Reproducing & Extending Results: *Towards Formally Verifying Congestion Control Behavior*

Alex Kwiatkowski
School of Computing and Information
University of Pittsburgh
Pittsburgh, United States
ajk175@pitt.edu

Birju Patel
School of Computing and Information
University of Pittsburgh
Pittsburgh, United States
bsp22@pitt.edu

***Abstract*—In this paper, we recreate and confirm the results found in the case studies from *Towards Formally Verifying Congestion Control Behavior*. Additionally, we attempt to find another use for the tool created in the paper, CCAC, by identifying lower bounds on utilization for the AIMD, BBR, and COPA congestion control algorithms. We also use formal verification to test TCP Reno.**

I. Introduction

How does a developer know whether or not their congestion control algorithm (CCA) will perform well enough and meet their expectations? For large content providers, they are able to collect a significant amount of realistic data simply by switching a portion of their users over to their algorithm and comparing how it performs against the current approach. However, for researchers, it is likely infeasible to test the performance of their CCA in this manner, and, even if they could, it is unlikely they would be able to capture anywhere near all of the edge cases and tail end performance that could be seen across the network.

With these problems in mind, the authors of *Towards Formally Verifying Congestion Control Behavior* [1] developed the Congestion Control Anxiety Controller (CCAC), a tool which seeks to use formal verification to prove facts about CCA performance even under some of the worst-case scenarios in a network. CCAC utilizes a theoretical path model that captures a wide variety of possible network behavior that a prover can reason about. This theoretical model utilizes an abstraction of a long path consisting of a single hop between a sender and a server with a token bucket filter. CCAC can simulate a path with either one or several servers that could cause queuing delays. Given assumptions about how the CCA performs, a formal description of the CCA, and the aforementioned network model (all expressed using first order logic), CCAC attempts to find counterexamples, or network states, where these assumptions do not hold through the use of the Z3 Theorem Prover. This counterexample is a scenario that could theoretically occur on the network and would also break the performance assumptions for a particular CCA.

To demonstrate the utility of their tool, the authors used CCAC to recreate known issues and identify previously undocumented issues with three different CCAs: BBR, COPA, and AIMD. For BBR and COPA, the authors were able to identify scenarios where the CCA encountered low utilization. For AIMD, they were able to find a scenario where packet loss occurred even when the window size was small. For each of the issues, they were then able to propose solutions that could be formally verified as correct.

For our project, we first recreated all of the results found in the three case studies mentioned above. These experiments allowed us to gain familiarity with the each of the CCAs and with CCAC. We then sought to extend the results in two ways. First, we tried to used CCAC to test a CCA that we discussed in class, TCP Reno. TCP Reno utilizes AIMD as a base and incorporates two new phases: slow start and fast recovery. With these two new components, we wanted to see if the issue that the authors identified with AIMD could be resolved. Then, we attempted to find lower bounds on the performance of the three original CCAs and of TCP Reno. In the original paper, the authors mainly used CCAC for uncovering bugs in a CCA. However, through the use of the utilization query from the BBR and COPA case studies, we thought it might be possible to use this query that identifies under-utilization to identify lower bounds on CCA performance. We could then use these lower bounds to compare CCAs. Thus, if successful, this could be another valid use case for CCAC.

When recreating results, the code provided by the authors was already capable of running the queries mentioned above, so we just set up CCAC and ran the provided queries. To test TCP Reno, we first had to express the algorithm in first order logic. We decided to use the author's existing description of AIMD and modify it by adding the slow start and fast recovery phases. For the performance comparison test, we also modified a query provided by the author and extended it to cover all of the CCAs discussed above. We then ran these modified queries on each of the CCAs several times to try and identify a realistic lower bound on the percentage of utilization.

We were able to recreate the original results identified by the authors in their three case studies and did not encounter

any conflicting findings. We found that TCP Reno does solve the aforementioned bug. We discovered that this was due to the addition of the fast recovery phase. We also found lower bounds on all CCAs above. At first, we discovered that the CCA with the greatest lower bound on utilization was AIMD, which approached 60%. After implementing TCP Reno, we found that the only CCA with a greater lower bound was AIMD with the addition of Fast Recovery with a 64-68% utilization. Interestingly, CCAC could find cases where BBR and COPA experienced utilization rates approaching 0%.

II. Experimental Setup and Methods

A. Recreating Results

The original code contained a Python script to run the queries for the three case studies mentioned in the original paper. Thus, this experiment mainly consisted of running the script with the appropriate CCA and verifying that the results produced matched the results in the paper. To recreate the BBR and COPA results, when we wanted to test if CCAC could find different utilization cases, we added a section so that users could enter any arbitrary utilization rate between 0.0 and 1.0. For COPA, we also needed to adjust the mode of the model between composing and non-composing which could also be set by the user. In order to recreate the BBR results, the model could be kept in composing mode. For an explanation of the difference between composing and non-composing, please see Section 2C.

B. TCP Reno Setup

Before we could implement TCP Reno in CCAC, we had to determine what additions needed to be made to the first order logic of AIMD. In particular, we had to determine the first order logic for the slow start and fast recovery phases.

We maintain an additional variable, $ssthresh$. When the congestion window is smaller than $ssthresh$, we are in the slow start phase. When it is larger, we transition into the congestion avoidance phase. We start in slow start, so ssthresh is initially set to infinity. The congestion window is initially set to some arbitrary constant, α.

In slow start, the congestion window doubles with each time step. This continues until Reno detects loss, i.e. the sender received 3 duplicate ACKs, at which point $ssthresh$ is set to half of the current congestion window and the congestion window is reduced to $ssthresh + 3\alpha$.

After receiving 3 duplicate ACKs, we now enter fast recovery. The congestion window increases by α each time a duplicate ACK is received, so that Reno can keep sending despite not receiving new ACKs. When a new ACK is received, we reset the congestion window back down to $ssthresh$, and we enter congestion avoidance. We will remain in the congestion avoidance phase until a loss is detected, at which point we reenter into fast recovery, or a timeout occurs, and we reset the congestion window to α and reenter slow start. Below, we have expressed this behavior in first order logic.

$$ssthresh_{t=0} = \infty \wedge cwnd_{t=0} = \alpha \quad (1)$$

$$cwnd_{t-1} \leq ssthresh_{t-1} \rightarrow cwnd_t = cwnd_{t-1} * 2 \quad (2)$$

$$cwnd_{t-1} > ssthresh_{t-1} \rightarrow cwnd_t = cwnd_{t-1} + \alpha \quad (3)$$

$$is\_timeout_{t-1} \rightarrow ssthresh_t = \frac{cwnd_{t-1}}{2} \quad (4)$$
$$\wedge\, cwnd_t = \alpha$$

$$3\_duplicate\_acks_t \rightarrow ssthresh_t = \frac{cwnd_{t-1}}{2} \quad (5)$$
$$\wedge\, cwnd_t = ssthresh_t + 3 * \alpha$$
$$\wedge\, fast\_recovery_t = true$$

$$fast\_recovery_t \wedge new\_ack_t \rightarrow cwnd_t = ssthresh_t \quad (6)$$
$$\wedge\, fast\_recovery_t = false$$

Statement 1 corresponds to the initial setup, statement 2 corresponds to the slow start phase, statements 3 and 4 express the AIMD phase, and statements 5 and 6 correspond with the fast recovery phase. With the underlying first order logic figured out, the next step was converting it to logic that could be handled by CCAC. Using the initial code for AIMD as a base, the only variables we needed to add to CCAC were $ssthresh$ and $fast\_recovery$; although new_ack and $3\_duplicate\_ack$ are not implemented as individual variables, there were already conditionals configured to handle this behavior. With TCP Reno implemented in CCAC, we could then run the existing AIMD premature loss query with the selected CCA as Reno. Alongside the complete TCP Reno, we also implemented 2 separate CCAs containing only slow start or fast recovery combined with the original AIMD logic to determine whether or not both phases were capable of addressing the premature loss case.

C. Lower Bound Performance Comparison

In order to compare the performance of the different CCAs, we used the "is there an 10% utilization case" query from the BBR and COPA case studies as a starting point. To simplify manually changing this value, we modified this query to allow the user to input any arbitrary value between 0 and 1.0, so CCAC will find a case where the algorithm has a utilization lower than $x\%$. Through the use of this query, we would be able to identify a lower bound on the utilization that the CCA could achieve, and we would be able to compare different CCAs based on this theoretical worst case performance.

Alongside allowing the user to specify the utilization that they want to find, we also allow the user to select whether to use CCAC's composing or non-composing model. With a composing model, CCAC is able to capture behavior that might occur with several path servers placed serially; this allows it to capture additional network behavior such as jitter. With a non-composing model, CCAC only captures the behavior that can occur within a singular path server.

When CCAC finds a case for a certain utilization, we need to ensure that this result could actually occur in a

real network. This is because CCAC can create unrealistic examples, as it is allowed to set its parameters to any arbitrary value. For instance, CCAC can identify an examples with either an astronomically large congestion window or a case where the sender sends an infinitesimal fraction of a byte. This analysis occurred on a case-by-case basis for each CCA and is explained in Section 3B.2.

III. Results

If CCAC identifies a case where the CCA fails a given performance query, it produces two graphs that model the network activity between the sender and the receiver that caused this failure to occur. To understand what caused any specific failure, we must interpret the graph according to the framework outlined in the original paper. For both graphs, the x axis indicates the time step, and the y axis indicates the amount of data, expressed in terms of the bandwidth delay product (BDP), which is assumed to be constant.

The top graph captures the flow of packets from the sender to the receiver. It shows the amount of packets sent by the sender (arrival curve, dark blue), the amount of packets accepted into the receiver's queue (accepted curve, light blue), and the amount of ACKs sent by the server (service curve, red). The receiver can process new packets at a constant rate, and must acknowledge a packet in its queue before a certain amount of delay. These serve as the bounds of the service curve, and are indicated as the two black curves. When the arrival curve and the arrival accepted curve diverge, it is because packets reach the receiver but are not accepted by the receiver. This means that a packet was lost. Loss occurs when the amount of packets in the receiver's queue exceeds the limit. Loss is depicted with two separate curves; the orange curve represents when loss occurred in the network, and the yellow curve represents when the sender detects that loss.

The bottom graph can vary depending on the CCA. It can show the congestion window the sender maintains over time as a dashed black line. Additionally, this graph also captures the pacing rate displayed as an orange curve. When running CCAC on COPA, the queue delay is presented as an inequality curve shaded in light blue with a separate scale on the right. The units are the same as the top graph.

A. Recreating Original Results

For each of the below experiments, we ran the original queries on our machines, and verified that we could replicate their results.

1) BBR Low Utilization: In the original paper, the author's demonstrated that it is possible for BBR to experience low utilization. In particular, CCAC identifies cases where the service is only a fraction of what should be possible. They found that this behavior occurs since, with BBR, the sender periodically increases its pacing rate to get a better estimate of bandwidth. However, the server continues to service the arrivals at the same rate as before due to limitations of the network. This results in an unchanging estimate of the bandwidth and BDP. In other words, BBR performs no closer to the upper bound and ends up with severe under-utilization.

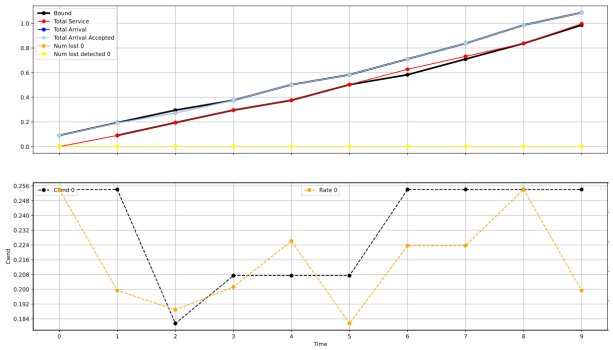

Fig. 1. BBR: 10% Utilization Case

We were able to recreate this result and ended up with the network trace as seen in Figure 1 where CCAC finds a scenario where BBR gets only 10% utilization. In this case, a failed probe can be seen between times t=3 and t=5. The sender increases its sending rate to try and probe for more bandwidth. However, as can be seen in the graph, there is a large difference between the service curve and upper bound due to the state of the network. This difference then causes the probe to fail, and the server continues to be under-utilized.

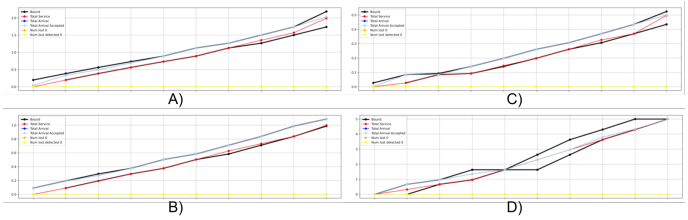

Fig. 2. BBR: Different Utilization Cases

In Figure 2, we also verified that CCAC is able to generate cases where BBR is able to experience various levels of under-utilization. In particular, for BBR, CCAC identified an example of 5% utilization in the top left graph labeled A, an example of 10% utilization in the bottom left graph labeled B, an example of 20% utilization in the top right graph labeled C, and an example of 50% utilization in the bottom right graph labeled D.

2) COPA Low Utilization: For COPA, the authors also used the same utilization query as they used to test BBR. In particular, they showed that CCAC could not identify a case where COPA's utilization fell under 50% when CCAC was set to test with a non-composing model. When set to test with the composing model, CCAC is able to generate cases where COPA encounters arbitrarily low utilization.

When attempting to generate an example where COPA faced 50% or less utilization, we were unsuccessful when testing with the non-composing model which was to be expected.

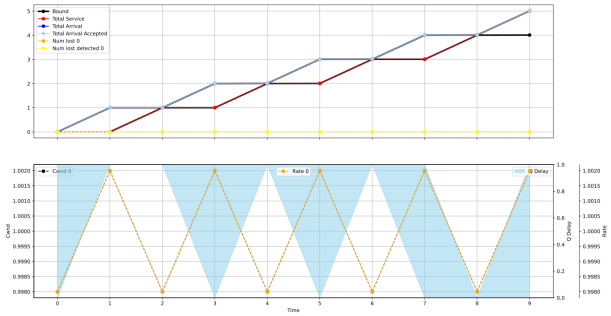

Fig. 3. COPA with Non-Composing Model: 50.1% Utilization

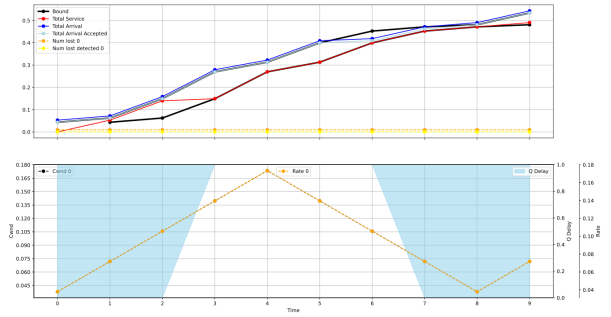

Fig. 4. COPA with Composing Model: 10% Utilization

However, when we used a utilization above 50%, CCAC was able to create an example which can be seen in Figure 3. Meanwhile, CCAC was capable of generating a 10% utilization case for COPA when set to use composing model which can be seen in Figure 4.

3) AIMD Premature Loss: The original paper demonstrated that it is possible for AIMD to experience loss even when the sender's congestion window is smaller than the receiver's packet buffer. This behavior is caused by two factors: an ACK burst and a loss burst. If ACKs arrive in a burst, which is possible due to internal network dynamics, AIMD will increase its congestion window sharply, and send new packets in a burst. This will overwhelm the receiver's buffer, causing loss. This effect is well know, but the paper also showed that it can also occur as a result of a sudden detection of loss. This should be impossible because AIMD cuts the window in half on a detection of loss, and its window should now be smaller than the receiver's buffer. With this in mind, it should not be able to overwhelm the receiver's buffer. However, CCAC found a case when this mechanism failed.

We were able to recreate this result, producing a network trace as seen in Figure 5 that closely resembled the trace presented in the original paper. In this scenario, AIMD starts with a large congestion window of 4 BDP. The receiver's buffer has a capacity of 2 BDP. By sending packets faster than the receiver can handle them, it fills up the buffer completely. This is indicated by fact that the arrival curve jumps above the upper bound of the service curve. This leads to some initial loss occurring at t=1, which can be observed as the

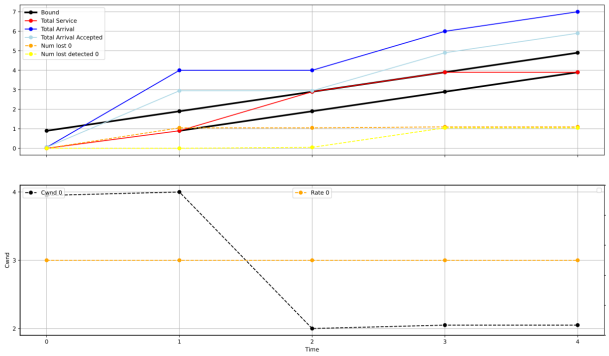

Fig. 5. AIMD Premature Loss

arrival curve and acceptance curve diverge. This loss is not immediately detected by the sender, as it has not received an ACK indicating it has occurred.

At t=2, the receiver sends a burst of 2 BDP ACKs. The sender now becomes aware of the initial burst of loss, and cuts its congestion window in half. The congestion window is now 2 BDP, which is equal to the size of the buffer. Since the congestion window is less than or equal to the buffer size, no more loss should occur. However, the sender also responds to the burst of ACKs by sending a new burst of 2 BDP packets. When this reaches the receiver at t=3, it again overwhelms the buffer, leading to more loss. Thus, CCAC is able to show AIMD encounters loss even when the sender's congestion window is not larger than the receiver's packet buffer, and we were successfully able to replicate this result from the paper.

B. Extending Results

1) TCP Reno Implementation: In the original paper, the authors suggest that the AIMD premature loss issue could be solved by implementing RFC6582, which describes an implementation of TCP Reno. This was proposed, but not verified. We tested TCP Reno with CCAC using their original query and verified that it is a correct fix. TCP Reno adds two additional phases to AIMD; slow start and fast recovery. To determine which of these additions mitigate the issue, we implemented three variations of the algorithm; one that used all three phases (1), one that combined slow start and AIMD (2), and another that combined fast recovery and AIMD (3). See our definition of TCP Reno in section 2B.

The exact same bug described in section 3A.3 occurs when running version 2. As you can see in Figure 6, from t=0 to t=4, the algorithm is in the slow start phase, and doubles its congestion window at each time step. However, it encounters loss at t=5, and transitions into the ordinary AIMD phase. At this point, the same issue occurs, as it cuts its window in half at t=8 upon discovering loss, and then detects loss again when the window is $\leq$ 2 BDP.

However, when attempting to rerun this query with either version 1 or 3, CCAC was unable to find a network state that satisfied the premature loss query. The addition of the fast

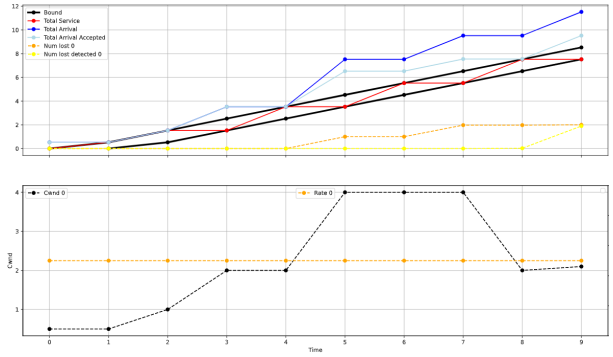

Fig. 6. Reno Slow Start Premature Loss

recovery phase prevented this issue. We believe this is because when an isolated incidence of loss occurs, Reno increases its congestion window for each duplicate ACK received, allowing it to continue sending new packets at a steady rate. This prevents it from sending packets in a large burst, which is what causes the second incident of loss.

2) Lower Bound Performance Comparison: We now attempted to discover lower bounds on the utilization for each CCA. Alongside this lower bound for AIMD and Reno, we also report the corresponding the maximum transmission unit α (the value used to increment the congestion window) found at this bound as a reference. Additionally, we provide two network trace that corresponds to that utilization for the composing and noncomposing model; the composing model trace will be labeled with A whilst the noncomposing model trace will be labeled with B. When attempting to do this, we encountered some problems concerning whether or not the output from CCAC was realistic. CCAC can generate any arbitrary example as long as it fits within the bounds we provide. In order to find a realistic lower bound on CCA performance, we had to fine tune the initial bounds used in the query. Please note, we will discuss what the bounds approached, but the precise lowest values we found will be listed in Table 1.

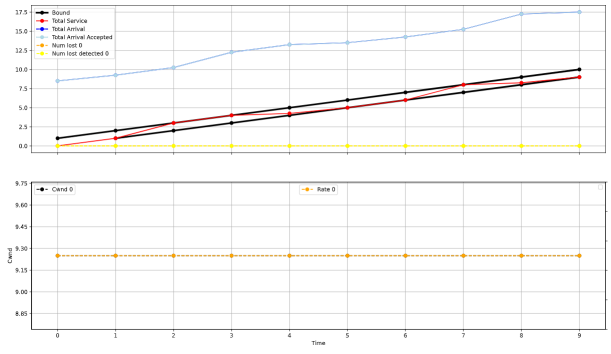

Fig. 7. AIMD with an unrealistically high congestion window

When attempting to find the lowest utilization for AIMD, we initially let the receiver's buffer vary arbitrarily. However, this caused CCAC to find cases where the receiver's buffer was unreasonably high allowing the sender to send 10 or more times the BDP worth of packets without penalty, so AIMD could achieve nearly 100% utilization by maintaining an enormous congestion window, as shown in Figure 7. We then limited the receiver's buffer to 2 BDP, and reran the query. We now encountered cases where CCAC set the maximum transmission unit, α, above 1 BDP. However, AIMD increments the congestion window by the MTU after receiving each ACK, so α should be small relative to the BDP. We then capped the range of possible α between 0 and 0.5 BDP. This now produced realistic traces. Through trial and error, we discovered the lower bound on utilization approached 60% for both the composing and non-composing models. Interestingly, as we got closer to this bound, α approached 1/3 BDP. The network traces corresponding to this bound are included in Figure 8.

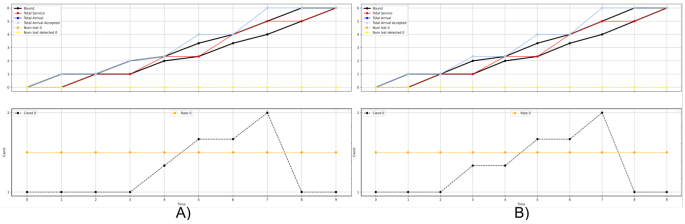

Fig. 8. AIMD Composing and Non-Composing 60% Utilization

When attempting to find the lowest utilization for Reno, we encountered the same unrealistic behavior that occurred with AIMD when we let the buffer and alpha be chosen arbitrarily. As a result, we used the same bounds we used for AIMD. For the version including only AIMD and slow start, we found that the lower bound on utilization approached 40% in the composing model and 40.63% in the non-composing model with α approaching 0.25 and 0.0625 for the two models. The network traces identified by CCAC with these utilization values for composing and non-composing are depicted in Figure 9. For the version including AIMD and fast recovery, the bounds approached 64% and 68% for the composing and non-composing models respectively, and α approached 0.4 as we got closer to those bounds for both composing and non-composing models. The network traces identified by CCAC with these utilization values for composing and non-composing are depicted in Figure 10. For the complete version of TCP Reno, the bounds were the same as slow start, approaching 40% and 40.65% and α approaching 0.125 and 0.0625 for composing and non-composing respectively. Even though the complete version of Reno had the same bounds as AIMD plus Slow Start, the examples produced were slightly different and utilized MTUs that were a factor of 2 apart as can be seen in graph. The network traces identified by CCAC with these utilization values for composing and non-composing are depicted in Figure 11.

When it came to finding the lower bounds on utilization for BBR, we found that the arbitrary nature of the buffer and α selection did not lead to any unrealistic cases. However,

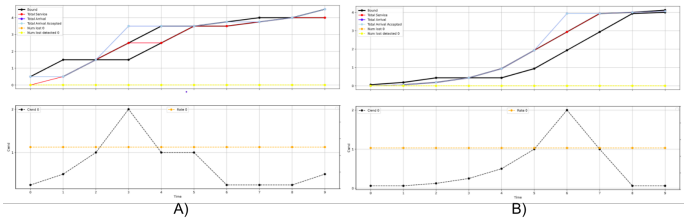

Fig. 9. AIMD and Slow Start Composing and Non-Composing

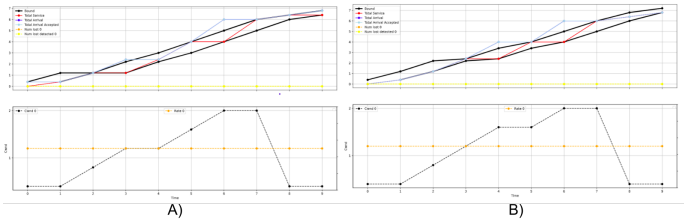

Fig. 10. AIMD and Fast Recovery Composing and Non-Composing

we did notice some unrealistic behavior when it came to the lower bounds of utilization on the composing model. With this model, we found that after a certain utilization value CCAC repeats the exact same example except the steps on the graph just get repeatedly smaller; this can be seen in Figure 12 for 0.00000001% utilization and Figure 13 for 0.7% utilization. Since AIMD and Reno produced different traces for examples only 0.1% utilization apart, we found it strange that the 0.00000001% utilization example was the exact same as the 0.7% example. Since CCAC's arrival and service functions are in terms of the BDP with arbitrary units, we were not able to identify any realistic bounds we could impose on these functions. As a result, we decided to choose the maximum utilization that produced this repeating behavior as our lower bound for the composing model. For the composing model, the repeating behavior starts to occur at 0.7% utilization. For the non-composing model, the lower bound on utilization was 40.25%. The network traces that produces the above utilization values are depicted in Figure 14.

When it came to finding the lower bounds on utilization for COPA, we had a similar problem in the composing case. We could also make the utilization arbitrarily small, and saw the same pattern repeating after we reached 0.00009% utilization. For the non-composing case, we discovered a lower bound at 50% utilization. Finally, the network traces that produces these utilization values are depicted in Figure 15.

Our results are summarized in Table 1. Each of the models performed the same or better in the non-composing model. Since the composing model is able to capture more varied behavior in the network that can negatively affect performance, it necessarily covers more cases, and therefore finds a lower bound. Interestingly, the CCAs the relied on AIMD as a base did not have as noticeable as a difference between the performance between the composing and non-composing cases. Meanwhile, BBR and COPA saw their utilization border on

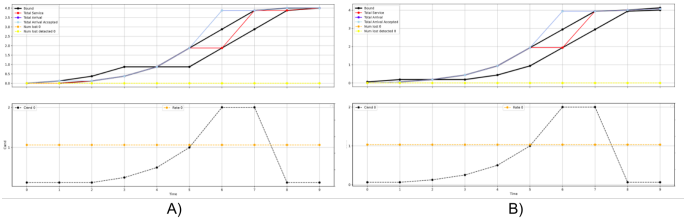

Fig. 11. TCP Reno Composing and Non-Composing

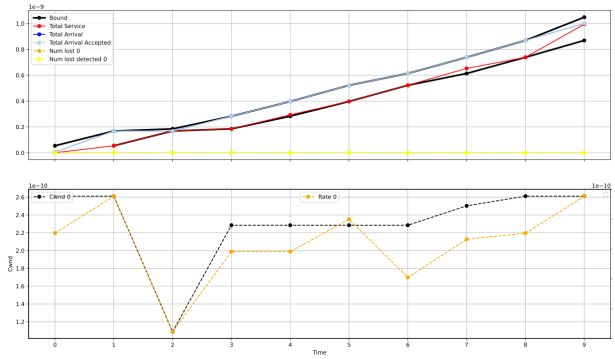

Fig. 12. BBR Composing Model 0.00000001% Utilization

0% for the composing case with a difference of nearly 40% or more when it came to the non-composing case. It is interesting to note that the simpler congestion control algorithms ended up with a greater lower bound. We hypothesize that this is because it takes into account fewer variables, so there is less for CCAC to manipulate leading to less antagonistic network traces. Overall, we found that AIMD and fast recovery had the highest lowest guaranteed performance while COPA had the lowest guaranteed performance when it came to the composing model and BBR had the lowest guaranteed performance when it came to the non-composing model.

IV. Limitations and Future Work

As the original paper stated, CCAC is unable to replicate behavior that might occur when multiple CCAC are competing for bandwidth. Additionally, the lack of an unambiguous unit system combined with the lack of a discrete time scale limits the analysis. These factors limit the types of queries that are possible with this tool. If we were able to model competing CCAs, we would be able to perform queries on fairness and

TABLE I
Lower Bounds on Utilization

CCA	Model	
	Composing	*Non-Composing*
AIMD	60.01%	60.01%
Complete Reno	40.01%	40.63%
AIMD + Slow Start	40.01%	40.63%
AIMD + Fast Recovery	64.01%	68%
BBR	0.7%	40.25%
COPA	0.00009%	50.01%

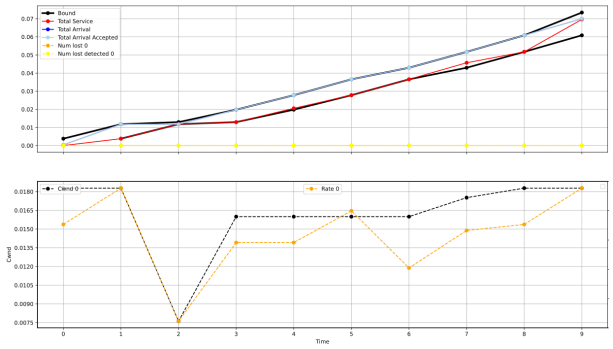

Fig. 13. BBR Composing Model 0.7% Utilization

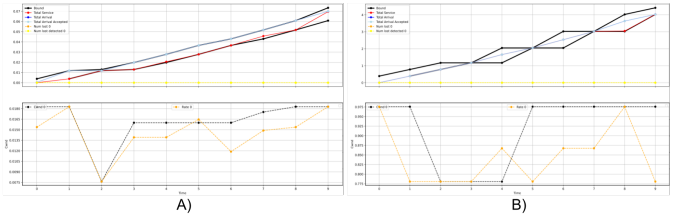

Fig. 14. BBR Composing Model and Non-Composing Lower Bound

improve our lower bound experiment to cover more cases, among other things. If we had a consistent set of units, we could compare our theoretical results with empirical results from the experiments of others. In particular, we would be able to perform analysis on what is likely to occur as opposed to only what can occur.

For the TCP Reno implementation, the fact that CCAC used a discrete time scale, not continuous time, did not present difficulties. However, for other potential CCAs, this would make it difficult to implement certain behavior that relies on a more precise timescale. For instance, there are other, more modern versions of TCP which we would have liked to investigate, such as TCP CUBIC. As a potential future experiment, one could implement additional versions of TCP - such as TCP CUBIC. However, CUBIC changes its congestion window as a function of the time, which may be impossible to implement in the present version of CCAC.

For the performance experiments, it is important to note that we calculated the theoretical lower bound on performance of a CCA, not the performance a user should expect. For instance, BBR tends to perform better than TCP Reno in many applications, but has a lower theoretical utilization bound. Additionally, we do not know the probability of any of these events occurring in a real network setting. It is possible that, while a CCA can reach arbitrarily low utilization, it almost never occurs in real life. However, with these bounds and edge cases identified, it would be possible to modify the algorithm to address the case that produced the lower bound similar to what was done in the original paper. With these additions, it would be interesting to see how much of a difference they make to the identified lower bound. Finally, it would also be interesting to identify these bounds

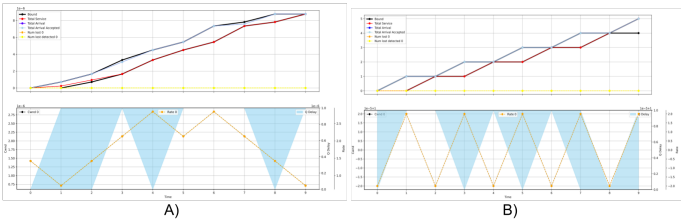

Fig. 15. COPA Composing Model and Non-Composing Lower Bound

on additional CCAs. As was mentioned above, it would be important to ensure that CCAC can model that CCA before performing any analysis.

V. Conclusion

We were able to recreate the results found by the authors in their case studies, and this allowed us to gain familiarity with CCAC in order to successfully perform our extensions. In particular, we were successfully able to use CCAC to test TCP Reno. We showed that the addition of the fast recovery phase to AIMD prevents the premature loss bug the authors identified in the paper from occurring, confirming their hypothesis. We also showed that CCAC could be used to find the theoretical lower bounds on channel utilization for a given CCA. Although CCAC has its limitations, it is a powerful tool for testing and debugging CCAs, and it allows one to view a CCA outside the limits of empirical testing. There is room for extending our results by modifying CCAC to test in continuous time and by using the framework to test different CCAs.

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 (SIGCOMM '21). Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3452296.3472912

VI. Appendix

CCAC is a Python tool that can be ran locally without the need to set-up any nodes on GENI. It makes use of the Z3 automated theorem prover.

A. Setting up CCAC

1) Approach 1: This is a more general approach to setting up CCAC:

1) Download our code. We have provided a zip file, and it is also available at https://github.com/AlexKwi/CS2520-Final-Project.
2) Install Python and Pip on your machine. The author recommends any version above 3.8.5. We used version 3.8.15 as Python 3.10 was causing problems.
3) Download Z3 from https://github.com/Z3Prover/z3 and follow the appropriate steps for installing it on your specific OS. We downloaded the .whl file of the 4.11.0 release.

4) Install the Python packages matplotlib, numpy and scipy with the command *python3.8 -m pip install matplotlib numpy scipy*.

2) Approach 2: This was the approached used to get CCAC working in Linux:

1) Download our code. We have provided a zip file, and it is also available at https://github.com/AlexKwi/CS2520-Final-Project.
2) Install Python and Pip on your machine. The author recommends any version above 3.8.5. We used version 3.8.15 as Python 3.10 was causing problems.
3) Create a python virtual environment as follows:
   python3 -m venv proj-venv
4) Enter the virtual environment as follows:
   source proj-venv/bin/activate
5) Install the dependencies as follows:
   pip install z3-solver matplotlib numpy scipy

B. Code Overview

The zip file contains all of the necessary code other than the aforementioned dependencies. The files that we modified for our experiments were:

- example_queries.py
  - Added necessary queries for testing AIMD and Reno under different utilization values
  - Added the ability to flexibly select utilization and choose between composing and non-composing
  - Added Reno premature loss query
  - Added different reno versions (aimd+slow-start, aimd+fast-recovery, and complete reno)
- model.py
  - Added necessary changes in order for CCAC to choose our Reno variants
- plot.py
  - Fixed a typo in the variables being plotted
  - Added all reno versions to the list of CCAs where alpha should be printed

Alongside the above files, the only completely new file is cca_reno.py which contains the necessary code for the three reno variants.

C. Recreating Our Results

After running a query, CCAC will print **unsat** if it could not find a case violating that property. If it does find a case, it will print a matrix containing a list of values. It will also save the graph displaying the network trace as **multi_flow_plot.svg**. Please note: there is some randomness in how CCAC selects the values it uses in the network trace, so some of the graphs might not be exact replicas of each other. However, the same kind of behavior should occur just at different time steps.

To recreate the AIMD premature loss results from the original paper:

1) Run the following command:
   python3 example_queries.py aimd_premature_loss
2) This will produce the graph as seen in Figure 5.

To recreate the BBR low utilization results from the original paper:

1) Run the following command:
   python3 example_queries.py bbr_low_util
2) When prompted for input, input 0.1.
3) When prompted for input again, input 1.
4) This will produce the graph as seen in Figure 1.
5) If you wish to reproduce Figure 2, simply repeat steps 1-4 but replace the value "0.1" with the desired utilization.

To recreate the COPA low utilization results (composing model) from the original paper:

1) Run the following command:
   python3 example_queries.py copa_low_util
2) When prompted for input, input 0.1.
3) When prompted for input again, input 1.
4) This will produce the graph as seen in Figure 3.

To recreate the COPA low utilization results (non-composing model) from the original paper:

1) Run the following command:
   python3 example_queries.py copa_low_util
2) When prompted for input, input 0.501.
3) When prompted for input again, input 2.
4) This will produce the graph as seen in Figure 4.

To recreate the Reno premature loss results for complete reno:

1) Run the following command:
   python3 example_queries.py reno_premature_loss
2) When prompted for input, input 1.
3) CCAC will output **unsat**.

To recreate the Reno premature loss results for slow start:

1) Run the following command:
   python3 example_queries.py reno_premature_loss
2) When prompted for input, input 2.
3) This will produce the graph as seen in Figure 6.

To recreate the Reno premature loss results for fast recovery:

1) Run the following command:
   python3 example_queries.py reno_premature_loss
2) When prompted for input, input 3.
3) CCAC will output **unsat**.

To recreate the performance results:

1) Start by running one of the following:
   - For BBR:
     python3 example_queries.py bbr_low_util
   - For COPA:
     python3 example_queries.py copa_low_util
   - For AIMD:
     python3 example_queries.py aimd_low_util
   - For Reno:
     python3 example_queries.py reno_low_util
2) When prompted for an input, enter the utilization specified in Table 1 that you wish to recreate.
3) When prompted again for an input, input 1 if you want to recreate the composing results, and input 2 if you want to recreate the non-composing results.
4) CCAC will then produce the appropriate graph.