallel Delay Flows)**.** *Two systems S_1 and S_2 are combined to a parallel system. S_1 emits a delay flow bound from below by Ψ_1^L, the equivalent case holds for S_2 and Ψ_2^L. The arrival flow is bounded from below by α_1^L and α_2^L. Then the lower delay bound for both systems processing in parallel in a fork/join scheme is $\Psi_P^L = \Psi_1^L \vee \Psi_2^L$. When an arbitrary routing scheme is used, the lower delay bound is given by $\Psi_P^L = \Psi_1^L \wedge \Psi_2^L$.*

Proof. For lower delay bounds we have to consider the best-case, thus all requests are distributed to the parallel setup in a way that response times are minimal.

Synchronization includes waiting for the slower service, thus the slowest service determines the overall delay. From the proof of Theorem 8.16 we know that

$$D_P(t) \leq \max\{D_1(t), D_2(t)\} \tag{8.62}$$

Delay flow D_1 is bound by Ψ_1^L and D_2 by Ψ_2^L from below, thus in the synchronized case we have

$$D_P \geq D_P \overline{\otimes} \left(\Psi_1^L \vee \Psi_2^L \right) \tag{8.63}$$

For the routing case the delay flow is given by the service that finishes first. The "and" becomes an "or" and by continuing with Equation 8.56 the minimum delay flow is

$$D_P(t) \leq \int_0^t \Big[(R_1^*)^{-1}(R_1(x)) - (R_1)^{-1}(R_1(x)) \text{ or } (R_2^*)^{-1}(R_2(x)) - (R_2)^{-1}(R_2(x))\Big]\, \mathrm{d}x \tag{8.64}$$

$$= \int_0^t \min\Big\{(R_1^*)^{-1}(R_1(x)) - (R_1)^{-1}(R_1(x)), (R_2^*)^{-1}(R_2x)) - (R_2)^{-1}(R_2(x))\Big\}\, \mathrm{d}x \tag{8.65}$$

$$= \min\{D_1(t), D_2(t)\} \qquad \text{Def. 7.1.1 and 7.1.3} \tag{8.66}$$

Using the lower bounds we receive

$$D_P \geq D_P \overline{\otimes} \left(\Psi_1^L \wedge \Psi_2^L \right) \tag{8.67}$$

$\square$

With these results the filter bank summation for SDPs is formulated. We start with parallel constructs featuring the routing of job requests to one of the services. Figure 8.5 includes the participating flows and services.

Theorem 8.18 (Parallel SDP Composition)**.** *$SLA_i = \{\alpha_i^U, \alpha_i^L, \Psi_i^U, \Psi_i^L\}$, $i \in \{1, \ldots, n\}$ is a set of SDPs. The parallel composition (Definition 8.2.3) of $SLA_1, \ldots, SLA_n$ to SDP $SLA_P = \{\alpha_P^U, \alpha_P^L, \Psi_P^U, \Psi_P^L\}$ is given by*

$$\alpha_P^U = \bigwedge\nolimits_{i=1}^n \alpha_i^U \tag{8.68}$$

$$\alpha_P^L = \bigvee\nolimits_{i=1}^n \alpha_i^L \tag{8.69}$$

$$\Psi_P^U = \bigvee\nolimits_{i=1}^n \Psi_i^U \tag{8.70}$$

$$\Psi_P^L = \bigwedge\nolimits_{i=1}^n \Psi_i^L \tag{8.71}$$

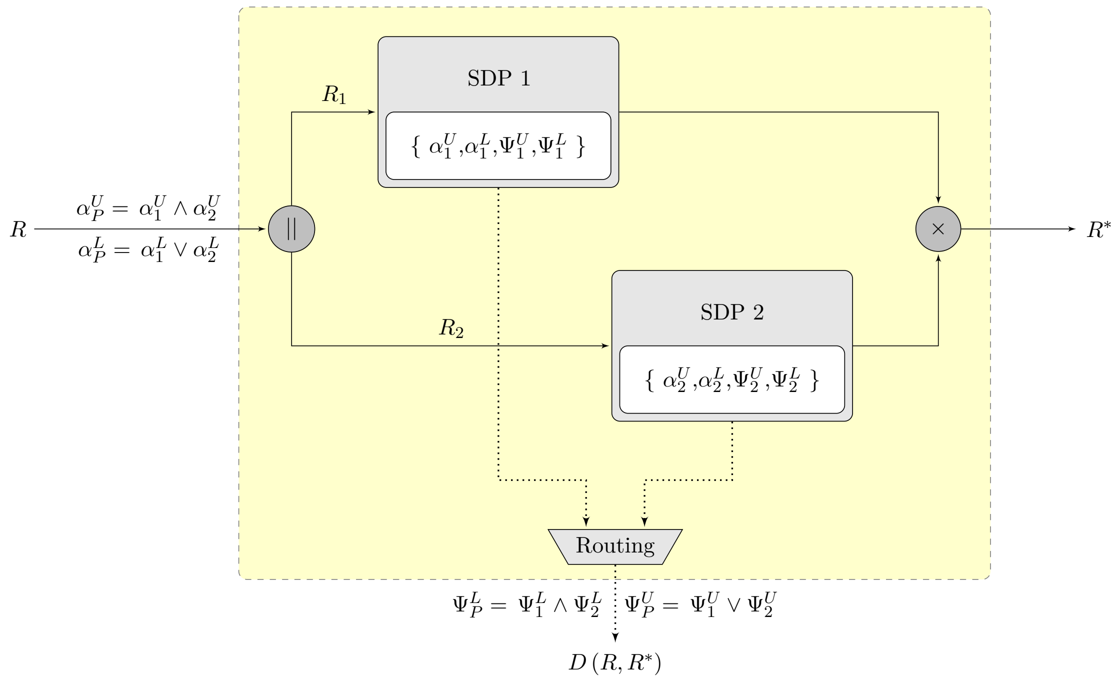

SDP 1
$\{ \alpha_1^U, \alpha_1^L, \Psi_1^U, \Psi_1^L \}$
SDP 2
$\{ \alpha_2^U, \alpha_2^L, \Psi_2^U, \Psi_2^L \}$
R
$\alpha_P^U = \alpha_1^U \wedge \alpha_2^U$
$\alpha_P^L = \alpha_1^L \vee \alpha_2^L$
$||$
R_1
R_2
$\times$
R^*
Routing
$\Psi_P^L = \Psi_1^L \wedge \Psi_2^L$
$\Psi_P^U = \Psi_1^U \vee \Psi_2^U$
$D(R, R^*)$

Figure 8.5: Parallel Composition of SDPs with job routing

Proof. Let R be a request arrival flow processed by n services in parallel.

1. Upper arrival bound α_P^U is equal to α_P^U shaper conformance. It follows from the filter bank summation of n shapers with a shaping curves α_i^U being an upper constraint on R_i (Equation 6.39). It limits the arrival flow to $R_i^* = R \underline{\otimes} \alpha_i^U$.

   The filter bank summation is given by Theorem 6.10 and the assiociativity of the minimum.

2. Lower arrival bound α_P^L follows from filter bank summation of prefetchers in Theorem 8.8 and the associativity of the maximum.

3. Upper delay bound Ψ_P^U follows from Theorem 8.16 and the associativity of the max operator.

4. Lower delay bound Ψ_P^L follows from Theorem 8.17 and the associativity of the min operator.

□

When two services are used in a fork/join scheme (see Figure 8.6) their combined SDP can be computed by the next theorem.

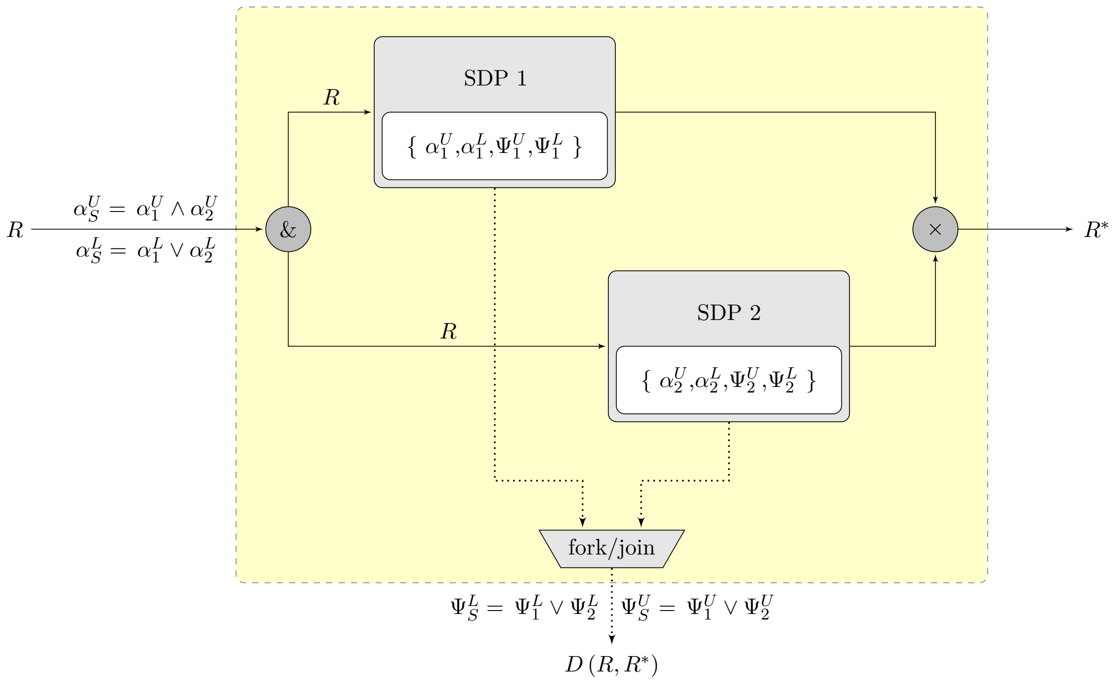

SDP 1
$\{ \alpha_1^U, \alpha_1^L, \Psi_1^U, \Psi_1^L \}$
SDP 2
$\{ \alpha_2^U, \alpha_2^L, \Psi_2^U, \Psi_2^L \}$
R
R
R
$\alpha_S^U = \alpha_1^U \wedge \alpha_2^U$
$\alpha_S^L = \alpha_1^L \vee \alpha_2^L$
&
$\times$
R^*
fork/join
$\Psi_S^L = \Psi_1^L \vee \Psi_2^L$
$\Psi_S^U = \Psi_1^U \vee \Psi_2^U$
$D(R, R^*)$

Figure 8.6: Synchronized composition of two SDPs

Theorem 8.19 (Synchronized Parallel SDP Composition)**.** *Given a set of SDPs $SLA_i = \alpha_i^U, \alpha_i^L, \Psi_i^U, \Psi_i^L$, $i \in \{1, \ldots, n\}$ the parallel composition offering fork/join semantics (Definition 8.2.4) to SDP $SLA_S = \{\alpha_S^U, \alpha_S^L, \Psi_S^U, \Psi_S^L\}$ is given by*

$$\alpha_S^U = \bigwedge\nolimits_{i=1}^{n} \alpha_i^U \tag{8.72}$$

$$\alpha_S^L = \bigvee\nolimits_{i=1}^{n} \alpha_i^L \tag{8.73}$$

$$\Psi_S^U = \bigvee\nolimits_{i=1}^{n} \Psi_i^U \tag{8.74}$$

$$\Psi_S^L = \bigvee\nolimits_{1}^{n} \Psi_i^L \tag{8.75}$$

Proof. The proof equals the proof of Theorem 8.18 except the result for the lower delay bound Ψ_S^L. It is replaced by the first result of Theorem 8.17 considering fork/join constructs. □

Both Theorems for parallel compositions do not include any reshaping delay. This is not necessary since every parallel composition can be merged into a single SDP and included in serial compositions.

Non-Concave Delay Curves

The general advise in Section 6.2.4 and 7.1.4 for instantiating upper bounds is to use concave piece-wise linear functions for their simple variable semantics. However, for two concave curves f, g the combination $f \vee g$ is not necessarily concave, for example, when Theorem 8.16 is applied. The interpretation of upper delay curves has to be extended to the non-concave case.

Figure 8.7 shows the phenomenon for delay curves f, g and $h = f \vee g$. Function h is constructed with affine functions γ_{r_i,b_i} with index $i = \{A \ldots D\}$. Line segment B limits a lower delay rate than segment D for the long-term delay rate, thus $r_B < r_D$. This curve form does not exclude delay bursts, they are still possible up to a size of b_D time units by the translation of the last affine segment. However, in accordance with the interpretation for concave delay curves, the burst is limited in its rate by preceding segments. By the first line segment, a burst is limited in its rate by r_A. The intersection point of segments B and C marks the preliminary end of a request processing slowdown switching to a phase of faster response times. Therefore, the lower rate in the second line segment can be interpreted as a guarantee that the system will recover after short delay bursts and catch up on long-term processing to some extent.

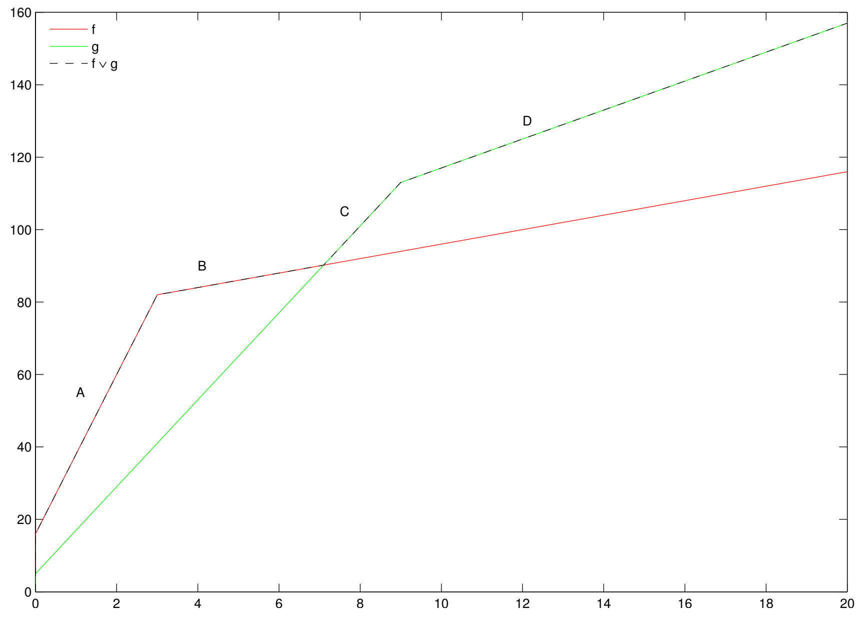


Figure 8.7: Pointwise maximum of two arrival curves is not necessarily concave.

8.4.6 Repeated Service Calls

We will derive arrival and delay bounds for services used in loops (Definition 8.2.4) as the last basic workflow pattern considered in this work, thus a single request is processed several times by the same resource. Figure 8.8 shows a representation of such a loop. For the maximum workload that can be processed by a service in a loop the following holds:

Theorem 8.20 (Arrival Flow Bound to Loops). *The request arrival flow entering a loop section is constrained by $\frac{\alpha^U}{v}$ from above and by $\frac{\alpha^L}{v}$ from below. When the arrival flow is looped through the service v times the overall arrival flow is constrained by α^U and α^L.*

Proof. An arrival flow constrained by $\frac{\alpha^U}{v}$ looped v times through the same service equals multiplexing v arrivals flows constrained by $\frac{\alpha^U}{v}$. Thus

$$\sum_{i=1}^{v} \frac{\alpha^U}{v} = v \cdot \frac{\alpha^U}{v} = \alpha^U \tag{8.76}$$

Lower bounds are shown by replacing α^U with α^L. □

Response time guarantees for SDPs are found by the next theorem.

Theorem 8.21 (Upper Delay Constraints in Loops). *A service S guarantees SDP $SLA = \{\alpha^U, \alpha^L, \Psi^U, \Psi^L\}$. An arrival flow limited by $\frac{\alpha^U}{v}$ is looped at most v times through the service. Then the delay flow D is constrained by $v \cdot \Psi^U$. A delay flow limit from below is given by $v \cdot \Psi^L$.*

Proof. Since the number of repetitions is at most v, the request arrival flow to S is conform to the SDP (by Theorem 8.20). The loop including service S can be unrolled to a sequence of v separate services S_i, $i = 1 \ldots v$, that are equal to S and also conform to the SDP. The arrival flow to S_1 is constrained by $\frac{\alpha^U}{v}$, due to Theorem 8.15 the upper arrival curve contract of the concatenation is

$$\alpha_C^U = \underline{\bigotimes}_{i=1}^{n} \frac{\alpha^U}{v} \tag{8.77}$$

Since curve contracts in SDPs are required to be sub-additive, α^U and $\frac{\alpha^U}{v}$ are sub-additive, hence α_C^U is an equal or lower bound than $\frac{\alpha^U}{v}$. Given that, the concatenation is conform to the arrival curve contract in the SDP.

Although each service S_i is processing a reduced workload compared to S Lemma 8.5 gives the original delay curves $\Psi_i^U = \Psi^U$ for all services i in worst-case. Furthermore, services S_i are in a serial concatenation and by Theorem 8.13 the upper delay bound is:

$$\sum_{i=1}^{v} \Psi_i^U = v \cdot \Psi^U \tag{8.78}$$

The lower limit follows from the same construction and Theorem 8.14. □

Theorem 8.22 (Looped SDPs). *A service modeled by SDP $SLA = \{\alpha^U, \alpha^L, \Psi^U, \Psi^L\}$ for a single request processes a job at most v times. Then the SDP SLA_v offered to the arrival flow is*

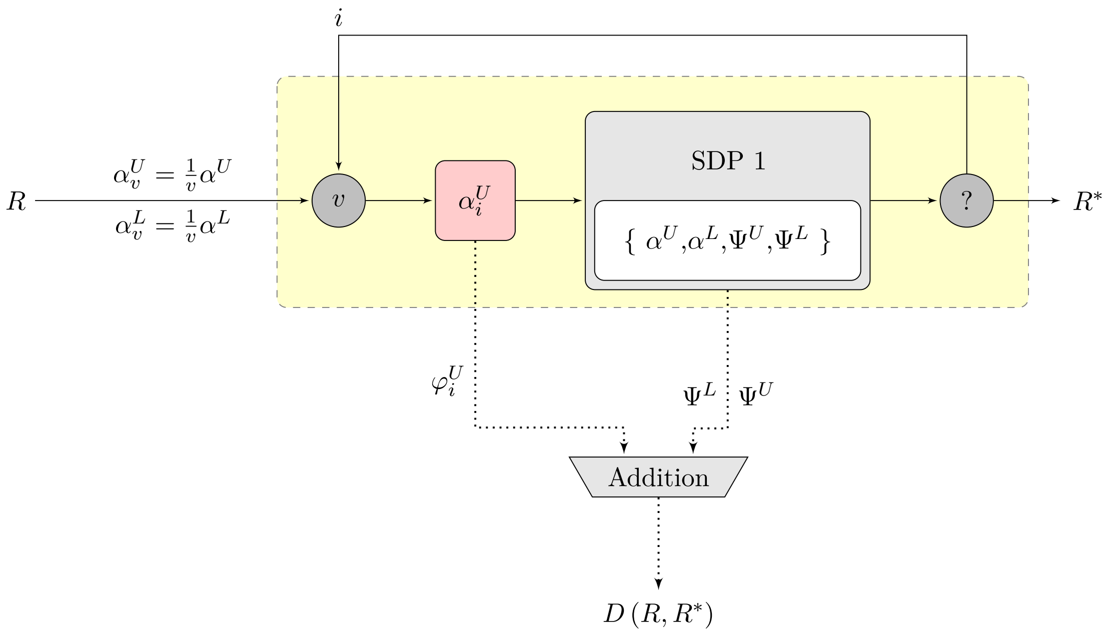
$$\alpha_v^U = \frac{1}{v} \cdot \alpha^U \tag{8.79}$$

$$\alpha_v^L = \frac{1}{v} \cdot \alpha^L \tag{8.80}$$

$$\Psi_v^U = v \cdot \Psi^U + \sum_{i=1}^{v} \varphi_i^U \tag{8.81}$$

$$\Psi_v^L = v \cdot \Psi^L \qquad \textit{if } \max_t \left\{ h_{early}\left(\alpha_{i-1}^{L'}, \alpha_i^L\right)(t) \right\} = 0 \ \forall i = 1 \ldots v \tag{8.82}$$

If the condition for Ψ_v^L does not hold, $v \cdot \Psi^L$ computes the intrinsic delay.



i
R
$\alpha_v^U = \frac{1}{v}\alpha^U$
$\alpha_v^L = \frac{1}{v}\alpha^L$
v
α_i^U
SDP 1
$\{ \alpha^U, \alpha^L, \Psi^U, \Psi^L \}$
?
R^*
φ_i^U
Ψ^L
Ψ^U
Addition
$D(R, R^*)$

Figure 8.8: A service modeled by an SDP is called v times.

Proof. For the constraints on the arrival flow Equation 6.84 originating from Real-Time Calculus is used. Delay bounds Ψ_v^U and Ψ_v^L follow from Theorem 8.21. $\square$

8.5 SOA Modeling with SLA Calculus

For building SOA models with SLA Calculus we start with the inclusion of SLAs (or more specific, the included SLOs) as they are the basic model element for a service in SLA Calculus. Then these single components will be joined to workflows using the theorems for SDP compositions.

8.5.1 Modeling SLAs with SLA Calculus

Modeling service performance is done in SLA Calculus by transferring arrival and delay contracts found in SLAs into SDPs. Each set of worst-case constraints becomes a performance characterization of the underlying processing functions. Arrival bounds are modeled with compositions of affine functions using the semantics described in Section 6.2.5. Rates, as well as, capacities for bursts can be considered. In a similar way the corresponding delay bounds are set up, as described in Section 7.1.4.

In the complementary modeling approach to SOAs with Network Calculus the relationship of arrival rates and resulting delay was split up. Instead, arrival and service rate bounds gave the maximum delay to decide on SLA conformity. With the precondition on the knowledge of service rates the approach to SOAs is open to providers only. Since SLAs are equally available to customers and providers modeling with SDPs in SLA Calculus can be used independently of the role in the business relationship.

Example 8.5.1. *Based on the arrival constraints and implicated delay commitments in Tables 2.1 and 2.2 the SDPs for the basic services in the ParcelSink workflow can be stated. For the external geocoder services the SLAs are known only, they give the SDPs*

$$SLA_{hollowearth} = \left\{ \begin{array}{ll} \alpha^U & = \min(\gamma_{25,10}, \gamma_{8,61}) \\ \alpha^L & = \beta_{3,5} \\ \Psi^U & = \min(\gamma_{30,5}, \gamma_{10,165}) \\ \Psi^L & = \beta_{4,3} \end{array} \right\}$$

$$SLA_{flatworld} = \left\{ \begin{array}{ll} \alpha^U & = \min(\gamma_{20,0}, \gamma_{15,10}) \\ \alpha^L & = \max(\beta_{1,0}, \beta_{2,1}) \\ \Psi^U & = \min(\gamma_{50,20}, \gamma_{5,112}) \\ \Psi^L & = \beta_{3,0} \end{array} \right\}$$

The remaining services, regardless if contracted externally or hosted at ParcelSink are described with three further SDPs.

$$SLA_{catalog} = \left\{ \begin{array}{ll} \alpha^U & = \min(\gamma_{12,10}, \gamma_{9,13}) \\ \alpha^L & = \beta_{1,5} \\ \Psi^U & = \gamma_{2,0} \\ \Psi^L & = \beta_{1,10} \end{array} \right\}$$

$$SLA_{fetchaddress} = \left\{ \begin{array}{ll} \alpha^U & = \gamma_{15,2} \\ \alpha^L & = \beta_{1,0} \\ \Psi^U & = \gamma_{10,5} \\ \Psi^L & = \beta_{2,4} \end{array} \right\} \text{ and } SLA_{printinvoice} = \left\{ \begin{array}{ll} \alpha^U & = \gamma_{25,6} \\ \alpha^L & = \beta_{6,4} \\ \Psi^U & = \gamma_{16,3} \\ \Psi^L & = \beta_{0.5,8} \end{array} \right\}$$

Finally the target SLA, the ParcelSink workflow, should conform to is given by

$$SLA_{Target} = \left\{ \begin{array}{ll} \alpha^U & = \min(\gamma_{100,5}, \gamma_{305,12}) \\ \alpha^L & = 0 \\ \Psi^U & = \gamma_{5,0} \\ \Psi^L & = 0 \end{array} \right\}$$

8.5.2 Mapping of BPEL Structures to SLA Calculus

SLA Calculus is specifically designed to model workflow structures in SOAs, so mapping BPEL workflows is straight forward. Each `<invoke/>` tag representing a service call interface is modeled with an SDP in SLA Calculus. Further, control structures forming workflows are mapped to SLA Calculus by applying the previously developed theorems whenever services are combined by the following BPEL tags:

<sequence/> constructs are formed by serial compositions of SDPs using Theorem 8.15.

<switch/> decisions are given by parallel services with routing described in Theorem 8.18.

<flow/> tags indicating a fork/join structure are modeled with Theorem 8.19.

<while/> with a known upper bound v for visits is given by Theorem 8.22.

Example 8.5.2. *The ParcelSink workflow uses five service instances, in Figure 8.9 each of these services is represented by its SDP as given in Example 8.5.1. We mapped the structure of the BPEL file into the model and connected the SDPs to a network. The request arrival flow R is fed into a network topology similar to the Network Calculus or Queueing Networks example, the departure flow is R^*. Here a distinction in parallelization is made between for the synchronized* `<flow/>` *tag for the geocoders and the routing to the catalog service stated by a* `<switch/>` *statement. Again, the abstract model does not include any variables or logic to perform the routing, a parallel construct is used instead. For the repetition count of the* `printinvoice` *service $v = 2$ is used.*

A new addition is the second network for delay flows (dotted lines). The flows are joined by delay multiplexers for serial, synchronized and routing constructs. They result in flow $D(R, R^*)$ leaving the system at the bottom.

8.5.3 SLA Validation

SLAs for services are validated for their performance conformance by comparing them to given requirements. Hence, to validate the performance of service compositions the overall SLAs have to be derived. SLA Calculus supports this step with results for composing SDPs to a single SDP. During composition, the upper and lower arrival bounds to the composition become narrowed and can be compared to the customer's workload requirements. In detail, the bounds are valid if the customer's arrival process is conform to the overall arrival bounds by Definition 8.3.6.

Contrary to limits on the arrival process, delay flow bounds become wider during composition. The providers part of the SLA can be checked by comparing the upper and lower delay bounds using Definition 8.3.7. When the customer and provider part of the workflow SDP are conform to the requirement SDP the SLA is valid.

The steps for an SLA Calculus workflow analysis are implicitly given by Definitions 8.2.5 and 8.4.1. Best practice here are the following steps:

1. The curves α^U_{common} and α^L_{common} to the whole modeled workflow have to be computed first using the respective composition theorems.

2. Parallel compositions are reduced to an SDP each, then their service curves are computed.

3. Using α^U_{common}, the service curves and input contracts all output bounds $\alpha^{U'}$ for services with a successor are computed.

4. Finally, bounds on delay can be derived using the respective composition theorems again.

Example 8.5.3. *Using the SDPs from Example 8.5.1 we can derive the overall SDP $SLA_{ParcelSink}$ for the workflow in Figure 8.9. For SLA validation it is compared to*

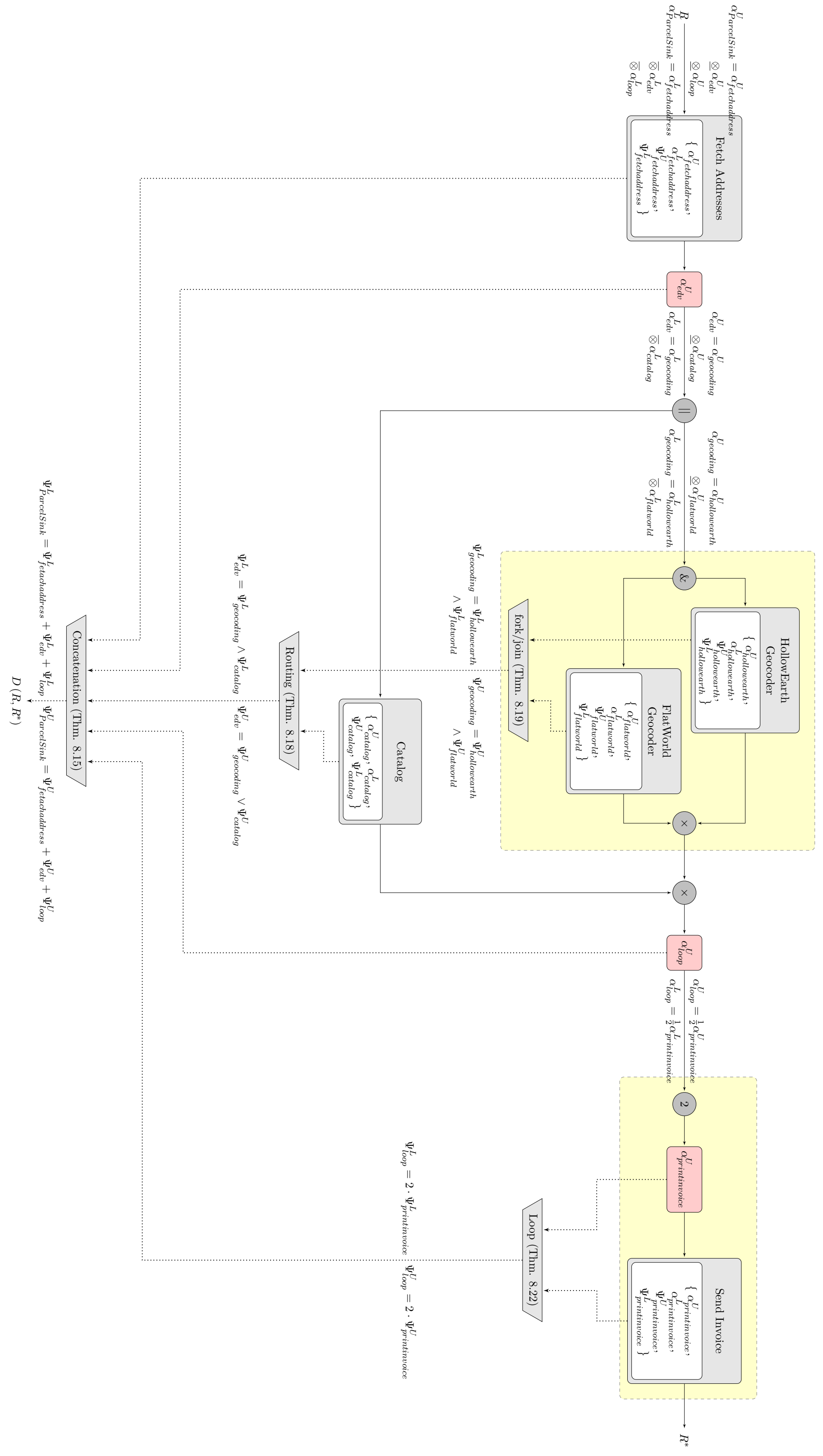

Fetch Addresses
HollowEarth Geocoder
FlatWorld Geocoder
Catalog
Send Invoice
fork/join (Thm. 8.19)
Routing (Thm. 8.18)
Concatenation (Thm. 8.15)
Loop (Thm. 8.22)
R
R'
$D(R, R')$
$\alpha^U_{ParcelSink} = \alpha^U_{fetchaddress} \otimes \alpha^U_{edc} \otimes \alpha^U_{loop}$
$\alpha^L_{ParcelSink} = \alpha^L_{fetchaddress} \otimes \alpha^L_{edc} \otimes \alpha^L_{loop}$
$\alpha^U_{edc} = \alpha^U_{geocoding} \otimes \alpha^U_{catalog}$
$\alpha^L_{edc} = \alpha^L_{geocoding} \otimes \alpha^L_{catalog}$
$\alpha^U_{geocoding} = \alpha^U_{hollowearth} \otimes \alpha^U_{flatworld}$
$\alpha^L_{geocoding} = \alpha^L_{hollowearth} \otimes \alpha^L_{flatworld}$
$\alpha^U_{loop} = \frac{1}{2}\alpha^U_{printinvoice}$
$\alpha^L_{loop} = \frac{1}{2}\alpha^L_{printinvoice}$
$\Psi^L_{geocoding} = \Psi^L_{hollowearth} \wedge \Psi^L_{flatworld}$
$\Psi^U_{geocoding} = \Psi^U_{hollowearth} \wedge \Psi^U_{flatworld}$
$\Psi^L_{edc} = \Psi^L_{geocoding} \wedge \Psi^L_{catalog}$
$\Psi^U_{edc} = \Psi^U_{geocoding} \vee \Psi^U_{catalog}$
$\Psi^L_{loop} = 2 \cdot \Psi^L_{printinvoice}$
$\Psi^U_{loop} = 2 \cdot \Psi^U_{printinvoice}$
$\Psi^L_{ParcelSink} = \Psi^L_{fetchaddress} + \Psi^L_{edc} + \Psi^L_{loop}$
$\Psi^U_{ParcelSink} = \Psi^U_{fetchaddress} + \Psi^U_{edc} + \Psi^U_{loop}$

Figure 8.9: ParcelSink example modeled with SLA Calculus

SLA_{Target}. Computations are done with MATLAB and the RTC Toolbox. In a first step we will derive the bounds without the reshaping delay, we will add it in a second step. We start with the fork/join construct for both geocoders:

$$SLA_{geocoder} = \left\{ \begin{array}{ll} \alpha^U & = \min(\gamma_{20,0}, \gamma_{15,10}, \gamma_{8,61}) \\ \alpha^L & = \max(\beta_{1,0}, \beta_{2,1}, \beta_{3,5}) \\ \Psi^U & = \begin{cases} \min(\gamma_{50,20}, \gamma_{5,112}) & t < 4.28 \\ \min(\gamma_{30,5}, \gamma_{10,165}) & t \geq 4.28 \end{cases} \\ \Psi^L & = \max(\beta_{3,0}, \beta_{4,3}) \end{array} \right\}$$

A plot of both initial SDPs and the resulting SDP with all four curves in each case is shown in Figure 8.10. A new upper bound with three segments is formed by convolution. Since the curves are sub-additive, the operation is equivalent to finding the point-wise minimum. Although the HollowEarth geocoding service accepts arrival bursts better than its competitor the overall burst capacity of the synchronized system is determined by FlatWorld. For the long-term capacity, the service rate of 8 of HollowEarth limits the system. The intersection point $t \approx 7.28$ between the second an third segment in $\alpha^U_{gecoding}$ marks the transition between bursts and long-term capacity. For the lower arrival bound $\alpha^L_{geocoding}$ the transition from phases with less than expected (modeled with two segments) to long-term minimum rate $r = 3$ is at $t = 13$. The maximum combination of upper delay curve shows results in a curve that is sub-additive, but obviously not concave anymore. At the intersection point of the second to the third segment ($t = 4.28$) the delay rate increases from 5 to 30 until it transits to the long-term delay guarantee of 10 ($t = 8$).

The alternative route using the internal address database is added with Theorem 8.18:

$$SLA_{edv} = \left\{ \begin{array}{ll} \alpha^U & = \min(\gamma_{20,0}, \gamma_{9,13}, \gamma_{8,61}) \\ \alpha^L & = \max(\beta_{1,0}, \beta_{2,1}, \beta_{3,5}) \\ \Psi^U & = \begin{cases} \min(\gamma_{50,20}, \gamma_{5,112}) & t < 4.28 \\ \min(\gamma_{30,5}, \gamma_{10,165}) & t \geq 4.28 \end{cases} \\ \Psi^L & = \max(\beta_{1,10}) \end{array} \right\} \tag{8.83}$$

Except of setting the lower delay curve to a rate of 1 the limited burst capacity of the internal database also reduces the arrival burst capacity of the subsystem.

Printing the invoice is done exactly two times, so the performance bounds in a loop are found using Theorem 8.22 and $v = 2$:

$$SLA_{loop} = \left\{ \begin{array}{ll} \alpha^U & = \min(\gamma_{12.5,3}) \\ \alpha^L & = \beta_{3,4} \\ \Psi^U & = \gamma_{32,6} \\ \Psi^L & = \beta_{1,8} \end{array} \right\} \tag{8.84}$$

The overall SDP is found by concatenation of $SLA_{fetchaddress}$ and SLA_{edv} cutting a small *"edge" from the arrival burst capacity and adding delay:*

$$SLA_{front} = \left\{ \begin{array}{ll} \alpha^U & = \min(\gamma_{20,0}, \gamma_{15,2}, \gamma_{9,13}, \gamma_{8,61}) \\ \alpha^L & = \max(\beta_{1,0}, \beta_{2,1}, \beta_{3,5}) \\ \Psi^U & = \begin{cases} \min(\gamma_{60,25}, \gamma_{15,117}) & t < 4.28 \\ \min(\gamma_{40,10}, \gamma_{20,170}) & t \geq 4.28 \end{cases} \\ \Psi^L & = \beta_{3,0} \end{array} \right\} \tag{8.85}$$

and finally by concatenation of SLA_{front} with SLA_{loop}:

$$SLA_{ParcelSink} = \left\{ \begin{array}{ll} \alpha^U_{ParcelSink} & = \min(\gamma_{20,0}, \gamma_{12.5,3}, \gamma_{9,13}, \gamma_{8,61}) \\ \alpha^L_{ParcelSink} & = \max(\beta_{1,0}, \beta_{2,1}, \beta_{3,4}) \\ \Psi^U_{ParcelSink} & = \varphi^U_{ParcelSink} + \begin{cases} \min(\gamma_{92,31}, \gamma_{47,123}) & t < 4.28 \\ \min(\gamma_{72,16}, \gamma_{52,176}) & t \geq 4.28 \end{cases} \\ \Psi^L_{ParcelSink} & = \max(\beta_{2,4}, \beta_{3,5.333}, \beta_{4,6.5}) \end{array} \right\} \tag{8.86}$$

Figure 8.11 shows both input SDPs and the result for ParcelSink. The overall guaranteed worst-case arrival capacity of the workflow is thus limited by a rate of 8. Arrival bursts are limited by a rate of 20, this is further decreased to 12.5 and 9. With α^L a minimum arrival rate of 3 is set, phases of lower activity are quite limited. The upper delay curve gives us a long-term delay rate of 52, for shorter periods even 92 is acceptable. Delay bursts up to a weight of 31 time units are not regulated at all. Noteworthy for Ψ^L is the interval of 4 time units with no arrivals at all.

Finally, when $\alpha^U_{ParcelSink}$ is known, the reshaping delay can be computed. Appropriate service curves have been given in Example 6.7.2 based on Theorem 8.10. The workflow is a serial composition of three major parts: the FetchAddress service, the routing and synchronization block in the middle and the looped PrintInvoice service. For the first passage between the first and second parts the burstiness increase for the arrival flow is

$$\alpha^{U'}_{fetchaddress} = \alpha^U_{ParcelSink} \underline{\oslash} \beta^L_{fetchaddress} \tag{8.87}$$

To be conform to the input bound of the second part the flow is reshaped to α^U_{edv} with a maximum delay of $\varphi^U_{fetchaddress} = h_{\max}\left(\alpha^{U'}_{fetchaddress}, \alpha^U_e dv\right) = 10s$. After reshaping and passing the second part the flow has the upper limit

$$\alpha^{U'}_{edv} = \left(\alpha^{U'}_{fetchaddress} \underline{\otimes} \alpha^U_{edv}\right) \underline{\oslash} \beta^L_{edv} \tag{8.88}$$

The maximum delay for reshaping to α^U_{loop} is $\varphi^U_{edv} = h_{\max}\left(\alpha^{U'}_{fetchaddress}, \alpha^U_{loop}\right) = 8.5556s$. Within the third part, the loop, and after the first iteration the bound is

$$\alpha^{U'}_{printinvoice} = \left(\alpha^{U'}_{edv} \underline{\otimes} \alpha^U_{loop}\right) \underline{\oslash} \beta^L_{printinvoice} \tag{8.89}$$

resulting in subsequent reshaping costs of $\varphi^U_{printinvoice} = h_{\max}\left(\alpha^{U'}_{printinvoice}, \alpha^U_{printinvoice}\right) = 16s$ to match the input bound for the second iteration. Thus, the necessary time for *reshaping the request flow in the composition sums up to 34.5556 seconds. As delay curve this expresses in*

$$\varphi^U_{fetchaddress} + \varphi^U_{edv} + \varphi^U_{printinvoice} = \varphi^U_{ParcelSink} = \lambda_{34.5556} \tag{8.90}$$

In Figure 8.11 the reshaping delay is plotted in blue. The workflow SDP can be completed:

$$SLA_{ParcelSink} = \left\{ \begin{array}{ll} \alpha^U_{ParcelSink} & = \min\left(\gamma_{20,0}, \gamma_{12.5,3}, \gamma_{9,13}, \gamma_{8,61}\right) \\ \alpha^L_{ParcelSink} & = \max\left(\beta_{1,0}, \beta_{2,1}, \beta_{3,4}\right) \\ \Psi^U_{ParcelSink} & = \begin{cases} \min\left(\gamma_{126.5556,31}, \gamma_{81.5556,123}\right) & t < 4.28 \\ \min\left(\gamma_{106.5556,16}, \gamma_{86.5556,176}\right) & t \geq 4.28 \end{cases} \\ \Psi^L_{ParcelSink} & = \max\left(\beta_{2,4}, \beta_{3,5.333}, \beta_{4,6.5}\right) \end{array} \right\} \tag{8.91}$$

By Definition 8.3.6 we can conclude that the workflow is conformant to the customer requirements.

$$b_{\max}\left(\alpha^U_{Target}, \alpha^U_{ParcelSink}\right) = 0 \tag{8.92}$$

On the other side, based on Definition 8.3.7 we can say that the workflow is (by far) not conformant from the perspective of the provider.

$$b_{\max}\left(\Psi^U_{Target}, \Psi^U_{ParcelSink}\right) = \infty \tag{8.93}$$

Still, there is the lower arrival and delay implication. Using Definition 8.3.2 we check for possible arrival contract breaches due to earliness in the service concatenation.

$$\max_t \left\{ h_{early}\left(\alpha^{L'}_{FetchAddress}, \alpha^L_{edv}\right)(t) \right\} = \infty \tag{8.94}$$

$$\max_t \left\{ h_{early}\left(\alpha^{L'}_{edv}, \alpha^L_{loop}\right)(t) \right\} = 6.0 \tag{8.95}$$

$$\max_t \left\{ h_{early}\left(\alpha^{L'}_{loop}, \alpha^L_{PrintInvoice}\right)(t) \right\} = 12.0 \tag{8.96}$$

Analysis shows that in every service transition there is a contract breach, Figure 8.12 shows the situation in detail. Therefore the service provider is not obliged to the lower delay bound, we carry the delay bound on as intrinsic delay only.

We did not take exact measurements on computation time, but the used *RTC Toolbox* interpreted in *MATLAB* delivered results in less than one second. So the analytic analysis of SLA Calculus models gives us performance figures within the same time magnitude as product-form Queueing Networks or Network Calculus do. The computation time of simulative analysis is undercut easily on same hardware (see Example 5.1.3 and Table D.3).

8.6 Discussion

Analysis with SLA Calculus provides us a distinct perspective on systems compared to previously presented SOA modeling methods. We used performance bounds from SLAs for simple services as input and received bounds for the worst-case characterization of the complete workflow. Thus, analysis has a different quality in result and it has to be

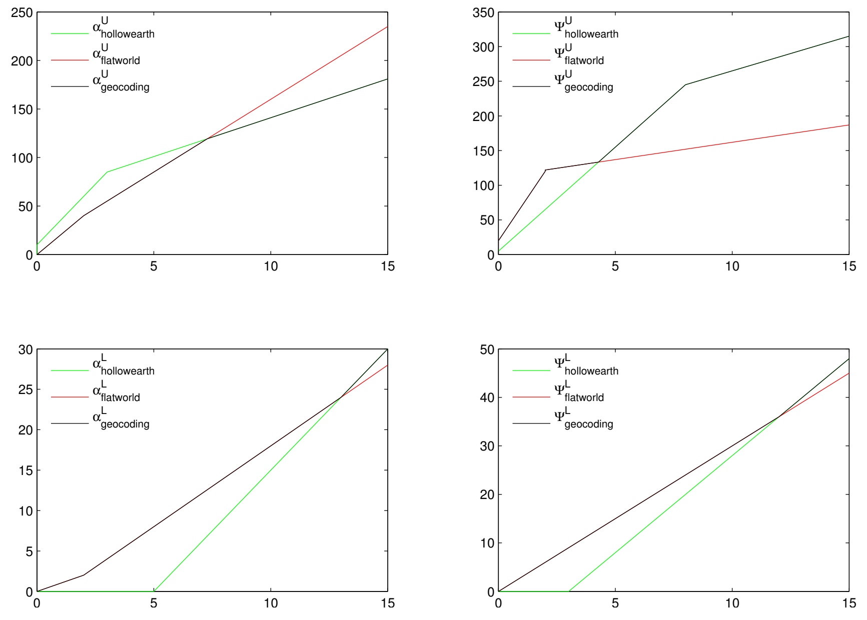


Figure 8.10: Plot of SDP for synchronized geocoders. α^U on the top left, Ψ^U on the top right and the lower curves in the bottom row.

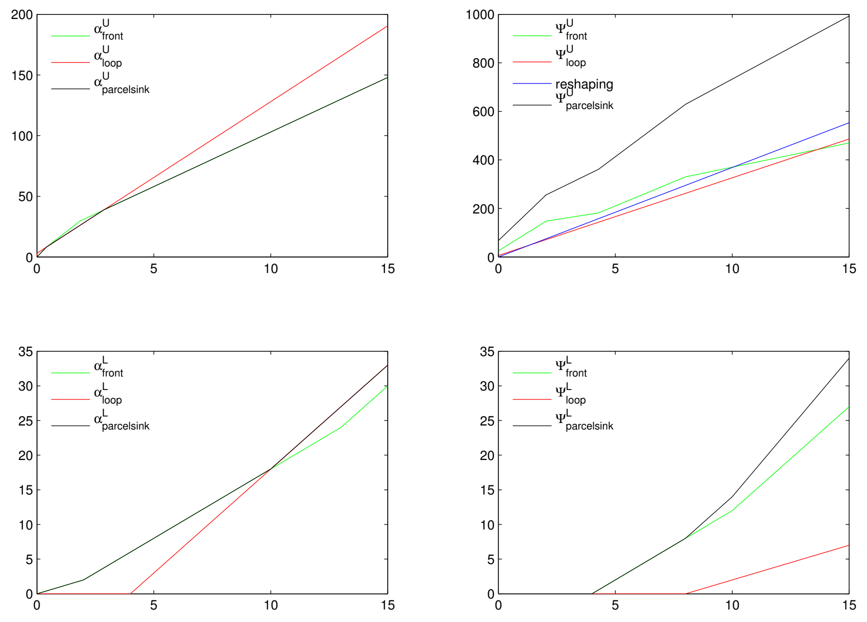


Figure 8.11: Plots for SLA_{front}, SLA_{loop} and $SLA_{ParcelSink}$.

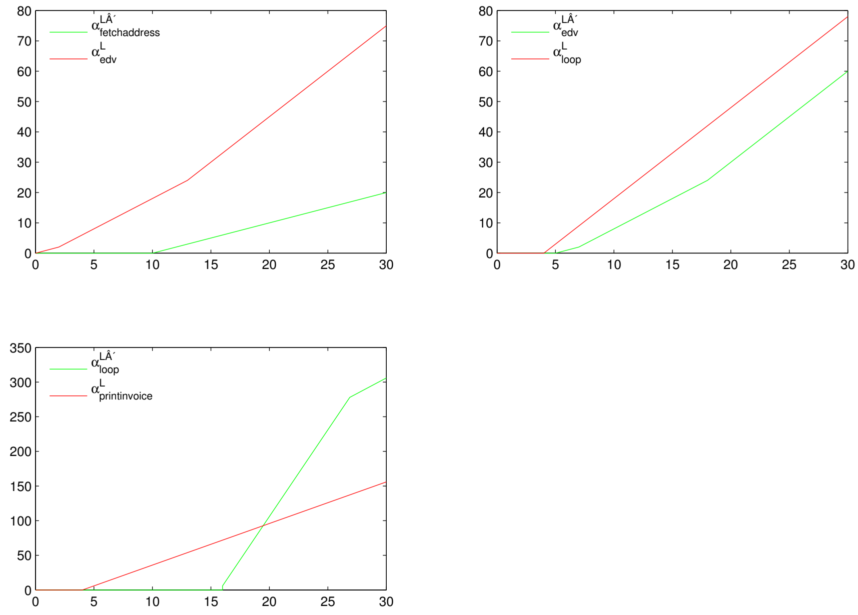


Figure 8.12: Lower departure envelopes show earliness towards subsequent services.

discussed, if SLA Calculus is a better SOA modeling method than Queueing Networks, Simulation or even Network Calculus. For the sketched SLA validation scenario we make the statement that it is definitively a well-suited approach.

With the secrecy policies in SOA eco-systems a modeling has to work frugally with information. SLA Calculus models have low information requirements, services are considered as black-box, system internals on scheduling, storage and especially available processing resources are not required. It is also irrelevant if the response time reacts linear to increasing workload. The SLA Calculus model with SDPs as service abstraction is simple and allows patterns and hierarchies. Therefore they are ideal to analyze system drafts when a simulative approach would bring overhead only. With just a small information set we were able to set up our exemplary analysis and, by modeling the worst-case, omit many details.

Since SLAs are available to all participants in an SOA eco-system, SLA Calculus suites both service customers and providers. Both sides can create workflow models, analyze them and validate SLAs for conformance. Independence on knowledge of service curves is SLA Calculus's advantage to pure Network Calculus. For our ParcelSink model we just used the readily available information in SLAs, no further derivation of service rates was necessary.

8.6.1 Deterministic SLA Validation

The strength of SLA Calculus is the deterministic but detailed modeling approach on workload and delay. Analysis gives upper and lower performance bounds without stochastic components. Although these bounds are wide in contrast to stochastic modeling analysis methods, they are reliable. If a system fullfills an SLA in any case or not, the decision for an SLA validation can be made in this way. Queueing Networks at least with efficient analysis, and simulation, can draw a better picture of average behavior, but result figures like delay are potentially unbounded. Validation results will always include some limited, but existing uncertainty. This is worsened by the process model of shown stochastic methods: The modeler asks for limits yet the models accept average rates only. In previous instances of the ParcelSink example we have seen that this may lead to false validation results. It has to be noted that SLA Calculus assumes valid guarantees for input and stochastic process models and may be more robust to errors.

Furthermore, bounds are given for workload and delay in a single analysis and even short-term results are found. In our example we instantly see the workload one can send to the ParcelSink workflow without overloading at least one participating service, it is bound by $\alpha^U_{ParcelSink}$. Even more, we additionally can read from the first line segments in $\alpha^U_{ParcelSink}$ that the long-term rate can be superseded for a short time. This allows the selection of services with a certain long-term rate and some spare capacity, instead of using fast (and expensive) providers, that allow continuous processing with the burst rate. In the above example, an arrival burst may have a duration of 48 time units and, for a short period of 0.4 seconds, a rate up to 20 is possible. With product-form Queueing Networks this result is unachievable, simulative analysis would require detailed statistics.

The claim to fame for SLA Calculus is the detailed delay model with upper and lower bounds. This is an advantage to pure Network Calculus, and in terms of delay limits, to Queueing Networks. When we include the analysis time as a factor, simulation falls behind, too. For our example, the analysis gave us the long-term delay rate under worst-case conditions by $\Psi^U_{ParcelSink}$ without any lengthy computation. The delay rate is rounded 86.6 time units which draws a bad picture on the workflow performance compared to the average results in Queueing Networks. However, one has to be aware that this is a worst-case figure and a service request will seldom take so much time. The modeler receives an analysis result based on service guarantees which allows him or her to judge whether this workflow is suitable for his own needs. In any case the limits are strict, time-invariant and not annotated with confidence intervals.

Even more interesting analysis results are the delay burst characteristics which show a delay flow is limited in short phases of lower or zero processing capacities. SLA Calculus analysis also gives these latency bounds for the short-term in a single run. We gain more insight into response time characteristics of systems compared to the maximum in Network Calculus or average response time in Queueing Networks. For the last one the transient analysis is partially substituted. Limited short-term variations in delay are known beforehand and can be integrated into further system capacity planning. A noteworthy detail in this context is the "dent" in $\Psi^U_{ParcelSink}$ having a line segment with a slower delay rate ($r \approx 81.6$) than the long-term rate. This is not an error, but the guidelines on non-concave curves in Section 8.4.5 have to be considered. They show how the system recovers after a delay burst. In interval $[0, 4.28)$ the burst is limited to rate $r \approx 126.6$, $t = 4.28$ marks the preliminary end of the request processing slowdown switching to a phase of faster response times. With SLA Calculus we did not only receive a deterministic upper bound for the long-term delay, we also got a detailed description on how the workflow reacts to fluctuations in processing or arrival bursts.

By deriving the overall SDP for a system we receive lower process bounds. Awareness of lower bounds is not given in product-form Queueing Networks, partially in Real-

Time Calculus and elaborative in simulation. Curves $\alpha^L_{ParcelSink}$ and $\Psi^L_{ParcelSink}$ are not symmetric to the upper ones, hence they include additional information. The arrival curve marks the minimum request flow accepted by the system. It can support service selection or enable the implication for lower response time contracts. However, such an implication for lower envelopes is dependent on the service ordering and fragile as we can see in the example. For the ParcelSink workflow (and in general) the results are still valuable: The minimum arrival curve ($\alpha^L \neq \beta_{0,0}$) as well as the minimum delay curve are not trivial. When analysis results in a curve with zero rate, the trivial lower arrival bound, the non-zero minimum delay shows the intrinsic delay of the service.

8.6.2 SLA Calculus has its Justification

There is still one question unanswered: When service curves can be computed easily from SDPs, why not use Network Calculus for SLA validation in general? This already worked in Chapter 6 for the ParcelSink workflow. Our criticism on this approach is that many detailed aspects of arrival and delay contracts including tolerances get lost on conversion. Network Calculus gives “end-to-end” network guarantees, any internals or even the ordering of system components are neglected. With SLA Calculus burstiness and lagginess of request flows between single services becomes visible to the modeler. To guarantee workload contracts within service compositions additional measures are necessary. SLA Calculus analysis exposes the costs of in-lined shapers in terms of response times. This is only possible due to the detailed awareness of burstiness increases at internal flows. As for the other performance figures, this extra time is the most pessimistic assumption and requires a classification in the model context.

For product-form Queueing Networks, limited to Poisson processes without variance information, this result is unreachable. Simulation models can be implemented in a way to log internal arrival contract violations. The exemplary *ProC/B* language and the shown SLA validation process offers enough descriptiveness. Still, not all contract breaches may be detected and the reshaping delay cannot be bounded. In any case, simulative analysis stays lengthy. Here, this SLA Calculus analysis part is as efficient as other Network Calculus based computations.

9 Service Curve Computation

On the one hand, it is hard to give reliable performance guarantees for SOAs with Queueing Systems, but for long-term results no information on service processes is required. The average processing rate of a service can be computed easily with performance figures provided by its SLA. On the other hand Network Calculus can provide deterministic worst-case guarantees, but is depending on service rates and their bounding curves. SLA Calculus gives worst-case bounds on delays without knowledge of the service process in SOAs, it is designed to model systems just by combination of other performance bounds. But due to the need for internal reshaping SLA Calculus analysis is dependent on service curves, too.

Furthermore, knowledge of the lower envelope of the service process in a model is helpful when results are implemented in a real world system. Therefore, computation of service curves for models can be a subsequent step to SLA Calculus SOA model analysis. This can be the case in system capacity planning (c.f. Section 2.4), when a known SDP is used to size the required real world computing infrastructure. Computations of lower service curves give the minimum required system performance to fulfill service guarantees. The reverse problem of capacity planning is to validate if a modeled workflow is executable in time with an existing system. Computed service curves can be compared to already known service curves and one can decide if the system will perform adequately. Another scenario that involves service curve computation is to switch from SLA Calculus to pure Network Calculus modeling and analysis. In many scenarios apart from SOA performance modeling, Network Calculus is a more mature and general modeling method than SLA Calculus, which evolved in two decades of research. An argument derived from the popularity of Network Calculus is that extensive software support exists [99, 110, 123]. Knowledge of service curves for SOA models gives the modeler access to these tools.

In [117] a numerical method for service curve computation is presented. It does give a rate-latency service curve and involves an optimizer. However, we have seen that in Queueing Systems the service rate μ can be easily computed by basic laws for Queueing Systems. In Example 4.2.1 the average service rate μ is found by combining Little's Law and Equation 4.7 for the average system population:

$$\mu = \lambda + \frac{1}{E[S]} \tag{9.1}$$

Obviously, this is not an optimization problem and can be solved with simple algebra. Since Network Calculus and SLA Calculus are purely analytic methods one should find a comparable capability here. In this chapter a similar relationship for (min,+)-based algebras is described [119]. It generalizes the derivation of service curves and arbitrary input curves are accepted. Due to its analytic foundation it is independent of the optimizer.

9.1 Problem Description

The problem of service curve computation in SLA Calculus is to find the minimal lower service curve a system has to provide in order to conform to a given SDP. Thus, with given curves α^U and Ψ^U a lower envelope β^L on the needed resource flow is calculated. The SDP implicates the system to limit the delay flow D as long as the arrival contract is valid. This has to hold even for the extreme case $R = \alpha^U$ with a worst-case delay flow of $D(R, R^*) = \Psi^U$.

In the SLA Calculus system model a delay flow $D(R, R^*)$ is given by the function of horizontal distance between arrival R and departure flow R^* (Definition 7.1.1). Flow R is limited by α^U and $D(R, R^*)$ by Ψ^U, two curves given by an SDP. Unknown is the departure flow R^*, but, under the condition that the system is stable, there has to be a minimum departure flow that results in maximum but conform delay flow. Recalling Definition 6.2.11 we know that for arrival flow R and lower service curve β^L

$$R^* \geq R \underline{\otimes} \beta^L \tag{9.2}$$

gives a lower bound for the departure flow R^*. When only bounds in form of an SDP $SP = \{\alpha^U, \Psi^U\}$ are at hand the actual input flow R is not available. However, to create a worst-case situation one can replace R in (9.2) with upper bound α^U. Using Definition 7.1.3 the service curve computation problem is then to find the minimum lower service curve β^L with

$$\beta^L = \min\left\{\beta \in \mathcal{F}_0 : D\left(\alpha^U, \alpha^U \underline{\otimes} \beta\right) \leq \Psi^U \underline{\otimes} D\left(\alpha^U, \alpha^U \underline{\otimes} \beta\right)\right\} \tag{9.3}$$

Before a solution is presented the consequences of optimal, approximative and or even wrong computation of β^L are discussed. Let SP be an SDP and β_C be the computed service curve, β_T is the theoretical minimal solution.

$\beta_C = \beta_T$ The curve is optimal. In this case we have $\Psi^U \geq D\left(\alpha^U, \alpha^U \underline{\otimes} \beta_C\right)$ and no better solution than β_C exist. For most situations, unless SP is limited to linear curves, usually an optimal solution $\beta_C = \beta_T$ is not achievable. Delay flows are formed by (horizontal) integration, therefore $D\left(\alpha^U, \alpha^U \underline{\otimes} \beta_C\right)(t)$ is continuous and a derivative exists for all t. Piecewise linear functions with more than one segment are naturally not continuous. Thus the identity between Ψ^U and $D\left(\alpha^U, \alpha^U \underline{\otimes} \beta_C\right)(t)$ as a function of higher degree is usually not achievable and a non-optimal solution has to be accepted. Figure 9.1 shows how the delay curve is approximated by the resulting delay flow.

$\beta_C > \beta_T$ This leads to $D\left(\alpha^U, \alpha^U \underline{\otimes} \beta_C\right) < \Psi^U \underline{\otimes} D\left(\alpha^U, \alpha^U \underline{\otimes} \beta_C\right)$. The curve is not the minimum solution and belongs to a faster system than SP requires. In practice, the system would hold more processing power than necessary. This becomes visible when comparing Figure 9.2 to Figure 9.1. Processing is faster than necessary and delay flow D keeps away from its bound Ψ^U.

An invalid solution is $\beta_C < \beta_T$. A system implementing service envelope β_C is not conform to SP as $D(\alpha^U, \alpha^U \underline{\otimes} \beta_C) > \Psi^U \underline{\otimes} D(\alpha^U, \alpha^U \underline{\otimes} \beta_C)$. Arrivals are delayed too long, hence D exceeds Ψ^U by far.

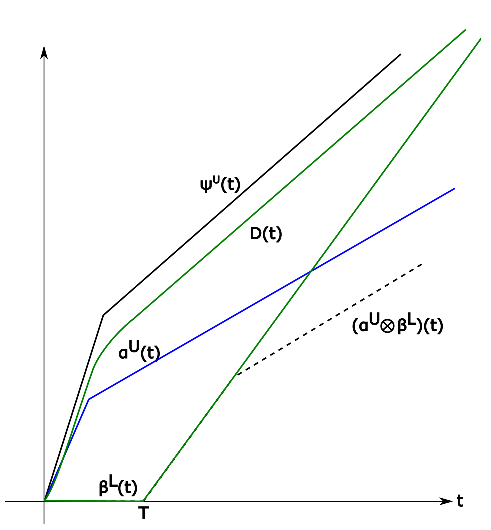


Figure 9.1: Good service curve estimation: Resulting delays are conform to Ψ^U.

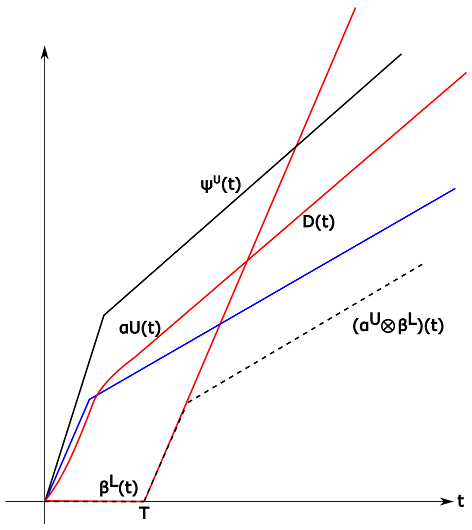


Figure 9.2: Bad estimation: Rate in β^L is too high, Ψ^U is not exhausted.

9.2 Revisiting (min,+)-Deconvolution

Equation (9.1) gives the service rate for known arrival rates and latencies in Queueing Systems. In SLA Calculus models information on these figures is available in form of upper bounds. By Definition 6.2.13 the departure flow of a server system is limited from below by

$$R^* \geq R \underline{\otimes} \beta^L \tag{9.4}$$

We define α^* as the lowest departure flow when the arrival flow is maximized, thus

$$\alpha^* = \alpha^U \underline{\otimes} \beta^L \tag{9.5}$$

For a moment let us assume that α^* is known or can be computed from the given data. Then the idea to compute the service curve can be sketched by using an inverse operation to (min,+)-convolution in order to unfold the known arrival bounds out of departure bounds. Thus, an operator $\underline{\otimes}^{-1}$ should exist with

$$\beta^L = \alpha^* \underline{\otimes}^{-1} \alpha^U \tag{9.6}$$

to represent the service curve.

The dual counterpart to (min,+)-convolution is (min,+)-deconvolution (c.f. Definition 6.1.7). Although (min,+)-deconvolution is not the inverse operation to (min,+)-convolution in general [76, 107], it can be shown for worst-case conditions in SLA Calculus that it gives similar results to an inverse operation. With deconvolution not being the inverse to convolution [76, 107] there is

$$f \neq (f \underline{\otimes} g) \underline{\oslash} g \tag{9.7}$$

for $f, g \in \mathcal{F}$. The bad news here is that for $f = \beta^L$ and $g = \alpha^U$ one cannot completely reconstruct service curves with the information encoded in the departure bound $\alpha^U \underline{\otimes} \beta^L$. However, a result in bandwidth estimation can be used to compute the service curve with deconvolution: In Network Calculus models, service curves are descriptions for achievable bandwidths of server elements. When a model for a server should be based on in- and output measurements the appropriate service curve has to be estimated. An estimation approach for lower service curves based on (min,+)-deconvolution is used in Liebeherr et al. [75, 76]. They show that deconvolution gives sufficient estimates for lower service curves when applied to input/output functions.

Estimation goal in [76] was to deduce the necessary resource flow C for measured arrival and departure flows, such that $R^*(t) \geq (R \underline{\otimes} C)(t)$ holds for all pairs of R and R^* and for all t. Since Equation 9.7 neglects the reconstruction of the exact resource flow C an approximation $\tilde{C}$ is introduced. Deconvolution gives [76]

$$\tilde{C} = R^* \underline{\oslash} R \tag{9.8}$$

with

$$\tilde{C} \leq C \tag{9.9}$$

The approximated resource flow $\tilde{C}$ has the property

$$R^* = R \underline{\otimes} \tilde{C} \tag{9.10}$$

So $\tilde{C}$ is a service curve reconstructed from the departure curve that can replace the unknown curve C.

The deconvolution approach gives valid results only if the system is linear and time invariant. Thus the output is linear to the input load and the estimated service curve is the impulse response of the server. For practical reasons network elements have a nonlinear input/output behavior, therefrom the deconvolution approach has limited use in [76]. Instead the authors present methods for measuring the service curve of network elements by stimulating the system with different load levels. Bandwidth estimation in [76] is based on traces of real systems with varying, not necessarily worst-case system load.

It is not specified if the system is linear or not with SDPs describing a system in SLA Calculus. Arrival bounds implicating a delay bound are descriptions for the worst-case, this single observation point is only important for service guarantees. There are no measurements of input and output streams comparable to [76], there are only SDPs $\{\alpha^U, \Psi^U\}$ they conform to. Neglecting the system behavior at average load levels, a special case can be constructed where system linearity is irrelevant. For this worst-case situation, deconvolution can be applied as an inverse operator to compute a service curve based on an SDP with pure algebraic methods.

9.3 Solution Steps

With deconvolution an algebraic tool is found, but still some additional preparations for service curve computation are required. Given an SDP $SP = \{\alpha^U, \Psi^U\}$ two steps and an optional third one have to be performed.

1. Find a maximum output function α^* with $\Psi^U(t) \geq D(\alpha^U, \alpha^*)(t)$.
2. Use (min,+)-deconvolution to derive a lower service curve: $\beta^L = \alpha^* \underline{\oslash}\, \alpha^U$.
3. Replace β^L with its super-additive closure

While the second and third steps are simple the first one causes some difficulties.

9.3.1 Departure Flow from SLA Delay Properties

A maximum departure flow α^* can be constructed by the information given by upper arrival and delay bounds in an SDP. This can be done by shifting the arrival curve by the maximum allowed delay to the right. Based on Definition 7.1.4 and Theorem 7.2 the delay in t is:

$$h\left(\alpha^U, \alpha^*\right)(t) = \frac{\mathrm{d}}{\mathrm{d}t} D(t) \tag{9.11}$$

Departure bound α^* is unknown, but can be derived in return by shifting $\alpha^U(t)$ by the horizontal distance for all t. We also know that D is limited by Ψ^U, for the worst-case situation the flow is exchanged by the curve. Figure 9.3 illustrates the idea. For the figure and the following equations, operator $\frac{\mathrm{d}}{\mathrm{d}t} f(t) = \mathrm{rt}(f, t)$ gives the derivation of f in t.

This approach is quite simple, but does only work for delay bounds given by a linear function. Delay curves including more than one line segment with pair-wise different rates are non-continuous in intersection points. With the sudden change of the derivative in these points the shift width also changes. Figure 9.4 includes a situation with a delay curve Ψ^U with two line segments Ψ_1 and Ψ_2. Since $\mathrm{rt}(\Psi_1) \neq \mathrm{rt}(\Psi_2)$ a jump is introduced in the departure flow. These gaps will introduce an error into the service curve computation. Derived curves will be valid, but not optimal. Sizing and limiting the error is subject to future research.

Theorem 9.1 (Departure Flow from Upper Bounds). *Let $SP = \left\{\alpha^U, \Psi^U\right\}$ be an SDP, α^U and Ψ^U are concave. A valid output flow $B(t)$ with $\Psi^U \geq D\left(\alpha^U, B\right)$ is given by*

$$B(t) = \alpha^U(t - \mathrm{rt}(\Psi^U, t)) \tag{9.12}$$

Proof. To prove that $B(t)$ is an outgoing arrival flow which results from a system that complies with SP, one has to show that

$$D\left(\alpha^U, B\right)(t) = \int_0^t h\left(\alpha^U, B\right)(x)\,\mathrm{d}x \leq \Psi^U(t) \text{ for all } t \geq 0 \tag{9.13}$$

Deriving both sides of (9.13) we get

$$h\left(\alpha^U, B\right)(t) \leq \mathrm{rt}(\Psi^U, t) \tag{9.14}$$

Let $\Delta = \tau + t$ for $\tau > 0$. Arrival curve α^U is wide-sense increasing, so $\alpha^U(\Delta) \geq \alpha^U(t)$ holds. Delay curve Ψ^U is required to be concave, so for the delay rate we have $\mathrm{rt}(\Psi^U, t) \leq \mathrm{rt}(\Psi^U, \Delta)$.

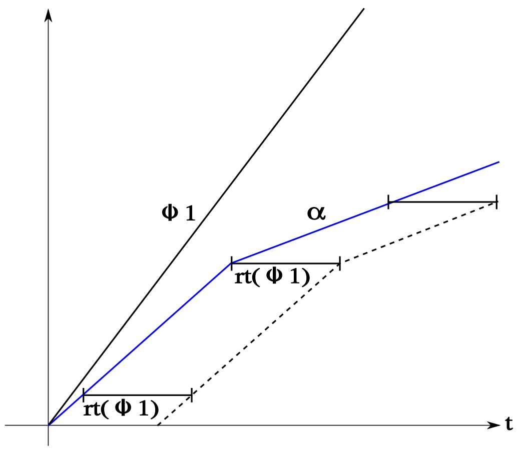


Figure 9.3: Departure flow for a linear delay curve

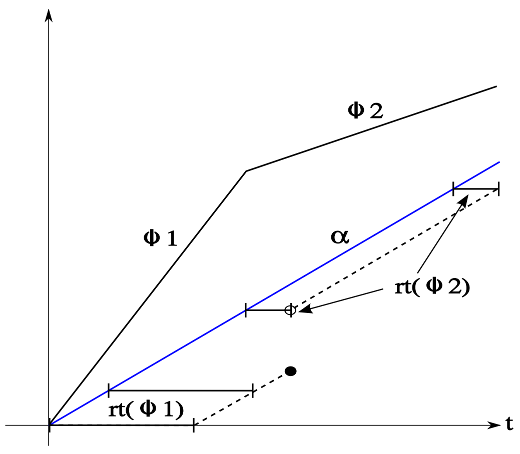


Figure 9.4: Departure flow for a delay curve with discontinuities.

First we consider the case of a constant delay rate $\mathrm{rt}(\Psi^U, t) = \mathrm{rt}(\Psi^U, \Delta)$.

$$\begin{aligned}
\mathrm{rt}(\Psi^U, t) &\geq d\left(\alpha^U, B\right)(t) && (9.15)\\
&= (B)^{-1}(\alpha^U(t)) - (\alpha^U)^{-1}(\alpha^U(t)) && (9.16)\\
&= (\alpha^U)^{-1}(\alpha^U(t)) + \mathrm{rt}(\Psi^U, \Delta) - (\alpha^U)^{-1}(\alpha^U(t)) \qquad \textit{Lemma } 7.3 && (9.17)\\
&= \mathrm{rt}(\Psi^U, \Delta) && (9.18)
\end{aligned}$$

Now follows the case when the rate decreases: $\mathrm{rt}(\Psi^U, t) > \mathrm{rt}(\Psi^U, \Delta)$. Let r^* be the value of $\mathrm{rt}(\Psi^U, \Delta)$ with $r^* = \mathrm{rt}(\Psi^U, \Delta) < \mathrm{rt}(\Psi^U, t)$, thus $B(t) = \alpha^U(t - r^*)$.

$$\begin{aligned}
\mathrm{rt}(\Psi^U, t) &\geq d\left(\alpha^U, B\right)(t) && (9.19)\\
&= (B)^{-1}(\alpha(t)) - (\alpha^U)^{-1}(\alpha^U(t)) && (9.20)\\
&= (\alpha^U)^{-1}(\alpha^U(t)) + r^* - (\alpha^U)^{-1}(\alpha^U(t)) \qquad \textit{Lemma } 7.3 && (9.21)\\
&= r^* && (9.22)
\end{aligned}$$

□

The first solution step can be done using Theorem 9.1 and gives us $B = \alpha^*$. The second step is supported by Theorem 8.10 that can be proofed by application of results.

Proof of Theorem 8.10. We apply the results of [76], summarized in Equation 9.8, and replace flows with worst-case bounds. For the workload to the system we set $R = \alpha^U$. A minimum departure flow $\alpha^* = R^*$ under consideration of maximum delay contract Ψ^U is given by Theorem 9.1. Since Equation 9.8 provides the minimum service flow to achieve the output (Equation 9.10) we set $\tilde{C} = \beta^L$. □

The service curve resulting from deconvolution in the second step is not necessarily convex. In general, this is not a restriction for service curves, but when a non-convex curve is used in further computations not all line segments will contribute. This redundant information can be removed from β^L when the curve is replaced by its super-additive closure, thus $\beta^L = \underline{\beta^L}$ [72].

Example 9.3.1. *For the ParcelSink workflow there have been five SDPs given for single services in Example 8.5.1. The necessary service capacities are to be computed with the approach shown above. This can be a step in sizing the actual hardware by getting the minimal service rate and, due to latency information in service curves, the maximum startup or maintenance time. We already used the results in Example 6.7.2 for Network Calculus ahead of time. Actual computation was done with MATLAB and the RTC Toolbox, additionally we added an implementation for the rate function rt().*

For the externally hosted services the service curves are

$$\begin{aligned}\beta^L_{HollowEarth}(t) &= \alpha^U_{HollowEarth}(t - \mathrm{rt}(\Psi^U_{HollowEarth}, t)) \underline{\oslash} \alpha^U_{HollowEarth}(t)\\ &= \begin{cases}\max(\beta_{8,10}(t), \beta_{25,13.4}(t)) & t < 18\\ \beta_{8,2.375}(t) & t \geq 18\end{cases}\end{aligned} \tag{9.23}$$

and

$$\begin{aligned}\beta^L_{FlatWorld}(t) &= \alpha^U_{FlatWorld}(t - \mathrm{rt}(\Psi^U_{FlatWorld}, t)) \underline{\oslash} \alpha^U_{FlatWorld}(t)\\ &= \begin{cases}\beta_{20,5}(t) & t < 7\\ \beta_{15,4.33}(t) & t \geq 7\end{cases}\end{aligned} \tag{9.24}$$

Figures 9.5 and 9.6 show the input, the service curves as well as the departure flow used in the deconvolution step.

For the remaining services the lower service curves are given by

$$\begin{aligned}\beta^L_{Catalog}(t) &= \alpha^U_{Catalog}(t - \mathrm{rt}(\Psi^U_{Catalog}, t)) \underline{\oslash} \Psi^U_{Catalog}(t)\\ &= \begin{cases}0 & t < 2\\ \beta_{12,1.1667}(t) & t < 3\\ \beta_{9,0.556}(t) & t \geq 3\end{cases}\end{aligned} \tag{9.25}$$

$$\begin{aligned}\beta^L_{FetchAddress} &= \alpha^U_{FetchAddress}(t - \mathrm{rt}(\Psi^U_{FetchAddress}, t)) \underline{\oslash} \alpha^U_{FetchAddress}(t)\\ &= \beta_{15,9.8667}(t)\end{aligned} \tag{9.26}$$

$$\begin{aligned}\beta^L_{PrintInvoice} &= \alpha^U_{PrintInvoice}(t - \mathrm{rt}(\Psi^U_{PrintInvoice}, t)) \underline{\oslash} \alpha^U_{PrintInvoice}(t)\\ &= \beta_{25,15.76}(t)\end{aligned} \tag{9.27}$$

To verify the results we have to show that a system $i \in \{HollowEarth, FlatWorld, \dots\}$ implementing the computed service curve β^L_i does not violate Ψ^U_i. Therefore we compute departure flow $R^*_i = \alpha^U_i \underline{\otimes} \beta^L_i$ and delay flow $D_i = D\left(\alpha^U_i, R^*_i\right)$. The service curve is suitable if for the resulting delay flow

$$D_i \leq D_i \underline{\otimes} \Psi^U_i \tag{9.28}$$

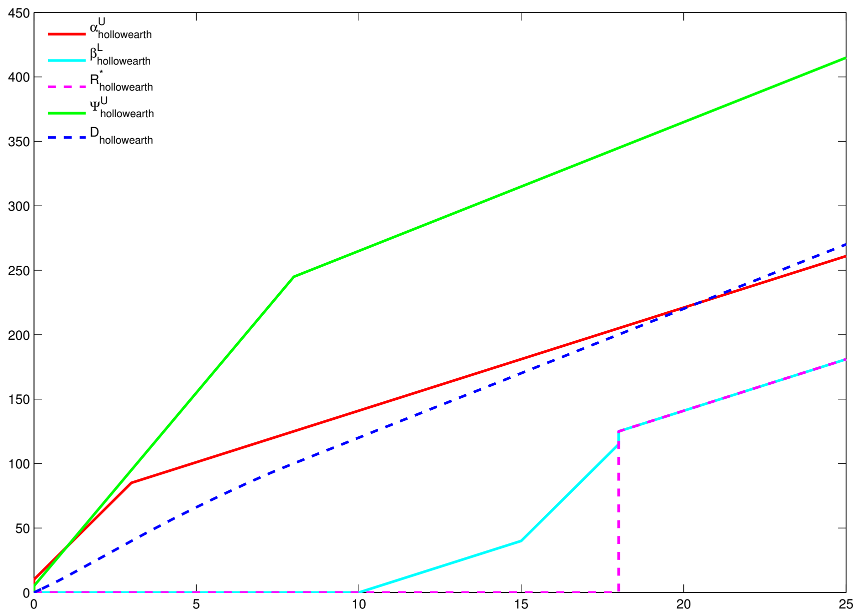


Figure 9.5: HollowEarth service curve computation and verification

holds. This procedure is implemented using *MATLAB* and the *RTC Toolbox*, too. Figures 9.5, 9.6 and 9.7 also include the used curves and flows. The delay flows do not violate the delay contracts and show that the computed results are valid.

Maybe the most noteworthy results are related to the HollowEarth service (Figure 9.5). The computed departure flow R^* includes a step at $t = 18$ influenced by the change of the delay rate at $t = 8$. While the initial delay rate shifts the arrival flow 25 time units to the right, the second delay rate gives a shift of 10 only and starts a line segment at $t = 18$. To avoid ambiguous departure flows the conservative second segment is used. The computed service curve for the HollowEarth service describes a maximum time interval of 10 units with no processing and a rate of 8.0 on the long-term. Due to the rate increase in interval $[10, 18]$ $\beta^L_{HollowEarth}$ is not convex, but it is still super-additive since $\beta^L_{HollowEarth} = \underline{\beta^L_{HollowEarth}}$ [72]. The short-lived rate increase can be interpreted as additional processing capacity to recover from phases with no or reduced processing. We also note that deconvolution gave us the minimum stable system regarding long-term rates.

When the delay flow $D_{HollowEarth}$ is computed for verification one can see that it has an equal rate to the delay curve on the long-term, thus

$$\frac{\mathrm{d}}{\mathrm{d}t}D(t) = \frac{\mathrm{d}}{\mathrm{d}t}\Psi^U_{HollowEarth} = 10 \text{ for } t \to \infty$$

On the short-term the maximum visible rate is found at

$$\max_t \left(\frac{\mathrm{d}}{\mathrm{d}t}D_{HollowEarth}(t)\right) = 13.8$$

indicating the maximum delay for this service curve. Still, the burst capacity offered by $\Psi^U_{HollowEarth}$, as visible in Figure 9.5, is not fully used. Here we can give two reasons for

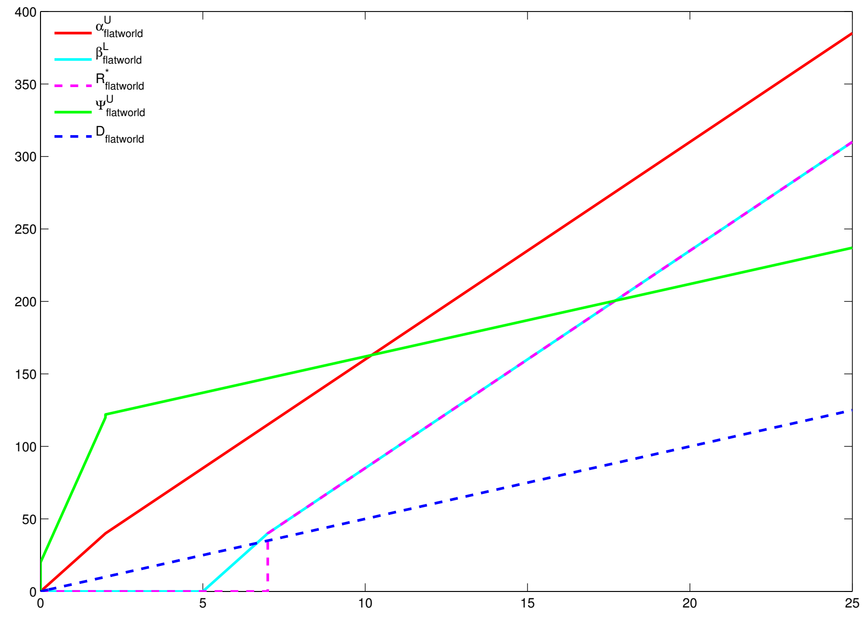


Figure 9.6: Flatworld service curve computation and verification

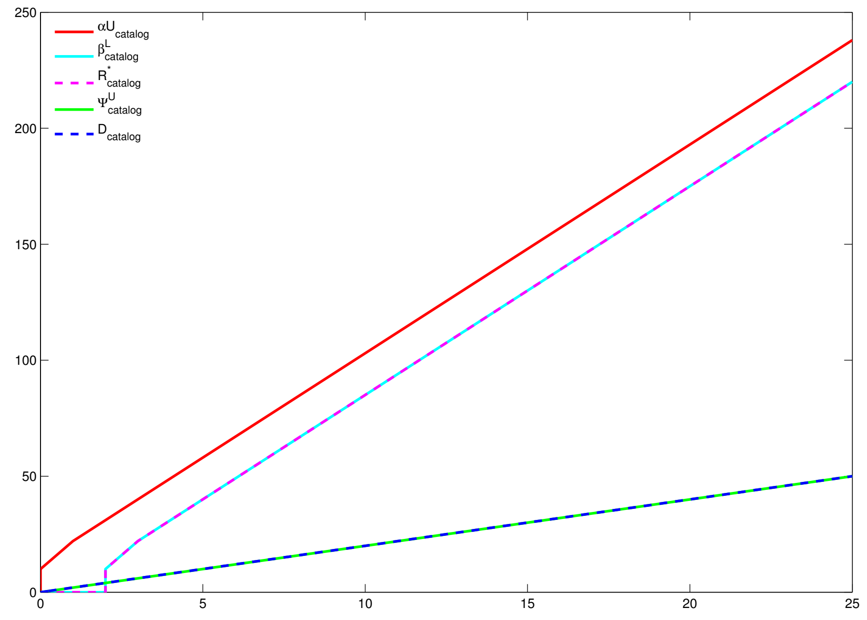


Figure 9.7: Catalog service curve computation and verification

this result: First, no identity between the non-continuous delay curve contract and the flow can be achieved since the flow is a function from higher degree (c.f. Section 9.1). Remember, the delay curve is combined of affine functions due to their burst-rate semantics and not for their properties in integral calculus. Second, when the intermediate departure flow for service curve computation is constructed, the information on a high delay rate of 25 is dropped in this special case in favour of the long-term rate.

The results for the FlatWorld service are found under comparable conditions. Service curve $\beta^L_{FlatWorld}$ describes zero processing up to 5 time units and a long-term rate of 15. Again the curve is not convex, but super-additive and gives a stable system. During the construction of the intermediate delay flow the delay burst capacity described in $\Psi^U_{FlatWorld}$ is completely lost and not included in $\beta^L_{FlatWorld}$. As a result, when the system is verified, the delay flow has a constant rate of 5. A constant delay curve contract $\Psi^U_{Catalog} = \gamma_{2,0}$ was specified for the catalog component. This results in a service curve being a shifted arrival curve, thus

$$\beta^L_{Catalog}(t) = \max\left(0, \alpha^U_{Catalog}(t-2)\right)$$

In this special case we also notice the identity of $D_{FlatWorld}$ and $\Psi^U_{FlatWorld}$ due to the constant rates.

10 Curve Estimation

With SLA Calculus one is able to determine theoretical worst-case performance bounds of systems with components characterized by SLAs. We already applied this in examples like the ParcelSink Web Services with given and synthetic SDPs. In other modeling situations the SLAs are unknown. Curve contracts have to be filled with parameters first according to the reactions and behavior of real or planned services. Model fitting problems are common to all modeling methods, therefore various approaches exist. For instance, rates in Queueing System are determined from traces by averaging performance figures.

A complete modeling method for SOAs has to provide solutions for two situations: In the first case, the SLA for a service is known or available. A mapping from SLA values, or more specific, their SLOs, to curve contracts in SDPs exists. This work will not provide a scheme to transfer parameters from WSLA or similar formats. Here, we will consider the second case with unknown SLAs and present a curve fitting method based on recorded input and output traces. It is a solution for providers when a services are new, undocumented or exist as simulation models only. These systems can be surveyed at runtime with statistical methods and descriptive SDPs can be estimated. The case considered here is a continuation of previous works [15–17] with already existing SOA simulation models and appropriate tools for evaluation.

10.1 The Curve Estimation Problem

The problem solved here is finding a characterizing SDP for a SOA service or a general system based on input and output traces. In detail, for an SDP altogether four upper and lower bounding curves have to be estimated in such a way, that the initial traces are conform to the curve contracts again. Formally, for a system an arrival flow R and a departure flow R^* are given in form of traces with time marks when jobs entered and left the system. Required are the curves $\alpha^L, \alpha^U, \Psi^L$ and Ψ^U with $R \geq \alpha^L \,\overline{\otimes}\, R$ and $R \leq \alpha^U \,\underline{\otimes}\, R$. Delay flow D is computed from R and R^* by their horizontal deviation (see Section 7.1.3). The respective delay curves should be estimated in such a way that $D \geq \Psi^L \,\overline{\otimes}\, D$ and $D \leq \Psi^U \,\underline{\otimes}\, D$.

Modeling SOAs with performance constraints and guarantees adds further conditions on the estimated curves. Despite of being valid bounds for arrival and delay flows the estimated functions should be kept simple. Parameter sets of piece-wise linear functions are small, have clear semantics and separate short-term variations. An estimation approach for SLA Calculus should deliver this type of bounds.

10.1.1 Network Calculus Curve Estimation

Curve estimation for Network Calculus has been published in several works [37, 52, 67, 107]. In the following we will summarize the approaches and show that they are not suitable to compute SDPs from measured traces, due to some specifics found in SOAs.

Deconvolution Approach

In general arrival curves can be computed from arrival flows directly using (min,+)-deconvolution. In [107, Chapter 1.2.4] Le Boudec and Thiran show that deconvolving a flow with itself gives the so-called minimum arrival curve. In this context it is the smallest of all valid upper arrival curves for a flow, not a lower bound. Hence the minimum arrival curve α^U for a flow R is

$$\alpha^U(t) = \max_{\lambda \geq 0} \{R(t+\lambda) - R(\lambda)\} = (R \underline{\oslash} R)(t) \tag{10.1}$$

The deconvolution approach was used in [67] to fit arrival curves. (min,+)-deconvolution includes the maximum vertical distance between input and output flows. In [66] this is exploited for curve estimation in a comparable sliding window approach.

In a similar way the maximum lower arrival curve can be determined. Replacing (min,+) with (max,+)-deconvolution gives the upper bounds for R [67]:

$$\alpha^L(t) = \min_{\lambda \geq 0} \{R(t+\lambda) - R(\lambda)\} = (R \overline{\oslash} R)(t) \tag{10.2}$$

At a first glance the deconvolution approach has some advantages. The implementation in a curve fitting tool is easy as (min,+)-deconvolution is a standard operation in Network Calculus. No further knowledge on the arrival flow or the properties of the underlying trace is necessary. Also the resulting curve is the best fitting curve for a flow instance. At a second glance the approach is not well suited for SLA modeling in SLA Calculus. The curve results are very detailed, the resulting curve will be a step function if the flow is a step function. This misfits the idea of a simple representation with piecewise-linear functions. Thus, the deconvolution approach will remain unused for curve estimation in SLA Calculus.

Estimation of Affine Functions

A single affine function $\gamma_{r,b}$ can serve as arrival curve with two parameters only. Although its descriptiveness is rather limited, it is worth to have a look at estimation schemes for r and b. The two parameters differ in estimation complexity. Rate r can easily be found by inspection of arrival flow R. In Queueing Systems the average arrival rate for a system is the sum of all arrivals n_a divided by time [35, Equation 6.23]:

$$\lambda(t) = \frac{n_a(t)}{t} \tag{10.3}$$

An arrival flow $R(t)$ is also the sum of all arrivals up to time t, so we can use $n_a(t) = R(t)$. When an arrival flow is based on measurements it is finite, there must be a point of time t_f that belongs to the last measurement. For the arrival rate r we can conclude

$$r = \frac{R(t_f)}{t_f} \tag{10.4}$$

More elaborate is buffer size b. In Heckmann et. al. [52] b is computed for a given arrival flow. It exploits the relationship between affine curves and leaky buckets, where b indicates the bucket size.

T-SPEC Estimation

T-SPEC contracts are a combination of two affine curves often used for network traffic specification [100, 107]. Their four parameters p, M, r, b can be mapped to term $\gamma_{p,M} \wedge \gamma_{r,b}$. As demonstrated in Section 6.2.5 parameters r and b can be used to bound arrival flows in the long-term, while p and M take responsibility for short-term behavior. The algorithm of Heckmann et. al. is not capable to separate these flow properties.

To find T-SPEC contracts for network routers a scheme of how to translate observations based on arrival traces is given in [37]. Measurements include size and interarrival times of packets to a network interface. For short-term bounds the packet sizes are relevant. The largest packet found in the trace marks the maximum arrival size M. Then continuous sequences of M-sized packets are located, their maximum is used for the peak rate p. Long-term rates r are based on Equation 10.4. Publication [37] does not indicate how the last parameter b is found. We assume they use the duration of the longest M-sized packet sequence to adjust b.

This translation scheme can, with small modifications, be employed to find lower boundaries for flows. For the convex variant of T-SPEC curves minimum packet sizes and the intervals with no arrivals are relevant. Parameter M is used to size the maximum time interval with no traffic. Value p indicates the minimum short-term arrival rate. It is found by locating the longest sequence of minimum-sized packets. Again [37] does not reveal how to find b, but we can assume it is dependent on the length of the small packet sequence. The average long-term arrival rate r is still based on Equation 10.4 and thus, equal to the concave curve.

10.1.2 Noteworthiness for SOA models

In summary, the method for T-SPEC presented in [37] depends on maximum and minimum packet sizes and the occurrence of sequences of these packets. The scheme can take advantage of technical restrictions that exist in communication networks, a modeling domain suitable for Network Calculus operating on packet or even byte stream level. In the TCP/IP world one would say that modeling takes place at the transport, Internet or even link layer [68]. End-to-end connections or lines transmit data packets one after the other and so packet sequences can easily be identified. Another aspect of packet oriented transmission systems is their specified upper packet size limit. For instance, in Ethernets this is the Maximum Transfer Unit (MTU) giving the maximum length of a single Ethernet frame. Network layer protocols like IP fit in their own packet lengths into a frame, hence the MTU directly influences the size of a packet arrival in Network Calculus models. Both properties can be exploited to estimate upper arrival curves.

In SOAs both preconditions do not hold. Job requests to services are not necessarily sequential as packet arrivals to a network interface. Communication in SOAs and thus for Web Services is on application level, so several requests to a single service can arrive almost parallel. Multiple job requests, SOAP messages for example, might arrive in interleaved packets in several TCP/IP connections. On bit and packet level these arrivals are still sequential, but parallel from application perspective. When several small jobs arrive in this way they can induce more system load than a larger single arrival (Figure 10.1). For

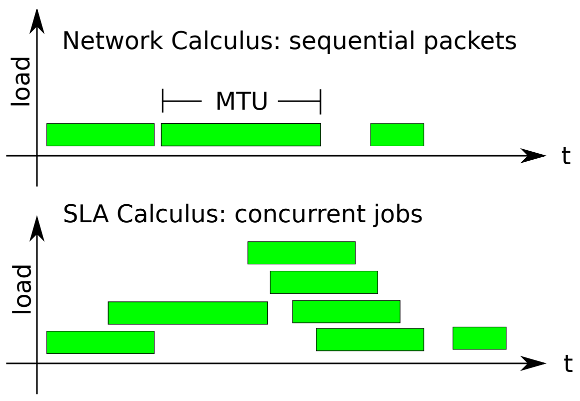

Network Calculus: sequential packets
load
MTU
t
SLA Calculus: concurrent jobs
load
t

Figure 10.1: Concurrent arrivals are possible in SOA: Workload induced by several small request is greater than a single serial arrival

delay arrivals, being a theoretic construct without any technical limitation, such parallel arrivals in the delay flow are seen if jobs finish at the same time. As a result, peak arrival rate p cannot be determined as in [37] by observing a sequence of maximum sized packets in a single stream. Differences exist also in the identification of maximum job sizes equal to MTUs. Job sizes may not be limited for processing times and thus delay arrivals. There may be no limiting timeout. Another aspect is that it is very unlikely to find two delay arrivals of the same size in sequence, so utilizing the correlation of arrival sizes as in Network Calculus is not possible. The special situation in SOA models renders the curve estimation method in [37] unsuitable for SLA Calculus.

10.2 Curve Estimation for SLA Calculus

The following curve estimation scheme is tailored towards arrival processes and delay flows in SOA. No prior knowledge on sizes and sequences of arrivals is required, input and output traces are accepted. It is independent of the measurung unit, so curves for request and delay arrivals to create SDPs are found. Estimated curves are compositions of piece-wise linear functions. Matching a set of functions of type $\gamma_{r,b}$ to bound a flow can be elaborative, since there are at least two parameters for each function to fit. The problem is simplified by splitting measured flows into sub-flows with single long-term rates and burst sizes each. As a side effect, the number of affine functions in each curve is determined based on the trace characteristic.

The curve estimation scheme is a shift in the paradigm of this work and SLA Calculus. Previously, the focus was on deterministic performance guarantees for (composed) systems, absolute knowledge on input bounds was assumed. Now a real system or running simulation model is measured, and for practical reasons, the observation period has to be limited. Not every state may be reached, thus one can not assume that the worst-case situation is included in the trace. As a consequence, all curves derived from trace data can only set bounds based on their measurements, other, not observed situations cannot be considered. There will be an error the user should be aware of, and bounds for one trace might over- or underestimate the limits for other system runs. Furthermore, when curves are estimated from arbitrary traces, a second trace measurement under same conditions might not be conform. An option to avoid such violations is to formulate very loose-fitting traffic contracts. This over-fitting might lead to very pessimistic assumptions on system performance in a following analytic system analysis. The approach taken here is to accept curve contract violations, quantify them and select the "best" fitting curve with respect to several measurements.

10.2.1 Event Traces and Delay Flow Computation

Arrival and delay flows are based on events (c.f Sections 7.1.1 and 8.1.1): For the ith event $e_i = (w_i, t_i)$ there is a weight w_i describing the job size and t_i marks the arrival time (c.f. [17]). Curve estimation can work on event traces recordable at every system with an input and departure process. For curve estimation the minimum data set contains arrivals and departure times, the job size is the arrival event weight, default is $w_i = 1$. An event trace E is a set of events $e_1, \ldots, e_n$ with $t_i \leq t_j$ for $i < j$. With the ordering event traces and arrival flows can be used synonymously. An arrival flow $R(t)$ is then a step-function indicating the cumulated weights of arrival trace E:

$$R(t) = \sum_{i=1}^{k} w(e_i) \text{ where } k = \max\{i | t_i \leq t\} \text{ and } e_i \in E \tag{10.5}$$

Delay flows can be constructed from input and departure traces. When flow A is the arrival flow and B is the departure flow, delay flow D is formed by computing the horizontal distance (c.f. Section 6.2.2) of matching input and output events. Let a_i, b_i be a pair of events with $a \in A, b \in B$ generated on behalf of job request i. The new events $(t_i, w_i) \in D$ are given by $t_i = t(b)$ and $w_i = t(b) - t(a)$.

10.2.2 The Onion Bucket Algorithm

Our estimation scheme consists of an algorithm to determine curves from traces and a validation process to discard not feasible solutions. The algorithm is called the onion bucket algorithm since it shapes and cuts the flow like onion bulbs with the principles of leaky buckets [1]. For our description we will apply estimations to arrival flows, the reader is free to apply them to delay flows, too.

The basic idea of the algorithm is the trivial fact that all arrival flows have an average arrival rate when they include at least two arrival events. A rate can easily be computed for a flow or trace R of limited length by Equation 10.4. When r_1 is the long-term arrival rate for R, one can see that there are arrivals conform to an arrival curve $\lambda_{r_1}(t) = r_1 t$, thus, they arrive at average rate, and some which are not arrive at a higher rate. When conform and non-conform arrivals are identified the arrival flow can be split into two sub-flows. Figure 10.2 shows the separation of R into two flows by λ_{r_1}: the light blue area are the conform arrivals (R_1), the darker blue area are the non-conform ones (R_2). From a global perspective, flow R_2 is an arrival burst on top of R_1. Arrival events that exceed

[1] For the record, there is no other serious reason for this name.

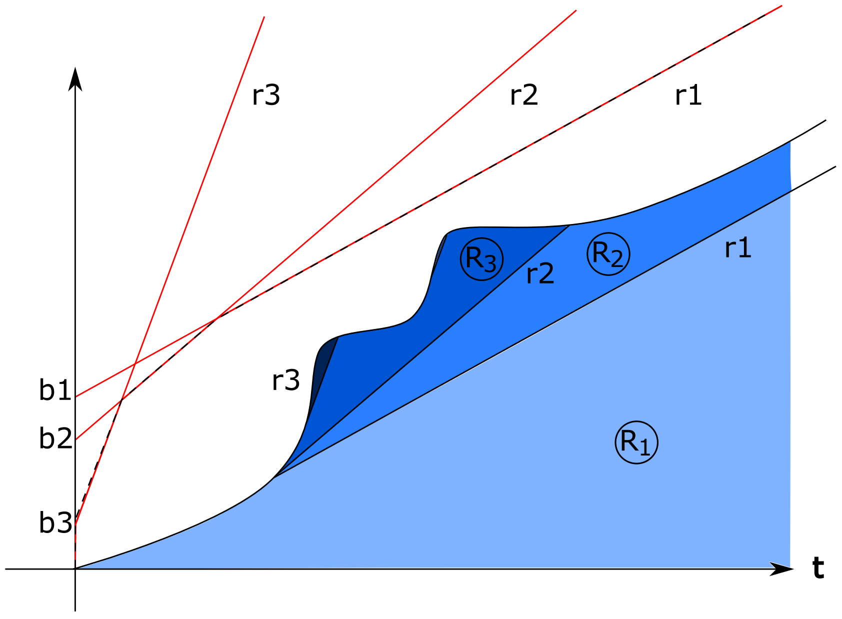


Figure 10.2: Onion Bucket Algorithm: Every flow R_i has an average rate r_i and burst size b_i. Non-conform arrivals form a residual flows that is bounded again with r_{i+1} and b_{i+1}.

the long-term rate are arrival bursts that can be limited by affine functions as described in Section 6.2.5. Or, using the metaphor of leaky buckets, the flow is conform to a leaky bucket described by γ_{r_1,b_1} with refill rate r_1 and a still to be determined bucket size b_1. In Figure 10.2 this is visible as the red line with rate r_1 and vertical shift b_1.

A single affine curve may not be sufficient as an arrival limit, the rate of the bursts can be controlled with curve compositions (c.f. Section 6.2.5). Since the bursts are isolated in R_2 we can again identify an average rate r_2 and a bucket size b_2 for R_2. This gives us a second affine curve γ_{r_2,b_2} usable to limit the bursts in the original flow with a composed arrival curve $\gamma_{r_1,b_1} \wedge \gamma_{r_2,b_2}$. We also receive a residual flow of higher-than average arrivals R_3. Figure 10.2 shows R_3 as dark-blue area. Obviously we can repeat the step and receive an even more detailed burst limit γ_{r_i,b_i} for residual flow R_i. The iteration stops either when the residual flows contain one or zero arrivals or the bucket size for an iteration is smaller than the weight of the largest arrival event.

The informal algorithm description above requires an approach to separate flows by a rate and to determine the buffer size for a flow. Both tasks can be done by Network Calculus greedy shaper elements implementing a linear shaping curve. Remember, the greedy shaper buffers non-conform arrivals and outputs them as soon as possible. When the shaper starts to buffer, there are arrivals destined for the buffer depicted as gray area in Figure 10.3. This flow towards the buffer can be marked as non-conform and separated from the main flow. We also see in Figure 10.3 that there are discontinuations in the flow towards the buffer when the arrival rate drops below the rate of the shaping curve.

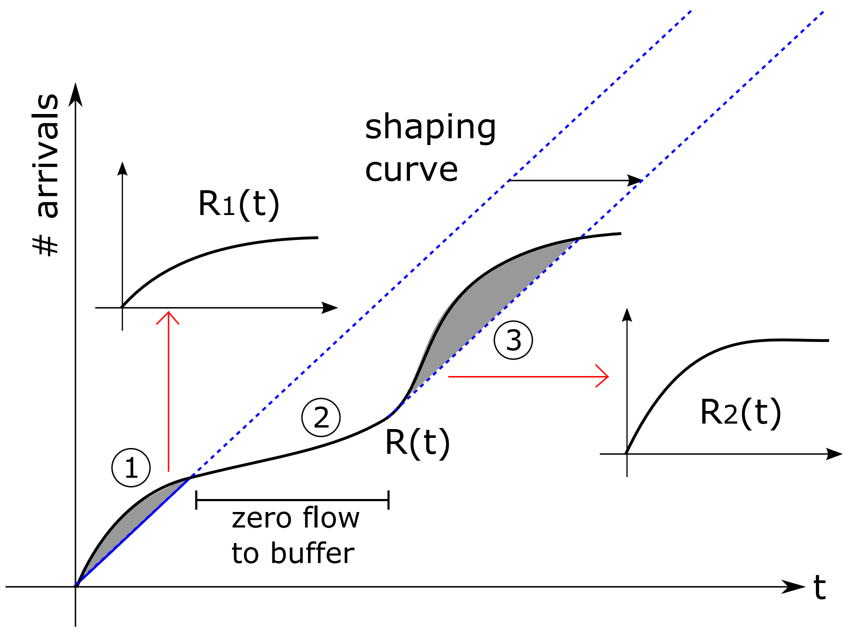


Figure 10.3: Shaping for Onion Bucket Algorithm: Curve α separates burst phases (1 and 3) as new flows R_1 and R_2.

These flow fragments are dropped from the input to the next shaping iteration except the fragment with the highest average rate. Flow fragments with lower rates will not contribute to the arrival curve estimation and, as a bonus, the algorithm converges faster.

The buffer or bucket size is the difference between input and output given by Equation 6.40.

Concave Implementation

Further details and specifics of the estimation algorithm are added by the comments on pseudo-code. Listing 1 contains the main loop generating affine functions and Listing 2 a shaper for a set of flows. Initial input of Listing 1 is a measured arrival (or delay) flow R. It is included in the first bulb, a data structure containing the sub-flows of each loop iteration (Line 2). The output will be stored in the variable *curve*, a set of affine functions with no initial content. The maximum arrival weight M is saved for the stopping condition (Line 3).

The main algorithm runs in an infinite loop whose iterations are identified by counter i: For all flows in the onion bulb the mean rates are computed and their maximum is saved in r_i (Line 8). Additionally the maximum buffer requirement b_i for a shaper with a linear curve of rate r_i is estimated for all flows in the bulb. Buffer estimation function $x(flow, rate)$ is discussed below. Now the first ($i = 1$) upper bound is known, function $\gamma_i(t) = r_i \cdot t + b_i$ is added to data structure *curve* (Line 16). γ_i limits the flow R in the long-term, the next iteration will give limits for a shorter time horizon. To find rate limits for these short phases, the sub-flows with a higher rate than r_i have to be shaped of from

R. As a small optimization, in Line 22 all flows with buffer requirements smaller than M are removed.

The shaping itself is performed in function SHAPEANDCUT (Listing 2): Input is the set *onion* of arrival flows and the rate parameter r for a shaping curve. Return value *bulb* is initialized as an empty set of flows. For all flows R_i in *onion* the following procedure is repeated: k is the iteration counter and *start* marks the beginning of a backlogging phase. The implementation uses an affine shaping curve whose y-axis intercept is variated with b, this will be covered later. In the inner loop over events $e_j \in R_i$ the cumulated weight $R_i(t(e_j))$ is compared to the value of the shaping curve $\gamma(t(e_j))$ to decide if backlogging is required. If a backlogging phase starts, a new flow data structure $lossFlow_k$ is created (Line 4). It will contain arrivals in R_i that are non-conform for the shaper and thus have to be backlogged. The offset *start* is set to the arrival time of the last event e_{j-1} before backlogging started (Line 14). This is necessary to preserve the rate induced by e_j in the new flow. All succeeding events in the backlogging phase are transformed to new events e' using offset *start* and their full weight (Line 18). If not backlogging, the shaping curve is lowered by adjusting b to simulate the effect of a (min,+)-convolution.

We return to Listing 1. The next loop iteration starts with the new set of flows generated by the shaper in Line 23 giving the next affine function. The loop runs as long as there are flows in set *bulb* that can be bound by an affine function and shaped to sub-flows (Line 24). Maximum arrival size M was saved at the beginning to set all b_i to at least M. We decided to do so because when there is an arrival of weight M the minimum bucket size has to be greater or equal to M, otherwise the arrival could never pass the system. For this reason condition *endPhase* is set in Line 21 to ensure that no buffer requirement is smaller than M. It decides in Line 15 if a new affine curve is added or the last one is just updated with a new rate.

The maximum buffer size estimation is based on Equation 6.40 (Figure 6.17), but takes care of the step function nature of R and discrete event arrivals:

$$x(R, r) = \max_{t>0} \left\{ \sup_{0 \le s \le t} \{R(t) - R(s) - w(s) - r(t - s)\} \right\} \tag{10.6}$$

When evaluating $x(R, r)$ at $t(e_i)$ only the left beginning of each step in $R(t)$ is considered (as symbolized in Figure 10.5). To include the time interval between two succeeding events e_i, e_{i+1} term $w(s)$ is subtracted. When an event has to be buffered, it is buffered as a whole, thus the maximum of the required buffer and the event is used (Line 12).

Convex Implementation

A convex version of the algorithm for lower arrival curves can be implemented by the usage of minimum instead of maximum operations for rates and replacing bucket sizes b_i with time-axis intercepts T to get a set of rate-latency curves $\beta_{R,T}$. Instead of maximum arrival M, the maximum interarrival time can be used (see Figure 6.12). Listing 1 has to be changed in some lines:

- replace maximum weight with the maximum interarrival time $maxIA$ (line 3)
- and transform it to a y-axis intercept: $M = -maxIA \cdot r$

Algorithm 1 Fitting upper (concave) arrival curves

```
 1: $curve \leftarrow \{\}$
 2: $bulb \leftarrow \{R\}$
 3: $M \leftarrow maxWeight(bulb)$
 4: $endPhase \leftarrow$ false
 5: $loopActive \leftarrow$ true
 6: $i \leftarrow 1$
 7: repeat
 8:     $r_i = \max(rate(bulb))$
 9:     $b_i \leftarrow t \leftarrow 0$
10:     for all $R \in bulb$ do
11:         $myBuffer \leftarrow x(R, r_i)$
12:         $b_i \leftarrow \max(myBuffer, b_i)$
13:     end for
14:     $b \leftarrow \max(b_i, M)$
15:     if $\neg endPhase$ then
16:         $curve \cup (r_i, b_i)$
17:         $t \leftarrow i$
18:     else
19:         $r_t \leftarrow r_i$
20:     end if
21:     $endPhase \leftarrow b_i \leq M$
22:     $bulb \leftarrow \{R : R \in bulb, x(R, r_i) \geq M\}$
23:     $bulb \leftarrow$ SHAPEANDCUT$(bulb, r_i)$
24:     $loopActive \leftarrow elements(bulb) > 0$
25:     $i \leftarrow i + 1$
26: until $\neg loopActive$
27: return $curve$
```

Algorithm 2 Shaper implementation

```
 1: function SHAPEANDCUT$(onion, r)$
 2:     $bulb \leftarrow \{\}$
 3:     for all $flow \in Onion$ do
 4:         $lossFlow \leftarrow \{\}$
 5:         $start \leftarrow 0$
 6:         $b \leftarrow 0$
 7:         $k \leftarrow 0$
 8:         $backLogging \leftarrow$ false
 9:         for all $e_j \in flow$ do
10:             if $R(t(e_j)) > r \cdot t(e_j) + b$ then
11:                 if $\neg backLogging$ then
12:                     $backLogging \leftarrow$ true
13:                     $k \leftarrow k + 1$
14:                     $start \leftarrow t(e_{j-1})$
15:                     $lossFlow_k \leftarrow \{\}$
16:                     $bulb \leftarrow bulb \cup lossFlow_k$
17:                 end if
18:                 $e' = (t(e_j) - start, w(e_j))$
19:                 $lossFlow_k \leftarrow lossFlow_k \cup e'$
20:             else
21:                 $backlogging \leftarrow$ false
22:                 $b \leftarrow R(t(e_j)) - r \cdot t(e_j)$
23:             end if
24:         end for
25:     end for
26:     return $bulb$
27: end function
```

- search for minimum rate (line 8) and minimum buffer (line 12 and 14)

In the shaper implementation of Listing 2 some lines have also to be changed:

- begin backlogging if $R(t(e_i)) < r \cdot t(e_j) + b$ (line 10)
- replace line 14 with $startTime \leftarrow \min\left\{\frac{(r \cdot t(e_j)+b)-w(e_{j-1})}{r}, t(e_j) - t(e_{j-1})\right\}$
- replace line 22 with $b \leftarrow R(t(e_{j-1})) - r \cdot t(e_j)$

The weight of the previous event is subtracted from the shaper function for *startTime* to include the beginning of every step in flow R.

10.3 Model Selection

A single set of curves for an SDP can be found with the Onion Bucket Algorithm based on a trace set. The functions are fitted to exactly bound the arrival and delay flows given by the traces, but with finite measurement length not all situations are considered. As a consequence, the curve from one trace might be too tight, when applied, other traces under the same conditions might violate it.

An obvious solution here would be to record several traces from a system, estimate their bounds and use their maxima and minima. This approach could approximate or even yield the real bounds. Unfortunately, this works for systems with existing process limits only, but this cannot be assumed for an estimation method that does not require a priori information. For example, if the arrival model is Poisson there is no absolute bound on the arrival process and thus on bursts. For this reason we estimate curves for a larger set traces and select the candidates that characterize all arrival and delay flows best. The decision on the "best" curve is based on the fitting criteria intended to minimize the number of curve violations when applied to all trace sets. The tool for this curve selection used here is Hold-Out Validation [61].

10.3.1 Calibration and Validation

The selection process of Hold-Out Validation is applied to a set of traces recorded from a system under measurement. It is split into phases for model calibration and validation. In the calibration phase a group of curves for an SDP is estimated for a smaller subset of traces, in the validation phase it has to show its fitting quality according to the remaining traces. Phase separation in hold-out validation avoids over-fitting of curves.

Let T be a set of arrival and departure flow pairs $\{R_i, R_i^*\}$ respective traces. For calibration a subset $T_C \subset T$ is used: In our experiments we used the first 10% of traces in T. The Onion Bucket Algorithm is executed for all $\{R_i, R_i^*\} \in T_C$ resulting in curve groups $\{\alpha_i^U, \alpha_i^L, \Psi_i^U, \Psi_i^L\}$. Goal of the calibration phase is to identify a single SDP $SP_C = \{\alpha_C^U, \alpha_C^L, \Psi_C^U, \Psi_C^L\}$ that characterizes and limits the arrival and delay processes in T_C best. We exemplarily continue with α_i^U, the same procedure is applied to the other three curve types individually. With α_i^U we have a valid bound for the arrival process described by R_i. For other flows the bound might be too loose or too tight and curve contract violations might occur. This error is quantified for all traces in T_C with a cost or error function $C(\alpha_i^U, R_j)$ assessing the fitting quality when α_i^U is applied to limit R_j with $j \neq i$ and $\{R_j, R_j^*\} \in T_C$. A curve α_i^U is selected as a candidate α_C^U for a minimum sum of costs for all traces in T_C.

With candidate set SP_C the validation phase is started to check if SP_C is also a valid characterization of the remaining traces $T_V = T \setminus T_C$. For each of the four curves in SP_C the fitting quality with $\{R_j, R_j^*\} \in T_V$ is evaluated using the cost function. The results are averaged and their standard deviation is computed. A curve candidate is accepted, if the confidence interval of its validation results shows a significant distance to the error values of discarded curves estimated in the calibration phase.

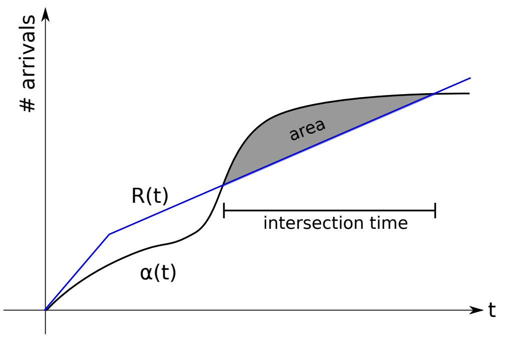


Figure 10.4: Intersected area and intersection time criteria

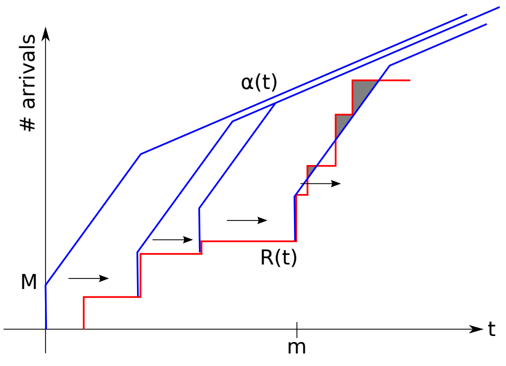


Figure 10.5: Curve violations and time invariance for arrival curves

10.3.2 Cost Function and Fitting Criteria

To assess the level of conformance of a flow and a curve, three criteria are combined in the cost function C. Avoiding curve contract violations is the primary goal for curve estimation in SLA Calculus and so the two criteria are based on intersections of flows and curves. The severeness of a violation is captured by the intersection area, its duration by the intersection length. Figure 10.4 shows both criteria applied to an exemplary arrival flow. The third criterion is introduced as an antagonist to the first two to gratify tight bounds. It is based on the distance between flow and curve.

The indicator function is used to indicate curve contract violations:

$$1_{(a,b)}(x) = \begin{cases} 1 & a \leq x \leq b \\ 0 & otherwise \end{cases} \tag{10.7}$$

Function $\omega(R)$ returns the arrival time of the last event relative to the first event, thus when $e_1 \in R$ is the first and $e_z \in R$ is the last event $\omega(R) = t(e_z) - t(e_1)$ holds. The criteria functions are designed in a way that f is the curve and R is the flow. For assessing lower bounds f and R have to be exchanged.

Intersection Time

The intersection time between arrival flow f and curve R indicates the fraction of time a curve contract is violated by an arrival process within the observation interval.

$$it(f, R) = \int_0^{\omega(R)} x \cdot 1_{(0,\infty)}(R(x) - f(x))\, \mathrm{d}x \tag{10.8}$$

The indicator function includes time intervals with intersections in the sum only.

Intersected Area

As the second fitting criterion the area between flow and curve (Figure 10.4) spanned by curve contract violations is used:

$$ia(f,R) = \int_0^{\omega(R)} (R(x) - f(x)) \cdot 1_{(0,\infty)}(R(x) - f(x))\,\mathrm{d}x \tag{10.9}$$

This criterion is included in our estimation method to anticipate short but severe curve contract violations.

Mean Squared Distance

As a measure of distance between curve and flow the mean squared distance is included. Criteria based on intersection return a value of 0 if there is no curve contract violation at all. From the opposite perspective there is no way to quantify with intersection criteria on how tight the limits imposed by estimated curves are. A measure giving reward on a tight fit between curve and flow is the cumulated distance, or as used here, the squared distance eliminating negative distance figures.

$$mse(f,R) = \sum_{t \in E_R} (R(t) - f(t))^2 \tag{10.10}$$

Time-Invariant Criteria

With the three criteria as described above one determines the fitting quality in interval $[0, \omega(R)]$. The claim of curve contracts as described in Section 6.2.3 is that they apply to all intervals regardless of their length. As a consequence, criteria have to be time-invariant, too. Considering the interval for the complete trace only brings the risk of not including curve violations that occur when the curve is "moved" along the trace. Figure 10.5 shows such a situation for an upper curve, where only one computation of the fitting criteria would ignore a contract violation. Applying the curve at $t = 0$ will indicate no contract violation as $R(t) < \alpha^U(t)\ \forall t$. If we apply the arrival curve at the interval starting at m, later arrivals clearly violate the curve contract.

To simulate the time invariance of curve contracts in fitting criteria we decided to fix the curve to the origin and move the flow instead. This is achieved by the successive removal of the first event in the input trace, shifting the time indices of remaining events and computation of the criterion. Our decision to implement time invariance this way is based on the performance gain when the number of events is reduced. Let E be an event trace for an arrival or departure flow, notation $|E|$ gives the number of included events. Operator S shifts the trace by time span Δ:

$$S(E,\Delta) = \{e' | e' = [t(e) - \Delta, w(e), a(e)], e \in E, t(e) - \Delta \geq 0\} \tag{10.11}$$

The average value of each criterion $c \in \{ia, it, mse\}$ under consideration of time-invariance is then

$$AVG(c,f,R) = \sum_{e \in R} \frac{c(f,R)}{p_c \cdot |S(R,t(e))| \cdot \omega(S(R,t(e)))} \tag{10.12}$$

Function results from the three criteria are dependent on the input trace length. Different experiments might yield different trace sizes and operator $S(E, \Delta)$ does also variate their length. To compare criteria figures independent of trace lengths, values are normalized to interval $[0, 1]$ using the theoretic worst-case.

Worst-case intersection time is equal for concave and convex curves:

$$p_{it}(\alpha, R) = \omega(R) \tag{10.13}$$

For concave curves (used for upper bounds) the worst-case intersected area is given by

$$p_{ia}(\alpha, R) = \int_0^{\omega(R)} R(x)\,\mathrm{d}x \tag{10.14}$$

and for convex curves for lower bounds it is

$$p_{ia}(\alpha, R) = \int_0^{\omega(R)} \alpha(x)\,\mathrm{d}x \tag{10.15}$$

The worst-case for the mean squared distance criterion is to have no arrivals at all. This is equivalent to an arrival flow R_Z with arrivals of zero weight. For practical implementation R_Z has to include arrivals, we use the intersection points of the line segments in curve contracts. Let x_i be the abscissa value for the intersection point of line segment γ_i and γ_{i+1}.

$$E_{R_Z} = \{0, x_1, x_2, \ldots, x_{n-1}, \omega R\} \tag{10.16}$$

is the set of events based on intersection points. Then the flow is given by

$$R_Z = \{e_i | e_i = (t_i, 0, i) : t_i \in E_{R_Z}\} \tag{10.17}$$

and the worst-case is computed by

$$p_{mse}(\alpha, R) = mse(\alpha, R_Z) \tag{10.18}$$

The overall cost function is given as a weighted sum of all three criteria being normalized:

$$C(\alpha, R) = a_{it} \cdot \frac{AVG(it, \alpha, R)}{p_{it}} + a_{ia} \cdot \frac{AVG(ia, \alpha, R)}{p_{ia}} + a_{mse} \cdot \frac{AVG(mse, \alpha, R)}{p_{mse}} \tag{10.19}$$

with $a_{it} \geq 0$, $a_{ia} \geq 0$, $a_{mse} \geq 0$ and $a_{it} + a_{ia} + a_{mse} = 1$.

10.4 Implementation and Experiment

The goal of the testbed is to estimate an SDP for a model analyzed in a simulator. The Onion Bucket Algorithm and the described model selection were implemented in an SDP fitting tool. *sc_estimator* accepts a set of trace file pairs that have been recorded at the same system at different runs. Each pair consists of files with measured arrival and departure times. Output are four curves bounding arrival and delay for each trace file pair, additionally, for each kind of limiting curve the best fitting instance is marked. *sc_estimator* is implemented in *C++* featuring a command line interface. This simple interface allows integration in model fitting workflows based on the Processes Fitting Toolkit Dortmund (ProFiDo) tool [20] as a future work.

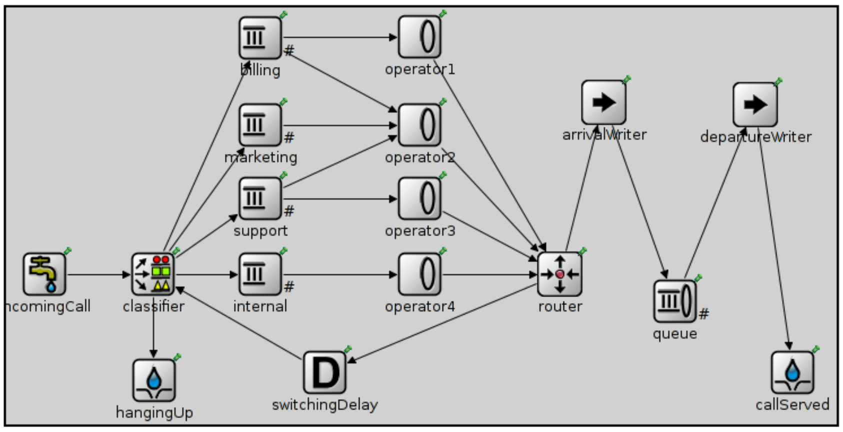


Figure 10.6: Modified *OMNeT++* call center model with trace output

10.4.1 Model and Experiment Setup

Compared to the previous chapters a different model is set up for the following experiment. To generate a set of input and output traces the *OMNeT++* simulator [112, 113] was used. The model under test is a variation of the call center model provided in the examples of the *OMNeT++* queueing library examples. Figure 10.6 shows a screenshot of the model as presented in the *OMNeT++* GUI. The original call center ends in the router component in the right part of the model, its output drives a simple server system. Arrivals to the model are customer calls with exponential distributed interarrival times ($\mu = 15s$). A classifier routes the calls to different waiting lines, after talking the customer either hangs up (thus leaves the system) or is forwarded, after some delay, to the classifier again. For our experiment we are interested in the arrival process of served customers hanging up being observable at the router, it forms the arrival flow. We have chosen this model configuration to include a source with a non-trivial interarrival time distribution in the system. The exponential distributed interarrival times of calls are modified while passing the call center model.

To support our experiment setup we added two trace writers and a queue element. The arrival writer records every customer hanging up as an arrival to a file. Before the calls are removed from the model they are sent to the added queue. Its server offers service times given by a truncated normal distribution ($\mu = 5s, \sigma = 2s$). Finally, the server output is recorded as departure flow by the second trace writer to another file. With the described experiment setup the arrival flow to be bound with the Onion Bucket algorithm is determined by the call center model, while the departures flow (and thus the delay) is given by the queue.

For the experiment we generated 100 pairs of traces with identical parameters for 100000s of model time each. Unique trajectories due to different system runs were ensured by varying random number generator seeds. To avoid the transient phase the arrivals of the first 2000 time units were dropped. All three criteria were equally weighted for curve selection ($a_{it} = a_{ia} = a_{mse} = \frac{1}{3}$). The initial trace set is divided into two subsets for calibration (first 10 runs) and validation (remaining 90 runs). For each estimated curve a computation time of about one hour was necessary.

10.4.2 Experiment Results

The estimated SDP describing the arrival process to and the delay process of the server element in the model is

$$SLA = \left\{ \begin{array}{ll} \alpha^U & = \min(\gamma_{1229.6,1.0}, \gamma_{0.5931,3.1065}, \gamma_{0.0656,90.6407}) \\ \alpha^L & = \max(\beta_{0.0323,-4.5749}, \beta_{0.0532,-8.7103}, \beta_{0.0628,-71.7494} \\ \Psi^U & = \min(\gamma_{215.749,35.4441}, \gamma_{2.4737,106.168}, \gamma_{0.4220,645.733}) \\ \Psi^L & = \max(\beta_{0.1995,-29.1075}, \beta_{0.3980,-874.584}) \end{array} \right\}$$

Detailed tables with all estimated, but not selected curves and their calibration findings can be found in Appendix D.2. The algorithm assigned up to four line segments to every curve type, the selected curves feature three. For Ψ^L the first segment, $\beta_{0,0}$, is redundant due to the definition of $\beta_{R,T}$ and thus omitted here (compare Table D.11).

The last segments in α^U and α^L limiting the long-term arrival rate are determined by the arrival rate of incoming calls to the call center. An average rate of $r_1 \approx \frac{1}{15} = 0.0\bar{6}$ is almost exactly mirrored by α^U and approximated by α^L from below. For the third segment in α^U the computed buffer size is 1, this equals the default arrival weight. The delay curves limit the long-term delay rate by 0.422 from above and 0.398 from below.

Next to curves and statistical information *sc_estimator* outputs *gnuplot* scripts to plot flows and estimated curves. All plots f can be optionally transformed to g by $g(t) = f(t) - r_1 \cdot t$ to make them appear horizontal. We have chosen this way of presentation to point out estimation quality and errors. In Figure 10.7 α^U is plotted together with the arrival flow (#8) it is based on. The estimated curve always stays above the arrival flow independent of the starting point s and we have

$$R(t) \leq \inf_{0 \leq s \leq t} \{R(t-s) + \alpha^U(s)\} = (R \underline{\otimes} \alpha^U)(t) \tag{10.20}$$

Indeed α^U is a upper arrival curve for flow R, the algorithm even considered extrema in the input data. The dashed line in Figure 10.7 depicts such a situation where bucket size b_1 is used completely to fulfill this relationship. In Figure 10.8 α^L is shown together with its corresponding flow, the selected curve instance for Ψ^U and Ψ^L are shown in Figures 10.9 and 10.10. For the readers delight, Figure 10.11 shows a situation found by the estimation tool during calibration when Ψ^U is not a valid upper bound on another flow. Obviously, when Ψ^U is applied at $t \approx 25000$ the curve intersects the flow.

The curves forming SDP SLA have been selected by the algorithm for their fitting quality during calibration. In Tables D.6, D.8, D.10 and D.12 the fitting results during the calibration phase are listed. Rows corresponding to curves selected for SDP SLA are printed bold. The mean squared distance criterion has an interesting impact on curve selection. On the one hand, from the calibration set the upper arrival curve computed for flow 8 was selected (Table D.6) for its low overall mean. Although it shows a high mean squared error value it has no (or less than the six decimal place's resolution) intersections. On the other hand, upper delay curve 8 has virtually no curve violations for the calibration

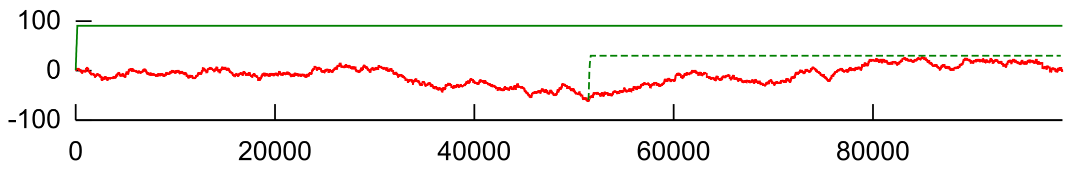


Figure 10.7: Green function: Estimated upper arrival curve α^L with arrival flow (#8, red). The dashed line shows the burst situation leading to $b_1 = 90.64$.

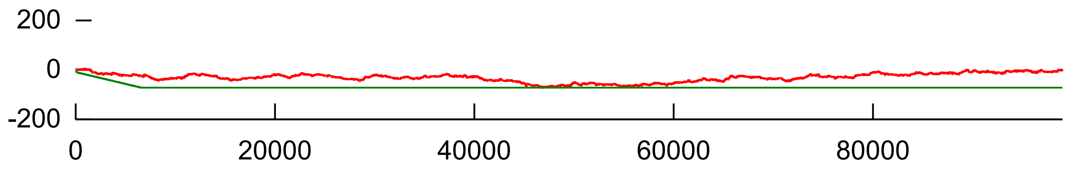


Figure 10.8: Flow #1 with estimated lower arrival curve α^L.

set (c.f. Table D.8), but remains unselected due to its high distance to the flows. Instead, curve 2 was preferred despite curve violations. The explanation for these selections is the low variance in the arrival and delay traces, so the estimated long-term contract (r_1, b_1) becomes important. Arrival Curve 8 has the highest r_1 of all curve candidates, additionally r_2 and b_1 are also high. Also for the selected upper delay curve based on flow 2 the long-term delay rate r_1 outstands together with a large, but not largest buffer capacity b_1. This segment parameter combination results in less local curve violations, but if a flow has a low average rate the mean squared distance increases. The addition of the mean squared distance criteria helps to balance between local and global fitting. As an example, in Figure 10.12 curve α^U is plotted together with a flow 1 from the calibration set. This combination showed the worst combined criteria results during calibration. Although α^U is a valid upper bound for this flow it is not a tight bound. Again, no curve violations can be found, but the difference between long-term rate and average flow rate leads a high distance error.

The four selected curve candidates are applied to the validation set in the second phase. Table 10.4.2 includes the validation results when the curves are applied to the remaining 90 traces. The results for each criteria are averaged and finally a weighted overall error including a 95% confidence interval is computed. For α^U the average error value 0.0014071 is $[0.001399, 0.0014152]$. When compared to the calibration results in Table D.6, the selected curve has a decent distance to the mean error of other curves in the calibration set. The same holds for α^L, Ψ^U and Ψ^L, hence the estimated curves pass the validation phase.

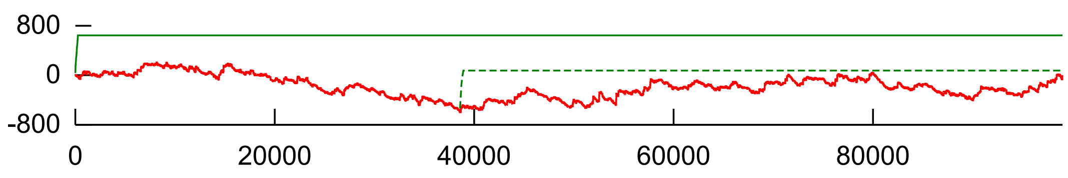


Figure 10.9: Delay flow with estimated Ψ^U (curve #2), the dashed line shows the maximum burst situation.

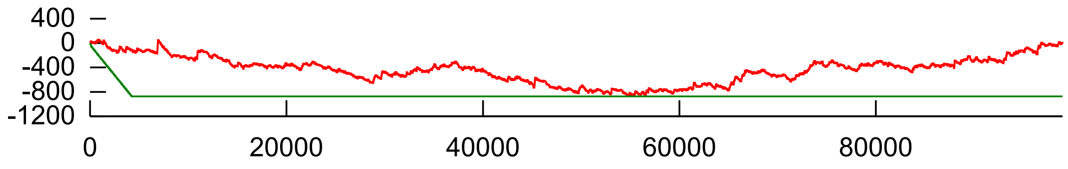


Figure 10.10: Flow #1 with estimated lower delay curve Ψ^L.

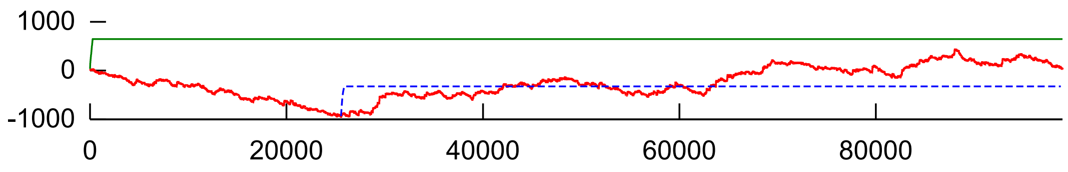


Figure 10.11: Ψ^U with worst fitting flow. Using Ψ^U as delay bound leads to curve violations.

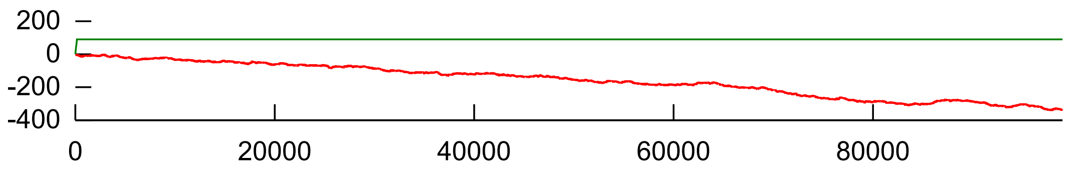


Figure 10.12: Arrival flow #1 with α^U: Estimated bound is too loose.

Table 10.1: Validation findings

Curve	Intersected Area		Intersection Time		Mean Squared Error		Combined		
	mean	std.	mean	std.	mean	std.	mean	std.	95% Confidence
α^U	0.00000	0.00000	0.00004	0.00036	0.00418	0.00112	0.00141	0.000368	[0.001399, 0.0014152]
α^L	0.00000	0.00001	0.01960	0.01308	0.00490	0.00131	0.00817	0.004197	[0.008076, 0.0082605]
Ψ^U	0.00002	0.00013	0.00283	0.01292	0.00605	0.00203	0.00297	0.004205	[0.002878, 0.0030627]
Ψ^L	0.00000	0.00000	0.00244	0.00429	0.01474	0.00357	0.00573	0.001651	[0.005689, 0.0057618]

10.4.3 Discussion

A single parameter was neither discussed nor optimized: the weight of each fitting criterion in cost function (10.19). Curve selection during the calibration phase is heavily dependent on these weights. In the experiment we distributed the weight equally for the curve selection, depending on the field of application a different weighting could have been chosen. For example, the intersected area criterion might be preferred but results in small numbers. In other situations, when there is plenty of computing capacity available, the weight of the mean squared error can be reduced. So, for same input data but different weights the tool will yield different SDPs. Future research should develop parameter policies for different fields of applications.

This brings us to another, more general aspect on curve estimation when it is used to compute SDPs: While SLA Calculus assumes that all SDPs represent deterministic performance guarantees even in worst-case, the estimation tool accepts fitting errors. Furthermore, it is based on the input data quality and length since it can consider situations only that appear in traces. In the end, the curve estimation approach presented suffers from the same faults as simulation. This opens another research field to quantify the estimation error.

Yet we think the estimation scheme is a suitable tool to derive SDPs for systems. It shows its qualities for black box systems. It also is practical when it is to elaborative to derive bounding curves out of detailed system specifications as done in Real-Time Calculus. However, the modeller using SLA Calculus should keep in mind that estimated curves have limited precision.

11 Conclusions

In this thesis SLA Calculus as a novel method based on Network Calculus to model quantitative requirements in SLAs for SOAs was introduced. It allows to compute and validate performance requirements given by SLAs. Service descriptions are purely based on contracted bounds originating from SLAs. For service compositions deterministic and thus, dependable results on workload capacity and delay are computable. Key features are the exclusion of any stochastic model element and to analyze for worst-case situations. Still, performance descriptions may extend beyond simple maximum rates and hard deadlines. Arrival processes with bursts and response times in systems with downtimes can be described in detail for short- and long-term using Network Calculus curve contracts.

The enabling factors for SLA validation are the definition of delay flows and their envelopes. With delay curves the latency of services within a time interval can be bounded from above and, if necessary, from below. Long- and short-term goals for delay rates can be set, when piece-wise linear functions are used, the semantics are simple. Since SLAs do not include detailed information on service rates, delay curves serve as a substitute for service curves. This is further supported by the introduction of SDPs as basic system model. SDPs combine arrival and delay curves to represent an implication found in SLAs: A maximum delay can only be guaranteed when the workload is limited. Hence, in SLA Calculus services are black boxes with upper and lower bounds for arrival and delay processes.

To support service composition we provide performance bounds for basic workflow structures. Based on individual SDPs global SDPs are computable and can be used to validate given SLAs by comparison. The calculus features theorems for concatenation, routing, fork/join and loops serial and parallel composition of SDPs, upper and lower bounds are considered in all compositions. Workloads bounds are narrowed in compositions to avoid overloading individual services. Delay bounds, due to the nature of passed time not being distributable, are summed up or maximized to include the worst-case. Found limits are pessimistic, but can help to reduce the risk of contract breaches significantly. By focusing on deterministic delay guarantees based on arrival contracts, we also got new insights in serial workflows: To enable service guarantees for workflows, requests have to be buffered at every service. Without buffers, the increased output burstiness of one service might violate the arrival contract of subsequent services. Since buffering introduces additional delay, a correction term based on required buffer sizes is added to all composition theorems. On the negative side some properties dedicated to Network Calculus are lost, for example, service sequences cannot be reordered due to lost linearity. Limiting buffer delays and linking Network Calculus to SLA Calculus requires the initially unknown service curve provided by a SDP. We solve this problem with service curve computation. The approach is, using (min,+)-deconvolution, purely analytic. With curve estimation for SLA Calculus we provide a method to find the characterizing SDPs for existing services. Arrival and delay curves are extracted out of traces, no prior knowledge on the measured system is required. The new method is necessary since methods for Network Calculus are based on some limitations in computer networks that do not apply to SOAs. However, since a single trace limited in length does not include all aspects of system behavior, we also provide a selection and validation workflow based on the set of traces. With this statistical element fitting errors have to be considered.

The research on a novel analytic modeling method for SOAs with SLA is justified. Due to the information hiding approach in SOAs SLAs are often the only available quantitative system description, models should be modest with parameters. Yet, a fast analysis is required for drafted models, fast service selection and enabling advanced functionalities in service brokers. When workflows are composed, reliable performance bounds are required to issue own SLAs. Still some form of tolerance is desired to compensate the stochastic nature of networks. At these points, many established system performance models fail. They are unable to derive deterministic limits, cannot consider worst-case scenarios in analysis or require long computation times. Based on our evaluation we consider Queueing Networks, Simulation and Network Calculus as unsuitable modeling methods for SLAs validation.

In an running example the benefits and downsides of each modeling method for SLA validation are demonstrated. Queueing Networks and simulation give optimistic results on the expected average response time. With Network Calculus bottlenecks in the composition are detected, still the commutative service curves give no information on internal contract breaches. The suggested SLA Calculus gives valid but pessimistic results for the worst-case. In this context, it is up to the modeler to decide which method is suitable to him or her, yet with SLA Calculus it is the first time that nonfunctional requirements on performance in SLAs can be used in a method that offers efficient analysis and deterministic bounds at the same time.

The work presented in this thesis can be extended in calculus and software support. Customer classes and priorities known to Real-Time Calculus remained unconsidered, their inclusion would enable QoS levels. Furthermore, Real-Time Calculus makes use of upper service curves for better departure bounds. Service curve computation could be extended in this direction using lower arrival and delay contracts.

For the underlying method a step away from 100% deterministic analysis could improve a modeler's perspective on systems. In a first step, product-form Queueing Systems and SLA Calculus could be combined for SOAs performance analysis. Both methods feature efficient analysis, but SLA Calculus cannot give average results and performance limits with Queueing Systems analysis are elaborative. A joined analysis would still be efficient, but delivers performance results with lower, average and upper figures. This way, modelers get a wider perspective on systems. The second step would be the dissolution of deterministic bounds in SLA Calculus using the theory of Stochastic Network Calculus [56]. In Stochastic Network Calculus, every curve envelope is related to a conformance probability. The idea is that stochastic envelopes with a probability smaller than 1 can be defined tighter. As an effect, modeling of worst-case situations get lost, one gets "almost worst-case" results only. The advantage of this trade-off are less pessimistic results for real world applications.

For practical applications the software support for SLA Calculus can be extended. A GUI for service compositions or integration with the *ProC/B* toolkit would extend the user base. Also, a formal or even automatic transformation from SLAs given in an XML dialect to SDPs is lacking.

When service workflows are modeled analyzed with SLA Calculus and finally, go into operation one might also monitor their performance at runtime. Service providers need continuous monitoring to respond to complaints, customers will control their workload level to see bottlenecks early. The customer can control if he reacts conform to the SLA and can also see if the provider is conform. Further development of curve estimation in Chapter 10 could enable a monitoring tool. Based on live input data the bounding curves for a flow can be computed and compared to contracted ones.

We also envision a new kind of delay-aware service composition for service brokers. It aggregates several providers with limited capacity to serve a certain workload. With variating workload the providers are exchanged to keep monetary costs as low as possible. The novelity is that deterministic delay guarantees are possible, when the selection algorithm is based on SLA Calculus models. Key element is the theorem for parallel services with routing. In first sketches we found the underlying optimization problem to be non-linear, thus for efficient solutions some research is necessary.

A Notations and Functions

A.1 Basic Functions for Curves

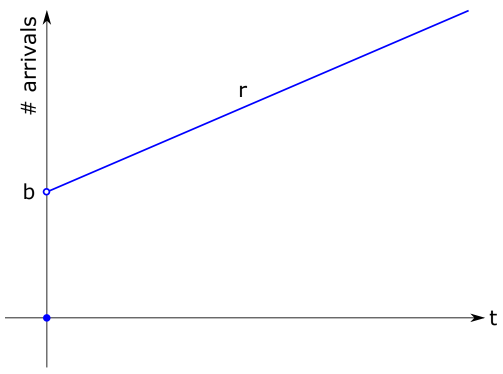


Figure A.1: Affine function $\gamma_{r,b}(t)$ with rate r and burst b

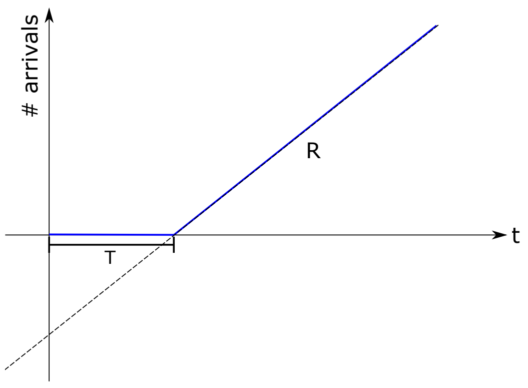


Figure A.2: Rate-latency function $\beta_{R,T}(t)$ with rate R and latency T

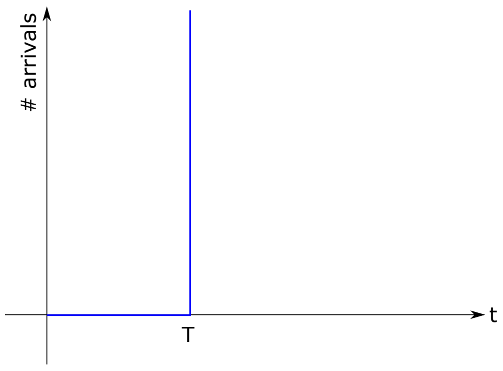


Figure A.3: Burst-delay function $\delta_T(t)$ for usage in delay elements

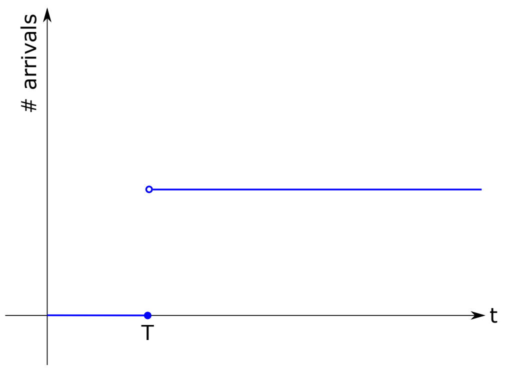


Figure A.4: Step Function $u_T(t)$

A.2 Overview of Used Symbols

Symbol	Description	See also	Page
$\mathcal{G}$	Set of wide-sense increasing functions	Definition 6.1.4	73
$\mathcal{F}$	Subset of $\mathcal{G}$ with $f(t) = 0$ for $t < 0$	Definition 6.1.4	73
$\mathcal{F}_0$	Subset of $\mathcal{F}$ with $f(t) = 0$ for $t \leq 0$	Definition 6.1.4	73
$f_1 \wedge f_2$	Pointwise minimum of functions f_1 and f_2	Section 6.1	73
$f \underline{\otimes} g$	(min,+)-Convolution	Definition 6.1.6	76
$f \underline{\oslash} g$	(min,+)-Deconvolution	Definition 6.1.7	77
$f \overline{\otimes} g$	(max,+)-Deconvolution	Definition 6.1.8	78
$f_1 \vee f_2$	Pointwise maximum of functions f_1 and f_2	Section 6.1.7	78
$\overline{f}$	Sub-additive closure of function f	Definition 6.1.11	79
$\underline{f}$	Super-additive closure of function f	Definition 6.1.11	79
f°	Function f with explicit $f(0) = 0$	Lemma 6.3	79
$R(t)$	Arrival flow: cumulative arrivals	Definition 6.2.1	80
$R^*(t)$	Departure flow: cumulative departures	Section 6.2.1	80
$v\left(f, g\right)(t)$	Vertical deviation between f, g	Definition 6.2.2	82
$h\left(f, g\right)(t)$	Horizontal deviation between f, g	Definition 6.2.3	82
$b_{\max}\left(f, g\right)$	Maximum backlog between f and g	Section 6.2.2	82
$h_{\max}\left(f, g\right)$	Maximum delay between f and g	Section 6.2.2	83
α^U, α^L	Upper/lower arrival curve	Definition 6.2.7	84
$C(t)$	Resource flow: cumulative service	Definition 6.2.10	93
β^U, β^L	Upper/lower service curve	Definition 6.2.11	93
$d\left(R, R^*\right)(t)$	Delay function between R and R^*	Definition 7.1.1	116
$D(t)$	Delay flow, cumulative delay up to time t	Definition 7.1.3	117
$\Psi^U(t)$	Upper delay curve	Definition 7.1.4	118
$\Psi^L(t)$	Lower delay curve	Definition 7.1.5	118
$b(R, R^*)(t)$	Backlog function between R and R^*	Definition 7.4.1	125
$B(t)$	Backlog flow, cumulative backlog up to time t	Definition 7.4.2	125
$h_{early}\left(f, g\right)(t)$	Earliness of f to lower contract g	Definition 8.3.2	132
$\alpha_i^{U'}$	Upper departure bound of service i	Definition 8.4.1	140
φ_i^U	Delay curve due to reshaping for service i	Definition 8.4.2	141
$\alpha_i^{L'}$	Lower departure bound of service i	Definition 8.4.3	142

B Proofs for Network Calculus

B.1 Proof of Corollary 6.1 (See page 78)

Corollary 6.1 (Sub-additivity of concave functions). *Any concave function f with $f(0) = 0$ is sub-additive.*

Proof. The proof is based on the result that concave functions are star-shaped [107, Thm. 3.1.4]. Let s, t be two variables with $s \leq t$. Obviously $s \leq s + t$ and since f is star-shaped we can write

$$\frac{f(s+t)}{s+t} \leq \frac{f(s)}{s} \tag{B.1}$$

$$f(s+t) \leq (s+t)\frac{f(s)}{s} \tag{B.2}$$

$$= f(s) + \frac{tf(s)}{s} \tag{B.3}$$

Using the star-shaped property again, we know that

$$\frac{f(s)}{s} \leq \frac{f(t)}{t} \tag{B.4}$$

$$\frac{tf(s)}{s} \leq f(t) \tag{B.5}$$

Combining both results one can state

$$f(s+t) \leq f(s) + f(t) \tag{B.6}$$

□

B.2 Proof of Corollary 6.2 (See page 78)

Corollary 6.2 (Super-additivity of convex functions). *Any convex function f with $f(0) = 0$ is super-additive.*

Proof. Recalling the definition of a convex function f and by precondition $f(y) = 0$ for $y = 0$ we can identify the following rule:

$$f(zx + (1-z)\cdot 0) \leq zf(x) + (1-z)f(0) \tag{B.7}$$

$$f(zx) \leq zf(x) \tag{B.8}$$

Now, for two variables s, t:

$$\begin{aligned}
f(s) + f(t) &= f\left((t+s)\frac{t}{t+s}\right) + f\left((t+s)\frac{s}{t+s}\right) && \text{(B.9)}\\
&\leq \frac{t}{t+s} f(t+s) + \frac{s}{t+s} f(t+s) \qquad \text{using (B.8)} && \text{(B.10)}\\
&= f(t+s) \cdot \left(\frac{t}{t+s} + \frac{s}{t+s}\right) && \text{(B.11)}\\
&= f(s+t) && \text{(B.12)}
\end{aligned}$$

□

C Proofs for SLA Calculus

This appendix contains the proofs for some non-central theorems in Chapter 8.

C.1 Proof of Theorem 8.1 (See page 131)

Theorem 8.1 (Input-Output Characterization of Prefetchers). *Consider a prefetcher implementing a lower envelop α^L for arrival flow R. At $t = 0$ there is no debt towards R and $R(t) > 0$ for $t \geq 0$ holds such that prefetching is possible. The output flow is given by*

$$R^*(t) = (R \,\overline{\otimes}\, \alpha^L)(t) \tag{8.5}$$

Proof. Consider a system that accepts flow R as input and has a departure flow S with constraints

$$S \geq R \text{ and } S \geq S \,\overline{\otimes}\, \alpha^L \tag{C.1}$$

A system fulfilling the first condition above has to emit arrivals before the corresponding "real" arrival since the output can be higher than the input. The output has a lower envelope of α^L by the second condition. Although (C.1) is fulfilled this is not necessarily a prefetcher. The counterexample is a departure flow $S(t) = \infty$ indicating the unlimited accumulation of debt towards the arrival flow, instead of stopping prefetching as soon as possible. A system stopping as soon as possible would produce an output that is the minimal solution to (C.1).

One can show that $R \,\overline{\otimes}\, \alpha^L$ is the optimal solution (minimal bound) to (C.1). The departure flow of a prefetcher shall be $S^* = R \,\overline{\otimes}\, \alpha^L$.

Let S' be another solution to (C.1), so it has α^L as lower traffic envelope. By definition of the prefetcher, $S' = S' \,\overline{\otimes}\, \alpha^L$. Then we have an output that at least equals the input $S' \geq R$. By monotonicity of $\overline{\otimes}$ follows

$$S' \geq R \,\overline{\otimes}\, \alpha^L = S^* \tag{C.2}$$

Now, with a minimal solution, one can show that $R^* = S^*$. Input R is wide-sense increasing, so is the output S^*.

R^* as departure flow has to be a solution for a system with conditions C.1 and so $R^*(t) \geq S^*(t)$ $\forall t$ holds. In case of $R^*(t) > S^*(t)$ there would be a situation with accumulation of debt towards the arrival flow R. This is inconsistent with the definition of a prefetcher that has to avoid arrival flow debt, thus solution S^* has to be equal to departure flow R. □

C.2 Proof of Theorem 8.7 (See page 137)

Theorem 8.7 (Convolution of Super-Additive Functions). *When two functions f and g are super-additive then their convolution can be simplified to the point-wise maximum.*

$$\underline{f} \,\overline{\otimes}\, \underline{g} = \underline{f \vee g} \tag{8.28}$$

Proof. (⇒) To proof one direction we start with the observation

$$f \vee g \;\leq\; f^\circ \vee g^\circ \;\leq f^\circ \overline{\otimes} g^\circ \tag{C.3}$$

We also know by the monotonicity of the super-additive closure that if $f \leq g$ then $\underline{f} \leq \underline{g}$ holds.

$$\underline{(\,f \vee g\,)} \geq \underline{(f^\circ \overline{\otimes} g^\circ)} = \underline{f}\,\overline{\otimes}\,\underline{g} \tag{C.4}$$

So for the right side of the theorem we can write

$$\underline{\,f \vee g\,} \leq \underline{(f^\circ \overline{\otimes} g^\circ)} = \underline{f}\,\overline{\otimes}\,\underline{g} \tag{C.5}$$

(⇐) The other direction of the proof starts using the monotonicity of $max()$ (see [38, Section 6.1]):

$$f \leq\; f \vee g \tag{C.6}$$
$$g \leq\; g \vee f \tag{C.7}$$

and in combination with (C.4) we get

$$\underline{f} \leq\; \underline{f \vee g} \tag{C.8}$$
$$\underline{g} \leq\; \underline{g \vee f} \tag{C.9}$$

By the monotonicity of $\overline{\otimes}$ [38, Section 6.1] and $g \overline{\otimes} g = g$ if and only if g is super-additive one can write for the left side

$$\underline{f}\,\overline{\otimes}\,\underline{g} \leq \underline{(\,f \vee g\,)}\,\overline{\otimes}\,\underline{(\,f \vee g\,)} = \underline{\,f \vee g\,} \tag{C.10}$$

□

D Experiment Results

D.1 *ProC/B* Simulation Results

Measurement	Simulation Time				
	10000	20000	50000	100000	1000000
Utilization					
Mean	0.9912	0.9930	0.9875	0.9885	0.9885
Standard Deviation	0.0971	0.0841	0.1123	0.1084	0.1066
Confidence 90%	2.2269%	1.9925%	1.7134%	1.2703%	0.4522%
Throughput					
Mean	8.0159	8.0150	7.9922	7.9956	8.0043
Standard Deviation	0.1250	0.1247	0.1251	0.1251	0.1250
Confidence 90%	0.7687%	0.5520%	0.3105%	0.2438%	0.1982%
Population					
Mean	57.2328	83.1361	73.1683	77.5544	83.3045
Standard Deviation	39.9496	69.0944	65.1497	68.6656	88.0788
Confidence 90%	22.7902%	30.9151%	26.9257%	20.1518%	9.8191%
Turnaround Time					
Mean	7.1342	10.3724	9.1541	9.6954	10.4076
Standard Deviation	4.9155	8.6129	8.1038	8.5316	10.9411
Confidence 90%	21.8159%	30.3666%	22.9645%	17.5227%	8.5635%
CPU Time					
	8.88s	18.18s	45.55s	91.40s	922.70s

Table D.1: Simulation results for HollowEarth service (*M/M/1* queue, $\lambda = 8.0$, $\mu = 8.1$)

Measurement	Simulation Time				
	10000	20000	50000	100000	1000000
Utilization					
Mean	0.6854	0.6867	0.6846	0.6808	0.6790
Standard Deviation	0.4643	0.4638	0.4647	0.4662	0.4669
Confidence 90%	1.5630%	1.0678%	0.6785%	0.3856%	0.4997%
Throughput					
Mean	5.5440	5.5238	5.5209	5.5008	5.4994
Standard Deviation	0.1798	0.1802	0.1810	0.1818	0.1819
Confidence 90%	0.8835%	0.7058%	0.4647%	0.3100%	0.2640%
Population					
Mean	2.2041	2.2295	2.1955	2.1393	2.1187
Standard Deviation	2.7052	2.7815	2.6960	2.6140	2.5770
Confidence 90%	4.3064%	3.4377%	1.8835%	1.8455%	0.5390%
Turnaround Time					
Mean	0.3975	0.4036	0.3976	0.3889	0.3853
Standard Deviation	0.4051	0.4217	0.4066	0.3941	0.3867
Confidence 90%	3.5691%	4.1099%	1.4588%	1.5846%	0.6488%
CPU Time					
Seconds	5.93	11.84	30.00	63.27	632.14

Table D.2: Simulation results for HollowEarth service (*M/M/1* queue, $\lambda = 5.5$, $\mu = 8.1$)

Measurement	Simulation Time				
	10000	20000	50000	100000	1000000
Throughput					
Mean	11.9967	11.9834	11.9865	11.9810	11.9965
Standard Deviation	0.0830	0.0831	0.0831	0.0832	0.0832
Confidence 90%	0.5626%	0.6289%	0.3894%	0.2818%	0.0413%
Population					
Mean	40.4172	37.5964	37.3352	36.6339	38.0554
Standard Deviation	27.2867	25.2071	26.1456	25.4182	26.3926
Confidence 90%	15.3680%	10.2368%	6.0530%	4.5811%	1.7581%
Turnaround Time					
Mean	3.3683	3.1371	3.1141	3.0574	3.1722
Standard Deviation	2.3343	2.1791	2.2393	2.1870	2.2820
Confidence 90%	13.8751%	9.7538%	4.4433%	4.8048%	1.5044%
Turnaround > 5s					
Probability	0.2110	0.1696	0.1669	0.1597	0.1793
Confidence 90%	0.9156%	0.7413%	0.4733%	0.3437%	0.2263%
Turnaround > 15s					
Probability	0	0	$1.5684 \cdot 10^{-4}$	$1.0016 \cdot 10^{-4}$	$6.346 \cdot 10^{-4}$
Confidence 90%	-	-	16.9140 %	14.9703%	2.5592%
CPU Time					
Seconds	31.49	64.18	160.84	329.23	3196.69

Table D.3: Simulation results for ParcelSink Workflow

Measurement	Simulation Time				
	10000	20000	50000	100000	1000000
Throughput					
Mean	12.0128	11.9654	11.9845	11.9911	11.9992
Standard Deviation	0.0832	0.0835	0.0834	0.0834	0.0833
Confidence 90%	0.7005%	0.5277%	0.3428%	0.3434%	0.1382%
Population					
Mean	33.0432	31.0452	31.0903	31.0672	31.3333
Standard Deviation	17.9811	17.3725	17.4608	17.6523	17.9985
Confidence 90%	6.4335%	4.3699%	2.7011%	2.1758%	0.7074%
Turnaround Time					
Mean	2.7500	2.5941	2.5942	2.5908	2.6113
Standard Deviation	1.6083	1.5604	1.5585	1.5660	1.5811
Confidence 90%	7.2543%	4.7214%	2.6022%	2.2886%	0.6015%
Counter					
Served Requests	120129	239309	599226	1199107	11999249
Timeouts	13428	21574	55163	111481	1154474
Ratio	11.1780%	9.0151%	9.2057%	9.2970%	9.6212%
CPU Time					
Seconds	46.46	96.82	229.78	470.77	4248.86

Table D.4: Simulation results for ParcelSink Workflow with Timeouts. Requests older than 5 seconds are terminated, thus statistics on population and turnaround time are influenced.

D.2 Curve Estimation Results

Table D.5: Estimated parameters for upper arrival curve $\alpha^U = \min_i(\gamma_{r_i,b_i})$

Curve	Segment 4		Segment 3		Segment 2		Segment 1	
	b_4	r_4	b_3	r_3	b_2	r_2	b_1	r_1
1	1.0000	705.582	1.57781	297.893	4.98795	0.197502	74.094	0.0627804
2	-	-	1.0000	2065.46	2.14333	4.28695	82.5346	0.0649811
3	1.0000	537.008	1.98774	6.58638	16.6349	0.0777795	98.7674	0.0636619
4	-	-	1.0000	978.786	2.7071	1.09763	58.8953	0.0640002
5	-	-	1.0000	385.448	3.87977	0.258886	56.8743	0.0639058
6	-	-	1.0000	727.938	3.71806	0.374306	72.6915	0.06342046
7	-	-	1.0000	2572.86	2.84389	0.699311	107.222	0.0640278
8	**-**	**-**	**1.0000**	**1229.6**	**3.1065**	**0.593122**	**90.6407**	**0.0655885**
9	-	-	1.0000	1125.13	3.31477	0.767517	55.816	0.0639301
10	-	-	1.0000	230.345	2.84519	0.874648	74.8787	0.064284

Table D.6: Calibration findings for upper arrival curve α^U

Curve	Intersected Area		Intersection Time		Mean Squared Error		Overall
	mean	std.	mean	std.	mean	std.	mean
1	0.00322	0.00468	0.16297	0.14161	0.00170	0.00023	0.05597
2	0.00001	0.00003	0.00326	0.00978	0.00302	0.00069	0.00210
3	0.00045	0.00116	0.03173	0.06840	0.00235	0.00060	0.01151
4	0.00100	0.00225	0.06521	0.11510	0.00134	0.00032	0.02252
5	0.00124	0.00263	0.07732	0.12668	0.00119	0.00028	0.02658
6	0.00147	0.00292	0.08164	0.12112	0.00156	0.00028	0.02822
7	0.00014	0.00041	0.01565	0.04464	0.00350	0.00064	0.00643
8	**0.00000**	**0.00000**	**0.00000**	**0.00000**	**0.00401**	**0.00087**	**0.00134**
9	0.00124	0.00264	0.07786	0.12754	0.00120	0.00028	0.02677
10	0.00033	0.00091	0.02735	0.06549	0.00211	0.00048	0.00993

Table D.7: Estimated parameters for upper delay curve $\Psi^U = \min_i(\gamma_{r_i,b_i})$

Curve	Segment 4		Segment 3		Segment 2		Segment 1	
	b_4	r_4	b_3	r_3	b_2	r_2	b_1	r_1
1	-	-	28.4966	693.853	93.7272	2.19688	902.419	0.39801
2	**-**	**-**	**35.4441**	**215.749**	**106.168**	**2.47372**	**645.733**	**0.42198**
3	28.2806	26.7737	36.6414	9.63921	246.768	0.55034	950.943	0.40987
4	-	-	30.6942	45.1363	82.0811	1.57013	461.898	0.40217
5	-	-	31.4197	58.2452	116.512	1.36264	539.618	0.40394
6	-	-	34.9433	189.499	93.1401	3.17001	958.717	0.40184
7	-	-	29.2407	438.891	100.577	2.66975	912.093	0.40642
8	-	-	29.8506	36.4559	55.6746	2.56809	980.164	0.41886
9	-	-	39.1513	56.6582	392.27	1.08789	775.873	0.40641
10	-	-	29.9802	89.2779	41.8839	5.22601	592.197	0.40665

Table D.8: Calibration findings for upper delay curve Ψ^U

Curve	Intersected Area		Intersection Time		Mean Squared Error		Overall
	mean	std.	mean	std.	mean	std.	mean
1	0.00224	0.00372	0.08412	0.11258	0.00438	0.00064	0.03024
2	**0.00000**	**0.00001**	**0.00072**	**0.00217**	**0.00554**	**0.00151**	**0.00209**
3	0.00006	0.00015	0.00917	0.02369	0.00647	0.00163	0.00523
4	0.00480	0.00662	0.17138	0.16479	0.00171	0.00025	0.05930
5	0.00292	0.00468	0.11699	0.14593	0.00221	0.00043	0.04071
6	0.00097	0.00191	0.04912	0.08179	0.00535	0.00092	0.01848
7	0.00038	0.00086	0.02915	0.05894	0.00549	0.00110	0.01168
8	0.00000	0.00000	0.00000	0.00000	0.00847	0.00180	0.00282
9	0.00076	0.00157	0.04478	0.07918	0.00423	0.00090	0.01659
10	0.00155	0.00284	0.07586	0.11574	0.00287	0.00065	0.02676

Table D.9: Estimated parameters for lower arrival curve $\alpha^L = \max_i(\beta_{R_i,T_i})$

Curve	Segment 4		Segment 3		Segment 2		Segment 1	
	T_4	R_4	T_3	R_3	T_2	R_2	T_1	R_1
1	**-**	**-**	**-4.5749**	**0.032337**	**-8.71032**	**0.0532404**	**-71.7494**	**0.0627804**
2	-	-	-	-	-5.62666	0.0388455	-70.8507	0.0649811
3	-	-	-	-	-	-	-8.88804	0.0636619
4	-	-	-	-	-9.21112	0.049689	-50.631	0.0640002
5	-	-	-	-	-5.53154	0.0427862	-30.944	0.0639058
6	-	-	-	-	-6.62556	0.050524	-60.3973	0.0634204
7	-	-	-	-	0	0	-100.099	0.0640278
8	-	-	-	-	-5.90274	0.0374546	-62.6482	0.0655885
9	-	-	-	-	-5.29537	0.0416632	-30.6702	0.0639301
10	-	-	-	-	-5.53705	0.0397594	-27.7046	0.064284

Table D.10: Calibration findings for lower arrival curve α^L

Curve	Intersected Area		Intersection Time		Mean Squared Error		Overall
	mean	std.	mean	std.	mean	std.	mean
1	**0.00000**	**0.00000**	**0.01712**	**0.00875**	**0.00485**	**0.00086**	**0.00732**
2	0.00582	0.01329	0.06143	0.04403	0.00315	0.00039	0.02347
3	0.03074	0.03079	0.44114	0.08487	0.00161	0.00019	0.15783
4	0.00194	0.00572	0.08997	0.08470	0.00297	0.00035	0.03163
5	0.00952	0.01657	0.16843	0.09080	0.00210	0.00018	0.06001
6	0.00005	0.00015	0.04105	0.04243	0.00303	0.00039	0.01471
7	0.00000	0.00000	0.00343	0.00882	0.02472	0.00175	0.00938
8	0.02921	0.02600	0.05982	0.01807	0.00289	0.00042	0.03064
9	0.01001	0.01696	0.16986	0.08850	0.00208	0.00018	0.06065
10	0.02602	0.03181	0.17538	0.07680	0.00204	0.00026	0.06781

Table D.11: Estimated parameters for lower delay curve $\Psi^L = \max_i(\beta_{R_i,T_i})$

Curve	Segment 4		Segment 3		Segment 2		Segment 1	
	T_4	R_4	T_3	R_3	T_2	R_2	T_1	R_1
1	**-**	**-**	**0**	**0**	**-29.1075**	**0.1995**	**-874.5840**	**0.3980**
2	-	-	-	-	0	0	-597.7840	0.4220
3	-	-	-	-	-33.4493	0.2361	-115.061	0.4099
4	-	-	-	-	-7.2791	0.0389	-413.749	0.4022
5	-	-	-	-	-7.0567	0.0543	-219.446	0.4039
6	-	-	-28.7016	0.2146	-434.5550	0.3923	-811.9700	0.4018
7	-	-	-	-	-15.1939	0.1199	-736.2410	0.4064
8	-	-	-	-	0	0	-721.4460	0.4189
9	-	-	-	-	-25.8145	0.2058	-275.4910	0.4064
10	-	-	-	-	-18.6053	0.1381	-292.8500	0.4066

Table D.12: Calibration findings for lower delay curve Ψ^L

Curve	Intersected Area		Intersection Time		Mean Squared Error		Overall
	mean	std.	mean	std.	mean	std.	mean
1	**0.00000**	**0.00000**	**0.00262**	**0.00244**	**0.01512**	**0.00311**	**0.00591**
2	0.03070	0.02677	0.08190	0.02620	0.02096	0.00207	0.04452
3	0.07368	0.06521	0.30758	0.09784	0.00210	0.00036	0.12779
4	0.00172	0.00515	0.07899	0.09361	0.01516	0.00230	0.03196
5	0.01223	0.02128	0.20450	0.09411	0.00530	0.00116	0.07401
6	0.00000	0.00000	0.01990	0.02243	0.00792	0.00210	0.00927
7	0.00033	0.00100	0.02960	0.04502	0.01066	0.00235	0.01353
8	0.00805	0.01364	0.04922	0.03551	0.02656	0.00241	0.02794
9	0.01389	0.02238	0.17251	0.09490	0.00297	0.00066	0.06312
10	0.01239	0.02043	0.15533	0.09306	0.00365	0.00081	0.05712

List of Figures

List of Tables

Bibliography

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: concepts, architectures and applications.* Springer, 2003.

[2] E. Altman, K. Avrachenkov, and C. Barakat. TCP Network Calculus: The case of large delay-bandwidth product. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 417–426. IEEE, 2002.

[3] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web services agreement specification (ws-agreement). In *Global Grid Forum*, volume 2, 2004.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of Cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721672.

[5] M. Arns. *Approximative Verfahren auf erweiterten fork/join-Warteschlangennetzen zur Analyse von Logistiknetzen.* PhD thesis, Universität Dortmund, 2006.

[6] F. Baccelli, G. Cohen, G. Olsder, and J. Quadrat. *Synchronization and Linearity.* Wiley New York, 1992.

[7] S. Balsamo, V. de Nitto Personé, and R. Onvural. *Analysis of queueing networks with blocking*, volume 31. Springer, 2001.

[8] J. Banks, J. S. Carson II, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation.* Pearson Prentice Hall, Upper Saddle River, US, fourth edition, 2005.

[9] A. P. Barros and M. Dumas. The rise of web service ecosystems. *IT professional*, 8 (5):31–37, 2006. ISSN 1520-9202.

[10] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM (JACM)*, 22(2):248–260, 1975.

[11] F. Bause and P. S. Kritzinger. *Stochastic Petri Nets.* Vieweg, Wiesbaden, 2nd edition, Aug. 2002. ISBN 3-528-15535-3.

[12] F. Bause, H. Beilner, and P. Kemper. Modellierung und Analyse von Logistiknetzwerken mit Prozeßketten. In *ASIM-Symposium Simulationstechnik*, pages 63–67, September 2000.

[13] F. Bause, H. Beilner, and M. Schwenke. Semantik des ProC/B-Paradigmas. Technical Report 03001, Sonderforschungsbereich 559 Modellierung großer Netze in der Logistik, 2004.

[14] F. Bause, P. Buchholz, and C. Tepper. The ProC/B-approach: From informal descriptions to formal models. In *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Method (ISoLA)*, 2004.

[15] F. Bause, P. Buchholz, J. Kriege, and S. Vastag. Simulating process chain models with OMNeT++. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools)*, Marseille, 2008.

[16] F. Bause, P. Buchholz, J. Kriege, and S. Vastag. A framework for simulation models of service-oriented architectures. In S. Kounev, I. Gorton, and K. Sachs, editors, *LLNCS*, volume 5119, Darmstadt, Germany, June 27-28 2008. SPEC International Performance Evaluation Workshop 2008, SIPEW 2008, Springer. doi: 10.1007/978-3-540-69814-2_14.

[17] F. Bause, P. Buchholz, J. Kriege, and S. Vastag. Simulation based validation of quantitative requirements in service oriented architectures. In M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, editors, *Proceedings of the 2009 Winter Simulation Conference*, pages 1015–1026. IEEE, 2009.

[18] F. Bause, J. Kriege, and S. Vastag. Efficient validation of process-based simulation models. *Simulation News Europe (SNE) Special Issue on Quality Aspects in Modeling and Simulation*, 19(2):30–38, 2009. ISSN 0929-2268.

[19] F. Bause, P. Buchholz, J. Kriege, and S. Vastag. A simulation environment for hierarchical process chains based on OMNeT++. *Simulation*, 86(5-6):291–309, 2010.

[20] F. Bause, P. Gerloff, and J. Kriege. ProFiDo - a toolkit for fitting input models. In B. Müller-Clostermann, K. Echtle, and E. P. Rathgeb, editors, *Proceedings of the 15th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems and Dependability and Fault Tolerance (MMB & DFT 2010)*, volume 5987 of *LNCS*, pages 311–314. Springer, 2010. doi: 10.1007/978-3-642-12104-3_25.

[21] H. Beilner, J. Mäter, and C. Wysocki. The hierarchical evaluation tool HIT. In *Short Papers and Tool Descriptions of 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 6–9, 1994.

[22] M. Bertoli, G. Casale, and G. Serazzi. Jmt: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/1530873.1530877.

[23] P. Bianco, G. Lewis, and P. Merson. Service level agreements in service-oriented architecture environments. Technical report, DTIC Document, 2008.

[24] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475: An architecture for differentiated services, dec 1998. URL `ftp://ftp.internic.net/rfc/rfc2475.txt`. Status: PROPOSED STANDARD.

[25] G. Bolch and H. Riedel. *Leistungsbewertung von Rechensystemen mittels analytischer Warteschlangenmodelle*. Teubner, Stuttgart, 1989. ISBN 3-519-02279-6.

[26] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains - Modelling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, Inc, 1998. ISBN 0-471-19366-6.

[27] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[28] A. Bouillard and E. Thierry. An algorithmic toolbox for Network Calculus. *Discrete Event Dynamic Systems*, 18(1):3–49, 2008.

[29] A. Bouillard, B. Gaujal, S. Lagrange, and E. Thierry. Optimal routing for end-to-end guarantees using network calculus. *Performance Evaluation*, 65(11-12):883–906, 2008.

[30] A. Bouillard, L. Jouhet, and E. Thierry. Service curves in network calculus: dos and don'ts. Technical Report 7094, INRIA, Nov. 2009.

[31] M. Boyer. NC-Maude: a rewriting tool to play with Network Calculus. In *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 137–151. Springer, 2010.

[32] L. Breuer and D. Baum. *An introduction to queueing theory and matrix-analytic methods*. Springer Verlag, Heidelberg, 2005.

[33] P. Buchholz and U. Clausen, editors. *Große Netze der Logistik. Die Ergebnisse des Sonderforschungsbereichs 559*. Springer, 2009.

[34] P. Buchholz, J. Dunkel, B. Müller-Clostermann, M. Sczittnick, and S. Zäske. *Quantitative Systemanalyse mit Markovschen Ketten*, volume 8 of *Teubner Texte zur Informatik*. Teubner, 1994. ISBN 3-8154-2056-3.

[35] C. Cassandras. *Discrete event systems: modeling and performance analysis*, volume 2. Irwin, 1993.

[36] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the 6th Design, Automation and Test in Europe (DATE)*, pages 190–195, 2003.

[37] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister. Performance evaluation of network processor architectures: Combining simulation with analytical estimation. *Computer Networks*, 41(5):641–665, 2003.

[38] C. Chang. Performance guarantees in communication networks. *European Transactions on Telecommunications*, 12(4):357–358, 2001.

[39] C.-T. Chen. *Linear System Theory and Design.* Oxford University Press, Inc., third edition, 1999. ISBN 0195117778.

[40] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. *W3C Candidate Recommendation,*, 2001. URL `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[41] D. R. Cox. A use of complex probabilities in the theory of stochastic processes. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 51, pages 313–319. Cambridge Univ Press, 1955.

[42] R. Cruz. A calculus for network delay, part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[43] R. Cruz. A calculus for network delay, part II: Network analysis. *, IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[44] A. Duda and T. Czachórski. Performance evaluation of fork and join synchronization primitives. *Acta Informatica*, 24(5):525–553, 1987.

[45] J. Eckert, K. Pandit, N. Repp, R. Berbner, and R. Steinmetz. Worst-case performance analysis of Web service workflows. In *Proceedings of the 9th International Conference on Information Integration and Web-based Application & Services*, 2007.

[46] J. Eckert, S. Schulte, N. Repp, R. Berbner, and R. Steinmetz. Queuing-based capacity planning approach for Web service workflows using optimization algorithms. In *Digital Ecosystems and Technologies, 2008. DEST 2008. 2nd IEEE International Conference on*, pages 313–318. IEEE, 2008.

[47] M. Fidler and S. Recker. Conjugate network calculus: A dual approach applying the legendre transform. *Computer Networks*, 50(8):1026–1039, 2006.

[48] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica. Above the clouds: A Berkeley view of Cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28, 2009.

[49] T. Frey and M. Bossert. *Signal- und Systemtheorie*, volume 2. Springer, 2008. ISBN 978-3-8351-0249-1.

[50] N. Gollan, F. A. Zdarsky, I. Martinovic, and J. B. Schmitt. The DISCO network calculator. In *Proceedings of the 14th GI/ITG Conference - Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB), 2008*, pages 291–293. VDE Verlag, Mar. 2008. ISBN 978-3-8007-3090-2.

[51] B. Hayes. Cloud computing. *Communications of the ACM*, 51(7):9–11, 2008. ISSN 0001-0782.

[52] O. Heckmann, F. Rohmer, and J. Schmitt. The token bucket allocation and reallocation problems (MPRASE token bucket). Technical Report TR-KOM-2001-12, Technische Universität Darmstadt, Multimedia Communications Lab (KOM), Dec. 2001.

[53] J. Huang. Simulative Bewertung von ProC/B Modellen. Master's thesis, Universität Dortmund, Mar. 2006.

[54] R. Jain. *The art of computer systems performance analysis*, volume 182. John Wiley & Sons New York, 1991.

[55] Y. Jiang. Network calculus and queueing theory: Two sides of one coin. *ICST ValueTools*, 2009.

[56] Y. Jiang and Y. Liu. *Stochastic network calculus.* Springer-Verlag New York Inc, 2008.

[57] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, et al. Web services business process execution language version 2.0. *OASIS Standard*, 11, 2007.

[58] K. Kant and M. M. Srinivasan. *Introduction to computer system performance evaluation.* McGraw-Hill, 1992.

[59] J. Kaufmann. Entwicklung und Realisierung einer Anbindung des INET-Frameworks an Prozesskettenmodelle. Master's thesis, TU Dortmund, May 2009.

[60] P. Kemper and C. Tepper. A Petri net approach to debug simulation models of logistic networks. In *Proceedings of the 5th Mathmod Vienna*, volume 30, 2006.

[61] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International joint Conference on artificial intelligence*, volume 14, pages 1137–1145, 1995.

[62] J. Kriege. *Fitting Simulation Input Models for Correlated Traffic Data.* PhD thesis, Technische Universität Dortmund, Fakultät für Informatik, 2012.

[63] J. Kriege and P. Buchholz. Simulating stochastic processes with OMNeT++. In *Proceedings of the 4th International OMNeT++ Workshop (OMNeT++ 2011).* ICST, 2011. doi: 10.4108/icst.simutools.2011.245512.

[64] J. Kriege and P. Buchholz. Traffic modeling with a combination of phase-type distributions and ARMA processes. In *Proceedings of the Winter Simulation Conference (WSC) 2012*, 2012. doi: 10.1109/WSC.2012.6465299.

[65] A. Kuhn. *Prozessketten in der Logistik - Entwicklungstrends und Umsetzungsstrategien.* Verlag Praxiswissen, Dortmund, 1995.

[66] S. Künzli and L. Thiele. Generating event traces based on arrival curves. In *Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB), 2006 13th GI/ITG Conference*, pages 1–18. VDE, 2006.

[67] S. Künzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *Design, Automation and Test in Europe, 2006. DATE'06. Proceedings*, volume 1, pages 1–6. IEEE, 2006.

[68] J. F. Kurose and K. W. Ross. *Computer networking*, volume 2. Addison Wesley, 2001.

[69] D. D. Lamanna, J. Skene, and W. Emmerich. SLAng: A language for defining service level agreements. In *Proceedings of the Ninth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003), 2003*, pages 100–106. IEEE Computer Society, 2003. doi: 10.1109/FTDCS.2003.1204317.

[70] A. M. Law and W. D. Kelton. *Simulation modeling and analysis.* McGraw-Hill Higher Education, 2000.

[71] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitative system performance: computer system analysis using queueing network models.* Prentice-Hall, Inc., 1984.

[72] J.-Y. Le Boudec and D.-C. Tomozei. Demand response using service curves. In *Innovative Smart Grid Technologies (ISGT Europe), 2011 2nd IEEE PES International Conference and Exhibition on*, pages 1–8. IEEE, 2011.

[73] J.-Y. Le Boudec and D.-C. Tomozei. A demand-response calculus with perfect batteries. In *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, pages 273–287. Springer, 2012.

[74] M. Li, W. Zhao, Q. Li, S. Chen, and A. Xu. Sufficient condition for min-plus deconvolution to be closed in the service-curve set in computer networks. In *Proceedings of the WSEAS International Conference on Mathematics and Computers in Science and Engineering.* World Scientific and Engineering Academy and Society, 2008.

[75] J. Liebeherr, M. Fidler, and S. Valaee. A min-plus system interpretation of bandwidth estimation. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1127–1135. IEEE, 2007.

[76] J. Liebeherr, M. Fidler, and S. Valaee. A system-theoretic approach to bandwidth estimation. *Networking, IEEE/ACM Transactions on*, 18(4):1040–1053, 2010.

[77] J. Little. A proof for the queuing formula: L= λw. *Operations research*, 9(3):383–387, 1961.

[78] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck. Web Service Level Agreement (WSLA) language specification. http://www.research.ibm.com/wsla, 2003.

[79] D. Menascé, E. Casalicchio, and V. Dubey. On optimal service selection in service oriented architectures. *Performance Evaluation*, 67(8):659–675, 2010.

[80] D. A. Menascé and V. A. F. Almeida. *Capacity planning for Web performance: metrics, models, and methods.* Prentice-Hall, Inc., 1998.

[81] D. A. Menascé and V. A. F. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning.* Prentice Hall, 2000.

[82] D. A. Menascé and V. A. F. Almeida. *Capacity Planning for Web Services: metrics, models, and methods.* Prentice Hall Upper Saddle River, 2002.

[83] D. A. Menascé and V. Dubey. Utility-based QoS brokering in service oriented architectures. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 422–430. IEEE, 2007.

[84] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Performance by design: computer capacity planning by example.* Prentice Hall, Upper Saddle Riber, NJ, 2004. ISBN 0-13-090673-5.

[85] D. A. Menascé, H. Ruan, and H. Gomaa. QoS management in service-oriented architectures. *Performance evaluation*, 64(7-8):646–663, 2007.

[86] C. Molina-Jimenez, J. Pruyne, and A. van Moorsel. The role of agreements in it management software. *Architecting Dependable Systems III*, pages 36–58, 2005.

[87] V. Muthusamy, H. Jacobsen, T. Chau, A. Chan, and P. Coulthard. SLA-driven business process management in SOA. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 86–100. ACM, 2009.

[88] R. Nelson and A. N. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *IEEE Transactions on Computers*, 37(6):739–743, 1988.

[89] OASIS. UDDI online community, 2011. URL `http://uddi.xml.org/`.

[90] K. Pandit. *Quality of service performance analysis based on network calculus.* PhD thesis, TU Darmstadt, 2006.

[91] M. P. Papazoglou and W.-J. van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415, 2007.

[92] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1:344–357, 1993.

[93] A. Paschke and E. Schnappinger-Gerull. A categorization scheme for SLA metrics. *Service Oriented Electronic Commerce*, 80:25–40, 2006.

[94] C. Peltz. Web services orchestration and choreography. *Computer*, pages 46–52, 2003. ISSN 0018-9162.

[95] B. Rimal, E. Choi, and I. Lumb. A taxonomy and survey of Cloud computing systems. In *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pages 44–51. IEEE, 2009.

[96] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Pearson Higher Education, 2nd edition, 2004. ISBN 0321245628.

[97] J. Schmitt and I. Martinovic. Demultiplexing in network calculus-a stochastic scaling approach. In *Quantitative Evaluation of Systems, 2009. QEST'09. Sixth International Conference on the*, pages 217–226. IEEE, 2009.

[98] J. Schmitt and U. Roedig. Sensor Network Calculus – a framework for worst case analysis. *Distributed Computing in Sensor Systems*, pages 141–154, 2005.

[99] J. Schmitt and F. Zdarsky. The disco network calculator: a toolbox for worst case analysis. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, pages 8–es. ACM, 2006.

[100] S. Shenker and J. Wroclawski. RFC 2215. *General Characterization Parameters for Integrated Service Network Elements*, 1997.

[101] W. M. Sofack and M. Boyer. Generalisation of GPS and P-GPS in Network Calculus. In *Proceedings of the 9th IEEE International Workshop on Factory Communication Systems (WFCS), 2012*, pages 169–172. IEEE, 2012.

[102] S. H. Stefan König and T. Eymann. Socio-economic mechanisms to coordinate the Internet of Services: The simulation environment SimIS. *Journal of Artificial Societies and Social Simulation*, 13(2):6, 2009.

[103] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, 2000.

[104] L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert. Embedded software in network processors – models and algorithms. In *Embedded Software*, pages 416–434. Springer, 2001.

[105] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. *Network Processor Design: Issues and Practices*, 1: 55–89, 2002.

[106] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. A framework for evaluating design tradeoffs in packet processing architectures. In *Proceedings of the 39th annual Design Automation Conference*, pages 880–885. ACM, 2002.

[107] P. Thiran and J.-Y. Le Boudec. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet.* Number 2050 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, May 2004. ISBN 3-540-42184-X.

[108] L. Touseau, D. Donsez, and W. Rudametkin. Towards a SLA-based approach to handle service disruptions. In *Proceedings of the IEEE International Conference on Services Computing (SCC'08), 2008*, volume 1, pages 415–422. IEEE, 2008.

[109] J. J. M. Trienekens, J. J. Bouman, and M. van der Zwan. Specification of service level agreements: Problems, principles and practices. *Software Quality Journal*, 12 (1):43–57, 2004. doi: 10.1023/B:SQJO.0000013358.61395.96.

[110] A. Undheim, Y. Jiang, and P. J. Emstad. Network Calculus approach to router modeling with external measurements. In *Communications and Networking in China, 2007. CHINACOM'07. Second International Conference on*, pages 276–280. IEEE, 2007.

[111] W. Van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–76, 2003.

[112] A. Varga. OMNeT++. *Modeling and Tools for Network Simulation*, pages 35–59, 2010.

[113] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pages 1–10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[114] E. Varki. Mean value technique for closed fork-join networks. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, pages 103–112, New York, NY, USA, 1999. ACM. ISBN 1-58113-083-X. doi: 10.1145/301453.301484.

[115] E. Varki, A. Merchant, J. Xu, and X. Qiu. An integrated performance model of disk arrays. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003 (MASCOTS 2003)*, pages 296–305. IEEE, 2003.

[116] S. Vastag. ProC/B for networks: Integrated INET models. In Klaus and R. E. Bruno Müller-Clostermann AND Echtle, editors, *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, volume 5987 of *Lecture Notes in Computer Science*, pages 315–318. Springer Berlin / Heidelberg, 2010. doi: 10.1007/978-3-642-12104-3_26.

[117] S. Vastag. Modeling quantitative requirements in SLAs with Network Calculus. In *Proceedings of the 5th International ICST Conference on Performance Evaluation Methologies and Tools (ValueTools)*, ENS, Cachan, France, May 17th-20th 2011 2011. ICST.

[118] S. Vastag. Arrival and delay curve estimation for SLA Calculus. *Proceedings of the 2012 Winter Simulation Conference*, Dec. 2012.

[119] S. Vastag. A calculus for SLA delay properties. In J. Schmitt, editor, *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, volume 7201 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-28539-4. doi: 10.1007/978-3-642-28540-0_6.

[120] M. Völker, H. Beilner, F. Bause, P. Kemper, and M. Fischer. The ProC/B toolset for modelling and analysis of process chains. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *TOOLS 2002*, number 2324 in Lecture Notes in Computer Science, pages 51–70. Springer Verlag, 2002.

[121] W3C. Web services activities, 2011. URL `http://www.w3.org/2002/ws/`.

[122] E. Wandeler. *Modular performance analysis and interface-based design for embedded real-time systems*. PhD thesis, Swiss federal institute of technology Zürich, 2006.

[123] E. Wandeler and L. Thiele. Real-Time Calculus (RTC) toolbox, 2006. URL `http://www.mpa.ethz.ch/Rtctoolbox`.

[124] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, 2006.

[125] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Terminology for policy-based management. Technical report, RFC 3198, Nov. 2001.

[126] R. Yassin Kassab and A. van Moorsel. Formal mapping of WSLA contracts on stochastic models. *Computer Performance Engineering*, pages 117–132, 2011.

[127] L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of Cloud computing. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE, 2008.

[128] Q. Zhu. Beschreibung von ProC/B-Modellen zur simulativen Bewertung. Master's thesis, Universität Dortmund, Mar. 2006.