The DiscoDNC v2 – A Comprehensive Tool for Deterministic Network Calculus

Steffen Bondorf
University of Kaiserslautern
Distributed Computer Systems Lab (DISCO)
Germany
bondorf@cs.uni-kl.de

Jens B. Schmitt
University of Kaiserslautern
Distributed Computer Systems Lab (DISCO)
Germany
jschmitt@cs.uni-kl.de

ABSTRACT
In this paper, we present the Disco Deterministic Network Calculator v2 – our continuation in the effort to provide open-source tool support for worst-case performance analysis. The major achievement of this release is the provision of a well-documented network calculus implementation that enables for straight-forward application as well as easy extensibility. Apart from a fair amount of bug fixes that improve the tightness of the derived bounds, our efforts resulted in two main differences to the previous versions: the first is the rigorous modularization of network calculus analyses into their distinct components and the second is a comprehensive set of functional tests for our tool.

Categories and Subject Descriptors
C.2 [**Computer Systems Organization**]: Computer Communication Networks; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling Techniques*

General Terms
Performance

Keywords
Network Calculus

1. INTRODUCTION
We believe that in the provision of tool support for network calculus it matters to open up the usage of our offerings in a multitude of application scenarios. Therefore we decided to use the Java programming language to implement the core network calculus concepts as well as the analyses based on them [12]. In the second version of the Disco Deterministic Network Calculator (DiscoDNC)[1], we continue to pursue this goal by tackling further obstacles on the way to a tool commonly adopted and refined. Our efforts have been twofold: The first aim was to modularize our codebase along the lines of the network calculus theory. We provide a library consisting of the implementation of the network calculus framework – from simple curves to complex analyses. Thus, we allow for easy adaptability and extensibility of our toolbox – a goal we already achieved with an in-development version [7]. The second is to provide the classes necessary to use DiscoDNC v2 as a stand-alone tool. This allowed us to check our network calculus implementation against network configurations whose bounds were manually verified. In order to do so, we have created an extensive, well-documented set of functional tests that we provide alongside the source code. Further, we have re-licensed the entire code base under the GNU Lesser General Public License (LGPL) version 2.1 and the test cases under a permissive Creative Commons license to facilitate usage of both. The progress of our endeavor can be traced online [2] and in this paper, where we present the analytical background of network calculus we implemented as well as the design and the use of the DiscoDNC v2.

[1]Formerly known as *DISCO Network Calculator* we decided to rename it to Disco *Deterministic* Network Calculator to contrast with our other offering, the Disco *Stochastic* Network Calculator (DiscoSNC) [3, 6].

Outline. We structured the remainder of this paper according to the code base's modularization: Section 2 introduces the worst-case performance evaluation at network calculus' foundation and in Section 3 we present the derived concepts and operations available in our tool. Section 4 depicts the classes to create a network configuration to perform one of the analyses of Section 5 on. Section 6 presents the set of functional tests we created for our tool and Section 7 exemplifies the tool usage. Section 8 concludes the paper.

2. PERFORMANCE EVALUATION
In this section, we present the basic model for performance evaluation as used by network calculus. It is based on the concepts of data flows and their transformations, both described by functions cumulatively counting data. These curves belong to the following set

Definition 1. [`unikl.disco.curves.`$Curve$`.java`]
The set $\mathcal{F}_0$ of non-negative wide-sense increasing functions is defined by

$$\mathcal{F}_0 = \left\{ f : \mathbb{R} \to \mathbb{R}^+ \mid f(0) = 0,\ \forall s \leq t : f(s) \leq f(t) \right\},$$
$$\mathbb{R}^+ := [0, +\infty].$$

The class `unikl.disco.curves.Curve.java` provides methods to create piecewise linear functions of $\mathcal{F}_0$.

In particular, we are interested in the functions $A(t)$ and $A'(t)$ cumulatively counting a flow's data put into a system $\mathcal{S}$ up until time t and put out from $\mathcal{S}$ until t. Both functions are naturally assumed to be in $\mathcal{F}_0$. Throughout the paper, we assume that systems are abstractly characterized by their input and output functions; both are assumed to be left-continuous.

These curves allow us to define worst-case performance characteristics of flows as follows:

Definition 2. (Backlog and Delay) Assume a flow with input function A traverses a system $\mathcal{S}$ and results in the output function A'. The backlog *of the flow at time t is defined as*

$$B(t) = A(t) - A'(t).$$

The (virtual) delay *for a data unit arriving at $\mathcal{S}$ at time t is defined as*

$$D(t) = \inf\left\{\tau \geq 0 \mid A(t) \leq A'(t+\tau)\right\}.$$

3. NETWORK CALCULUS FRAMEWORK

Network calculus establishes its worst case semantics by defining the functions bounding data flows over the duration of an observation d instead of the time instance t.

Definition 3. [`unikl.disco.curves.`*ArrivalCurve*`.java`] *Given a flow with input function A, a function $\alpha \in \mathcal{F}_0$ is an arrival curve for A iff*

$$\forall\, 0 \leq d \leq t\, :\, A(t) - A(t-d) \leq \alpha(d)$$

The definition is independent from the actual time t the measurement takes place. It is used to express the worst-case data arrival within any duration of length d. We make use of $(\wedge, +)$-algebra to take advantage of these semantics of curves with the following operations. A detailed treatment of $(\wedge, +)$-algebra and network calculus can be found in [5], [8], and [10], respectively.

The operations presented in the following are closed in $\mathcal{F}_0$ and in the set of ultimately affine piecewise linear curves that is used in the DiscoDNC (see [9]). Their explicit solutions are implemented in the packages and classes referenced in brackets.

Definition 4. [`unikl.disco.`*minplus*] *The $(\wedge, +)$-algebraic convolution and deconvolution of two functions $f, g \in \mathcal{F}_0$ are defined as*

$$\text{convolution: } (f \otimes g)(d) = \inf_{0 \leq s \leq d} \{f(d-s) + g(s)\},$$
$$\text{deconvolution: } (f \oslash g)(d) = \sup_{u \geq 0} \{f(d+u) - g(u)\}.$$

Convolution and deconvolution enable us to characterize flow transformations using curves.

Definition 5. [`unikl.disco.curves.`*ServiceCurve*`.java`] *If the service provided by a system $\mathcal{S}$ for a given input function A results in an output function A' we say that $\mathcal{S}$ offers a service curve β iff*

$$A' \geq A \otimes \beta.$$

A number of systems fulfill, however, a stricter definition of service curve [8], which is particularly useful as it permits certain derivations that are not feasible under the more general minimum service curve model.

Definition 6. [`unikl.disco.curves.`*ServiceCurve*`.java`] *Let $\beta \in \mathcal{F}_0$. System $\mathcal{S}$ offers a* strict *service curve β to a flow if, during any backlogged period of duration d, the output of the flow is at least equal to $\beta(d)$.*

Strictness is implicitly checked in the DiscoDNC by applying the permitted derivations only.

Having curves for different purposes, we can give the performance bounds of Definition 2 for network calculus.

Theorem 1. [`unikl.disco.nc.`*BacklogBound*`.java`] [`unikl.disco.nc.`*DelayBound*`.java`] *Consider a system $\mathcal{S}$ that offers a service curve β. Assume a flow f traversing the system which has an arrival curve α. Then we obtain the following performance bounds:*

$$\text{backlog: } \forall t \in \mathbb{R}^+ :\ B(t) \leq (\alpha \oslash \beta)(0)$$
$$\text{delay: } \forall t \in \mathbb{R}^+ :\ D(t) \leq \inf\{d \geq 0 \,|\, (\alpha \oslash \beta)(-d) \leq 0\}$$

Graphically, the backlog bound is the maximum vertical deviation between arrival curve α and service curve β. Similarly, the delay bound is the maximum horizontal deviation between α and β. Note, that the delay bound only holds for systems that preserve the order of packets within a flow – a property we call *FIFO per micro-flow*. Non-FIFO per micro-flow systems require a different treatment [11].

In addition to the performance bounds derivation, the $(\wedge, +)$-deconvolution also enables to derive an arrival curve that bounds $A'(t)$ at the output of a system.

Theorem 2.[`unikl.disco.minplus.`*Deconvolution*`.java`] *(Output Arrival Curve) Assume a flow f has an arrival curve α and consider f traversing the system $\mathcal{S}$ offering a service curve β. After being transformed by $\mathcal{S}$, i.e., at the system's output, f is bounded by the arrival curve*

$$\alpha'(d) = \left(\alpha \dot{\oslash} \beta\right)(d) = \begin{cases} 0 & \text{if } d = 0 \\ (\alpha \oslash \beta)(d) & \text{otherwise} \end{cases}.$$

Note, that the deconvolution as given in Definition 4 does not guarantee $(\alpha \oslash \beta)(0) = 0$ and is thus not closed in $\mathcal{F}_0$. While this property allows for backlog bounding, a slight augmentation of the operation is needed in our tool in order to enforce closure and fulfill the arrival curve definition.

THEOREM 3. [unikl.disco.nc.*LeftOverService*.java] *Consider a system $\mathcal{S}$ that offers a strict service curve β and that serves two input flows, f_1 and f_2 with arrival curves α^{f_1} and α^{f_2}, respectively. The minimum service f_1 is guaranteed to receive is lower bounded by the so-called left-over service curve $\beta^{l.o.f_1}$.*

In case of arbitrary multiplexing of flows crossing $\mathcal{S}$, it holds that

$$\beta^{l.o.f_1} = \beta \ominus^{ARB} \alpha^{f_2}$$

with $\left(\beta \ominus^{ARB} \alpha\right)(t) = \sup_{0 \leq s \leq t} (\beta - \alpha)(s)$ being the non-decreasing upper closure of $(\beta - \alpha)(t)$.

In case of FIFO multiplexing, the left-over service curve is

$$\beta^{l.o.f_1} = \beta \ominus^{FIFO} \alpha^{f_2}$$

where $\ominus^{FIFO}$ computes left-over service curve with the smallest latency T in a worst-case FIFO multiplexing scenario. T is defined as the first time instance when α's burst is worked off and its arrival rate is smaller than β's service rate. At this time it can be safely assumed that the system has spare capacity that, in the FIFO multiplexing scheme, will be used to serve f_1's data that arrived in the meantime.

One of the strongest results of network calculus (albeit being a simple consequence of the associativity of $\otimes$) is the concatenation theorem that enables us to investigate tandems of systems as if they were single systems:

THEOREM 4. [unikl.disco.minplus.*Convolution*.java] *(Concatenation Theorem) Consider a flow f that traverses a tandem of systems $\mathcal{S}_i$, $i = 1, \ldots, n$. and that $\mathcal{S}_i$ offers a service curve $\beta_{\mathcal{S}_i}$ to f. Then the concatenation of the n systems offers a service curve $\bigotimes_{i=1}^{n} \beta_{\mathcal{S}_i}$ to the flow.*

4. NETWORK CONFIGURATION

In addition to the network calculus foundation, we also provide the classes to create a network configuration to be analyzed. They are found in the package `unikl.disco.network`. In contrast to the previous versions of our tool, we do not import external libraries for network creation and are thus able to tightly integrate the concepts needed for the network calculus analysis into the respective parts of the network. This section provides an overview of the network classes by presenting how they fulfill this purpose.

The finest granularity a system $\mathcal{S}$ can have is a single server. Hence, we provide a `Server.java` class. Besides the server's service curve, it also holds information about its multiplexing strategy. The modularization of the network calculator therefore made it possible to mix servers with different multiplexing strategies in a single network. The servers are connected by directed links of type `Link.java` and the instances of both classes are combined to a topology created via and stored by an object of `Network.java`.

`Network.java` acts as the exclusive central entity allowing to create a topology – including servers and links. It is responsible to provide the information needed by analyses. For that purpose, methods such as querying servers or their incident links are given. Moreover, `Network.java` allows to add flows – instances of `Flow.java` – to the network. For analyses, methods to request specific sets of flows, e.g., those crossing a given link or originating in a certain server, also exist. The flow class itself only knows its arrival curve and its path. `Path.java` is a new class introduced in the DiscoDNC v2. It supports the analyses by providing an abstracted view of the network, only consisting of the topological data known to a flow – the links and servers it crosses – and used in the derivation of its performance bounds.

Code 1 depicts the Creation of network configurations.

5. NETWORK CALCULUS ANALYSES

A network calculus analysis combines the operations given in Section 3 in order to compute bounds on end-to-end delays of flows and their maximum backlog along their path. Yet, there are different ways for this combination and therefore three distinct analysis methods: The Total Flow Analysis (TFA), the Separate Flow Analysis (SFA) and the Pay Multiplexing Only Once analysis (PMOO). These analyses are included in the package `unikl.disco.nc`. Their modularization according to the structure established in this section constitutes a major achievement of our efforts. Due to the focus on a single flow of interest, the analyses divide their approach into two distinct steps:

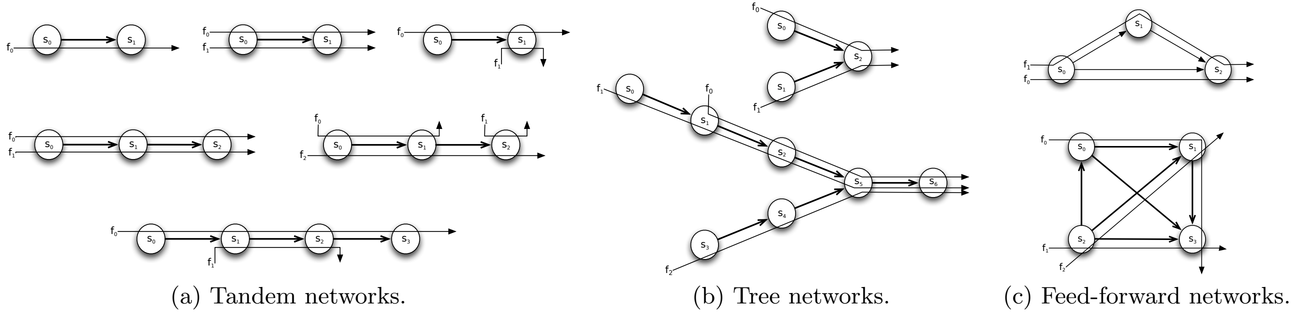
1. **Arrival Bounding:** First, the information needed to examine the flow of interest's resource share has to be derived. To do this, the bounds on data arrivals on its path are computed.
2. **Performance Bound Derivation:** In the second step, the aforementioned arrival bounds are used to derive the performance bounds of the flow of interest.

Both steps differ considerably between TFA and the other analysis methods, whereas the approaches of SFA and PMOO both follow the same general principle.

TFA [`TotalFlowAnalysis.java`]**.** This analysis bounds the totality of flows present at a server s on the flow of interest's path $\mathcal{P}$ in step 1. It then derives the local performance bounds at s, delay D_s and backlog B_s, for this flow aggregate (step 2). This whole procedure is repeated, hop by hop, from the flow of interest's source to its sink. In the concluding step, the end-to-end delay and backlog bounds are derived from the server-local bounds:

$$B = \max_{s \in \mathcal{P}} B_s \quad , \quad D = \sum_{s \in \mathcal{P}} D_s \, .$$

It is important to note, that the TFA can violate the FIFO per micro-flow assumption Theorem 1 relies on for its delay bound derivation. When bounding an aggregate of multiple flows, the applied multiplexing strategy of a server matters. Under arbitrary multiplexing, FIFO per micro-flow does not translate to FIFO per *macro-flow*, i.e., per flow aggregate. Thus, the delay bound is defined by the intersection of α and β instead of their maximum horizontal deviation. In contrast, FIFO per micro-flow can be assumed for FIFO multiplexing servers in the total flow analysis. Both types of systems can be mixed in TFA due to its per-hop approach.



(a) Tandem networks. (b) Tree networks. (c) Feed-forward networks.

Figure 1: Excerpt of the network configurations we test against.

SFA [`SeparateFlowAnalysis.java`]**.** In contrast to the TFA, the SFA models the network as if the flow of interest was not present in order to quantify the share of service that can be assumed to be at its disposal – we say the flow of interest is *separated* from its cross-traffic. Therefore, step one constitutes a **cross-traffic arrival bounding** only and step two needs to be divided into two separate parts:

- **Left-Over Service Derivation:** First, the network is abstracted to the flow of interest's view. To achieve this, the cross-traffic arrival bounds from step 1 are used to derive the left-over service curve $\beta_s^{\text{l.o.}}$ (Theorem 3) for every server $s \in \mathcal{P}$. They are combined to the end-to-end service curve $\beta_{\text{e2e}}^{\text{l.o.}}$ using Theorem 4.

- **Bound Computation:** Secondly, Theorem 1 is applied to the flow of interest's arrival curve and $\beta_{\text{e2e}}^{\text{l.o.}}$ to derive the bounds on delay and backlog. FIFO per micro-flow is guaranteed when deriving bounds in this fashion as the single flow of interest was separated first.

The server-local derivation of $\beta_s^{\text{l.o.}}$ allows to mix FIFO multiplexing and arbitrarily multiplexing servers in the SFA.

Further, SFA leverages a phenomenon called Pay Bursts Only Once (PBOO). It refers to the fact that the intermediate burstiness increase of the flow of interest – as locally perceived by the servers on its path – does not impact its performance bounds. This improvement can be attributed to the derivation of $\beta_{\text{e2e}}^{\text{l.o.}}$. It is not present in the TFA.

PMOO [`PmooAnalysis.java`]**.** The Pay Multiplexing Only Once analysis [13] proceeds along the lines of SFA, i.e., it bounds cross-traffic only, derives an end-to-end left-over service curve for the flow of interest and computes the bounds based on this information. Thus, the PMOO analysis possesses the PBOO property. Yet, it changes the order during the derivation of $\beta_{\text{e2e}}^{\text{l.o.}}$ – it concatenates before subtracting cross-traffic arrivals. PMOO does not apply Theorem 4 and it is only proven to be correct for arbitrary multiplexing servers whose service is given as a piecewise linear curve. In a PMOO analysis, the end-to-end semantic is established first and cross-traffic arrival bounding also differs from SFA's. In order to correctly account for demultiplexing on the flow of interest's path cross traffic needs to be grouped accordingly.

Arrival Bounding

The analyses presented above define the procedure on the flow of interest's path as well as the demand on the arrival bounding method conducted in step 1. The provisioning of arrival bounds as needed by the different analyses – e.g., including the flow of interest or bounding cross-traffic only – is coordinated by the class `ArrivalBounds.java` in `unikl.disco.nc`. It provides the ability to use multiple of the alternative methods for arrival bounding simultaneously:

- `PbooArrivalBounding_*.java`: Three alternatives to compute an arrival bound with the PBOO property exist – directly derived from Theorem 2 (`Output_PerHop`) or structured according to SFA; either applying Theorem 2 iteratively (`PerHop`) or deriving the end-to-end left-over service curve first (`Concatenation`).

- `PmooArrivalBounding.java`: This arrival bound alternative proceeds according to the PMOO analysis.

- Per-flow bounding: `ArrivalBounds.java` also provides the possibility to carry out a SFA or a PMOO analysis for every flow whose arrival needs to be bounded. A left-over service curve from the flow's source to the point of bounding is derived and Theorem 2 is used to bound the flow. The individual arrival bounds are then summed up to the aggregate arrival bound. This alternative leads to looser bounds compared to the others.

Independent of the actual analyses, the arrival bounding may be repeatedly executed in order to bound the cross-traffic of the flows to bound – this scheme even repeats recursively for cross-traffic of cross-traffic etc. For that reason the network configuration to be analyzed needs to possess the feed-forward property, i.e., there are no cyclic dependencies among its flows. Otherwise, such interdependencies will escalate in the arrival bounding process and thus eventually cause unboundable arrivals, i.e., infinite bounds.

6. FUNCTIONAL TESTS

We have established an extensive set of functional tests whose documentation reenacts the derivation of the bounds as done in the DiscoDNC itself. In the package `unikl.disco.tests` there are 18 different network configurations. Each is analyzed with every combination of analysis – TFA, SFA or PMOO – and arrival bounding alternative. The test suite covers a wide range of network configurations: Simple single

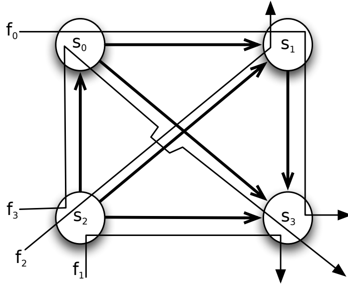


Figure 2: `FeedForward_1SC_4Flows_1AC_4Paths.java`

server, single flow settings test basic operations and complex feed-forward network configurations test different effects of the analyses like the positive and negative impact of the PMOO – either on the flow of interest or in the arrival bounding. Figures 1 and 2 show an excerpt of the network configurations we test against. They illustrate the inductive steps we take from tandems to trees to feed-forward networks. The delay and backlog bounds we check our toolbox against are all derived manually and provided in the 159 pages of supplementary material in `unikl.disco.tests`'s Single.pdf, Tandem.pdf, Tree.pdf and FeedForward.pdf.

Additionally, we test the different code paths that can be triggered in `unikl.disco.nc.Configuration.java`, e.g., the removal of duplicate arrival bounds if multiple arrival bounding methods are tested at every recursion level of this process. In total, we currently provide 1800 functional tests with the DiscoDNC v2. They all give insight into the algorithmics of deriving performance bounds with network calculus as well as the the inner workings of our calculator.

In order to allow for testing, we enabled the code to run on two alternative number representations: real values and rationals. The former is based on the `double` data type, and thus suffers from rounding errors that prevent their use for tests. Hence, we extended our code base with the ability to operate on rational numbers. The choice of the number representation to use is made at compile time by using the according file for `unikl.disco.misc.Num.java` and depends on the scenario the DiscoDNC is used for. Whereas the rational numbers deliver exact results that enable verification, they are based on integer values for the numerator and denominator and are thus prone to overflows. Moreover, the computations are more demanding than the with the `double`-based numbers. Therefore, analyses take potentially longer to finish. We recommend to resort to the rationals for verification and to reals for simulation.

7. USING THE DISCO DNC

Depending on the purpose the Disco Deterministic Network Calculator is used for, it might depend on external libraries. For testing purpose, using the rational number representation requires Apache Commons Math [1] as well as the JUnit testing framework [4] and its dependencies. For the use as stand-alone tool operating on real numbers there are no external dependencies to run the DiscoDNC v2.

In this section, we present the functional test `FeedForward_1SC_4Flows_1AC_4Paths.java` (See Figure 2). Besides the functional tests, we provide demo classes without JUnit testing in the package `unikl.disco.demos`.

Using the DiscoDNC will be illustrated by the network creation, running different analyses on the network and accessing the generated information. The code samples shown resemble the actual code of the test class – there is no pseudocode abstraction required to depict how to execute an analysis with our network calculator, yet, we omitted variables' types where it is clear from the context.

Network Creation. Our objective was to improve the network creation such that it is as straight-forward as possible. Therefore, we decided to require the user to provide a feed-forward network instead of transforming any given one with turn prohibition [14] as it was commonly done in the previous version. Additionally, this outright application of the analysis allows for easier relation of results to their corresponding impact factors. Code 1 shows how to use `Network.java` to create Figure 2's network configuration.

```
network = new Network();

s0 = network.addServer(service_curve);
s1 = network.addServer(service_curve);
s2 = network.addServer(service_curve);
s3 = network.addServer(service_curve);

l_s0_s1 = network.addLink(s0, s1);
l_s0_s3 = network.addLink(s0, s3);
l_s1_s3 = network.addLink(s1, s3);
l_s2_s0 = network.addLink(s2, s0);
network.addLink(s2, s1);
network.addLink(s2, s3);

List<Link> f0_path = new LinkedList<Link>();
f0_path.add(l_s0_s1);
f0_path.add(l_s1_s3);
List<Link> f3_path = new LinkedList<Link>();
f3_path.add(l_s2_s0);
f3_path.add(l_s0_s3);

f0 = network.addFlow(arrival_curve, f0_path);
f1 = network.addFlow(arrival_curve, s2, s3);
f2 = network.addFlow(arrival_curve, s2, s1);
f3 = network.addFlow(arrival_curve, f3_path);
```

Code 1: Create network configuration of Figure 2.

Running an Analysis. Section 5 presented a crucial part of our modularization efforts: The separation of the analysis into arrival bounding and performance bound derivation. The DiscoDNC v2 allows to combine alternatives for both and it is possible to set multiple arrival bounding methods that compete at every level of this process (see Code 2).

```
Configuration.setArrivalBoundMethod(
            ArrivalBoundMethods.PBOO_PER_HOP);

Configuration.addArrivalBoundMethod(
      ArrivalBoundMethods.PBOO_CONCATENATION);

Configuration.addArrivalBoundMethod(
                    ArrivalBoundMethods.PMOO);
```

Code 2: Setting the arrival bounding methods

	TFA	SFA	PMOO
Service curves [$\beta^{\text{l.o.}}_{\text{e2e}}$]	`[]`	`getLeftOverServiceCurves()`	`getLeftOverServiceCurves()`
Service curves [$\beta^{\text{l.o.}}_{s}$]	`[]`	`getServerLeftOverBetasMap()`	`[]`
Arrival bounds at s	`getServerAlphasMap()`† †entire traffic	`getServerAlphasMap()`‡ ‡cross-traffic only	`getServerAlphasMap()`$^{\ddagger *}$ *grouped according to demultiplexing
D	`getDelayBound()`	`getDelayBound()`	`getDelayBound()`
D_s	`getServerDelayBoundMap()`	`[]`	`[]`
B	`getBacklogBound()`	`getBacklogBound()`	`getBacklogBound()`
B_s	`getServerBacklogBoundMap()`	`[]`	`[]`

Table 1: Accessing analysis results.

Starting the entire analysis consists of two simple steps: First, create an instance of an analysis for the network configuration to be analyzed and then start it for a specific flow of interest – f_0 in our example in Code 3. Further options to tweak the analyses (in addition to arrival bounding shown in Code 2) can be found in `Configuration.java`.

```
tfa = new TotalFlowAnalysis(network);
tfa.performEnd2EndAnalysis(f0);

sfa = new SeparateFlowAnalysis(network);
sfa.performEnd2EndAnalysis(f0);

pmoo = new PmooAnalysis(network);
pmoo.performEnd2EndAnalysis(f0);
```

Code 3: Starting different analyses

Results. The last step of an analysis is accessing the results. As mentioned above, the network calculator operates on an unchanged network as provided by the user. This fact as well as the code modularization allowed us to make intermediate results user-accessible and directly relate them to the network configuration. Table 1 depicts each analysis's results together with their according function calls (or return value if constant). Note, that all analyses can produce sets of arrival bounds at a server s if multiple arrival bound methods are set similar to Code 2. Thus, SFA and PMOO both can result in a set of $\beta^{\text{l.o.}}_{\text{e2e}}$s. For example, in the network configuration we present here, there will be a total of 81 left-over end-to-end service curves for the SFA and 27 for the PMOO analysis. Yet, due to the relative simplicity of this network – service curves and arrival curves were chosen homogeneously – the configuration option `setRemoveDuplicateArrivalBounds(true)` of `Configuration.java` reduces the amount of arrival bounds on intermediate recursion levels of this procedure such that there is only one $\beta^{\text{l.o.}}_{\text{e2e}}$ remaining for each of the analyses. A similar reduction applies to SFA's $\beta^{\text{l.o.}}_{s}$ results.

8. CONCLUSION

In this paper, we presented the Disco Deterministic Network Calculator v2 – an extensively tested, simple to use and easy to extend toolbox for deterministic network calculus. In order to depict these properties, we provided insight in our test set, illustrated the straight-forward use of the DiscoDNC v2 and treated alternative arrival bounding methods provided with our source code distribution. We achieved all this by overhauling our code base; by modularizing it and documenting its functionality. Thus, our tool is not only suitable for straight-away use and integration in existing projects but due to the comprehensive set of tests it can also be used for educational purposes.

9. REFERENCES

[1] The Apache Commons Mathematics Library. Online. http://commons.apache.org/proper/commons-math/.

[2] The Disco Deterministic Network Calculator. Online. disco.cs.uni-kl.de/index.php/projects/disco-dnc.

[3] The Disco Stochastic Network Calculator. Online. disco.cs.uni-kl.de/index.php/projects/disco-snc.

[4] The JUnit Testing Framework. Online. http://junit.org.

[5] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley, 1992.

[6] M. A. Beck and J. B. Schmitt. The DISCO Stochastic Network Calculator Version 1.0 - When Waiting Comes to an End. In *ValueTools*, 2013.

[7] K. Blaiech, O. Mounaouar, O. Cherkaoui, and L. Beliveau. Runtime Resource Allocation Model over Network Processors. In *IEEE International Conference on Cloud Engineering (IC2E)*, 2014.

[8] J.-Y. L. Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[9] A. Bouillard and E. Thierry. An Algorithmic Toolbox for Network Calculus. *Discrete Event Dynamic Systems Journal, Springer*, 2008.

[10] C.-S. Chang. *Performance Guarantees in Communication Networks*. Springer, 2000.

[11] J. B. Schmitt, N. Gollan, S. Bondorf, and I. Martinovic. Pay Bursts Only Once Holds for (Some) Non-FIFO Systems. In *IEEE INFOCOM*, 2011.

[12] J. B. Schmitt and F. A. Zdarsky. The DISCO Network Calculator - A Toolbox for Worst Case Analysis. In *ValueTools*, 2006.

[13] J. B. Schmitt, F. A. Zdarsky, and I. Martinovic. Improving Performance Bounds in Feed-Forward Networks by Paying Multiplexing Only Once. In *GI/ITG MMB*, 2008.

[14] D. Starobinski, M. Karpovsky, and L. A. Zakrevski. Application of Network Calculus to General Topologies Using Turn-Prohibition. *IEEE/ACM Transactions on Networking*, 2003.