Semester Project Report

Formal Verification of Real-time Networks with Coq Proof Assistant

Author
Andrea Gilot

Professor
Jean-Yves Le Boudec
Supervisor
Hossein Tabatabaee

Fall 2020

Contents

Chapter 1

Introduction

Deadline violations in real-time networks such as avionics, automotive, and industrial networks can cause catastrophic damage; hence, it is necessary to provide formal delay certifications for such networks. An objective of formal timing verification approaches is to find trusted upper bounds on worst-case delays.

Network Calculus is a standard approach to find formal delay bounds for the worst case. However, a common problem with all these approaches is that they all rely on human-made proofs, i.e., they are viewed and verified by humans. Complex, subtle hypotheses used in such proofs imply a risk of error. There are several real-time analysis that were reviewed and considered to be valid at the time only to be refuted later on [1].

A proof assistant, such as Coq [2], can automate the proof-reading process [3]. We can use Coq to formally define mathematical objects, enunciate theorems, and finally describe proofs of these theorems. A computer is then able to automatically check these proofs. Despite not being extremely popular, Coq has demonstrated its capabilities, especially with the proof of the four colour theorem [4].

Coq provides several advantages. First, it increases the confidence in the correctness of proofs. Second, Coq enables us for a safe reuse of previous results. Specifically, people may mistakenly use previous results while some implicit hypotheses are missed; however, Coq only reuses previous results if all the hypotheses hold. Third, with Coq, we know which hypotheses are used where and how in proofs; in some cases, some hypotheses are found to be redundant, i.e., Coq still verifies the proof even if we remove these hypotheses. We can thus achieve simpler, more generalized results.

Authors in [3], with Coq, formalize a subset of the theory that is large enough to handle a complete proof of network calculus bounds on a simple case study. They first formalize the dioid algebraic structure in the mathematical library of Coq. This enables them to formally define network calculus definitions, such as arrival curves and service curves, and network calculus theorems, such as delay bounds and output arrival curves. Then, with Coq, they formally verify the proofs of these theorems. Note that only an expert in network calculus can approve whether definitions and theorems are well defined.

To use Coq to provide certification for a network like AFDX (Avionics Full-DupleX), we only need to define packetizer and non-preemptive static priority scheduler. My work is an extension of [3]. In this project, first, I defined packetizer and I formally verified several theorems about the impact of packetizer. Specifically, consider bit-based system that is followed by a packetizer; then, for the combined system, I have verified in Coq the impact of packetizer in the network calculus framework [5, Theorem 1.7.1], namely, the per packet delay, the maximum backlog, a minimum service curve, and an output arrival curve. Second, I formally defined a non-preemptive static priority scheduler with two inputs, that can be furthe generalized to any number of inputs. However, this part is still in progress and is not completed.

My project is also a base to automatize proofs and provide them on-demand in a function of the network topology. In fact, Coq is divided in few sub-languages, one of them being ltac [6], which is aimed to build programs that seek to prove theorems or part of them without human interaction. This powerful tool greatly facilitates the maintenance of the program and makes it very flexible.

Chapter 2

Work done

2.1 Problem

Safety critical networks, such the as ones used in avionic and automotive industry, must ensure guaranteed delays. In fact, unexpected delays in such applications may have fatal consequences. Network calculus is a theory providing bounds on network delays.

This mathematical theory currently relies on, human-made, pen and paper proofs. However, avionic applications must go through a certification process before being deployed on the field. Such certifications require a formal, rigorous verification process aimed to reduce as much as possible the risk of error. Coq proof assistant [2] is an ideal solution to meet this requirement. By checking the correctness of such proofs, it can certify calculated delays for these networks. So, the goal is to provide automated certifications that are formally checked by a computer.

Authors in [3] have formalized a subset of the theory large enough to handle a complete proof of network calculus bounds on a simple case study; this is a great step toward providing formal, automated certifications. However, there are some shortcoming. First, [3] only implements FIFO policy at servers. Second, it does not taken into account the impact of a packetizer. Lastly, any changes in the network topology or flows requires changing the Coq code manually.

My work is in the extension of the work done in [3] to enable Coq to provide automated certifications for networks such as Airbus networks. To do so, I first define, in Coq, a packetizer and prove the network calculus theorems regarding its impact [5, Theorem 1.7.1]. Second, I define a non-preemptive static prority (NP-SP) scheduler for two inputs in Coq. Generalizing the NP-SP for n inputs, and providing automated on-demand certification in a function of the network topology are left for further study.

2.2 Solution

My project work basically consisted in defining mathematical objects and proving several theorems about them. Most of those proofs concern basic mathematics and are a basis to define more complex objects. It is thus composed of 10 files. This section will mainly focus on the main features of my project, namely on the definition of the packetizer, including the formal verification of theorems about it, and the formal definition of a non-preemptive static priority scheduler.

2.2.1 Packetizer

When we usually work with a L-packetizer (see Appendix B for more information), L is a sequence of cumulative packets length, that does not converge and such that $L(0) = 0$. In this project, I decided to work with the sequence of non cumulative packets lengths, l. We can however easily recover L as $\forall n \in \mathbb{N},\ L_n = \sum_{i=0}^{n} l_i$. For the sake of simplicity, I assumed that l was a natural sequence and that it did not contain any term equal to 0 expect for the first one. In fact, a term equal to 0 would be meaningless and would not modify the behavior of a packetizer if removed. All the results are also valid if l is a real sequence as L, its partial sum does not converge. Another important assumption is the existence of a maximum length $l_{\max}$ which will be useful throughout the rest of the section.

Let's define the set of all the terms of L that are lower than a given value:

$$\begin{aligned} B_L : \mathbb{R}^+ &\to \mathcal{P}(\mathbb{R}^+) \\ x &\mapsto \{L_n : L_n \leq x\} \end{aligned} \tag{2.1}$$

By the completeness axiom of the real numbers, a subset of $\mathbb{R}$ has a supremum if and only if it is bounded and non empty. Hence, B_L has a supreme if and only if

$$\exists v \in \mathbb{R}^+ : \forall x \in \mathbb{R}^+, B_L(x) \leq v \ (B_L \text{ is bounded}) \tag{2.2}$$

$$\forall x \in \mathbb{R}^+, 0 \in B_L(x) \ (B_L \text{ is not empty}) \tag{2.3}$$

The supremum of this set is the greatest term of L that is lower than a given value. If $\forall n \in \mathbb{N},\ L_n = 1$, this is just a floor function. P_L represents such a function.

$$\begin{aligned} P^L : \overline{\mathbb{R}}^+ &\to \overline{\mathbb{R}}^+ \\ x &\mapsto \begin{cases} \sup\{B_L(x)\}, & \text{if } x \in \mathbb{R}^+ \\ \infty, & \text{if } x = \infty \end{cases} \end{aligned} \tag{2.4}$$

In Coq, the definition of P_L is:

```
Definition packetization (l: nat → nat) (ls: LengthSequence l) (v: Rbar): Rbar :=
  match v with
  |Finite x ⇒ last_cumul l im_0 x
```

```
|p_infty ⇒ p_infty
|m_infty ⇒ 0
end.
```

where *ls* is a proof that *l* has the correct properties (it is never equal to 0 expect for its first term...) and *last_cumul l im_0 x* is the supremum of $B_L(x)$ when $x \geq 0$ and 0 otherwise.

An L-packetizer is just a system that applies P_L to the input. In Coq it gives

```
Definition packetizer (a: R → Rbar) (l: nat → nat) ls : R → Rbar := fun t ⇒ packetization l ls (a t).
```

where a is the input flow.

However, here *packetizer* is a function where as formally, a system is a binary relation from $\mathcal{C}$ to $\mathcal{C}$ that fulfils given conditions (see Appendix A.2). In fact it should be

$$(A, D) \in S \iff D = P_L(A) \tag{2.5}$$

```
Definition packetizer_serv `(ls: LengthSequence l) := fun a d ⇒ Cumulative a ∧ d = packetizer a l ls.
```

We first need to prove that this relation actually describes a server. So I proved the following properties:

$$\forall A \in \mathcal{C}, P_L(A) \in \mathcal{C} \tag{2.6}$$

$$\forall x \in \mathbb{R}^+,\ x - l_{\max} < P_L(x) \leq x \tag{2.7}$$

The property (2.7) implies that $D = P_L(A) \leq A$ while (2.6) ensures that the output is always cumulative. Thus, our definition describes a server. The identities (2.7) is also really interesting as it define bounds for P_L which is a goal of network calculus.

Moreover, we seek to analyze the behavior of a system combined with a packetizer and for this I needed to prove the following propositions:

$$\forall x \in \mathbb{R}^+\ \exists n \in \mathbb{N} : P_L(x) = L(n) \tag{2.8}$$

$$\forall x \in \mathbb{R}^+,\ P_L(x) = L(n) \iff L(n) \leq x < L(n+1) \tag{2.9}$$

The identity (2.8) is an other way to say that the image of P_L is a subset of the image of L. The equivalence (2.9) is an alternative definition of P_L and comes to be convenient when proving theorems in relation to the packetizer.

Everything is now set up to study the behavior of a combined system, where two systems are concatenated (i.e., the input of the first system is the output of the second one). So formally, the binary relation describing the concatenation of two systems S_1 and S_2 is:

$$(A, D) \in S \iff \exists D' \in \mathcal{C} : (A, D') \in S_1 \wedge (D', D) \in S_2 \tag{2.10}$$

Even if at first glance it seems complex and difficult to work with, this definition is actually easily transposable and easy-to-use in Coq.

```
Definition server_concat `(S: Server s) `( T: Server t) := fun a c ⇒ ∃ b, (s a b) ∧ (t b c).
```

Let S_{BB} be a server with a minimal service curve β, S_L a packetizer and S the concatenation between the two servers. Let A be a packetized input of S_{BB}, D' the output of S_{BB} and thus the input of S_L and D the output of S_L. Let be α an arrival curve of D'.

$$d(A, D') = d(A, D) \tag{2.11}$$

$$b(A, D') \leq b(A, D) \leq b(A, D') + l_{\max} \tag{2.12}$$

$$[\beta(t) - l_{\max}]^+ \text{ is a minimal service curve of } S \tag{2.13}$$

$$\alpha'(t) = \begin{cases} 0 \text{ if } t = 0 \\ \alpha(t) + l_{\max} \text{ otherwise} \end{cases} \quad \text{is an arrival curve for } D. \tag{2.14}$$

The property (2.11) shows that if the input packets are already packetized, repacketize them after going through the server has no cost in terms of delay (see Appendix A.3). However, it has an impact on the backlog of the combined system. A bound for this backlog is given by the identity (2.12). Finally, we can easily recover the service curve of the combined system (see Appendix A.4) and an arrival curve of the output flow which are essential elements of Network Calculus.

In Coq the statements of the theorems (2.12), (2.13) and (2.14) respectively are:

```
Theorem combined_backlog `(S: Server s) `( ls : LengthSequence l) : ∀ a d d', is_finite_f a →
s a d → packetizer_serv ls d d' → Rbar_le(backlog a d) (backlog a d') ∧
Rbar_le (backlog a d') (Rbar_plus (backlog a d) (INR(lmax ls))).
```

where *is_finite_f a* means that $\forall t \in \mathbb{R}^+ A(t)$ is finite and *packetizer_serv ls d d'* means that $(D', D) \in S_L$ and *INR* is the function that converts natural numbers into real ones.

```
Theorem combined_service `(S: Server s) `( ls : LengthSequence l)
`( Beta: NonDecreasingPositive beta): is_minimal_service S Beta →
is_minimal_service (CombinedSystem S ls) (NonDecrFPlus (non_decr_plus non_decr (−INR (lmax ls)))).
```

```
Theorem packetizer_arrival_curve `(A: Cumulative a) `( alpha: NonDecreasingPositive al)
`( ls: LengthSequence l) : is_finite_f a → is_arrival_curve a alpha →
is_arrival_curve (packetizer a l ls) (NonDecrPlusLmax alpha ls).
```

where the content of the last parentheses in both statements is a proof that respectively $[\beta(t) - l_{\max}]^+$ and α' are non decreasing functions with positive values.

I have proven those propositions for finite input flows. For instance, in proposition (2.12) if A and D are equal to ∞ then my definition in Coq of backlog is not defined.

One of the step of the proof of the identity (2.11) uses the fact that the arrival times of the packets partition $\mathbb{R}^+$. Formally it means that

$$\bigcup_{i=0}^{\infty} [t_i, t_{i+1}[= \mathbb{R}^+ \tag{2.15}$$

However, in Coq I did not defined unions of an infinite number of sets. Thus, I could not prove this proposition.

2.2.2 Non preemptive static priority scheduler

The behavior of a non-preemptive static priority scheduler (see Appendix C) can easily be defined with words. At time t, if there are high priority packets that need to be served the system will serve them, otherwise it will serve the low priority ones. But how can this be defined formally? This section studies the formal definition of a non-preemptive static priority scheduler with constant rate and two inputs. This is a work that is still in progress and I did not have the time to test it and to check whether the theorems related to it work with this definition. Thus I cannot assert the correctness of this definition.

We start from the following observation: given a time $t \in \mathbb{R}^+$, a packetized input flow A and an output flow D, there is always an interval starting from t where A is either always equal to D or always greater than D:

$$\forall t \in \mathbb{R}^+, \exists \epsilon > 0 : (\forall \tau \in [t, t+\epsilon[, A(\tau) = D(\tau)) \vee (\forall \tau \in [t, t+\epsilon[, A(\tau) > D(\tau)) \tag{2.16}$$

This section studies a non-preemptive static priority scheduler with 2 inputs, A_h a high priority one and A_l a low priority one. Let be $\epsilon_h^*(t)$ the greatest ϵ such that the statement (2.16) holds at time t for A_h:

$$\epsilon_h^*(t) = \sup\{\epsilon > 0 : (\forall \tau \in [t, t+\epsilon[, A_h(\tau) = D_h(\tau)) \vee (\forall \tau \in [t, t+\epsilon[, A_h(\tau) > D_h(\tau))\} \tag{2.17}$$

The definition of $\epsilon_l^*(t)$ is analogous.
We can then define the sequence $(t_n)_n \in \mathbb{N}$ such that

$$\begin{cases} t_0 = 0 \\ t_{i+1} = t_i + \min(\epsilon_h(t_i)^*, \epsilon_l(t_i)^*) \end{cases} \tag{2.18}$$

where $\epsilon_h^*(t_i)$ and $\epsilon_l^*(t_i)$ are defined (2.17). The sequence starts at $t_0 = 0$. We compute $\epsilon_h^*(0)$ and $\epsilon_l^*(0)$ and we choose the smallest of the two: this is t_1. In the interval $[t_0, t_1[$, the proposition (2.16) holds for both A_h and A_l. We repeat the process but this time starting from t_1: we compute $\epsilon_h^*(t_1)$ and $\epsilon_l^*(t_1)$, we choose the smallest between the two. This will be repeated such that the intervals $[t_i, t_{i+1}[$ partition $\mathbb{R}^+$.

In each of these intervals 4 possibilities may arise:
1) $A_h = D_h \wedge A_l = D_l$: this is the simplest case, the server is not serving any packet.

2) $A_h = D_h \wedge A_l > D_l$: only the low priority packets are being served. If the scheduler is serving packets at a constant rate c then $D_l(t) = D_l(t_i) + c \cdot (t - t_i)$ and D_h stays constant.

3)$A_h > D_h \wedge A_l = D_l$: only the high priority packets are being served. In this case $D_h(t) = D_h(t_i) + c \cdot (t - t_i)$ and D_l stays constant.

4) $A_h > D_h \wedge A_l > D_l$: this is the most complex case. In fact the scheduler is non-preemptive, thus if a low priority packet is being served then it will wait until the whole packet has been served before serving a high priority one.
Let call L_l the sequence describing the cumulative low priority packet lengths. If we are in this situation, the low priority packet will be served when $D_l(t)$ will be equal to a term of L_l. This is equivalent to say that $D_l(t) = P_{L_l}(D_l(t))$. Thus the whole packet will be served at time:

$$\tau^* = \inf\{t_i \leq \tau \leq t_{i+1} : D_l(t) = P_{L_l}(D_l(t))\}\} \tag{2.19}$$

When $t \in [t_i, \tau^*]$, a low priority packet is being served thus $D_l(t) = D_l(t_i) + c \cdot (t - t_i)$ and D_h stays constant.
When $t \in [\tau^*, t_{i+1}]$, the high priority packets are being served, so $D_h(t) = D_h(\tau^*) + c \cdot (t - \tau^*)$ and D_l stays constant.
This definition works even if no low priority packet is being served. Indeed τ^* would be equal to t_i and we would be in the situation 3). However, if at the end of the interval the low priority packet has not been served yet, τ is not defined (because the set is empty). This problem can be circumvented by noting that we are in the same situation than the second one.

Although it seems quite cumbersome, this definition is valid as long as a system is defined as a binary relation between $\mathcal{C}$ and $\mathcal{C}$. In fact, provided a cumulative input and output we just need to check whether the relation holds.

2.3 Project achievements

As the source code of the paper [3] was unavailable, a part of the project was to rewrite the basic definitions and theorems that would have been useful for the rest of the project. As they have already been proven, I admitted them to save time.
The project is composed of 10 files:

- RBarComp: a list of additional theorems about $\mathbb{R}^+$
- SetTheory: definitions and theorems about sets, set operations and set maxima and minima
- FunImage: theorems about the image of a function
- InfSup: definitions and theorems about suprema and infima in $\mathbb{R}$ and $\mathbb{R}^+$
- Seq: definitions and theorems about sequences and partial sums
- Definitions: definitions and theorems about min-plus algebra
- Main: basic definitions and theorems of network calculus
- Packetizer: basic definitions and theorems about the packetizer
- Scheduler: formal definition a two inputs non-preemptive static priority scheduler
- Examples: examples and tests

	Definitions	Theorems	Lines of code
RBarComp	0	11	240
SetTheory	20	25	537
FunImage	0	19	490
InfSup	4	49	994
Seq	19	10	310
Definitions	24	24	826
Main	12	7	521
Packetizer	10	31	1257
Scheduler	10	0	69
Total	99	176	5244

Table 2.1: Number of theorems, definitions and lines of code written (admitted theorems excluded)

Where as many of them concern basic maths, the theorems about the packetizer have not been proven in the paper [3] and the scheduler has never been defined this way.

2.4 Skills acquired by the project

Although I was already keen on formal proofs, this project really sharpened my deductive reasoning. It taught me to always be careful to hasty conclusions even when they seem evident. Many times, my proofs were fine on paper but they would not work once transposed on Coq because of subtle details. For instance, the addition is not always defined in $\overline{\mathbb{R}}$ (when adding ∞ and $-\infty$) , and thus I needed to be careful to such corner cases.

In addition, I discovered Coq and a new paradigm: dependently typed programming. In fact, proving a theorem in Coq is equivalent to declare a new type. Coq is also a functional language where we can navigate through types with functions. As propositions are types, an implication is just a function that maps a proposition to an other. Learning this language did not only teach me a new paradigm but also strengthened my skills in functional programming.

Unexpectedly, the only way to learn Coq was through the documentation and very specialized articles. Despite this difficulty and the fact that it took me more time than expected, this way of learning gave me a deeper understanding of the language and made me more autonomous. Finally I expanded my mathematical knowledge and learnt network calculus.

2.5 Project approach and major steps

The first steps of the project were to learn how to program in Coq, get familiar with Network calculus, and the paper [3]. As the source code of the paper was not available, I spent some time to rewrite basic network calculus theorems and definitions. I also spent few weeks setting up the basic mathematics notions that would have been useful throughout the rest of the project (image, infima and suprema, sets...).

To save time I also used expedients in Coq that would have avoided me defining cumbersome structures. For instance, we cannot really define $\mathbb{R}^+$ or $\overline{\mathbb{R}}^+$. Actually, we would need to define a new set and prove all the theorems and properties valid in $\mathbb{R}$ or $\overline{\mathbb{R}}$ for this new set. A more clever trick would be to state that the domain of our functions is $\mathbb{R}$ but the theorems about them are valid if the input is greater than 0. At this point, I began to think about how to formally define the scheduler. Meanwhile, I realized that I needed to define a more basic component first: the packetizer. Once I defined it, I came back to the scheduler and defined it as well. Finally I spent the remaining time proving theorems about the scheduler and specifically about combined systems.

2.6 Self-assessment

I believe that I succeeded the most in learning Coq quickly. Actually, I started writing definitions and theorems since the first week, which really sped up the project. However, I could not familiarize myself with the libraries as they were really technical. I believe that if I had done so, I would have avoided writing many basic mathematics definitions and theorems and thus saved time. With this extra time, I could have written the proofs of theorems I did not have to demonstrate in Coq.

Bibliography

[1] Davis, R.I., Burns, A., Bril, R.J. et al. *Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised.* Real-Time Syst 35, 239–272 (2007).
`https://doi.org/10.1007/s11241-007-9012-7`

[2] Paulin-Mohring, Christine. (2012). Introduction to the Coq Proof-Assistant for Practical Software Verification. 10.1007/978-3-642-35746-6_3.
`https://www.lri.fr/ paulin/LASER/course-notes.pdf`

[3] Lucien Rakotomalala, Marc Boyer, Pierre Roux. *Formal Verification of Real-time Networks.* JRWRTC2019, Junior Workshop RTNS 2019, Nov 2019, TOULOUSE, France.
`https://hal.archives-ouvertes.fr/hal-02449140/document`

[4] Georges Gonthier, *Formal Proof - The Four-Color Theorem*, 2008
`http://www.ams.org/notices/200811/tx081101382p.pdf`

[5] Le Boudec, Jean-Yves & Thiran, Patrick. (2004). Network Calculus: A Theory of Deterministic Queuing Systems for the Internet. 2050. 10.1007/3-540-45318-0.

[6] Delahaye, David. (2000). A Tactic Language for the System Coq. 85-95. 10.1007/3-540-44404-1_7.
`http://www.lirmm.fr/ delahaye/papers/ltac%20(LPAR%2700).pdf`

Appendix A

Network Calculus Definitions

A.1 Cumulative functions

$f : \mathbb{R}^+ \to \overline{\mathbb{R}}^+$ is cumulative if:

1. $\forall t, s \in \mathbb{R}^+, t < s \implies f(t) \leq f(s)$ (f is non-decreasing)
2. $f(0) = 0$
3. $\forall t \in \mathbb{R}^+, \lim_{x \to t^+} f(x) = f(t)$ (f is right-continuous)

The right continuity is a convention. For instance, the paper [3] uses left-continuity instead. In the context of the project I decided to use right continuity, since the packetizer described in the book [5] is right continuous. The set of cumulative functions is denoted $\mathcal{C}$. The code below describes a cumulative function in Coq where *NonDecreasingPositive f* means that the function is both non-decreasing and always greater than 0.

```
Class Cumulative (f: R → Rbar) := {
  non_decr_pos :> NonDecreasingPositive f;
  right_cont: ∀ x,  x >= 0 → filterlim f (at_right x) (Rbar_locally (f x));
  start_0 :  f(0)  = 0;
}.
```

A.2 System/Server

A server or a system $S \subset \mathcal{C} \times \mathcal{C}$ is a binary relation that meets the following conditions:

1. $\forall A \in \mathcal{C}, \exists D \in \mathcal{C} : (A, D) \in S$
2. $(A, D) \in S \implies D \leq A$

In Coq a server is defined as follows:

```
Class Server (s : (R → Rbar) → (R → Rbar) → Prop) := {
  subset_c_c: ∀ a d : R → Rbar, s a d → Cumulative a ∧ Cumulative d;
  completeness: ∀ a : R → Rbar, Cumulative a → ∃ d, s a d;
  causality: ∀ a d t, s a d → t >= 0 → Rbar_le (d(t)) (a(t))
}.
```

where *s a d* means that $(A, D) \in S$ and *Rbar_le* is the "$\leq$" operator in $\overline{\mathbb{R}}$.

A.3 Backlog and Delay

Given two cumulative flows A and D, the backlog at time $t \in \mathbb{R}^+$ between A and D is

$$b(A, D, t) := A(t) - D(t)$$

Geometrically it can be seen as the vertical distance between A and D at time t. The backlog or horizontal deviation between them is

$$b(A, D) := \sup_{t \in \mathbb{R}^+} b(A, D, t)$$

It is a tight bound for the backlog at any time.

In Coq they are defined as follows:

```
Definition backlog_t (a d: R → Rbar)(t: R) := Rbar_minus (a t) (d t).
Definition backlog (a d: R → Rbar) := Rbar_lub(Im R Rbar (fun x ⇒ x >= 0) (backlog_t a d)).
```

where *Rbar_minus* is the minus operation in $\overline{R}$ and *Rbar_lub* represents the supremum of a set.

The delay at time t between A and D is given by:

$$d(A, D, t) := \inf\{\tau : A(t + \tau) \geq D(t)\}$$

It is the amount of time taken by A to reach D starting from time t. Thus it is the horizontal distance between A and D. The delay or vertical deviation is:

$$d(A, D) := \sup_{t \in \mathbb{R}^+} d(A, D, t)$$

The definition of the delay in Coq is:

```
Definition delay_t (a d: R → Rbar) (t: R): Rbar := Glb_Rbar(fun delta: R ⇒ Rbar_le (a t) (d(t + delta)
Definition delay (a d: R → Rbar): Rbar := Rbar_lub(Im R Rbar (fun x ⇒ x >= 0) (delay_t a d)).
```

where *Glb_Rbar* indicates the supremum of a set.

A.4 Arrival and Service curves

A non-decreasing function $\alpha : \mathbb{R}^+ \rightarrow \overline{\mathbb{R}}^+$ is an arrival curve for a flow $A \in \mathcal{C}$ if:

$$R \leq R \otimes \alpha$$

where $(R \otimes \alpha)(t) = \inf_{s \geq 0}(R(s) + \alpha(t-s))$ is the min-plus convolution between R and α. α gives an upper bound on how much data the flow can receive in a given interval of time. In fact for continuous flows this definition is equivalent to:

$$\forall t \in \mathbb{R}^+, \forall s \in [0,t], A(t) - A(s) \leq \alpha(t-s)$$

A server S offers a minimal service curve $\beta : \mathbb{R}^+ \rightarrow \overline{\mathbb{R}}^+$ if:

$$\forall (A, D) \in S, A \otimes \beta \leq D$$

The code below describes an arrival curve:

```
    Definition is_arrival_curve (a: R → Rbar) ‘(alpha: NonDecreasingPositive al) :=
    ∀ x, Cumulative a → x >= 0 → Rbar_le (a(x)) (conv al a x).
```

where *conv al a x* is the convolution between α and A evaluated at time x.

This time, β gives a lower bound for the outputs of the system. For continuous flows this definition is equivalent to:

$$\forall t \in \mathbb{R}^+, \exists s \in [0,t] : D(t) \geq A(s) + \beta(t-s)$$

In Coq the definition of a minimal service curve is:

```
    Definition is_minimal_service ‘(S: Server s) ‘( Beta: NonDecreasingPositive beta) := ∀
a d t, s a d → t >= 0 → Rbar_le (conv a beta t) (d t).
```

Appendix B

Packetizer

A L-Packetizer is a system which discretizes a continuous input. It takes a cumulative data flow as input and outputs a staircase function where the height of the steps is defined by L, a sequence of cumulative packet lengths. Formally a sequence of cumulative packets lengths $L : \mathbb{N} \to \mathbb{R}^+$ meets the following conditions:

1. $\forall n \in \mathbb{N}, L(n) \leq L(n+1)$ (f is non-decreasing)
2. $L(0) = 0$
3. $\lim_{n\to\infty} L(n) = \infty$
4. $(L(n+1) - L(n))_{n\in\mathbb{N}}$ is bounded above

By the completeness of $\mathbb{R}$, the last condition implies that
$l_{\max} = \sup_{n\in\mathbb{N}}\{L(n+1) - L(n)\}$ exists.

A packetizer is a system which applies P_L to the input flow, a function that takes a value x and maps it to the greatest term of L smaller than x:

$$\begin{aligned} P^L : \overline{\mathbb{R}}^+ &\to \overline{\mathbb{R}}^+ \\ x &\mapsto \sup_{n\in\mathbb{N}}\{L(n) : L(n) \leq x\}) \end{aligned}$$

The output of the packetizer D is thus $P_L(A)$

Appendix C

Scheduler

A scheduler is a server that reorders packets according to priorities. A static priority scheduler receives several input packets with different priorities and output the high priority ones. If there is no high priority packet waiting to be served, it will serve the packets with the second highest priority and so on... While it is serving a low priority packet, a high priority one may be received. If the server waits until the whole low priority packet has been served, then it is called a non-preemptive static priority scheduler. It can be seen has a server accepting several inputs flows and outputting the same number of flows, where each flow has a different priority level. The global flow is the superposition of all of them. In particular, in the project I studied a non-preemptive static priority scheduler with a constant output rate and two levels of priority as shown in the figure C.1.

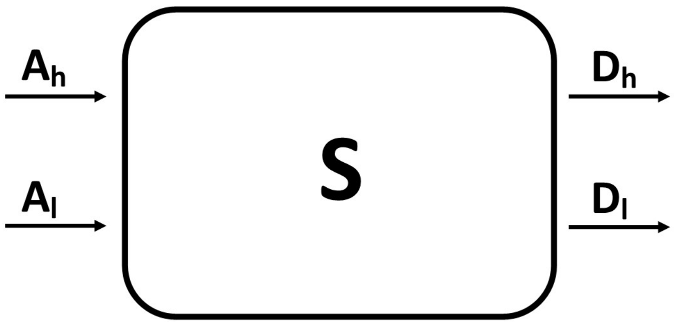

Figure C.1: Diagram of a two inputs scheduler