CBSAT

Release 0.2.0

September 07, 2015

CONTENTS

CBSAT contains the relevant code for design-time network performance analysis and run-time network traffic generation, measurement, detection, and management. It has been incorporated for both design-time analysis of models and run-time management of networked applications into ROSMOD.

The documentation has been broken down into varoius sections by relevance. The starting point is the *Getting Started* page which provides an overview of the parts of the project, and should point you to the specific section relevant to your interests.

CHAPTER

ONE

GETTING STARTED

Note: This page and many of the others that are part of this documentation use terminology such as *profiles*, *convolution*, *network analysis*, etc. These terms are used without explanation in many places to simplify much of the documentation. Readers are encouraged to read through all of *Background, Theory, and Results* for a complete understanding of these terms as well as the rest of the theory and background which acts as the foundation for all of this work.

This repository code and its documentation covers a novel technique for analyzing and predicting network performance of distributed applications, called **Precise Performance Prediction for Networks** (**P3N**), which has been implemented into a design-time analysis tool that analyzes application and system models. Furthermore, a run-time library has been developed which enables application code to generate network traffic that adheres to a specified network profile. Metrics about this traffic are automatically recorded for offline analysis. Furthermore the library enables management of such networked applications by forcing them to adhere to their profiles; in this way it acts as a lightweight layer on top of the communications middleware layer used by the applications on the system. Finally, anomalous traffic can be detected by receivers and disabled through out-of-band (OOB) communication to the sender-side middleware.

- *Background, Theory, and Results* : for readers interested in learning the theory behind this analysis and implementation; covers all of the relevant background, theory, and results pertaining to **P3N**.
- *Using the Code* : for readers interested in using **P3N**; provides an explanation the interfaces provided by the tools and walks through some examples to demonstrate how they can be used.
- *The Inner Workings* : for readers interested in extending this work or learning more about the complete implementation of the theory; provides a launching point which directs them to the APIs that have been developed for interfacing between the code's modules.

CHAPTER

TWO

BACKGROUND, THEORY, AND RESULTS

2.1 Background: Network Performance Analysis

Networking systems have been developed for over half a century and the analysis of processing networks and communications networks began even earlier. As computing power has increased, the field of network performance analysis at design-time has evolved into two main paradigms: (1) network performance testing of the applications and system to be deployed to determine performance and pitfalls, and (2) analytical models and techniques to provide application network performance guarantees based on those models. The first paradigm generally involves either arbitrarily precise network simulation, or network emulation, or sub-scale experiments on the actual system. The second paradigm focuses on formal models and methods for composing and analyzing those models to derive performance predictions.

We focus on the second paradigm, using models for predicting network performance at design-time. This focus comes mainly from the types of systems to which we wish to apply our analysis: safety- or mission-critical distributed cyber-physical systems, such as satellites, or autonomous vehicles. For such systems, resources come at a premium and design-time analysis must provide strict guarantees about run-time performance and safety before the system is ever deployed.

For such systems, probabilistic approaches do not provide high enough confidence on performance predictions since they are based on statistical models. Therefore, we must use deterministic analysis techniques to analyze these systems.

2.1.1 Min-Plus Calculus

Because our work and other work in the field, e.g. Network Calculus, is based on Min-Plus Calculus, or (min,+)-calculus, we will give a brief overview of it here.

Min-plus calculus, $(\mathbb{R} \cup \{+\infty\}, \wedge, +)$, deals with *wide-sense increasing functions* :

$$F = \{f : \mathbb{R}^+ \rightarrow \mathbb{R}^+, \forall s \leq t : f(s) \leq f(t), f(0) = 0\}$$

which represent functions whose slopes are always ≥ 0. Intuitively this makes sense for modeling network traffic, as data can only ever by sent or not sent by the network, therefore the cumulative amount of data sent by the network as a function of time can only ever increase or stagnate. A wide-sense increasing function can further be classified as a sub-additive function if

$$\forall s, t : f(s + t) \leq f(s) + f(t)$$

Note that if a function is concave with $f(0) = 0$, it is sub-additive, e.g. $y = \sqrt{x}$.

The main operations of min-plus calculus are the convolution and deconvolution operations, which act on sub-additive functions. Convolution is a function of the form:

$$(f \otimes g)(t) \equiv inf_{\{0 \leq s \leq t\}}\{f(t - s) + g(s)\}$$

Note that if the functions f, g are concave, this convolution simplifies into the computation of the minimum:

$$(f \otimes g)(t) = min(f, g)$$

Convolution in min-plus calculus has the properties of

- closure: $(f \otimes g)(t) \in F$,
- Associativity,
- Commutativity, and
- Distributivity

2.1.2 Network Calculus

Network Calculus provides a modeling and analysis paradigm for deterministically analyzing networking and queueing systems. Its roots come from the desire to analyze network and queuing systems using similar techniques as traditiional electrical circuit systems, i.e. by analyzing the *convolution* of an *input* function with a *system* function to produce an *output* function. Instead of the convolution mathematics from traditional systems theory, Network Calculus is based on the concepts of *(min,+)-calculus*, which we will not cover here for clarity, but for which an explanation can be found in my proposal and thesis.

By using the concepts of *(min,+)-calculus*, Network Calculus provides a way to model the application network requirements and system network capacity as functions, not of time, but of *time-window size*. Such application network requirements become a cumulative curve defined as the *maximum arrival curve*. This curve represents the cumulative amount of data that can be transmitted as a function of time-window size. Similarly, the system network capacity becomes a cumulative curve defined as the *minimum service curve*. These curves bound the application requirements and system service capacity.

Note that sub-additivity of functions is required to be able to define meaningful constraints for network calculus, though realistically modeled systems (in Network Calculus) will always have sub-additive functions to describe their network characteristics (e.g. data serviced or data produced). This sub-additivity comes from the semantics of the modeling; since the models describe maximum data production or minimum service as functions of *time-windows*, maximum data production over a longer time window must inherently encompass the maximum data production of shorter time-windows.

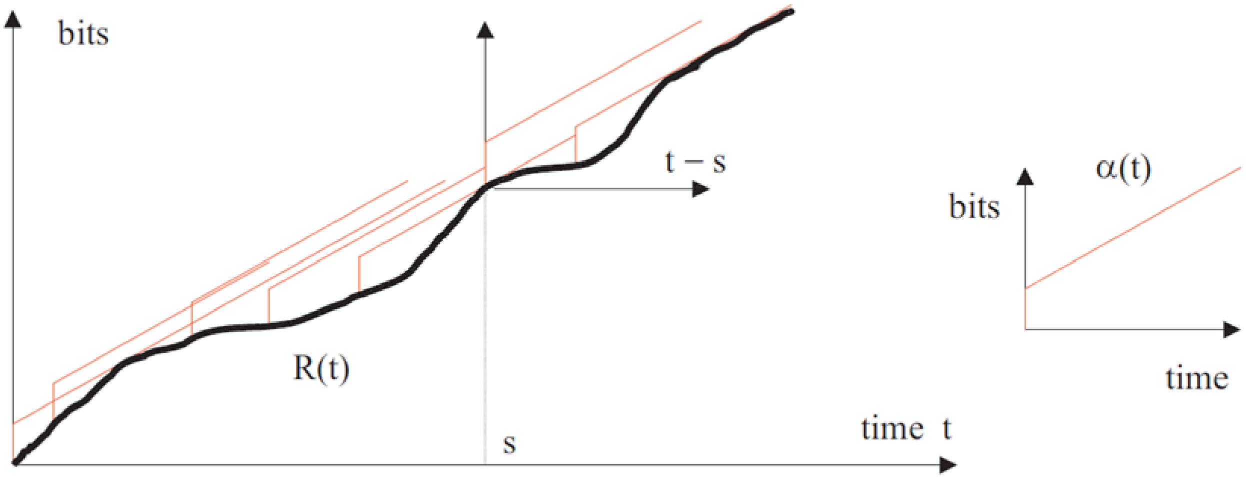


Fig. 2.1: Network Calculus arrival curve (α)

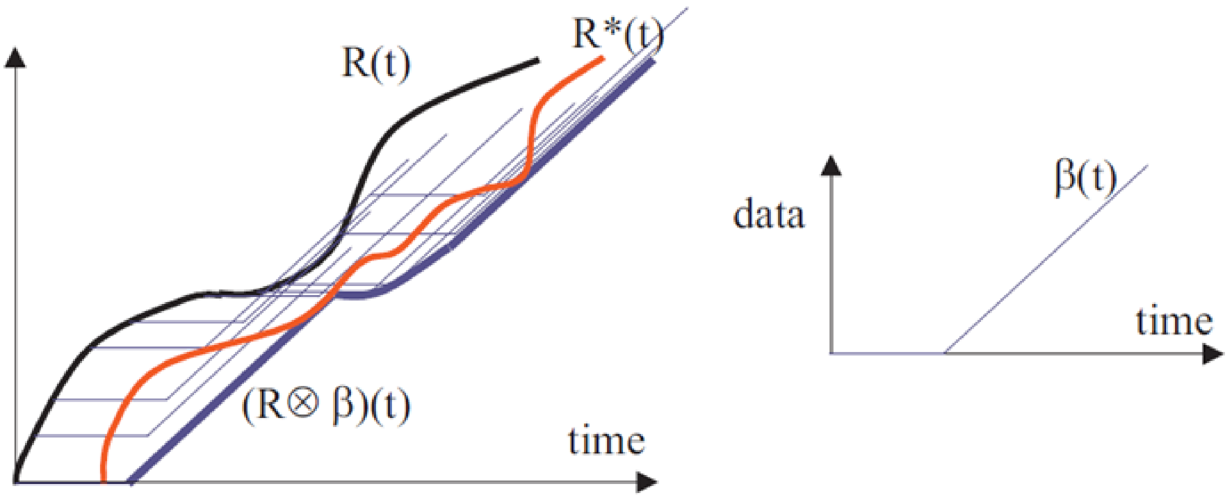


Fig. 2.2: Network Calculus service curve (β)

Network calculus uses *(min,+)-calculus convolution* to compose the application requirement curve with the system service curve. The output of this convolution is the maximum data arrival curve for the output flow from the node providing the service. By analyzing these curves, bounds on the application's required buffer size and buffering delay can be determined.

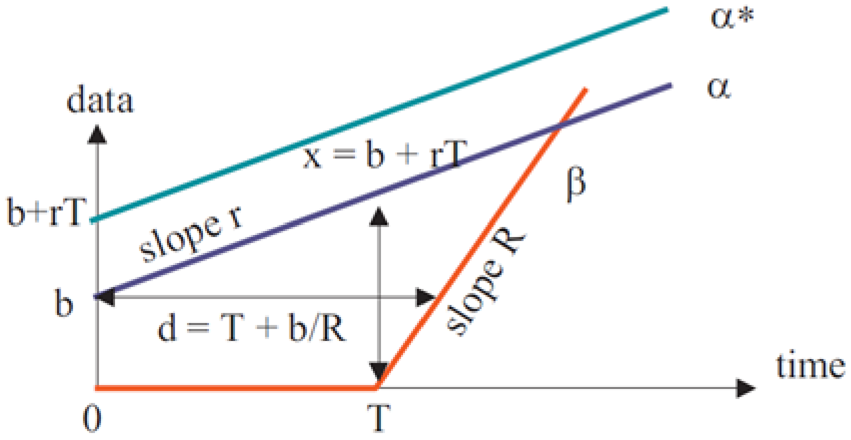


Fig. 2.3: Schematic deptiction of the buffer size (vertical difference) and delay (horizontal difference) calculations in Network Calculus.

With these bounds and the convolution, developers can make *worst-case* performance predictions of the applications on the network. These bounds are *worst-case* because the curves are functions of *time-window size*, instead of directly being functions of time. This distinction means that the worst service period provided by the system is directly compared with the maximum data production period of the application. Clearly such a comparison can lead to over-estimating the buffer requirements if the application's maximum data production does not occur during that period.

2.1.3 Real Time Calculus

Real-Time Calculus builds from Network Calculus, Max-Plus Linear System Theory, and real-time scheduling to analyze systems which provide computational or communications services. Unlike Network Calculus, Real-Time

Calculus (RTC) is designed to analyze real-time scheduling and priority assignment in task service systems. The use of (max,+)-calculus in RTC allows specification and analysis not of only the arrival and service curves described above for Network Calculus, but of upper and lower arrival curves ($\alpha^u(\Delta)$ and $\alpha^l(\Delta)$) and upper and lower service curves ($\beta^u(\Delta)$ and $\beta^l(\Delta)$). These curves represent the miniumum and maximum computation requested and computation serviced, respectively. An overview of RTC is shown below.

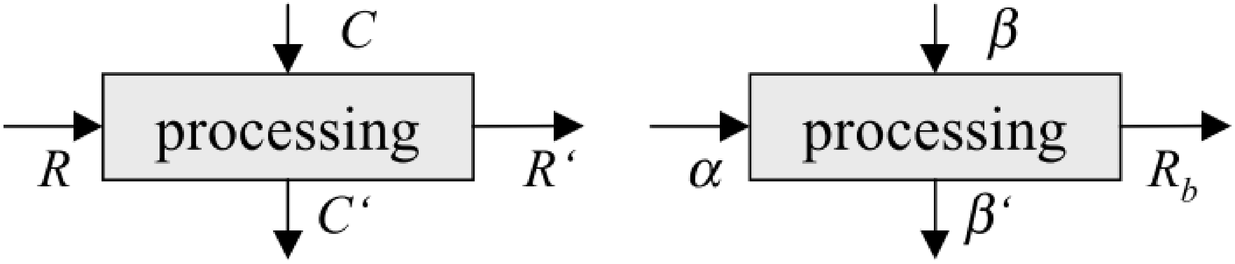


Fig. 2.4: Overview of Real-Time Calculus' request, computation, and capacity models.

$R(t)$ is the request function that represents the amount of computation that has been requested up to time t, with associated minimum request curve, α. $R'(t)$ is the total amount of computation delivered up to time t, with associated delivered computation bound $R_b(t)$. C and C' are the capacity function and remaining capacity functions which describe the total processing capacity under full load and the remaining processing capacity, respectively. C and C' are bounded by the delivery curve β and the remaining delivery curve β'.

RTC allows for the analysis of task scheduling systems by computing the request curve for a task model which is represented as a directed acyclic graph (DAG), the task graph $G(T)$. The graph's vertices represent subtasks and each have their own associated required computation time $e(u)$, and relative deadline $d(u)$ specifying that the task must be completed $d(u)$ units of time after its triggering. Two vertices in $G(T)$ may be connected by a directed edge (u, v) which has an associated parameter $p(u, v)$ which specifies the minimum time that must elapse after the triggering of u before v can be triggered. RTC develops from this specification the minimum computation request curve α_r and the maximum computation demand curve α_d. Finally, the schedulability of a task T_i is determined by the relation:

$$\beta'(\Delta) \geq \alpha_d^i(\Delta) \quad \forall\Delta$$

which, if satisfied, guarantees that task T_i will meet all of its deadlines for a static priority scheduler where tasks are ordered with decreasing priority. Note that the remaining delivery curve $\beta'(\Delta)$ is the capacity offered to task T_i after all tasks $T_{1 \leq j < i}$ have been processed. Similarly to Network Calculus, RTC provides analytical techniques for the computation of performance metrics such as computation backlog bounds:

$$\text{backlog} \leq sup_{\{t \geq 0\}}\{\alpha^u(t) - \beta^l(t)\}$$

which is equivalent to the network backlog bound derived in Network Calculus.

2.2 P3N : Theory

This page describes the mathematical formalization behind the network analysis techniques used by **P3N**.

2.2.1 Formalism for Precise Performance Prediction for Networks

To model the network capability of the system and the application traffic patterns, we have developed a network modeling paradigm similar to Network Calculus' traffic arrival curves and traffic shaper service curves.

Similarly to Network Calculus' arrival curves and service curves, our network profiles model how the network performance or application traffic generation changes with respect to time. Whereas Network Calculus' modeling transforms application data profiles and network service profiles into min and max curves for data received vs. size of time-window, our models take a simpler, deterministic approach which models exactly the data generated by the application and the data which could be sent through the network, allowing our performance metrics to be more precise. Specifically, the bandwidth that the network provides on a given communication link is specified as a time series of scalar bandwidth values. Here, bandwidth is defined as data rate, i.e. bits per second, over some averaging interval. This bandwidth profile can then be time-integrated to determine the maximum amount of data throughput the network link could provide over a given time. The bandwidth profile for the application traffic similarly can be time-integrated to determine the amount of data that the application attempts to send on the network link as a function of time.

Having time-integrated the bandwidth profiles to obtain data vs. time profiles that the application requires and that the system provides, we can use a special type of convolution ($\otimes$), *(min,+)-calculus convolution*, on these two profiles to obtain the transmitted link data profile as a function of discrete time. The convolution we define on these profiles borrows concepts from the min-plus calculus used in Network Calculus, but does not use a sliding-window and instead takes the transformed minimum of the profiles. For a given application data generation profile, $r[t]$, and a given system link capacity profile $p[t]$, where $t \in \mathbb{N}$, the link transmitted data profile $l[t]$ is given by the convolution equation (2.1). The difference $(p[t-1] - l[t-1])$ represents the difference between the amount of data that has been transmitted on the link $(l[t-1])$ and the data that the link could have transmitted at full utilization $(p[t-1])$. As demonstrated by the convolution equation, $\forall t : l[t] \leq r[t]$, which is the relation that, without lower-layer reliable transport, the link cannot transmit more application data for the application than the application requests as there will be packetization and communication header overhead as well. The buffer and delay equations (2.1) use the output of the convolution with the input profile to predict the minimum required buffer size for lossless tranmission and the maximum delay experienced by the transmitted data, respectively. A representative convolution example is shown below for reference. These functions are implemented as: `networkProfile.Profile.Convolve()`, `networkProfile.Profile.CalcBuffer()`, and `networkProfile.Profile.CalcDelay()`.

$$\begin{aligned} y = l[t] &= (r \otimes p)[t] \\ &= min(r[t], p[t] - (p[t-1] - l[t-1])) \\ \text{buffer} &= sup\{r[t] - l[t] : t \in \mathbb{N}\} \\ \text{delay} &= sup\{l^{-1}[y] - r^{-1}[y] : y \in \mathbb{N}\} \end{aligned} \tag{2.1}$$

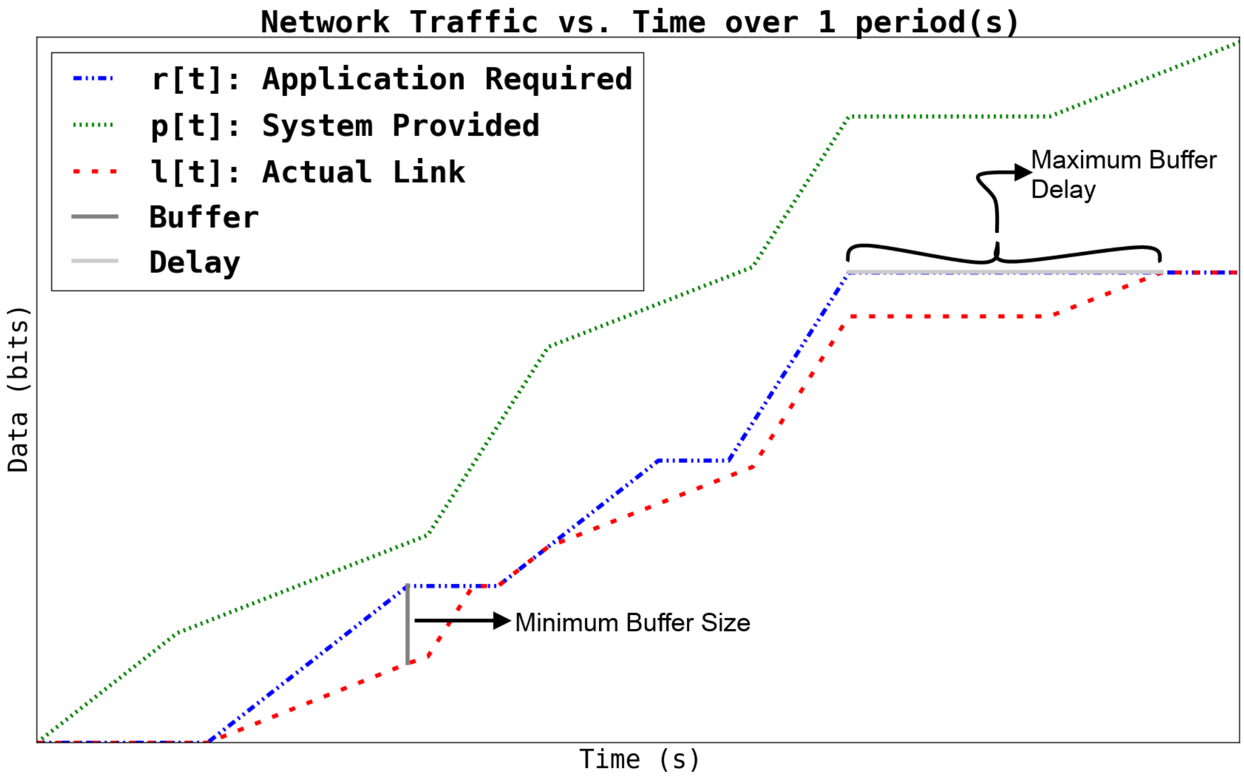
2.2.2 Assumptions Required for Analysis and Performance Prediction

As with any type of system modeling and analysis paradigm, it is important to remain aware of the types of systems the modeling/analysis is applicable to, the requirements imposed on the system by the model, and any edge cases or scenarios where the analysis or modeling paradigm breaks down.

The major assumption that we make with this type of system modeling and analysis is that we *can* know at design time what the system network capacity and the application data production will be as a (possibly periodic) function of time. Of course, this assumption is unrealistic for heavily data-dependent systems, but by performing some code analysis and/or doing some controlled experiments, models of the applications' behavior can be developed that can be analyzed.

Another key assumption and thus requirement of our modeling and analysis framework is a system-wide synchronized clock which all nodes use. By this we mean that if two nodes produce data for a third node at time $t = 3$ seconds, they will produce their data at exactly the same time. This is required for the composition of profiles as they traverse the network and are routed through nodes. This assumption restricts the types of systems for which our analysis can be most useful, but is not a critical hindrance, as many such critical systems, e.g. satellite constellations or UAVs have GPS synchronized clocks, which provide such a foundation.

Another restriction with our modeling paradigm is that data-dependent flows cannot be accurately represented, since we have no way of modeling data-dependence. A related assumption is processing power and the ability of the software to adhere to the profiles: we assume the applications are able to accurately and precisely follow their data production



profiles, regardless of the number of other components on their hardware node. Similarly, we assume that under all circumstances, the service profile of a hardware node will be adhered to.

2.2.3 Factors Impacting Analysis:

It is important when developing modeling and analysis techniques to analyze how the analysis time and results are affected by changes in the model. This is especially true when trying to determine how applicable new techniques are to large scale systems. Models are provided by the application and system developers and are described in the form of bandwdith (bps) vs time that the application requires or the system provides. These profiles are a time series that maps a given time to a given bandwdith. Between two successive intervals, the bandwidth is held constant. Clearly, to represent changing bandwidth over time, the developer must use sufficiently short enough time intervals to allow step-wise approximation of the curve. However, as with any system, there is a tradeoff between precision of the model and the analysis time and results.

Because the fundamental mathematics are linear for our convolution, our convolution scales with $O(n)$, where n is the total number of intervals in all of the profiles analyzed. It is worth noting that this complexity is not the same as the $O(n^2)$ or $O(n * log(n))$ complexity that traditional convolution has. This decrease in complexity is due to our convolution only requiring a single operation (comparison operation for the minimum) for each value of t. As such, each element in both of the profiles being convolved only needs to be operated on once.

Clearly, the overall system analysis complexity depends on the complexity of the system, so as the system scales and increases routing complexity, so too will the analysis complexity. However, for all systems there is an asymptotically increasing precision for a given increase in model precision and analysis time.

2.3 P3N : Results

These pages cover the results of my research as it applies to analysis of networked CPS. I will cover the research contributions in two aspects:

- *Design Time Results* : Details design-time network analysis contributions and improvements to network performance prediction
- *Run Time Results* : Details the run-time network monitoring and management contributions which have been based of the design-time work

2.3.1 Design Time Results

These results provide a methodology and a means for application developers and system integrators to determine conservative, precise, tightly bounded performance metrics for distributed networked applications and systems at design time. The contributions of this work are broken into sections by topic:

- *Periodic System Analysis*
- *Proving the Minimum Analysis for System Stability*
- *Comparison with NC/RTC*
- *Analysis of TDMA Scheduling*
- *Compositional Analysis*
- *Delay Analysis*
- *Routing Analysis*

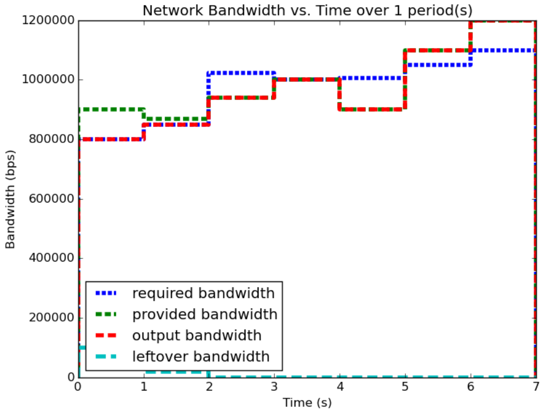
Periodic System Analysis

One subset of systems which we would like to analyze are periodic systems, since many systems in the real world exhibit some form of periodicity, e.g. satellites in orbit, traffic congestion patterns, power draw patterns. We define systems to be periodic if the data production rate (or consumption rate) of the system is a periodic function of time. The time-integral of these periodic data consumption/production rates is the cumulative data production/consumption of the system. These cumulative functions are called *repeating*.

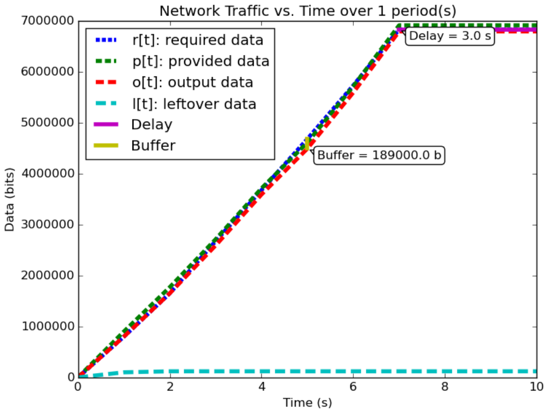
Given that the required data profile and system data service profile are *repeating*, we must determine the periodicity of the output profile. If we can show that the output profile similarly repeats, then we can show that the system has no unbounded buffer growth. First, let us look at the profile behavior over the course of its first two periods of activity.

We will examine two systems, *system (1)* and *system (2)*. Firstly, examine *(1)*, shown below (note: you can click on the images to open them in a larger format):

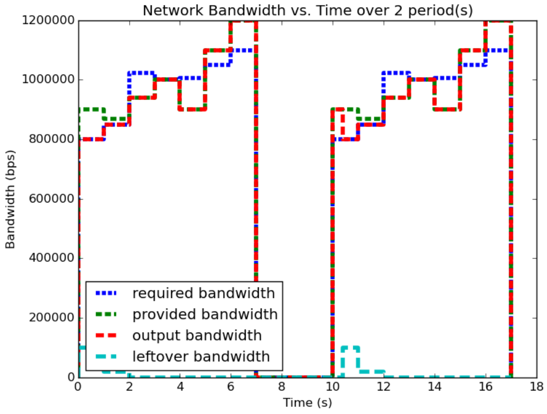
System *(1)* Bandwidth for 1 Period



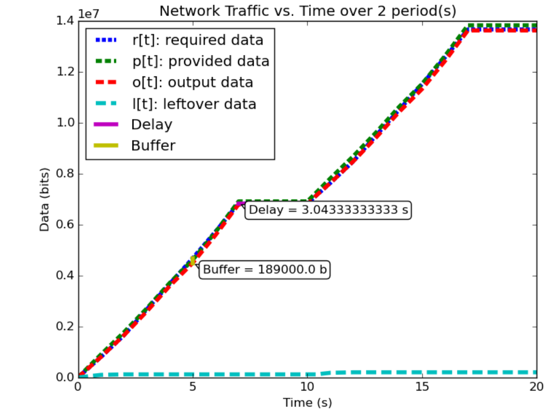
System *(1)* Data for 1 Period



System *(1)* Bandwidth for 2 Periods

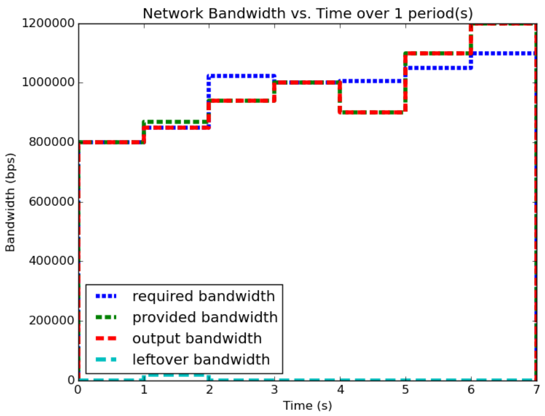
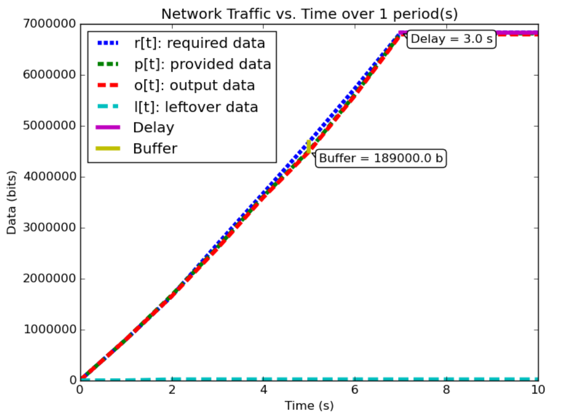


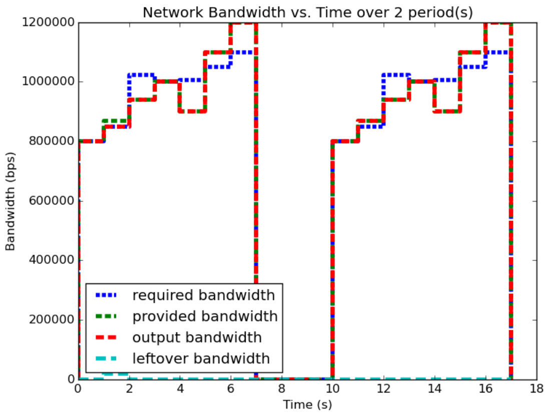
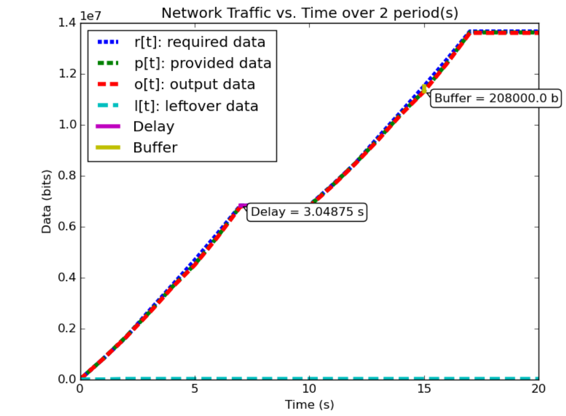
System *(1)* Data for 2 Periods



We notice that for this example system, the second period output profile is not an exact copy of the first (most easily seen by examining the bandwidth plots), and yet the required buffer size is still the same as it was when analyzing the system over one period. Furthermore, by running the analysis over even larger number of periods, we can determine (not plotted here for space and readability), that the predicted buffer size does not change no matter how many periods we analyze for this system.

Let us look at a system where this is not the case before we begin the analysis of such system characteristics.

System *(2)* Bandwidth for 1 Period	System *(2)* Data for 1 Period
	

System *(2)* Bandwidth for 2 Periods	System *(2)* Data for 2 Periods
	

Notice in system *(2)*, the first period analysis predicted the same buffer size and delay as system *(1)*, but when analyzing two periods the predicted buffer size changed. Clearly the behavior of the system is changing between these two periods. If we continue to analyze more periods of system *(2)*, as we did with system *(1)*, we'll find the unfortunate conclusion that the predicted buffer size increases with every period we add to the analysis.

We have discovered a system level property that can be calculated from these profiles, but we must determine what it means and how it can be used. First, we see that in system *(1)*, the predicted required buffer size does not change regarless of the number of periods over which we analyze the system. Second, we see that for system *(2)*, the predicted required buffer size changes depending on how many periods of activity we choose for our analysis window. Third, we see that the second period of system *(2)* contains the larger of the two predicted buffer sizes. These observations (with our understanding of deterministic periodic systems) lead us to the conclusion: system *(2)* can no longer be classified as periodic, since its behavior is not consistent between its periods. Furthermore, because the required buffer size predicted for system system *(2)* continually increases, we can determine that the system is in fact *unstable* due to unbounded buffer growth.

Proving the Minimum Analysis for System Stability

Let us now formally prove the assertion about system periodicity and stability which has been stated above. We will show that our analysis results provide quantitative measures about the behavior of the system and we will determine for how long we must analyze a system to glean such behaviors.

Typically, periodicity is defined for functions as the equality:

$$x(t) = x(t + k * T), \forall k \in \mathbb{N} > 0$$

but for our type of system analysis this cannot hold since we deal with cumulative functions (of data vs. time). Instead we must define a these functions to be **repeating**, where a function is repeating *iff*:

$$x(0) = 0 \text{ and}$$
$$x(t + k * T) = x(t) + k * x(T), \forall k \in \mathbb{N} > 0$$

Clearly, a repeating function x is **periodic** *iff* $x(T) = 0$. Note that repeating functions like the cumulative data vs. time profiles we deal with, are the result of **integrating** *periodic* functions, like the periodic bandwidth vs. time profiles we use to describe application network traffic and system network capacity. All periodic functions, when integrated, produce repeating functions and similarly, all repeating functions, when differentiated, procduce periodic functions.

Now we will consider a deterministic, *repeating* queuing system providing a data service function S to input data function I to produce output data function O, where these functions are *cumulative data versus time*. At any time t, the amount of data in the system's buffer is given by B_t. After servicing the input, the system has a remaining capacity function R.

- $S[t]$: the service function of the system, cumulative data service capacity versus time
- $I[t]$: the input data to the system, cumulative data versus time
- $O[t]$: the output data from the system, cumulative data versus time
- $B[t]$: the amount of data in the system's buffer at time t, i.e. $I[t] - O[t]$
- $R[t]$: the remaining service capacity of the system after servicing I, i.e. $S[t] - O[t]$

Because S and I are deterministic and repeating, they increase deterministically from period to period, i.e. given the period T_I of I,

$$\forall t, \forall n \in \mathbb{N} > 0 : I[t + n * T_I] = I[t] + n * I[T_I]$$

Similarly, given the period T_S of S,

$$\forall t, \forall n \in \mathbb{N} > 0 : S[t + n * T_S] = S[t] + n * S[T_S]$$

We can determine the hyperperiod of the system as the `utils.lcm()` of input function period and the service function period, $T_p = lcm(T_S, T_I)$.

At the start of the system, $t = 0$, the system's buffer is empty, i.e. $B[0] = 0$. Therefore, the amount of data in the buffer at the end of the first period, $t = T_p$, is the amount of data that entered the system on input function I but was not able to be serviced by S. At the start of the next period, this data will exist in the buffer. Data in the buffer at the start of the period can be compared to the system's remaining capacity R, since the remaining capacity of the system indicates how much extra data it can transmit in that period. Consider the scenario that the system's remaining capacity R is less than the size of the buffer, i.e. $R[T_p] < B[T_p]$. In this scenario, $B[2 * T_p] > B[T_p]$, i.e. there will be more data in the buffer at the end of the second period than there was at the end of the first period. Since the system is deterministic, for any two successive periods, $n * T_p$ and $(n + 1) * T_p$, $B[n * T_p] > B[(n + 1) * T_p]$, which extends to:

$$B[m * T_p] > B[n * T_p], \forall m > n > 0$$

Therefore the amount of data in the system's buffer increases every period, i.e. the system has *unbounded buffer growth*.

If however, there is enough remaining capacity in the system to service the data in the buffer, i.e. $R[T_p] >= B[T_p]$, then $B[2 * T_p] = B[T_p]$. Similarly to above, since the system is deterministic, for any two successive periods, $n * T_p$ and $(n + 1) * T_p$, $B[(n + 1) * T_p] = B[n * T_p]$. This extends to:

$$B[m * T_p] = B[n * T_p], \forall m, n > 0$$

Therefore the buffer size does not grow between periods, and the system has a *finite buffer*.

If we are only concerned with buffer growth, we do not need to calculate R, and can instead infer buffer growth by comparing the values of the buffer at any two period-offset times during the steady-state operation of the system ($t >= T_p$). This means that the system buffer growth check can resolve to $B[2 * T_p] == B[T_p]$. This comparison abides by the conditions above, with $m = 2$ and $n = 1$. Checking for system buffer growth occurs in `analysis.analyze_profile()`.

Comparison with NC/RTC

To show how our analysis techniques compare to other available methods, we developed our tools to allow us to analyze the input system using Network Calculus/Real-Time Calculus techniques as well as our own. Using these capabilities, we can directly compare the analysis results to each other, and then finally compare both results to the measurements from the actual system. The convenience function to generate a NC-based profile from our profile model is implemented in `networkProfile.Profile.ConvertToNC()`.

Taking the results from our published work, where our methods predicted a buffer size of 64000 bits / 8000 bytes, we show that Network Calculus predicts a required buffer size of 3155000 bits.

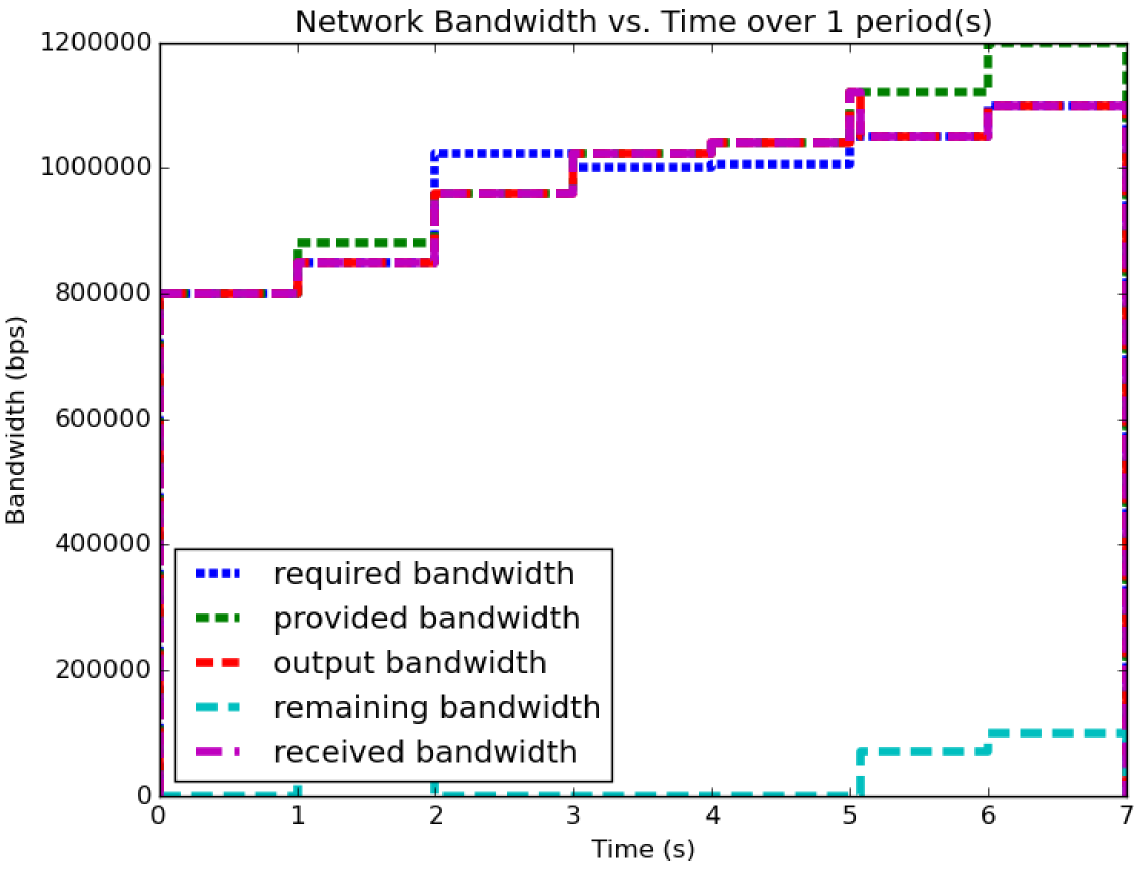


Fig. 2.5: Bandwidth profile describing the system and application.

The major drawback for Network Calculus that our work aims to solve is the disconnect from the real system that stems from using an approach based on time-window analysis. Such an approach leads to dramatically under-approximating

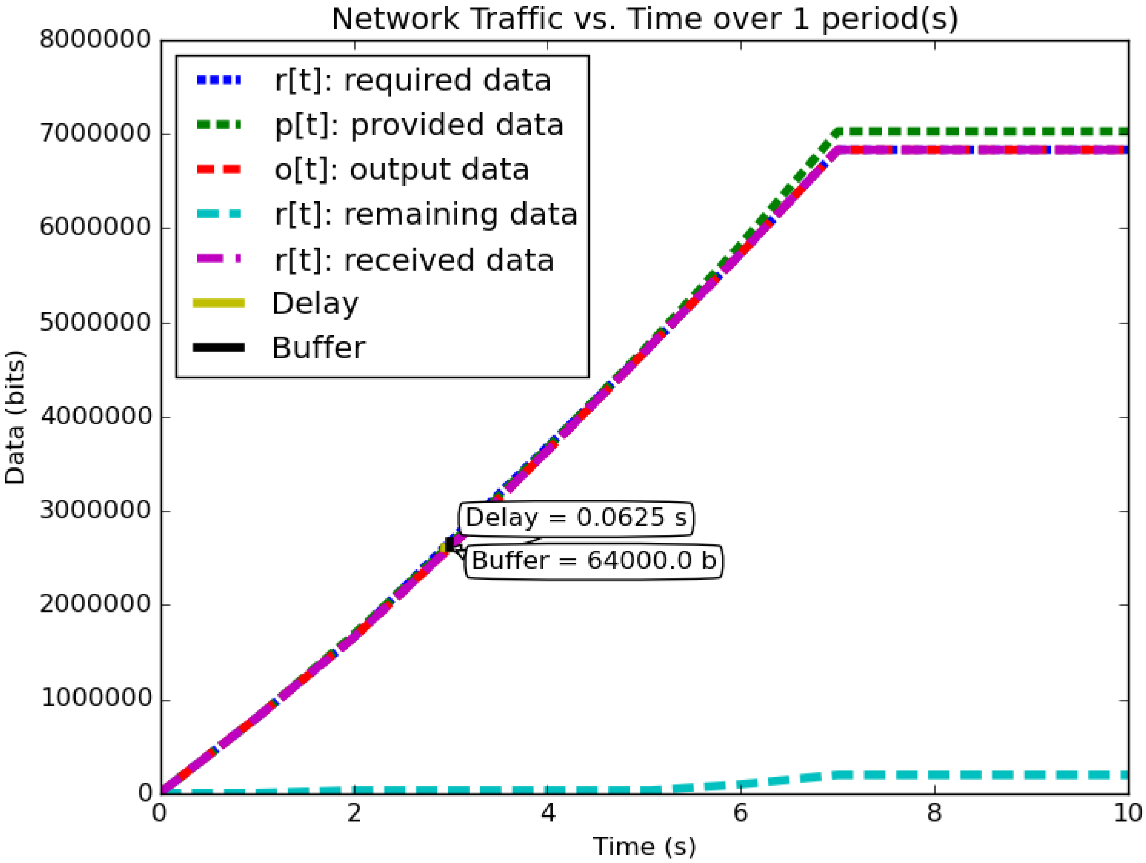


Fig. 2.6: Analysis of the system with our tools.

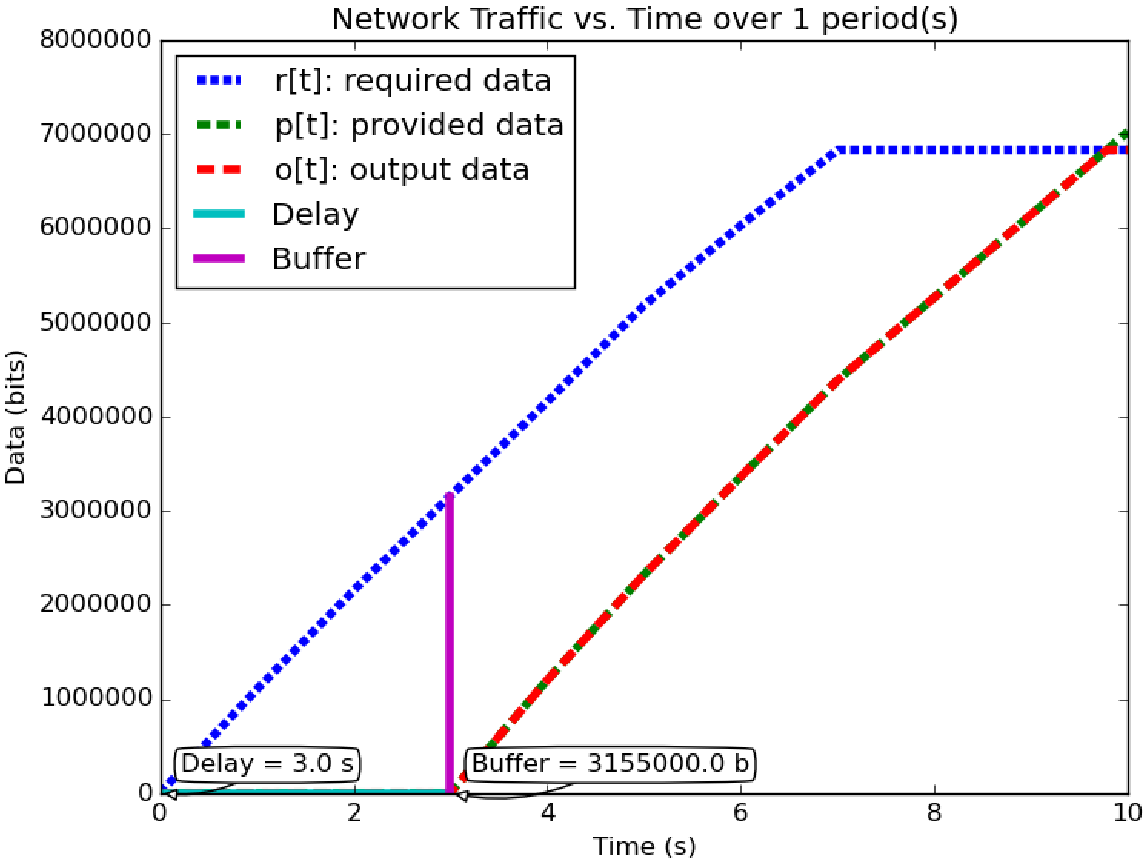


Fig. 2.7: Network-Calculus based analysis of the system.

the capacity of the network while simultaneously over-approximating the utilization of the network, since a known drop in network performance which is expected and handled by the application cannot be accurately modeled. In our case, the system is using a system profile which can service data during the period from $0 \leq t \leq 7$ seconds with a period of 10 seconds. The application is designed around this constraint and only produces data during that interval. Because our technique directly compares when the application produces data to when the system can service the data, we are able to derive more precise performance prediction metrics than Network Calculus, which compares the 3 seconds of system downtime to the 3 seconds of maximum application data production.

We developed software which produces data according to a supplied input profile and configured the system's network to provide the bandwidth profile described in the system configuration profile. Using this experimental infrastructure, we were able to measure the transmitted traffic profile, the received traffic profile, the latency experienced by the data, and the transmitter's buffer requirements. The results are displayed in the table below:

	Predicted	Measured (μ, σ)
Buffer Delay (s)	0.0625	(0.06003 , 0.00029)
Time of Delay (s)	3.0	(2.90547 , 0.00025)
Buffer Size (bytes)	8000	(7722.59 , 36.94)

Analysis of TDMA Scheduling

Medium channel access (MAC) protocols are used in networking systems to govern the communication between computing nodes which share a network communications medium. They are designed to allow reliable communication between the nodes, while maintaining certain goals, such as minimizing network collisions, maximizing bandwidth, or maximizing the number of nodes the network can handle. Such protocols include Time Division Multiple Access (TDMA), which tries to minimize the number of packet collisions; Frequency Division Multiple Access (FDMA), which tries to maximize the bandwidth available to each transmitter; and Code Division Multiple Access (CDMA) which tries to maximize the number of nodes that the network can handle. We will not discuss CDMA in the scope of this work.

In FDMA, each node of the network is assigned a different transmission frequency from a prescribed frequency band allocated for system communications. Since each node transmits on its own frequency, collisions between nodes transmitting simultaneously are reduced. Communications paradigms of this type, i.e. shared medium with collision-free simultaneous transmission between nodes, can be modeled easily by our MAReN modeling paradigm described above, since the network resource model for each node can be developed without taking into account the transmissions of other nodes.

In TDMA, each node on the network is assigned one or more time-slots per communications period in which only that node is allowed to transmit. By governing these timeslots and having each node agree upon the slot allocation and communications period, the protocol ensures that at a given time, only a single node will be transmitting data, minimizing the number of collisions due to multiple simultaneous transmitters. In such a medium access protocol, transmissions of each node affect other nodes' transmission capability. Because these transmissions are scheduled by TDMA, they can be explicitly integrated into the system network resource model.

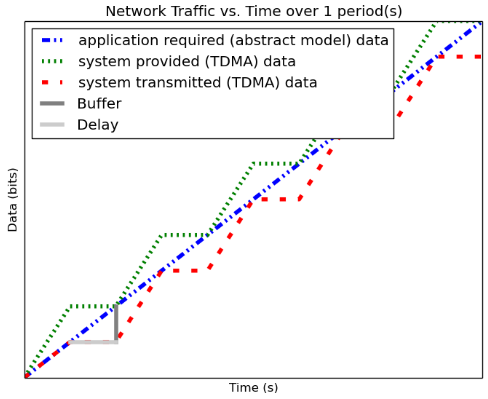
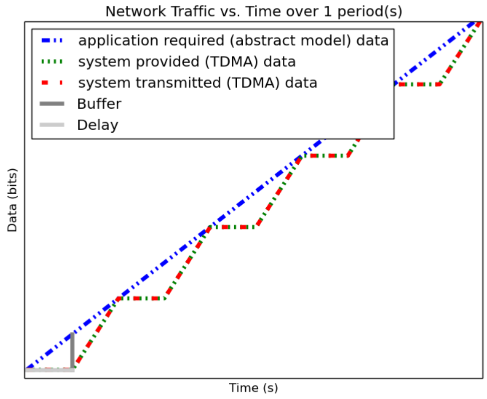
TDMA transmission scheduling has an impact on the timing characteristics of the applications' network communications. Because applications' network data production is decoupled from their node's TDMA transmission time slot, buffering may be required when an application on one node tries to send data on the network during the transmission slot of a different node. In this case, the data would need to be buffered on the application's node and would therefore incur additional buffering delay. If this TDMA schedule is not integrated into the analysis of the network resources, the additional buffer space required may exceed the buffer space allocation given to the application or the buffering delay may exceed the application's acceptable latency.

So far, the description of the system provided network service profile ($p[t] = y$), has been abstracted as simply the available bandwidth as a function of time integrated to produce the amount of data serviced as a function of time. We show how to model and analyze the network's lower-level TDMA MAC protocol using our network modeling semantics. We then derive general formulas for determining the affect TDMA has on buffer size and delay predictions.

As an example TDMA system which benefits from our analysis techniques, consider an application platform provided by a fractionated satellite cluster. A fractionated satellite cluster consists of many small satellites that may each have different hardware, computing, and communications capabilities. These capabilities are provided to distributed components of the satellite cluster's applications. Such a system has the combined challenges of (1) being expensive to develop, test, and deploy, (2) being very difficult to repair or replace in the event of failure, and (3) having to support mixed-criticality and possibly multiple levels of security applications. For this system, the network between these satellites is a precious resource shared between each of the applications' components in the cluster. To ensure the stability of the network resources, each satellite has a direct connection to every other satellite and is assigned a slot in the TDMA schedule during which the satellite may transmit. Each TDMA slot has a sinusoidally time-varying bandwidth profile which may differ from the other TDMA slot bandwidth profiles. The time-varying profile of the slot bandwidth comes from the coupling between the radios' inverse-squared bandwidth-as-a-function-of-distance and the satellites' sinusoidal distance-as-a-function-of-orbital-position.

Such a system and applications necessitates design-time guarantees about resource utilization and availability. Applications which utilize the satellite network need assurances that the network resources they require during each part of the orbital period will be satisfied. To provide these assurances, we provide the application developers and system integrators the ability to specify and analyze the network profiles as (possibly periodic) functions of time. Furthermore, the requirement for accurate predictions necessitates the incorporation of the TDMA scheduling and bandwidth profiling into the network modeling and analysis tools.

TDMA schedules can be described by their period, their number of slots, and the bandwidth available to each slot as a function of time. For simplicity of explanation, we assume that each node only gets a single slot in the TDMA period and all slots have the same length, but the results are valid for all static TDMA schedules. Note that each slot still has a bandwidth profile which varies as a function of time and that each slots may have a different bandwidth profile. In a given TDMA period (T), the node can transmit a certain number of bits governed by its slot length (t_{slot}) and the slot's available bandwidth (bw_{slot}). During the rest of the TDMA period, the node's available bandwidth is 0. This scheduling has the effect of amortizing the node's slot bandwidth into an effective bandwidth of $bw_{effective} = bw_{slot} * \frac{t_{slot}}{T}$. The addition of the TDMA scheduling can affect the buffer and delay calculations, based on the slot's bandwidth, the number of slots, and the slot length. The maximum additional delay is $\Delta_{delay} = T - t_{slot}$, and the maximum additional buffer space is $\Delta_{buffer} = \Delta_{delay} * bw_{effective}$. These deviations are shown below. Clearly, Δ_{delay} is bounded by T and Δ_{buffer} is governed by t_{slot}. Therefore, because t_{slot} is dependent on T, minimizing T minimizes both the maximum extra delay and maximum extra buffer space.

In-Phase TDMA profile vs abstract	Out-of-Phase TDMA Profile vs abstract
	

Following from this analysis, we see that if: (1) the TDMA effective bandwidth profile is provided as the abstract system network service profile, and (2) the TDMA period is much smaller than the duration of the shortest profile interval; then the system with explicit modeling of the TDMA schedule has similar predicted application network characteristics as the abstract system. Additionally, the maximum deviation formulas derived above provide a means for application developers to analyze the their application on a TDMA system without explicitly integrating the TDMA model into the system profile model.

Compositional Analysis

Now that we have precise network performance analysis for aggregate flows or singular flows on individual nodes of the network, we must determine how best to compose these flows and nodes together to analyze the overal system. The aim of this work is to allow the flows from each application to be analyzed separately from the other flows in the network, so that application developers and system integrators can derive meaningful perfomance predictions for specific applications.

We have implemented min-plus calculus based compositional operations for the network profiles which allow us to compose and decompose systems based on functional components. For network flows, this means we can analyze flows individually to determine per-flow performance metrics or we can aggregate flows together to determine aggregate performance. Profile addition and subtraciton are implemented in `networkProfile.Profile.AddProfile()` and `networkProfile.Profile.SubtractProfile()`. Using these functions we can aggregate or separate flow profiles and service profiles.

The composition is priority based, with each flow receiving a unique priority. This priority determines the oder in which the flows are individually analyzed, with the system's remaining capacity being provided to the flow with the next highest priority. This is similar to the modular performance analysis provided by Real-Time Calculus.

The basis for this priority-based interaction is the QoS management provided by many different types of networking infrastructure. DiffServ's DSCP provides one mechanism to implement this priority-based transmission and routing.

We are finalizing the design and code for tests which utilize the DSCP bit(s) setting on packet flows to show that such priority-based analysis techniques are correct for these types of systems.

Delay Analysis

When dealing with queueing systems (esp. networks) where precise design-time guarantees are required, the delay in the links of the network must be taken into account.

The delay is modeled as a continuous function of latency (seconds) versus time. In the profiles, the latency is specified discretely as $(time, latency)$ pairs, and is interpolated linearly between successive pairs.

Using these latency semantics, the delay convolution of a profile becomes

$$r[t + \delta[t]] = l[t]$$

Where

- $l[t]$ is the *link* profile describing the data as a function of time as it enters the link
- $\delta[t]$ is the *delay* profile describing the latency as a function of time on the link
- $r[t]$ is the *received* profile describing the data as a function of time as it is received at the end of the link

When analyzing delay in a periodic system, it is important to determine the effects of delay on the system's periodicity. We know that the period of the periodic profiles is defined by the time difference between the start of the profile and the end of the profile. Therefore, we can show that if the time difference between the **start time** of the *received* profile and the **end time** of the *received* profile is the same as the **period** of the *link* profile, the periodicity of the profile is unchanged.

- T_p is the period of the *link* profile
- $r[t + \delta[t]]$ is the beginning of the *received* profile
- $r[(t + T_p) + \delta[(t + T_p)]]$ is the end of the *received* profile

We determine the condition for which $(t_{end}) - (t_{start}) = T_p$:

$$\begin{aligned}(T_p + t + \delta[T_p + t]) - (t + \delta[t]) &= T_p\\ T_p + \delta[T_p + t] - \delta[t] &= T_p\\ \delta[T_p + t] - \delta[t] &= 0\\ \delta[T_p + t] &= \delta[t]\end{aligned}$$

Which is just confirms that the periodicity of the delayed profile is unchanged *iff* the latency profile is **periodic**, i.e.

$$\delta[t] = \delta[t + k * T_p], \forall k \in \mathbb{N} > 0$$

The profile delay operation is implemented in `networkProfile.Profile.Delay()`.

Routing Analysis

By incorporating both the latency analysis with the compositional operations we developed, we can perform system-level analysis of flows which are routed by nodes of the system. In this paradigm, nodes can transmit/receive their own data, i.e. they can host applications which act as data sources or sinks, as well as act as routers for flows from and to other nodes. To make such a system amenable to analysis we must ensure that we know the routes the flows will take at design time, i.e. the routes in the network are static and known or calculable. Furthermore, we must, for the sake of flow composition as decribed above, ensure that each flow has a priority that is unique within the network which governs how the transmitting and routing nodes handle the flow's data.

We have extended our network analysis tool to support such system analysis by taking as input:

- the sender flows and receiver functions in the network
- the provided service of each node in the network
- the network configuration specifying the nodes in the network and the routes in the network

where a flow is defined by (see `networkProfile.Profile.ParseHeader()`):

- Node ID of the profile
- Kind of the flow
- Period of the flow
- Flow type of the profile
- Priority of the flow
- flow properties vs time profile, see `networkProfile.Profile.ParseEntriesFromLine()`

and a route is specified as a list of node IDs starting with the source node ID and ending with the destination node ID. Any flows which have the respective source and destination IDs must travel along the path specified by the respective route. The route and the toplogy are implemented in `networkConfig.Route` and `networkConfig.Topology`, and the network configuration specification is found in `networkConfig.Config`.

We can then run the following algorithm to iteratively analyze the flows and the system:

```
analyze( profiles )
{
  profiles = sorted(profiles, priority)
  for required in profiles
  {
    transmitted_nodes = []
    for receiver in required.receivers()
    {
      route = required.route()
      for node in route
      {
        if node in transmitted_nodes and multicast == true
        {
          continue
        }
        provided = node.provided
        [output, remaining, received] = convolve(required, provided)
        node.provided = remaining
        required = received
        transmitted_nodes.append(node)
      }
      [recv_output, recv_remaining, ] = convolve(required, receiver)
    }
  }
}
```

In this algorithm, the remaining capacity of the node is provided to each profile with a lower priority iteratively. Because of this iterative recalculation of node provided profiles based on routed profiles, we directly take into account the effect of multiple independent flows traversing the same router; the highest priority flow receives as much bandwidth as the router can give it, the next highest priority flow receives the remaining bandwidth, and so on.

We take care of matching all senders to their respective receivers, and ensure that if the system supports multicast, a no retransmissions occur; only nodes which must route the flow to a new part of the network retransmit the data. However, if the system does not support multicast, then the sender must issue a separate transmission, further consuming network resources. In this way, lower-level transport capabilities can be at least partially accounted for by our analysis.

We have implmented these functions for statically routed network analysis into our tool, which automatically parses the flow profiles, the network configuration and uses the algorithm and the implemented mathematics to iteratively analyze the network. Analytical results for example systems will be provided when the experimental results can be used as a comparison. The analysis algorithm is implemented by *`analysis.analyze_config()`*.

We are finishing the design and development of code which will allow us to run experiments to validate our routing analysis results. They will be complete in the next two weeks.

2.3.2 Run Time Results

Middleware-integrated Measurement, Detection, and Enforcement

We have implemented these features based on our design-time results:

- Traffic generators according to profile generated into sender code
- Receiver service according to profile generated into receiver code
- Measurement of output traffic on sender side and input traffic on server side generated into code
- Detection of anomalous sending on sender side
- Mitigation of anoumalous sending on sender side
- Detection of anomalous sending on receiver side
- Push back to sender middleware through out-of-band channel for anomaly detection on server side

Each of these functions uses the same profiles which enable design-time system and application analysis. This integration not only helps with running experiments and data collection but also helps to ensure model to system consistency.

We have implemented profile-based traffic generators and traffic measurement into our code generators that we use with our model-driven design software. We developed this toolsuite to create distributed component-based software which uses ROS as the communications middleware. For publish/subscribe interactions between components, into the generated code we add generic traffic generators which read their associated profile from the deployment XML file and publish traffic on their publisher port according to that profile. Additionally, these publish operations are generated to use a small wrapper which can measure the publish rate and can decide to throw a *profile exceeded* exception if the application attempts to send too much data or if the receiver has pushed back to the sender informing it to stop.

This push back from the receiver occurs through the use of an out-of-band (OOB) channel using UDP multicast, which receivers use to inform specific senders that they are sending too much data to the receivers (and possibly overflowing the receiver buffers). This OOB channel provides a mechanism by which the secure middleware layer can protect the system from malicious or faulty applications.

Into the receiver code (for subscribers) we additionally generate a receive buffer and receiver thread which reads the receiver profile from the deployment XML file and pulls data from the buffer according to the profile. In this scenario, the receiver has a capacity with which it can handle incoming data, and it has a finite buffer so it must use the OOB channel and measurements on the incoming data stream to determine which senders to shut down to ensure its buffer does not overflow. When the buffer has had some time empty (so that it's not in danger of running out of buffer space), the receiver can use the OOB channel to inform the halted senders that it is alright to send again.

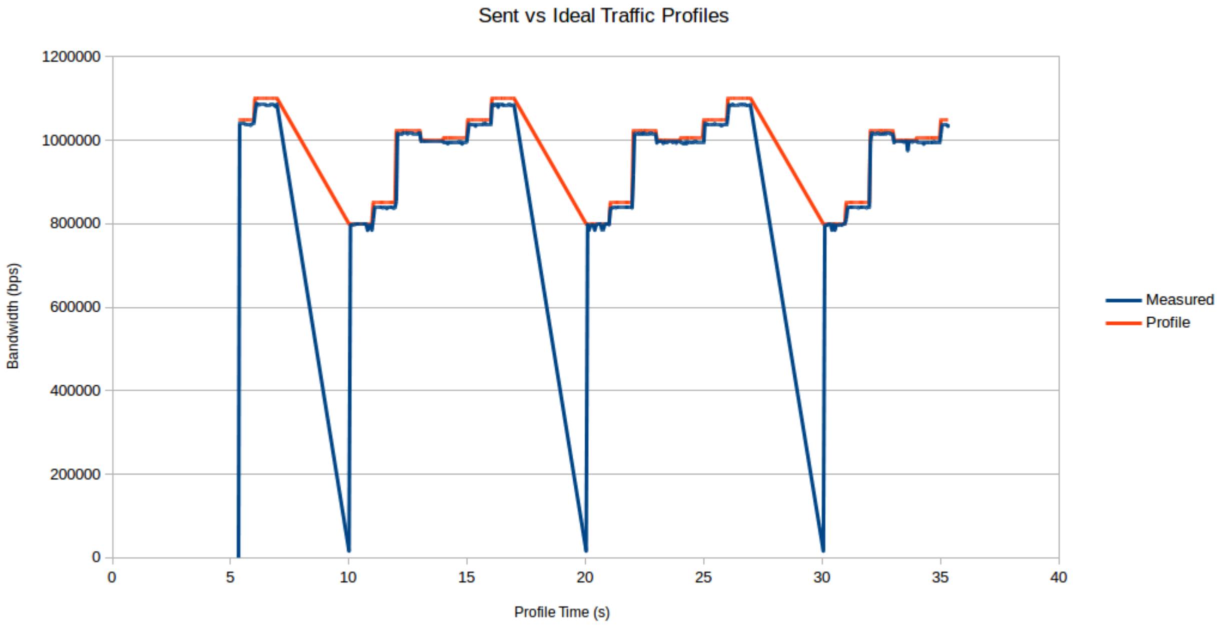


Fig. 2.8: Demonstration of the accuracy with which our traffic generators follow the specified profile.

Note: The measured bandwidth profile is calculated based on recorded time series data of $[reception\_time, message\_size]$, so the bandwidth drops to nearly 0 periodically since the Δt is so large between the messages.

Note: Our original implementation of traffic generators performed better since they did not utilize a middleware layer and relied instead on simple point to point ipv6 connections. However, that code was less useful for system analysis because it could do nothing aside from traffic generation and measurement; our current implementation which generates traffic generation code into component code is more versatile for several reasons:

- The component-based code integrates directly into our development toolsuite and deployment framework so it can be easily deployed on our cluster.

- Configuring different system topologies or component to host mappings (deployments) is simpler and more robust, allowing us to perform more and more varied experiments.
- The traffic generation code can be removed (or the code can be regenerated without the option selected) and the rest of the component-based and middleware code is still useful as an actual application.

Distributed Denial of Service (DDoS) Detection

DDoS attacks can take many forms, but are generally classified as excessive traffic from a large amount of (possibly heterogeneous) sources targeted towards a single point or a single group. Such attacks are common to machines on the internet, but can also become a hazard for machines on private networks which become infected or inadvertently expose a path for data traversal.

These private or semi-private systems must have mechanisms for detecting and mitigating such attacks, and the combination of our design-time analysis and run-time measurement, detection, and mitigation tools provides such capability.

If we relax the constraint from the design-time section that all sender profiles are absolute and the system behavior is completely known at design-time, then we not only expand the scope of applications that can be supported but also enable meaningful anomaly detection. Instead of treating the application profiles as the exact amount of data that the applications need to send, we assume they are the mean amount of data that the application will need to send over each respective interval. Additionally, we add an upper curve to the application profiles which is the guaranteed maximum that the application will ever try to send in each respective period.

For system analysis, we perform the design-time analysis on both the mean-data curve and the max-data curve which each give predicted buffer space requirements for the senders and receivers. Since buffer space on many of these systems comes at a premium, we cannot always take the maximum buffer size from this calculation as our implementation. For the senders, we are free to choose the allocated buffer space in the deployed system within this range as we see fit, since the senders have a large degree (or full) control over the sender rate and can meter themselves to not lose data. However, for the receivers, we can select a buffer size within the mean-max range based on our total system capacity and then rely on the measurement, detection, and mitigation code to ensure that we do not lose data due to buffer overflowing.

In this scenario, the receiver has been given knowledge of all of the sender profiles which might transmit data to it. When the receiver detects that the buffer is filling up quickly enough to overflow the buffer in the near future, it can use the measurements that have been gathered about the received traffic up to this point to determine which sender components to throttle temporarily.

We shown experimentally that, for example, a server side buffer size of 400000 bits, which would normally grow to 459424 bits because of excessive data pumps on the sender sides, is kept to 393792 by utilizing this out-of-band channel and secure middleware.

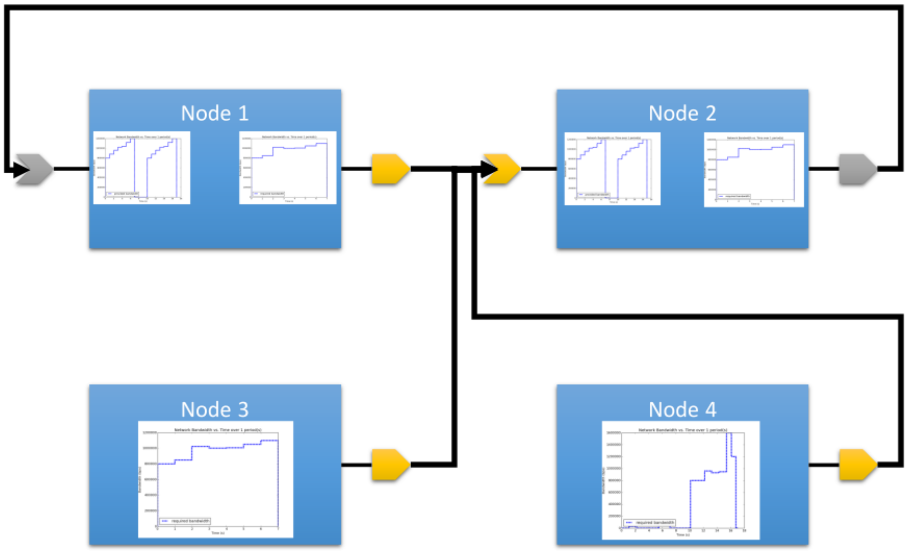


Fig. 2.9: The nodes in the network and how they communicate (using pub/sub).

CHAPTER

THREE

USING THE CODE

THIS SECTION IS NOT COMPLETE

Explain here how to use the code with specific references to the API and give an example to help people set up an example system and analyze it

3.1 The Analysis Tool

The analysis tool is a python library which implements both the Network Calculus and MAReN techniques described above.

3.1.1 Installation

0. Install tabulate for nice formatting of the output:

   ```
   sudo pip install tabulate
   ```

1. Download the analysis tool from the CBSAT repo.

3.2 The Middleware

Note: The middleware is C++ and supports Linux.

3.2.1 Compilation

1. To Build the client and server test of the middleware, run from a terminal:

   ```
   make
   ```

Congratulations! The set-up of the analysis tool and the middleware are complete!

CHAPTER

FOUR

THE INNER WORKINGS

This page acts as a launchpad for diving into the different aspects of the code, the interfaces provided by each submodule, and learning how everything works behind the scenes.

4.1 Analysis API

4.1.1 Analysis

This program is designed to analyze network performance of distributed applications in a networked system. Its analysis techniques are based on Network Calculus and provide deterministic analysis of networks and network applications. By analyzing the Quality of Service (QoS) that the system network provides to the applications and users, we can determine the buffer space required for the applications to communicate losslessly as well as the buffering delay experienced by the network traffic.

This program in particular implements these calculations and is able to load, parse, and analyze network profiles and configuration files describing the system, the network flows, and the time-dependent traffic generation or service profiles associated with the applications or the system, respectively.

`analysis.`**`analyze_profile`**`(`*required*, *provided*, *config*, *options*`)`

- Calculates the hyperperiod of the profiles
- Repeats the profiles for the specified number of hyperperiods in *options*
- Analyzes the requested profiles
- If more than one hyper-period has been specified it determines system stability
- Optionally plots the bandwidths and data for the profiles

Parameters

- **`required`** (*in*) – `networkProfile.Profile` describing the required profile
- **`provided`** (*in*) – `networkProfile.Profile` describing the provided profile
- **`config`** (*in*) – `networkConfig.Config` describing the configuration of the network
- **`options`** (*in*) – `Options` describing the program options for drawing and analysis

Returns a list of analysis results consisting of:

```
[ output, remaining, max delay, max buffer ]
```

- The output profile as a `networkProfile.Profile` generated by calling **required.** `networkProfile.Profile.Convolve()` (provided)

- The remaining capacity profile as a *networkProfile.Profile* which is determined as $remaining = (provided - output)$
- The delay structure generated by calling **required.** *networkProfile.Profile.CalcDelay()* (output)
- The buffer structure generated by calling **required.** *networkProfile.Profile.CalcBuffer()* (output)

analysis.**parse_profiles**(*config*, *options*)

analysis.**analyze_config**(*config*, *options*)

This function analyzes the system configuration in flow priority order, taking into account system-level concepts such as multicast capabilities. It performs the following steps:

- sort the sender profiles by priority
- retrieve from the system config all receiver profiles associated with this flow type
- for each receiver:
  - get the route the flow will take from the sender to the receiver
  - for each node along the route:
    - analyze the flow's profile with the node's provided profile
    - set the node's provided profile to the remaining profile
    - set the flow's required profile to the received profile
  - analyze the flow's profile with the receiver's profile

analysis.**main**(*argv*)

Performs the main analysis of the profiles using the following steps:

- Parses the command line options according to the *Options* specification.
- Loads the specified network configuration
- Parses the files in to separate profiles
- Analyzes the system configuration

class analysis.**Options**

--help	(to show this help and exit)
--nc_mode	(to run network calculus calcs)
--no_plot	(to not output any plots)
--no_profile_name	(to not plot 'profile_name', e.g. 'required')
--print	(to print the profiles as they are analyzed)
--required	<fileName containing the required profile>
--provided	<fileName containing the provided profile>
--receiver	<fileName containing the receiver profile>
--profile_folder	<path containing profiles to be loaded>
--network_config	<file containing network configuration>
--num_periods	<number of periods to analyze>

--nc_step_size <step size for time-windows in NC mode>

`plot_profiles` = **None**
plot the profiles?

`plot_dict` = **None**
dictionary with plot options generated

`print_profiles` = **None**
print the profiles?

`num_periods` = **None**
number of periods to analyze

`plot_line_width` = **None**
line width for plots

`font_size` = **None**
font size for plots

`nc_mode` = **None**
analyze using network calculus techniques?

`nc_step_size` = **None**
step size for network calculus analysis

`required_fileName` = **None**
what file to load as the required profile

`provided_fileName` = **None**
what file to load as the provided profile

`receiver_fileName` = **None**
what file to load as the receiver profile

`profile_folderName` = **None**
path to a folder which contains all the profiles to be analyzed

`network_configName` = **None**
file which contains the topology and configuration of the network

`parse_args` (*args*)

`print_usage` (*name*)

4.1.2 Network Profile

Network Profile implements the Profile class. This class provides all the members and functions neccessary to model, compose, and analyze network profiles for applications and systems.

class `networkProfile.`**`Profile`** (*kind=None*, *period=0*, *priority=0*, *node=0*, *flow_type=None*, *num_periods=1*, *sender_names=[]*)
Profile contains the information about a single network profie. A network profile has a kind (e.g. 'provided'), a period (in seconds), and a lists of relevant data vs time series (e.g. bandwidth, latency, data, etc.).

Parameters

- **`kind`** (*string*) – what kind of profile is it?
- **`period`** (*double*) – what is the periodicity (in seconds) of the profile
- **`priority`** (*int*) – what is the priority of the flow in the system
- **`source`** (*int*) – what is the node id from which the data on this profile will be sent

- **dest** (*int*) – what is the node id to which the data on this profile will be sent

field_delimeter = ','
Sepearates fileds in a line in a profile file

header_delimeter = '#'
Denotes headers (profile properties) in a profile file

comment_delimeter = '%'
Denotes commends in the profile file

line_delimeter = '\n'
Splits lines in a profile file

special_delimeters = ['#', '%']
Strip lines starting with these delimeters to get just profile data

interpolated_profiles = ['data', 'latency']
Which profiles are interpolated between points

kind = None
The kind of this profile, e.g. 'required'

period = None
The length of one period of this profile

priority = None
The priority of the profile; relevant for 'required' profiles

node_id = None
The node ID which is the source of this profile

flow_type = None
This flow is the reciever for which sender flows?

entries = None
Dictionary of 'type name' : 'list of [x,y] points' k:v pairs

ParseHeader(*header*)
Parses information from the profile's header if it exists:

- period
- priority
- node ID
- flow_type (for matching senders <–> receivers)
- profile kind (provided, required, receiver, output, leftover)

A profile header is at the top of the file and has the following syntax:

```
# <property> = <value>
```

ParseFromFile(*prof_fName*)
Builds the entries from a properly formatted CSV file. Internally calls `Profile.ParseFromString()`.

ParseFromString(*prof_str*)
Builds the entries from a string (line list of csv's formatted as per `ParseEntriesFromLine()`).

ParseEntriesFromLine(*line_str*)
Builds the [time, value] list for each type of value into entries:

- slope
- max slope
- latency

These values are formatted in the csv as:

```
<time>, <slope>, <max slope>, <latency>
```

`EntriesRemoveDegenerates`()

Remove duplicate entries by time stamp.

`AggregateSlopes`()

Remove sequential entries which have the same slope.

`EntriesStartFill`()

Make sure all entries have a start time of 0.

`Repeat`(*num_periods*)

Copy the current profile entries over some number of its periods.

`Integrate`(*time*)

Integrates the slope entries to produce data entries up to *time*

`Derive`()

Derives the slope entries from the data entries

`IsKind`(*kind*)

Returns True if the profile is of type *kind*, False otherwise.

`Kind`(*kind*)

Set the kind of the profile.

`Shrink`(*t*)

Shrink the profile to be <= *t*.

`AddProfile`(*profile*)

Compose this profile with an input profile by adding their slopes together.

Return type `Profile`

`SubtractProfile`(*profile*)

Compose this profile with an input profile by subtracting the input profile’s slopes.

Return type `Profile`

`MakeGraphPointsSlope`()

Return matplotlib plottable x and y series for the slope of the profile.

`MakeGraphPointsData`()

Return matplotlib plottable x and y series for the data of the profile.

`GetValueAtTime`(*key*, *t*, *interpolate=True*)

Return the value at time *t* from series *key*, optionally interpolating between.

`ToString`(*prefix=’‘*)

Returns a string version of the profile, with all values properly tabulated.

Return type `string()`

Parameters **`prefix`** (*in*) – string to be prepended to every line of the returned string.

ConvertToNC (*filterFunc*, *step=0*)

Perform time-window based integration to generate a Network Calculus curve from the profile. The conversion is configurable based on time-window step-size and a filter function (e.g. min or max). Passing `max()` will create an arrival curve, while passing `min()` will create a service curve.

Return type *`Profile`*, the network-calculus version of the *self* profile

Note: Requires the profile to have been integrated

CalcDelay (*output*)

Compute the maximum horizontal distance between this profile and the input profile.

This function implements the operation (see *Formalism for Precise Performance Prediction for Networks*):

$$delay = sup\{l^{-1}[y] - r^{-1}[y] : y \in \mathbb{N}\}$$

Where

- $l^{-1}[y]$ is the inverse map of the ouptut profile, e.g. a function mapping output data to time
- $r^{-1}[y]$ is the inverse map of the required profile, e.g. a function mapping required data to time

Return type

`list()` of the form:

```
[ <time>, <data>, <length of delay> ]
```

Parameters **`output`** (*in*) – a *`Profile`* describing the output profile

CalcBuffer (*output*)

Compute the maximum vertical distance between this profile and the input profile.

This function implements the operation (see *Formalism for Precise Performance Prediction for Networks*):

$$buffer = sup\{r[t] - l[t] : t \in \mathbb{N}\}$$

Where

- $l[t]$ is the output profile (see *`Profile.Convolve()`*)
- $r[t]$ is the required profile (*self*)

Return type

`list()` of the form:

```
[ <time>, <data>, <size of the buffer> ]
```

Parameters **`output`** (*in*) – a *`Profile`* describing the output profile

Delay (*delayProf*)

Compute the delayed profile composed of *self* profile and *delayProf*, received by a node for which this *self* profile is the output profile on the sender side. The delay profile describes the delay as a function of time for the link.

This function implements the operation:

$$o[t + \delta[t]] = l[t]$$

Where

- $\delta[t]$ is the delay profile
- $l[t]$ is the profile transmitted into the link (*self*)
- $o[t]$ is the output profile received at the other end of the link

Return type `Profile`, $o[t]$

Parameters **`delayProf`** (*in*) – `Profile` describing the delay

`Convolve`(*provided*)

Use min-plus calculus to convolve this *required* profile with an input *provided* profile.

This function implements the operation (see *Formalism for Precise Performance Prediction for Networks*):

$$y = l[t] = (r \otimes p)[t] = min(r[t], p[t] - (p[t-1] - l[t-1]))$$

Where

- $r[t]$ is the data profile required by the application (*self*)
- $p[t]$ is the data profile provided by the node's link
- $l[t]$ is the data profile transmitted onto the link

Return type `Profile`, $l[t]$

Parameters **`provided`** (*in*) – a `Profile` describing the node's provided link profile

4.1.3 Network Config

Network Config implements classes related to node-based flow/profile aggregation, routing, link management, and management of system level concerns such as multicast-capability.

class `networkConfig.`**`Node`**(*_id*)

Defines all the required information for a node in the network. This includes:

- All provided profiles (aggregated) whose node_id is this node

`id_type`

alias of `str`

`ID` **= None**

the ID of this node

`provided` **= None**

aggregate of all 'provided' profiles whose source ID is this node

`HasProfiles`()

`AddProfile`(*prof*)

`AddProvidedProfile`(*prof*)

class `networkConfig.`**`Route`**(*path=[]*)

Describes how a flow traverse the links of the system's network. This is specified as a list of nodes, with the source node at the front of the list and the destination node at the end of the list.

`header` **= 'route:'**

line header specifying a route in the config file

`path` **= None**

list of node IDs with a source, intermediate nodes, and a destination

AddDest (*dest*)
Append a node onto the end of the route.

AddSource (*src*)
Add a node onto the beginning of a route.

InsertNode (*node*, *pos*)
Insert a node into the route before the given position.

ParseFromLine (*line*)
Handles parsing of a route path from a line in the config file. A route is defined as:

```
route: src_node_id, hop_node_1, ... , hope_node_n, dst_node_id
```

Length ()

class networkConfig.**Topology** (*links={}*)
Describes the active links between nodes on the system's network. This is specified as a dictionary of node : list of nodes pairs.

header = 'topology:'
line header specifying a topology link in the config file.

ParseFromLine (*line*)
Handles parsing of a link from a line in the config file. A topology is defined as:

```
topology: src_node_id : direct_node_1, ... , direct_node_n
```

class networkConfig.**Config** (*nodes={}*, *multicast=False*, *retransmit=False*, *routes=[]*, *topology={}*)
Contains the routing and topology information to fully describe the system's network and provide a mapping between application data flows (logical) and the system's network links. It also provides interfaces for setting low-level communications considerations such as retransmission, multiple-unicast, multicast, etc.

addProfile (*prof*)

GetRoute (*src*, *dst*)
Returns the path for the flow from *src* to *dst*.

ParseHeader (*header*)
Parses information from the configuration's header if it exists:

- multicast capability
- retransmission setting

A profile header is at the top of the file and has the following syntax:

```
# <property> = <value>
```

ParseFromFile (*fName*)
Builds the entries from a properly formatted CSV file. Internally calls *Config.ParseFromString()*.

ParseFromString (*conf_str*)
Handles parsing of the header, topology, and routes in a config file.

networkConfig.**main** (*argv*)

4.1.4 Plotting

class plotting.**PlotOptions** (*profileList*, *labelList*, *dashList*, *line_width*, *annotationList*, *title*, *xlabel*, *ylabel*, *legend_loc*)
Options for setting up a plot in a figure.

Parameters

- **profileList** (*list*) – A list of [x,y] data series (profiles) to be plotted together
- **labelList** (*list*) – A list of strings which label the profiles
- **dashList** (*list*) – A list of integer lists which specify the dash properties for each profile
- **annotationList** (*list*) – A list of annotations to be added to the plot
- **line_width** (*int*) – The thickness of the lines on the plot
- **title** (*string*) – The title to be given to the figure
- **xlabel** (*string*) – The label for the x-axis
- **ylabel** (*string*) – The label for the y-axis
- **legend_loc** (*string*) – A string specifying the location of the legend, e.g. "upper left"

`plotting.`**plot_bandwidth_and_data**(*profList*, *delay*, *buffer*, *num_periods*, *plot_line_width*)

Parameters

- **profList** (*in*) – a list of `networkProfile.Profile` to be plotted
- **delay** (*in*) – a delay structure as generated from `networkProfile.Profile.Convolve()`
- **buffer** (*in*) – a buffer structure as generated from `networkProfile.Profile.Convolve()`
- **num_periods** (*in*) – how many periods the plot covers
- **plot_line_width** (*in*) – how thick the lines for each plot should be

`plotting.`**makeGraphs**(*pOptionsList*)

This function makes a figure for each PlotOptions object it receives in the list.

Parameters **pOptionsList** (*list*) – A list of `PlotOptions` describing the figures to be drawn

`plotting.`**clearAnnotations**()

`plotting.`**addAnnotation**(*annotation*)

Adds an annotation to the currently active figure.

Parameters **list annotation** (*in*) – a `list()` of the form [<string>, <x position>, <y position>]

`plotting.`**setFigureOpts**(*title*, *ylabel*, *xlabel*, *legend_loc*)

Configure the figure's options

Parameters

- **title** (*string*) – The title to be given to the figure
- **xlabel** (*string*) – The label for the x-axis
- **ylabel** (*string*) – The label for the y-axis
- **legend_loc** (*string*) – A string specifying the location of the legend, e.g. "upper left"

`plotting.`**disablePlotTicks**()

Disable the numbers on the x and y axes.

4.1.5 Utils

class `utils.`**`bcolors`**

Extended characters used for coloring output text.

`HEADER` **= '\x1b[95m'**

`OKBLUE` **= '\x1b[94m'**

`OKGREEN` **= '\x1b[92m'**

`WARNING` **= '\x1b[93m'**

`FAIL` **= '\x1b[91m'**

`ENDC` **= '\x1b[0m'**

`BOLD` **= '\x1b[1m'**

`UNDERLINE` **= '\x1b[4m'**

`utils.`**`lcm`**(*a*, *b*)

Returns the least-common-multiple (LCM) of *a* and *b* as

$$lcm = (a * b)/gcd(a, b)$$

`utils.`**`makeVLine`**(*v*)

Returns a list of [x,y] series for plotting a vertical line.

Parameters **`v`** (*list*) – A list of values of the form:

```
        [ <bottom x location>, <bottom y location>, <height> ]
```

`utils.`**`makeHLine`**(*h*)

Returns a list of [x,y] series for plotting a horizontal line.

Parameters **`h`** (*list*) – A list of values of the form:

```
        [ <left x location>, <left y location>, <length> ]
```

`utils.`**`remove_degenerates`**(*values*)

Make sure all value pairs are unique and sorted by time.

`utils.`**`repeat`**(*values*, *period*, *num_periods*)

Repeat values periodically for some number of periods.

`utils.`**`aggregate`**(*values*)

Remove any sequential entries with the same value.

`utils.`**`integrate`**(*values*, *t*)

Integrate all the values cumulatively and return the integrated values.

`utils.`**`derive`**(*values*)

Derive all the entries slopes from their data.

`utils.`**`split`**(*values*, *t*)

Remove and return every entry from *values* whose time > *t*.

`utils.`**`shift`**(*values*, *t*)

Add *t* to every value in *values*.

`utils.`**`get_index_containing_time`**(*values*, *t*)

Get the index of a value in *values* which contains time *t*

Parameters

- **values** (*list*) – a list() of [x,y] values
- **t** (*double*) – time value for indexing

utils.**get_value_at_time**(*values*, *t*, *interpolate=True*)
Get the value at the given time *t* from *values*

Parameters

- **values** (*list*) – list() of [x,y] values
- **t** (*double*) – time value
- **interpolate** (*bool*) – is the value interpolated or constant between values

utils.**get_times_at_value**(*values*, *value*, *interpolate=True*)
Get a list of times at which *values* match *value*.

Parameters

- **values** (*list*) – a list() of [x,y] values
- **value** (*double*) – value to test against
- **interpolate** (*bool*) – is the value interpolated or constant between values

utils.**subtract_values**(*values1*, *values2*, *interpolate=True*)
Subtract *values2* from *values1*, using either interpolated values or constant values.

utils.**add_values**(*values1*, *values2*, *interpolate=True*)
Add *values2* to *values1*, using either interpolated values or constant values.

utils.**max_vertical_difference**(*values1*, *values2*, *interpolate=True*, *epsilon=0.1*)
Get maximum vertical difference of *values2* - *values1*.

utils.**max_horizontal_difference**(*values1*, *values2*, *interpolate=True*, *epsilon=1e-06*)
Get maximum horizontal difference of *values2* - *values1*.

utils.**convert_values_to_graph**(*values*, *interpolate=True*)
Make the *values* plottable by separating the x,y values into separate lists.

utils.**get_intersection**(*p11*, *p12*, *p21*, *p22*)
Simple function to get a intersection of two lines defined by their endpoints

Parameters

- **p11** – list() [x,y] starting point of line 1
- **p12** – list() [x,y] ending point of line 1
- **p21** – list() [x,y] starting point of line 2
- **p22** – list() [x,y] ending point of line 2

4.1.6 Generate TDMA

class generateTDMA.**Options**(*tdmaPeriod=0.01*, *tdmaSlots=2*, *node='notdma'*, *outputFilename='tdma'*, *activeSlot=0*)

parseArgs(*args*)

class generateTDMA.**TDMA**(*period*, *slots*, *selectedSlot*)
A TDMA period describes when nodes can transmit in which slots on the network during a given period.

Parameters

- **period** (*double*) – the period of the TDMA profile
- **slots** (*int*) – number of slots this TDMA profile has
- **selectedSlot** (*int*) – which slot are we analyzing?

class generateTDMA.**ProfileEntry** (*start=-1*, *end=-1*, *bandwidth=-1*, *interface=''*)

fromLine (*line=None*)

generateTDMA.**generateNewProfile** (*oldProfile*, *tdma*)

Create a TDMA profile network Profile from a regular network Profile

Parameters

- **oldProfile** (*list*) – list of networkProfile.ProfileEntry describing the original profile
- **tdma** (*class*) – a *generateTDMA.TDMA* object describing the TDMA scheme

generateTDMA.**get_nodeProfiles** (*folder*)

generateTDMA.**generateProfile** (*txtProfile*)

generateTDMA.**main** ()

4.2 Middleware API

4.2.1 Network Profile

class profileMemBuf

This structuture creates a stream buffer for parsing csv profile files.

class ResourceEntry

time

Contains the start time for the resource entry.

Return type float

bandwidth

The bandwidth (bps) which is constant from the start of the entry to its end.

Return type unsigned long long

data

The cumulative data (bits) which have been sent by the end of this resource entry. Includes all summation of all previous entries' data.

Return type unsigned long long

latency

The latency (ms) for network traffic during this entry.

Return type unsigned long long

class NetworkProfile

A network profile contains a sorted list of time- and data-contiguous entries of type *ResourceEntry*. The profiles are periodic with a specific epoch-centric start-time.

resources

Return type std::vector<ResourceEntry>

start__time

Return type timespec

period

Return type double

initialize_from_file (*fname*)

Load in the profile specified by *fname*. Return 0 on success, -1 on error.

Parameters **char* fname** (*const*) – The filename containing a csv-delimited profile

Return type int

initialize_from_string (*buffer*)

Load in the profile contained in *buffer*. Return 0 on success, -1 on error.

Parameters **buffer** (*char**) – A string buffer containing the csv-delimited profile

Return type int

initialize_from_i_stream (*stream*)

Load in the profile contained in *stream*. Return 0 on success, -1 on error.

Parameters **stream** (*std::istream&*) – An istream containing the csv-delimited profile

Return type int

get_offset (*t*)

Returns the difference between *t* and the profile's start time, modulo the profile's period.

Parameters **timespec& t** (*out*) – epoch-centric time value

Return type double

get_next_interval (*start*, *bandwidth*, *latency*)

Returns as output parameters the next interval by comparing the current system epoch time to the profile's start epoch time. IF the profile has not been properly initialized, the call fails and returns -1, else it fills the output parameters and returns 0.

Parameters

- **timespec& start** (*out*) – epoch time when the next interval starts
- **unsigned long long& bandwidth** (*out*) – bandwidth during the next interval
- **unsigned long long& latency** (*out*) – latency value for the next interval

Return type int

delay (*data_len*, *sent_time*)

Returns the amount of time the program has to wait before sending again. This is calculated based using the input *data_lengh* that was last transmitted at *sent_time*, and takes into account the current system itme.

Parameters

- **unsigned long data_len** (*in*) – size of the message that was last sent
- **timespec sent_time** (*in*) – epoch time the message of length dataLen was sent

Return type double

initialized()

Returns true if the profile was properly initialized, false otherwise.

Return type bool

4.2.2 Network Buffer

class **NetworkBuffer**

A network profile contains a sorted list of time- and data-contiguous entries of type *ResourceEntry*. The profiles are periodic with a specific epoch-centric start-time.

max_size()

Return the maximum size that the buffer has reached.

Return type long

size()

Return the current size of the buffer.

Return type long

capacity()

Return the capacity of the buffer.

Return type long

capacity(*_capacity*)

Set the capacity of the buffer.

Parameters **long __capacity** (*in*) – new capacity for the buffer

Return type void

push(*data*)

Add data to the buffer if $data.size < (capacity - size)$. Return 0 on success, -1 on failure.

Parameters **message* data** (*in*) – message to be added to the buffer

Return type int

pop(*data*)

Returns 0 for successful data retrieval from the buffer, -1 otherwise.

Parameters **message*& data** (*in-out*) – Message pointer to data retrieved

Return type int

4.2.3 Network Middleware

4.2.4 Message

4.2.5 Client

4.2.6 Server

4.2.7 Connection Subsys

4.2.8 CSV Iterator

4.2.9 TC Wrapper

genindex

PYTHON MODULE INDEX

INDEX