(Toward) Formally Verifying AIMD Congestion Control Behavior

CS-428 Final Project

Julien de Castelnau
Martynas Rimkevičius
julien.decastelnau@epfl.ch
martynas.rimkevicius@epfl.ch

ABSTRACT

Congestion control algorithms (CCAs) are a mechanism within TCP to regulate data transmission, with the primary goal of maximizing usable bandwidth and avoiding *congestive collapse* due to packet delay/loss. Realizing and verifying generally effective CCAs is challenging, in large part due to the limited ability to simulate the wide range of conditions that can occur in real-world networks. In light of this, Arun et al. have introduced a formalization for CCAs, using SMT solvers to model properties such as steady-state packet loss or worst-case link utilization for a number of popular CCAs. However, they also make a number of claims on the properties of specific CCAs as well as of the model itself that are not machine checked, either being proven on paper or only conjectured to be true. In our project, we sought to bridge these gaps by mechanizing these proofs in a proof assistant. We focused in particular on verifying claims about Additive Increase/Multiplicative Decrease (AIMD), a classic congestion control scheme that still remains in use today. To this end, we successfully mechanized a number of properties/proofs from the paper. In addition, we identified an issue in the on-paper proof for a key property of the assumed network model, our counterexample for which we have also mechanized.

1 INTRODUCTION

Arun et al. introduce the *path-server model*[1], a formalization for the behavior of TCP network links (insofar as congestion control algorithms are concerned.) Figure 1 illustrates the model. It assumes a single *sender*, connected to a single *path server*, which has a token bucket filter. The cumulative number of packets arrived at the path server (the arrival curve) in bytes at a given time t is given by the function $A(t)$, while the number of packets actually sent by the server (the service curve) is given by $S(t)$. The *waste function* $W(t)$ captures the (generalized) behavior of a token bucket filter: packets can be sent only when there are enough tokens in the buffer, and tokens replenish at a constant rate, but extra tokens can be wasted at any time. Note that a token corresponds to one byte of packet data. The cumulative amount of tokens wasted up to time t is $W(t)$. The link rate C encodes the rate at which tokens are filled in the buffer, and in general $Ct - W(t)$ provides an upper bound for the service curve $S(t)$, as packets cannot be sent faster than the token buffer permits. Meanwhile, the lower bound is given by $C(t - D) - W(t - D)$, where D is the maximum delay/jitter permitted for a packet. This is because packets arrive into a queue, and assuming there are tokens available, they must be sent out after D timesteps. Therefore, the upper bound delayed by D timesteps provides the lower bound. These functions and their bounds are demonstrated visually in the example network trace of Figure 2.

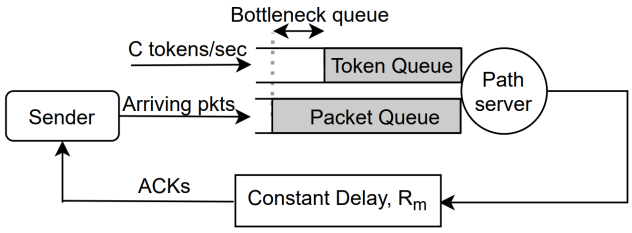

Figure 1: Path-server model.

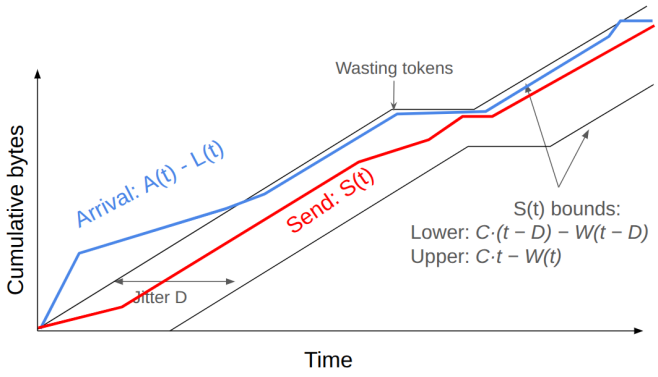

Figure 2: Graphical representation of path-server functions.

Properties like the size of the packet queue, the number of packets lost, the MTU of the packets, etc. are defined similarly. We omit a detailed explanation of each here, but provide a glossary of every symbol used in the model (from [1]) for reference (Figure 3). Practically speaking, in the paper, this model is used to find edge cases in congestion control algorithms by fixing some constant network conditions like jitter D and link rate C, and solving an SMT problem that asks if an edge case is possible given that $A(t)$, $W(t)$, $S(t)$, etc. are open variables for each timestep. Constraints are added to capture the requirement that these functions form a legal network trace, such as that the service curve satisfies the lower and upper bounds mentioned previously.

An important limitation of the path-server model is that it assumes only one server in the full network link, from the sender to its ACK return path. Realistic network links would, of course, include many servers in series. To compensate for this, Arun et al. state *composition theorems*: demonstrating that it is possible to construct a legal network trace of a single path server that captures the effects of composing two servers in series. In the paper, only specific cases

C - link rate	R_m - propagation delay
β - buffer size	D - max per-packet jitter
α - MTU size	$dupacks$ (threshold) - 3α
T - number of time steps	$cwnd(t)$ - congestion window
$Q(t)$ - packet queue length	$T(t)$ - token queue length
$A(t)$ - cumulative arrivals	$S(t)$ - cumulative service
$W(t)$ - cumulative waste	$L(t)$ - cumulative losses
$L^d(t)$ - cum. losses detected	$\tau_o(t)$ - timeout happened

Figure 3: Glossary of symbols.

of this theorem are considered. Namely, it is split based on whether the first server or the second server in sequence has a faster link rate. For the case where the second server is faster, and where its buffer size satisfies a specific lower bound[1], the authors provide a proof in Lean. For the other case (first server is faster), the proof assumes the servers have an infinite buffer size and thus no packet loss occurs, and it is only given on paper. Encoding all of these existing proofs in Rocq was thus a preliminary goal for our project.

For AIMD itself, as mentioned, Arun et al. use SMT solvers to find edge cases in the algorithm, using the network constraints discussed. The authors also extrapolate a *steady state* condition for the algorithm, where the congestion window (the maximum number of bytes sent per timestep) is expected to enter a certain range (calculated from the network parameters) and stay there. They prove using a mix of SMT solvers and on-paper proofs that this state does exist for any network and AIMD stays there (Theorem 1 in the paper). In addition, they claim that in this state, there are bounds to the packet loss that can happen (Theorem 2 in the paper), although there is no complete proof, so it is only conjectured. We explain this steady state and the associated theorems in further detail in 2.3, but mechanizing these properties of AIMD was another goal for our project.

2 APPROACH

In this section, we will describe in more detail the properties and proofs we encoded in Rocq. Our code is available on GitLab.

2.1 Encoding Trace

As discussed, the authors provided a Lean proof for the first composition theorem case. As part of this, they established the *Trace* record (a structure as Lean calls it), which holds each parameter of the network, the functions over time ($A(t), S(t), W(t)$, etc.), and the constraints on these functions such as the service curve bounds, the assumption that they start at zero, and are monotonic (since they are cumulative). Translating the definition to Rocq was a relatively straightforward process of changing the syntax, as it made use of relatively generic language features. A shortened version of the translated definition is given in Listing 1.

We note that time is encoded with natural numbers (a decision made by the authors to make SMT encoding easier[2]), while numbers of bytes are rational numbers. This gives each function of the network a signature of $\mathbb{N} \to \mathbb{Q}$. Differences between the Rocq and Lean rational numbers and their associated conversions to/from naturals were the main source of our initial trouble converting the Lean code to Rocq. One major difference is that in Rocq, rationals have their own equality relation (to handle something like $\frac{4}{6} = \frac{2}{3}$), while in Lean, the numerator and denominator are assumed to be coprime, so the standard equality is used. As a result, we primarily used rational equality == in our definitions as opposed to =. We discuss other challenges regarding numbers and our solutions in Section 2.4.

2.2 Encoding the Composition Theorem

Next, we encoded the composition theorem, which states that out of two path servers, we can model any network condition by creating a single path server that represents a concatenation of the two. With this theorem, we could model any network with any amount of path-servers. The proof of the composition theorem is split into two cases, based on which of the two servers we are composing has a faster link rate. We started with the part of the theorem where first server is slower than the second one. All of our work in the following sections is reflected in the Rocq file `Compose.v` in our project.

2.2.1 Second server is faster. We encoded the constraint that the first path server is connected to the second which is stated as follows:

$$\forall \tau_1, \tau_2. (\tau_1.S = \tau_2.A) \land (\tau_1.C \le \tau_2.C) \land (\tau_2.\beta \ge \tau_1.C \cdot \tau_1.D)$$
$$\implies \exists \tau_s, \tau_s.C = \tau_1.C \land (\tau_s.D = \tau_1.D + \tau_2.D) \land \tau_s.\beta = \tau_1.\beta$$
$$\land \tau_s.A = \tau_1.A \land \tau_s.S = \tau_2.S. \land \tau_s.L = \tau_1.L$$

τ_1 and τ_2 represent `Traces`, in other words, path servers. The theorem states that there exists a server that acts as the composition of these two with specific constraints which are shown above. The proof mainly involves showing that this composition still meets the constraints for a legal trace. We start constructing these constraints by showing that no packets can be lost at any time by the second server due to its faster rate, and the assumption of the buffer size ($\tau_2.\beta \ge \tau_1.C \cdot \tau_1.D$). This lemma shows that $\forall t. \tau_2.L(t) = 0$, and therefore $\tau_s.L = \tau_1.L$. Next, we had to prove that lower bound of the composed server uses the sum of jitter: $\forall t. \tau_1.C \cdot (t - (\tau_1.D + \tau_2.D)) - \tau_1.W(t - (\tau_1.D + \tau_2.D)) \le \tau_2.S(t)$, which holds because initial lower bound of the first server is higher. Then, we construct the witness out of properties of the first and the second server, and

[1]The authors note that the lower bound is based off realistic network conditions i.e. that the case where the buffer is smaller than the bound is an unlikely network scenario.
[2]It is worth noting that this decision also has consequences for the network behavior modeled. Since time is discretized, some scenarios modeled may not correspond to real-life networks. We do not address this limitation in our project, but the paper discusses this further.

```
Structure Trace := {
  C: Q; (* Link rate *)
  out: nat -> Q; (* Service curve *)
  wst: nat -> Q; (* Wastage *)
  ...
  upper_bound: forall t: nat, out t <= C * t - (wst t);
  ...
}.
```

Listing 1: (Shortened) Encoding of Trace in Rocq.

these lemmas. This side of the theorem was already encoded and proved by the authors in Lean, which helped in the proof structure.

2.2.2 First server is faster. The other case of the composition theorem, where the first server is faster, is stated as follows:

$$\forall \tau_1, \tau_2.(\tau_1.S = \tau_2.A) \land (\tau_1.C \geq \tau_2.C) \land (\forall t, \tau_1.L(t) = \tau_2.L(t) = 0)$$
$$\implies \exists \tau_s, \tau_s.C = \tau_2.C \land (\tau_s.D = \tau_1.D + \tau_2.D) \land (\forall t.\tau_s.L(t) = 0)$$
$$\land\ \tau_s.A = \tau_1.A \land \tau_s.S = \tau_2.S. \land \tau_s.W = \tau_1.W$$

Note the added condition that the loss of each server (and the composition) is zero. This corresponds to the assumption of infinite buffers.

Once again, to prove the statement, we must construct a witness trace τ_s, for which every other function is already bound except $\tau_s.W(t)$. Unlike the first case, we cannot simply use the waste of either of the two servers; intuitively, τ_1 may waste more tokens in a single timestep than the composition server has available, since τ_1 can replenish tokens faster ($\tau_1.C \geq \tau_2.C = \tau_S.C$). Likewise, τ_2 may have stored tokens in its buffer, but it does not mean that τ_1 has any tokens available (for example if $\tau_1.S$ is very delayed), and so the composition server would not be allowed to waste tokens overall.

In their proof, the authors construct a waste function $\tau_s.W(t)$ to complete the witness for the composed trace. We omit a full explanation of the algorithm used to construct this function, but intuitively, the function is intended to "track" the waste function of τ_1 when the composition server also has the same number of tokens in the buffer, otherwise it does not waste at all. Since the waste function determines the upper and lower bounds, 3 conditions need to be proven for τ_s to be a valid Trace:

(1) $\forall t.\tau_s.W(t) < \tau_s.W(t+1) \implies \tau_1.A(t+1) \leq \tau_2.C(t+1) - \tau_s.W(t+1)$ (when $\tau_s.W$ wastes it is allowed to do so)
(2) $\forall t.\tau_2.S(t) \leq \tau_2.Ct - \tau_s.W(t)$ (upper bound)
(3) $\forall t.\tau_2.S(t) \geq \tau_2.C(t - \tau_s.D) - \tau_s.W(t - \tau_s.D)$ (lower bound)

In our project, after having encoded the waste function in Rocq, we attempted to mechanize the proof given in the paper for these 3 conditions. However, we struggled to prove (2), the upper bound condition. Eventually, we actually showed that this condition has a counterexample. More precisely, we were able to show that there exist valid traces τ_1 and τ_2 such that the composition server formed with this waste function $\tau_s.W(t)$ is no longer a valid trace, due to violating this upper bound condition. Note, however, that this does not necessarily mean the composition theorem itself (or this case of it) is false, only that the authors' waste function cannot work as a valid witness for the proof.

To elucidate our counterexample, we visualize it in Figure 4. The graph is similar to the example in Figure 2, only now there are two servers and thus two sets of upper and lower bounds, represented with solid and dashed lines respectively. Recall that the upper and lower bounds are given by $u(t) = Ct - W(t)$ and $l(t) = C(t-D) - W(t-D)$ respectively. Note that the slope of τ_1 is greater, since we are in the case that the first server is greater ($\tau_1.C \geq \tau_2.C$). For the upper bound condition, we are interested in $\tau_s.u(t) = \tau_2.Ct - \tau_s.W(t)$ (red line), and $\tau_s.S(t) = \tau_2.S(t)$ (blue line). The condition says the blue line should always be contained by the red line, but we can see graphically that this is not true past $t = 1$. We explain how $\tau_s.u$ takes on these values.

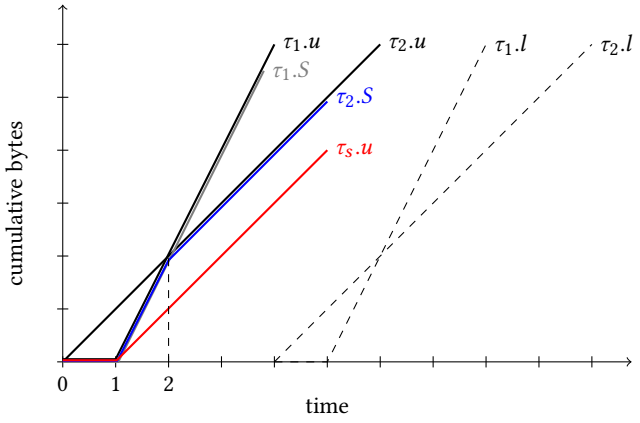

Figure 4: Network trace demonstrating counterexample where the composition trace τ_s using $\tau_s.W(t)$ is invalid.

At $t = 0$, everything starts at 0, as expected by the conditions of Trace. At $t = 1$, everything remains at 0 except for $\tau_2.u$. This means that a) nothing was sent to the first server (so nothing to the second either), b) τ_1 chose to waste all the tokens it would have gained ($\tau_1.C$ many tokens to be exact) and c) τ_2 chose *not* to waste its tokens, leading to the gap between the $\tau_2.u$ line and the rest. τ_s "tracks" τ_1 in this timestep as instructed by the $\tau_s.W(t)$ algorithm, since $\tau_1.u(1)$ has passed below what $\tau_s.u(1)$ would be had it not wasted anything. However, at $t = 2$, we break the upper bound condition: If τ_1 starts getting packets and outputting them with no delay at its full rate ($\tau_1.C$), then τ_2 can use the tokens it has stored from $t = 1$ to send a "burst" at the rate of $\tau_1.C$. This burst is shown in the steep blue segment from t=1 to t=2. Meanwhile, $\tau_s.u$ correctly stops tracking $\tau_1.u$, because τ_1 increased faster than its maximum rate of $\tau_2.C$. But, since it chose to waste tokens in $t = 1$ and τ_2 didn't, the upper bound is less than the output of τ_2, and the condition is violated.

We encoded this counterexample in Rocq, first creating a proof for our two witness `Traces` τ_1 and τ_2 that they are valid, then proving that at some timestep, the composition of these servers as determined by the $\tau_s.W(t)$ algorithm violates the upper bound condition. Listing 2 shows the Rocq statement we proved. `wst_compose` is a function that implements the $\tau_s.W(t)$ algorithm, taking the two `Traces` to compose and giving $\mathbb{N} \to \mathbb{Q}$. We have also written it to

```
Theorem compose_counterexample (IB: InfB):
 exists (tau1 tau2 : Trace), tau1.(C) >= tau2.(C) /\
  tau1.(out) = tau2.(inp) /\
  (forall t, tau1.(los) t = 0 /\ tau2.(los) t = 0) /\
  (exists (t: nat), tau2.(out) t > tau2.(C) * t
    - (wst_compose (wst_t1_track_ours tau1 tau2) tau1 tau2 t)) /\
  (exists (t: nat), tau2.(out) t > tau2.(C) * t
    - (wst_compose (wst_t1_track_paper tau1 tau2) tau1 tau2 t)).
```

Listing 2: Statement of the compose counterexample in Rocq.

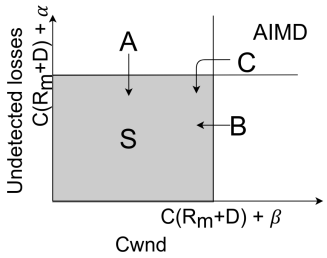

Figure 5: *Steady state* picture proof

take another function as a parameter: this determines exactly how to "track" one of the traces. We prove two versions of the statement with different tracking functions, as we had problems proving a different Trace condition (the waste condition, (1) from earlier) with the definition provided in the paper, and slightly adjusted it. We believe the definition simply contained a typo ($\tau_1.W(t+1) - \Delta Ct$ vs $\tau_1.W(t+1) - \Delta C(t+1)$), but we proved this counterexample on both for completeness. For added confidence, we have also ported our counterexample back to the original Lean encoding, which is available here.

2.3 AIMD

The paper presents theorems about TCP congestion control algorithm Additive Increase / Multiplicative decrease. This algorithm is used to pace the sender and utilize the available bandwidth. On each acknowledgement (ACK), indicating successful delivery, the sender increases its congestion window additively; on detecting a lost packet, it decreases the window multiplicatively

The authors embed AIMD congestion control mechanism in their path-server model by stating that ACK is said to be received if loss did not increase and loss was not detected. Loss increase is modeled by a function that indicates how many bytes have been lost up to certain time. Loss is detectable if $\forall t, \Delta t.\tau.S(t - R_m) \geq A(t - R_m - \Delta t) - L(t - R_m - \Delta t) + dupacks$. This definition is made for SMT solver, which explores all possible combinations of discrete time Δt to find a counter example for the formulas where AIMD is used. Although we modeled the behavior of AIMD as an inductive proposition in Rocq, we did not end up using it in our proofs as we ended up assuming lemmas from the paper that made working with the definition unnecessary. We explain these lemmas in further detail shortly.

Using the definition of AIMD, the authors examine a *steady state* of the CCA. *Steady state* is defined as a set of network states where as long as the network parameters remain unchanged, (1) if the network enters it, it will never leave the state, and (2) network will always enter it regardless of initial conditions. In the paper, Theorem 1 shows explicit bounds that define AIMD steady state

```
Lemma steady_state_AIMD:
  forall tau: TraceAIMD,
  BDP tau > 5 * alpha tau
  -> exists t : nat, cwnd tau t <= max_cwnd tau
    /\ undetected_loss tau t <= max_undetected_val tau
```

Listing 3: Encoding of steady state theorem.

```
Lemma steady_state_step_cwnd:
  forall tau: TraceAIMD,
  forall t: nat,
  BDP tau > 5 * alpha tau
  -> cwnd tau t > max_cwnd tau
  -> (cwnd tau (S t) < cwnd tau t - alpha tau
    \/ cwnd tau (S t) <= max_cwnd tau).
```

Listing 4: Congestion window step lemma.

```
Lemma steady_state_cwnd_ub:
  forall tau: TraceAIMD,
  forall t t' : nat,
  BDP tau > 5 * alpha tau
  -> cwnd tau t > max_cwnd tau
  -> (t < t')%nat
  -> (cwnd tau t' <= cwnd tau t - (alpha tau) * (t' - t)%nat)
    \/ (cwnd tau t' <= max_cwnd tau).
```

Listing 5: Congestion window upper bound lemma.

```
Lemma steady_state_loss_cwnd_stays:
  forall tau: TraceAIMD,
  forall t t' : nat,
  BDP tau > 5 * alpha tau
  -> cwnd tau t <= max_cwnd tau
  -> undetected_loss tau t > max_undetected_val tau
  -> (t < t')%nat
  -> cwnd tau t' <= max_cwnd tau.
```

Listing 6: Congestion window stays under maximum window lemma.

and they prove this theorem using Figure 5 picture illustrating the *steady state* bounds and four SMT-checked lemmas.

We encoded the AIMD constraints from the paper in Rocq using descriptions provided, our understanding of AIMD, and SMT solver code. We started the proof of the theorem by trying to prove the lemmas used in the image proof but we lacked information to prove them. However, as the lemmas had already been machine checked by SMT solvers, we decided to leave them as Admitted and focus on the proof of the main theorem. The proof is based on the witness of time t where AIMD is in the *steady state* as shown in Listing 3. We construct helper lemmas to create a witness.

First, we show that congestion window decreases if it exceeds maximum window size (defined by the network parameters) illustrated in Listing 4. Using this step lemma, we show that if the congestion window has exceeded the maximum window at time t, at the later time t' it is either under the maximum window, or has decreased by $t - t'$ steps. We show the upper bound of congestion window lemma in Listing 5. These lemmas together indicate that the congestion window will always end up lower than the maximum window after exceeding its upper bound.

Next, in Listing 6 we show that once the congestion window is lower than the maximum congestion window, it will not exceed it. We prove it by induction on time t using the SMT-checked lemmas.

We combine our congestion window lemmas together to show that the congestion window always enters the *steady state* bound. Intuitively, the congestion window will decrease after exceeding

```
Lemma steady_state_enter_cwnd:
  forall tau: TraceAIMD,
  BDP tau > 5 * alpha tau
  -> exists t: nat, cwnd tau t <= max_cwnd tau.
```

Listing 7: Congestion window entering the steady state lemma.

```
Lemma steady_state_loss_ub:
  forall tau: TraceAIMD,
  forall t t' : nat,
  BDP tau > 5 * alpha tau
  -> cwnd tau t <= max_cwnd tau
  -> undetected_loss tau t > max_undetected_val tau
  -> (t < t')%nat
  -> (undetected_loss tau t' <= undetected_loss tau t
        - (C tau) * (t' - t)%nat)
    \/ (undetected_loss tau t' <= max_undetected_val tau).
```

Listing 8: Bounded undetected losses lemma.

the maximum window since we know that loss will occur in this case. After enough decrease, congestion window will be less than the maximum window. Listing 7 shows the encoding of this lemma.

Similarly, we build a lemma to show how undetected loss decreases if it exceeds the maximum undetected loss count illustrated in Listing 8. In this lemma, we use SMT-checked lemmas as they define how the congestion window changes together with observed loss.

To complete the proof of the *steady state* theorem (Listing 3), we proceed with the lemma which states that congestion window always enters its upper bound (Listing 7). Then we proceed with case analysis at time t. In the first case, at time t undetected loss is within bounds, then time t is the steady state witness. Otherwise, using undetected loss upper bound lemma (Listing 8), we know that after $t' = (1+\tau.undetected\_loss(t) - max\_undetected\_loss)/(\tau.C)$ undetected loss drops below its bound. At time $t+t'$, the congestion window will still be in bounds as shown in congestion window stays lemma (Listing 6). Thus, $t + t'$ is also a witness for *steady state*.

2.4 Rocq Challenges

As mentioned, a source of issues throughout the project was in dealing with rationals and natural numbers in the same expressions. The multiplication of rationals and the conversion of a natural number, as in $C(t - D) - W(t - D)$ in the lower bound, was specifically problematic. Theorems often required us to reason about expanding such a formula and cancelling out common terms, such as if $C(t - D)$ is on one side of the inequality and $C(t - D + 1)$ is on the other (for an inductive proof). The first issue was that there is no built-in conversion function directly from naturals to rationals, so we defined our own, composing the built-in conversions from $\mathbb{N}$ to $\mathbb{Z}$ and then to $\mathbb{Q}$:

```
Definition qmake_nat (n: nat): Q :=
  inject_Z (Z.of_nat n).
Coercion qmake_nat: nat >-> Q.
```

```
Ltac qnormalize :=
  repeat (simpl; match goal with
    | _ => progress ring_simplify
    | _ => rewrite Qadd1_eq_nat_succ
    | _ => rewrite Qadd_eq_nat_add
    | _ => rewrite Qmult_0_r
    | _ => rewrite Qmult_1_r
    | _ => rewrite Qminus_0_r
  end).

Ltac qsmash := qnormalize;
  (lra + (unfold qmake_nat, inject_Z; qnormalize); lra).
```

Listing 9: Custom tactics for expressions on rationals/naturals.

This conversion function rendered Rocq's built-in tactics for dealing with rationals, namely `ring_simplify` and `lra`, ill-suited to deal with our mixed natural/rational expressions. We opted to create some custom tactics to deal with such expressions, shown in Listing 9. `qnormalize` attempts to simplify the goal using a variety of lemmas we proved that facilitate conversion of common expressions over $\mathbb{N}$ to $\mathbb{Q}$, as well as general simplification techniques (e.g $x \cdot 0 = 0$) that `ring_simplify` didn't pick up on for whatever reason. `qsmash` is a "hammer"-style tactic that first attempts to use `qnormalize` to put the goal into a form that can be solved by `lra`, and then completely unfolds the definition of the conversion if that fails. We add this second clause since there were situations that prematurely unfolding these definitions made `lra` *unable* to prove the goal.

We also introduced a tactic called `qcalc`, inspired by the `calc` tactic in Lean. As we were almost always dealing with inequalities in our goals, we usually were reasoning about transitivity: proving $a \leq b$ and $b \leq c$ for some intermediate b when the goal is $a \leq c$. Our initial proofs simply used `apply`/`eapply` with the appropriate theorem (`Qle_trans`, `Qle_lt_trans`, etc.), but this made them somewhat clunky/repetitive. Our tactic is more or less syntactic sugar over the applying the transitivity theorems, as it doesn't build the proof bottom up like `calc` in lean. It takes a function that manipulates either the LHS or RHS and applies the transitivity appropriately. For example, with a goal of `wst tau1 t < wst tau1 (S t)`,

```
apply Qlt_le_trans with (y := ((tau1.(wst) (S t)) - DC));
    [|lra].
```

becomes

```
qcalc _ < (fun R => R - DC); [lra|].
```

so that the goal is now `wst tau1 t < wst tau1 (S t) - DC`. The exact definitions for `qcalc`, as well as all the other tactics discussed in this section, are available in `Tactics.v` in the project.

3 RESULTS

We have formalized several theorems from the paper by Arun et al. about *path-server model* and AIMD congestion control. We proved one of the two composition theorems about concatenated path-servers, where the first server is slower of the two. We did not

manage to prove the second composition theorem, on account of having found problems in the on-paper proof. However, we have mechanized the problems we found into a reproducible counterexample in Rocq.

In addition to the composition theorems, we proved the *steady-state* theorem for the AIMD congestion control algorithm. We had to rely on Admitted definitions for lemmas defined (and checked with SMT) by the authors in the paper for this theorem, but we have successfully mechanized the integration of all these lemmas into the final theorem, which was only a proof by picture in the text. We did not manage to get to Theorem 2 from the paper - proving loss bounds on AIMD's steady state.

4 TIMELINE EVALUATION

Our original goal of proving composition theorems and proving two theorems about AIMD *steady state* was not achieved.

- Week 9 - 11. The goal was to mechanize existing proofs of composition theorems. The porting of existing proofs from Lean took longer than we initially planned due to Rocq difficulties while working with natural and rational numbers together. We both were involved with porting existing formalization to get familiar with the notion of Trace. Nonetheless, Martynas did prove the first composition theorem in time, and Julien had uncovered problems in the second theorem.
- 12 - 13. The goal was to already mechanize the proof of Theorem 1 and we split the four lemmas in half used in picture proof two for each of us. These weeks we together spent on working on trying to find gaps in the second composition theorem, and Julien came up with a counterexample on paper that we verified together. Julien encoded the counterexample in Rocq. After the counterexample, both of us tried to prove all *steady state* lemmas with no success.
- 13 - 15. The goal was to investigate AIMD *steady state* loss theorem (Theorem 2 in the paper). We did not have enough time for it, as we worked on the Theorem 1 lemmas that we defined and kept the paper lemmas admitted. Julien proved congestion window and undetected loss bounds lemmas and Martynas proved congestion window entering *steady state* lemma, and the general *steady state* theorem. Last week, we both worked on the presentation.

Last, we both worked on the report splitting the work equally.

5 RELATED WORK/DISCUSSION

Network calculus is an established theory for analysis and verification of TCP networks[2]. It is often used to prove statements about delay and loss properties of networks, for instance. Prior work has formalized network calculus in Coq, in the library `NCCoq` [3][4]. The path-server model of Arun et al., which our work uses throughout, is loosely based on network calculus, but makes modifications to facilitate analysis of congestion control algorithms specifically [1]. As a result, `NCCoq` and, to our knowledge, other formalization efforts for network calculus have not addressed individual congestion control algorithms like AIMD. By contrast, in Arun et al. and this work, the behavior of AIMD in terms of congestion window, loss, etc. is modeled and proven formally.

We were able to prove the one case of the composition theorem (with the added buffer assumption), but we found issues in the authors' proof of the other case. We briefly investigated alternate composition waste functions to complete the proof, but switched focus to AIMD in the interest of time. Future work can investigate this proof and ideally complete it. In the unlikely event that this case of the composition theorem turns out to be false, it would mean that the model is not expressive enough to capture a case of a faster server in front of a slower server. This does not affect the validity of the edge cases in CCAs found by Arun et al. (since it just means they can be replicated on the simplest networks), but would call into question the utility of any theorem about the correctness of a congestion control algorithm. If the model cannot express it with just one path-server, it is conceivable that AIMD could violate the steady state behavior on a multiple path-server setup, for example. In general, it would be interesting to investigate a full proof of the composition theorem (or see if it's even possible.) For this, related proofs on *concatenation* in network calculus may be applicable.

For AIMD, despite not having proved the lemmas showing that steady state can be reached, we have integrated their definitions into a complete proof. Since these lemmas were already proven by SMT, modulo differences in our Rocq encoding vs the SMT formula, the proof is now completely machine checked. We did not get time to work on it, but we believe proving the steady-state loss bounds of AIMD (Theorem 2 from the paper), if it is true, would make for an interesting result. To our knowledge, formal properties on congestion control algorithms have not been established or proven in the literature prior to Arun et al. In general, aside from the few lemmas Arun et al. did manage to prove using SMT, formal methods have only been used with CCAs to search inexhaustively for edge cases. Thus, a proof of Theorem 2 may be a first of its kind.

REFERENCES

[1] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. ACM, Virtual Event USA, 1–16. https://doi.org/10.1145/3452296.3472912

[2] Jean-Yves Le Boudec and Patrick Thiran. 2001. *Network Calculus*. Springer Verlag. https://infoscience.epfl.ch/handle/20.500.14299/173789

[3] Lucien Rakotomalala, Marc Boyer, and Pierre Roux. 2019. Formal Verification of Real-time Networks. In *JRWRTC 2019, Junior Workshop RTNS 2019*. Toulouse, France. https://hal.science/hal-02449140

[4] Lucien Rakotomalala, Pierre Roux, and Marc Boyer. 2021. Verifying Min-Plus Computations with Coq. In *13th NASA Formal Methods Symposium (NFM 2021) (Lecture Notes in Computer Science, Vol. 12673)*. Springer International Publishing, Online, United States, 287–303. https://doi.org/10.1007/978-3-030-76384-8_18