A Simulator and Benchmark for the RingWorld Protocol

Yichi Zhang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-121
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-121.html

May 13, 2022

A Simulator and Benchmark of the RingWorld Protocol

by Yichi Zhang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Scott Shenker
Research Advisor

5/10/2022

(Date)

* * * * * * *

[illegible]

Professor Sylvia Ratnasamy
Second Reader

5/13/2022

(Date)

A Simulator and Benchmark for the RingWorld Protocol

Abstract

RingWorld is a new consensus protocol aiming for deployment in datacenters that utilizes programmable top-of-rack (ToR) switches and multiple servers in a rack. The ToRs connect to each other and form a logical ring. This report attempts to build a simulator for RingWorld protocol's normal execution model, and to benchmark the protocol.

Introduction

Consensus protocols are protocols that make multiple systems agree on certain things. It could often be represented as machines agreeing on the order of an append-only log replicated on each machine.

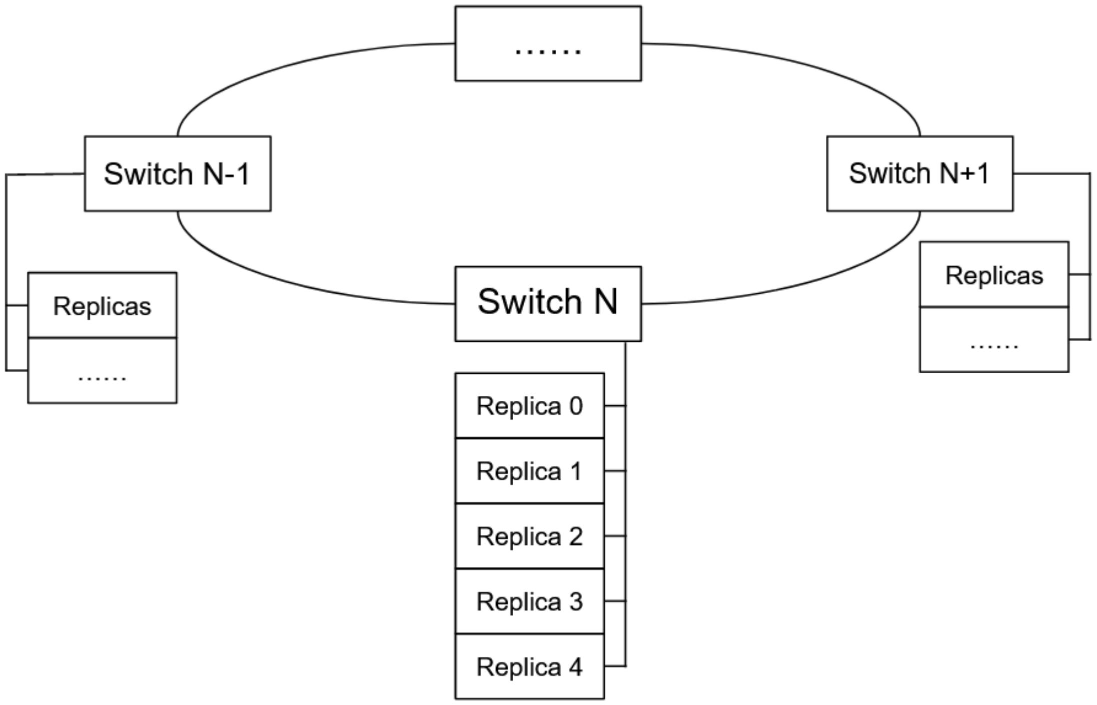
Traditionally, there are two paradigms of consensus protocols. One is the leader-based protocols, among them are Paxos [2] and Raft [3], etc. These protocols have low latencies during normal operations, but because of the communication overheads, the throughput is usually limited. In these protocols, although using more replicas increases the ability to tolerate fault, it brings a performance hit with lower throughput and higher latency. Furthermore, because of the existence of one single leader, the failure of it usually leads to a view-change operation that needs to be carried out to elect a new leader before they can resume normal operation.

Another is the ring-based protocols like Carousel [4]. These potentially have simpler failure-recovery models, and have high throughput. However, the latency of these systems scales up linearly with the number of replicas.

RingWorld [1] takes advantage of the programmable switches on top of each rack in datacenters, and gets the best of both worlds by running on a ring of racks model. On one hand, the latency is rather constant with the number of replicas because a constant number of racks are involved. On the other hand, the throughput is high thanks to the high speed of the switches. RingWorld also has a simple failure recovery protocol that only involves machines local to the failed point.

The next section will present the RingWorld protocol in more detail.

Background



RingWorld adopts the new topology of ring-of-racks. As shown in the image above, every switch is logically connected to two other switches in the ring - the predecessor and the successor. The connections between switches are extremely fast. Each switch is responsible for disseminating messages to the replicas under it, and Each switch only has a local view of the ring. Similarly, a replica does not need to know about other replicas' presence to operate normally.

For a message to be added to an append-only log in RingWorld, the following steps are taken. First, the incoming message from a client is routed to any one of the replicas. Upon receiving the message, the replica initiates a request to add it and sends it to its parent ToR. The ToR assigns the message an unique ID in the form of *(sequence number, switch ID)*, increments its sequence number counter, and starts the first round for the request - the Propose round.

In this round, every ToR that receives the message will propagate it to its children, as well as the next ToR. Replicas will need to keep a *pending* queue for messages that they have received but not yet delivered, sorted using the unique ID of the message. They will also need to acknowledge the message back to its parent, who will count the ACKs it has received for the message. When the Propose packet gets forwarded back to the original ToR, the Propose round ends and the second round, the Collect round, begins.

A Collect packet contains the information about how many machines under each ToR has acknowledged, and every ToR on the ring will forward the Collect packet after writing this field. The Collect packet could potentially travel the ring for multiple rounds, up to a configurable maximum value, because the time it takes for a packet to travel one round on the ring is significantly shorter than that for one to travel to and from a replica. After the Collect round is complete, the initial ToR can decide whether to proceed or abort based on the information carried in the Collect packet. If more than *f* racks have more than *m* acknowledgements each, the message is committed, and aborted otherwise.

Then the Commit round starts. Every rack will forward the Commit decision to its children and its successor. When a replica receives a Commit packet, it will mark the message in the pending queue to reflect the decision, and try to deliver some messages in the pending queue. A replica iterates from the beginning of the sorted queue and stops at the first undecided message. For each decided message in the queue, it either gets flushed to the delivered log, or gets deleted. The original replica that issued the request can reply to the client after a Commit packet is received.

The Simulator

As a simulator that runs on Linux, there are problems that it has to ignore and compromises to make. For example, it cannot run even nearly as fast as an actual programmable switch, and there is very little that can be done about it. In this section, I will discuss some design considerations of the simulator, and address some of the road blocks I have encountered.

Data Transport

First of all is the question of what layer and protocol to use to carry the payload. RingWorld, as a protocol acting on programmable switches, is supposed to be a L2 or L3 protocol. However, programming on lower layers is beyond my current knowledge and too complicated for the sake of this project, so they are out of consideration.

The closest thing to a layer 2 frame is a UDP packet. It communicates in packets instead of streams and it is not a reliable protocol, therefore the first version of the simulator is written using the UDP transport protocol. It ran well until it was hit with a large load, when it just dropped packets, and left the replicas and switches with partial information, and very few messages could get delivered, so UDP is out of consideration.

This issue in packet loss led to another readthrough of the RingWorld paper, and it was discovered that RingWorld actually assumes that the interconnects between switches neither drop nor reorder packets. Therefore a second version of the simulator was written using TCP. To reduce communication overhead, connections are made not for each request being made, but at the beginning of the simulation between each replica and its parent, and between the predecessors and successors on the ring. This way, no packet loss will be exposed to the simulator, and not much overhead incurs except for the potential packet retransmission.

Using TCP as the transport posed one roadblock as what I was taught in the internet courses usually does a blocking *listen* and *recv* for incoming data, while in the simulator we need to listen for multiple connections made from the children, the parent, or the predecessor depending on the role. The solution to this is to use *epoll*. *Epoll* allows a programmer to simultaneously listen for events on multiple file descriptors (fd) by “combining” the fds of interest and giving a new one to the programmer to listen to. In this scenario, every socket fd is added to the combined fd, after which we have the ability to listen on all the sockets for incoming messages.

Switches

The switches are straightforward, and one big difference from that described in Section (2) is that in the simulator, they also listen for Control messages. A control message could be from me, the runner of the simulator, to instruct the switch to establish a connection onto the port of its successor. It could also be from a replica which would bear the meaning that the replica is joining the switch as a child, or from a switch signaling that it is the predecessor. This is needed because of how the benchmark is set up - all instances of switches and replicas need to be up and running before they interconnect with each other. Only the runner of the simulator knows whether all machines are up, and only the runner knows when to terminate all instances.

When a switch is initialized, it sets up the *epoll* fd, and binds to a master port, on which it listens for incoming connections. Whenever a new connection is established, the new connection is added to the *epoll* structure, and the switch needs to keep track of the role of the connection - whether it’s from a child or a switch in its previous location. We achieve this by keeping a mapping from the fd to a role-type enumeration. Keeping this mapping could also serve as a debugging purpose since we can check for unexpected messages.

The switch further contains two mappings. One is from a Message ID tuple (sequence_number, switch_id) to the number of acknowledgements. This is to decide whether a message is getting committed or not when it is time to decide. The other is from a Message ID to the number of Collect rounds passed.

Replica

The replicas are quite similar to switches in terms of how they operate. They also have a master port to listen on, so that requests from clients can get to them. They also listen for Control messages to poke their parents about their presence. A replica also has the responsibility to reply to the original client for each committed message, so it needs to keep track of the messages it received. This is achieved by keeping a monotonically increasing sequence number and a mapping between the sequence number and a message at each replica. When it receives a Commit message targeting a message it broadcasted, it sends back either a success or failure reply.

The function of the pending queue is provided by a C++ std::map using Message IDs as keys. Since it is a custom struct, a custom comparator function, which prioritizes sequence number over switch ID, needs to be provided to the map. When a Propose message is received, it is inserted into the pending queue with the status marked as UNDECIDED. When a Commit message is received, the status is marked as PROCEED or ABORT. During delivery, a replica iterates from the beginning of the ordered map, stops at the first UNDECIDED message, and takes action on the decided messages.

Benchmark Setup

We run two groups of benchmarks with a different network setup. In both groups, we measure the latency of RingWorld with different numbers of switches and replicas. In these benchmarks, two computers are involved: (a) an AMD Ryzen 5 3600 CPU with 6 cores and hyperthreading, clocked at 4.0 GHz, and (b) an Intel i3-8100 CPU with 4 cores and no hyperthreading, clocked at 3.6 GHz. Both of them run Linux, and both of them are on the same wired LAN. All messages are sent from one python script on computer (a).

Setup 1

We have all instances of switches and replicas running on computer (a). This would be the theoretical fastest setup I will have access to: pinging localhost has an average round-trip time of 0.03 milliseconds.

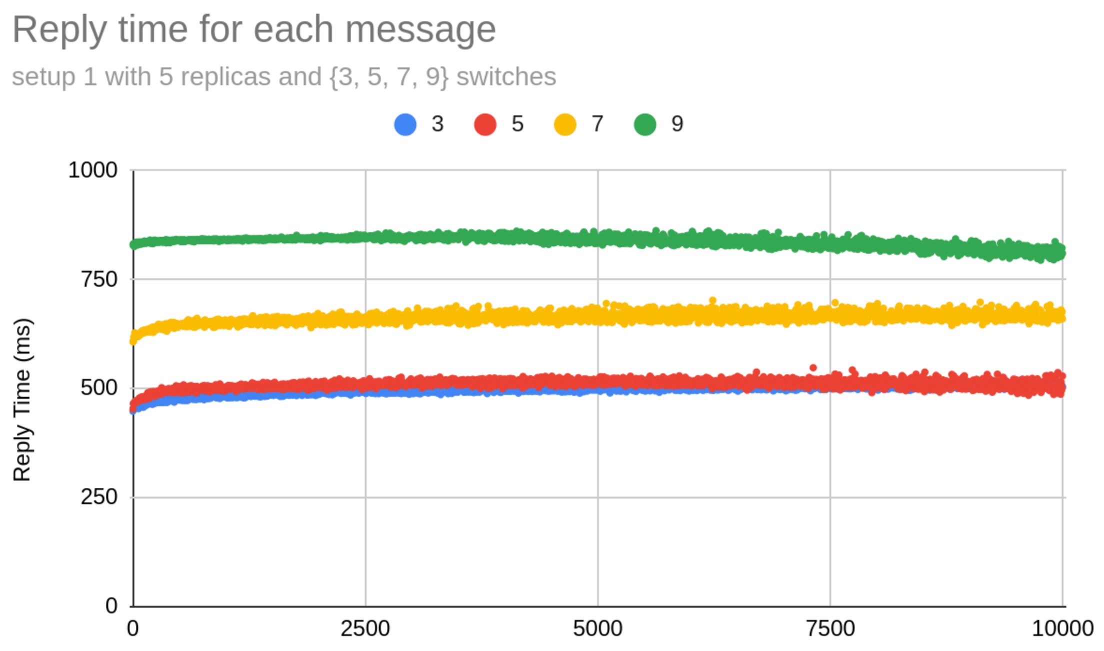
Setup 2

We have all instances of switches on computer (a), and all replicas on (b). Since they are on the same LAN, the connections between switches and replicas are quite small (0.16 milliseconds in ping), but connections between switches are even faster (still 0.03 milliseconds). The choice to make switches on (a) and replicas on (b) instead of the other way around is made because of the fact that switches process many more messages than replicas do even though there are more replicas than switches.

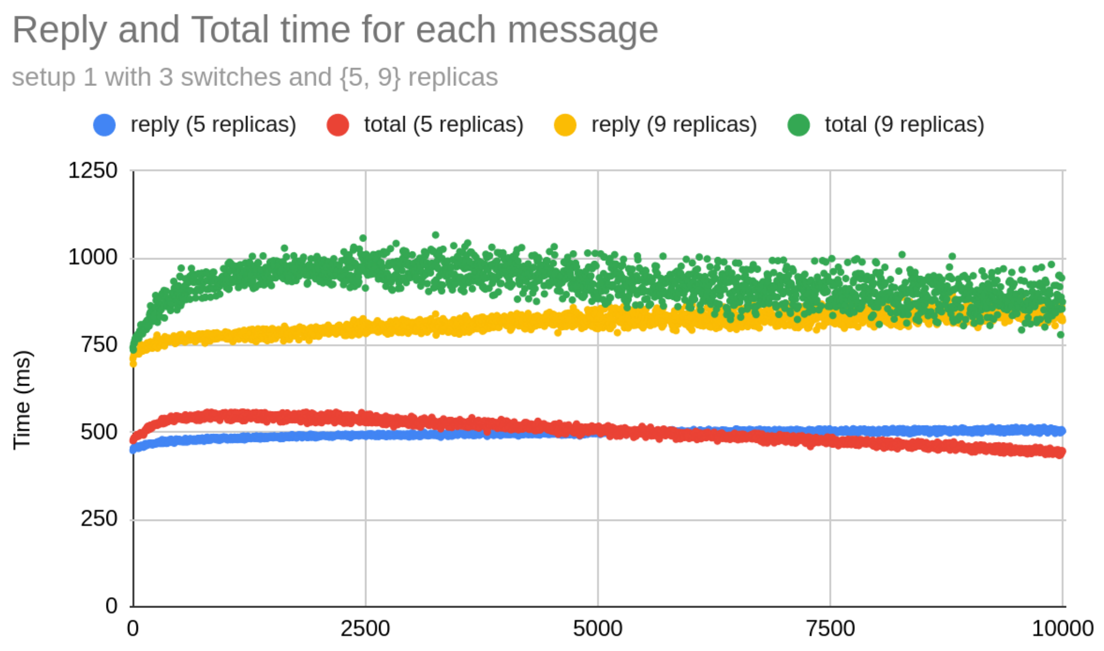
Results

We ran the two setups using these test suites. With 5 replicas for each switch, we test using {3, 5, 7, 9} switches, and with 9 replicas for each switch, we test using 3 switches. For each suite, we bombard the system with a total of 10,000 messages, each one about 8 bytes, and note down several time lengths: the time from a replica receives from a client to the time it replies back, and the time it delivers the message; the time from a switch starts a propose round to the end of the propose round, the end of all collect rounds, and the end of the commit round. All times are measured in milliseconds using C++ std::chrono::steady_clock. Because 10,000 data points make the plotting software slow, and to make the graph smoother, we take the average of five data points and use the resulting 2,000 to plot.

Setup 1



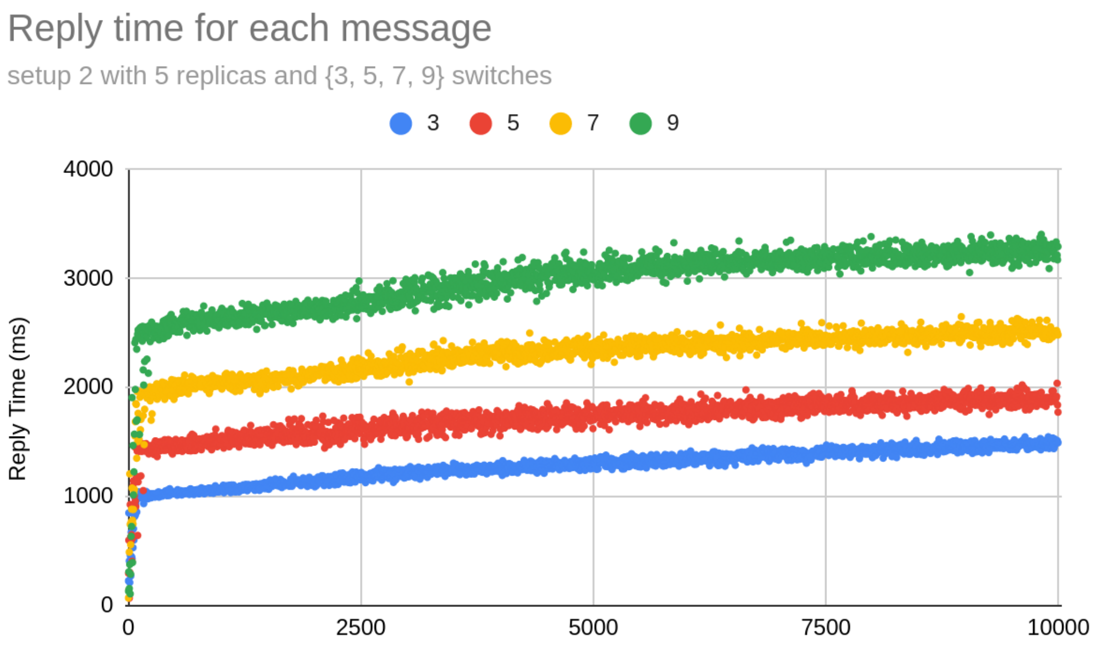
Here we observe one effect. Because of the low latency in this setup, the load on the machine as well as the processing time of the replicas affect more on the reply time of the system, and as a result, we see that it matters not that much to have either 3 switches on the ring or 5. This is mentioned in the RingWorld paper, and in real deployment, this could potentially extend to a larger number of switches.



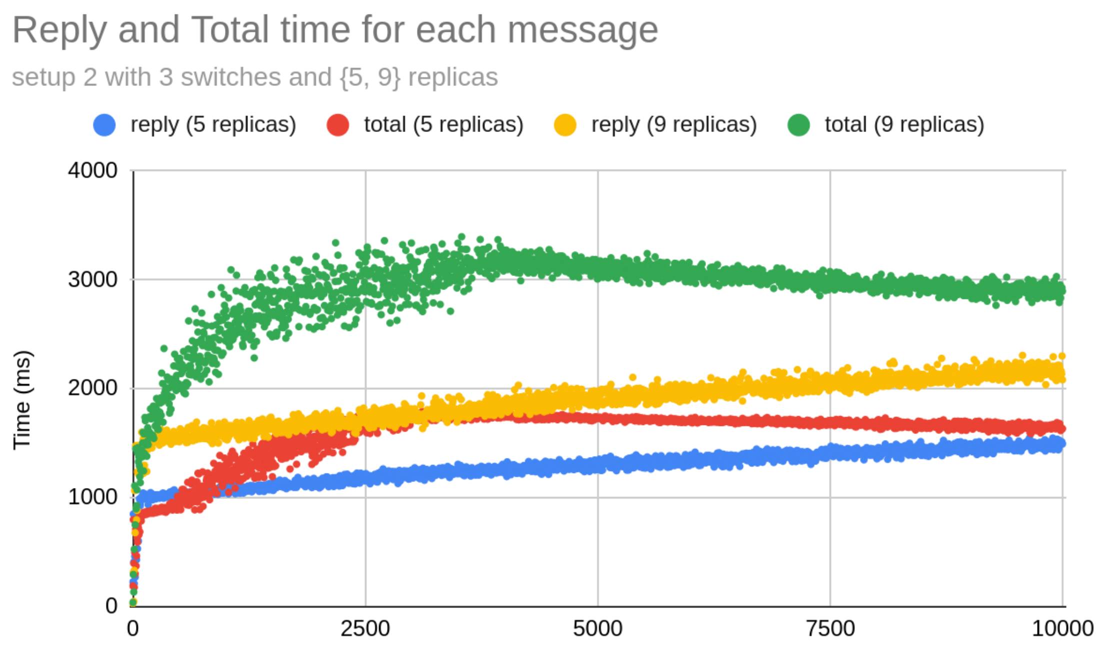
Again, we observed the effect mentioned above. Furthermore, we can see that with 9 replicas, the load on the system increases because of the increase in the number of packets to be processed. This is reflected in the graph as a wider “line”, showing an increase in variance.

We also try to explain a few things here. The gradual decrease in the total time could be caused by the gradual decrease in the number of packets flowing in the system, because the influx of messages stops rather early on. The crossover of total time and reply time is because we measure the two times on switches and replicas, respectively. This means that a portion of the network latency is not captured by these numbers, and results in the total time shorter than the reply time.

Setup 2



In setup 2, because everything gets slower - the network is slower because we are on the LAN, the replicas run slower because they are on a slower machine - a ring of size 3 as opposed to 5 actually has an effect on the time. We also see a great increase in reply times (in the 1300 ms area for 3 switches as opposed to 500 in the previous setup) There's also bigger variance compared to setup 1, and again, this could be because of the nature of networks.



Here, once again, we see the huge fluctuations in total times especially in the scenario with 9 replicas, and the gradual decrease in total times.

Future Works

This simulator could potentially benefit greatly from a total rewrite using C++'s Boost library. The asynchronous I/O should be more performant than the current single-threaded approach. It could also be implemented using only one host and add in artificial internet delay to mimic a real environment. Being asynchronous also gives the option to add in manual network delay, which makes running on one single machine more versatile.

Also, for some reason, the data gathering code stopped working at 10000 ms, and the benchmark had to stop at 9 switches with 9 replicas.

Conclusion

This report discusses the normal operation of RingWorld protocol, and ran a benchmark to investigate how RingWorld scales with an increased number of machines.

References

[1] Aisha Mushtaq. RingWorld: Consensus for Datacenters. 2022.

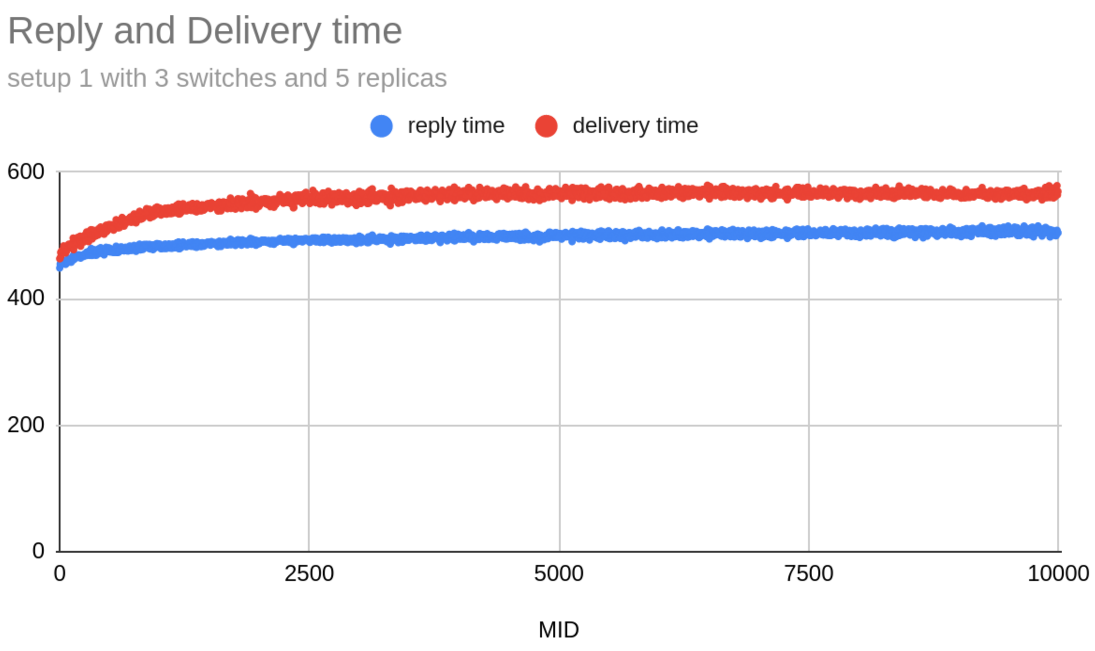
[2] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR),* 47(3):1-36, 2015.

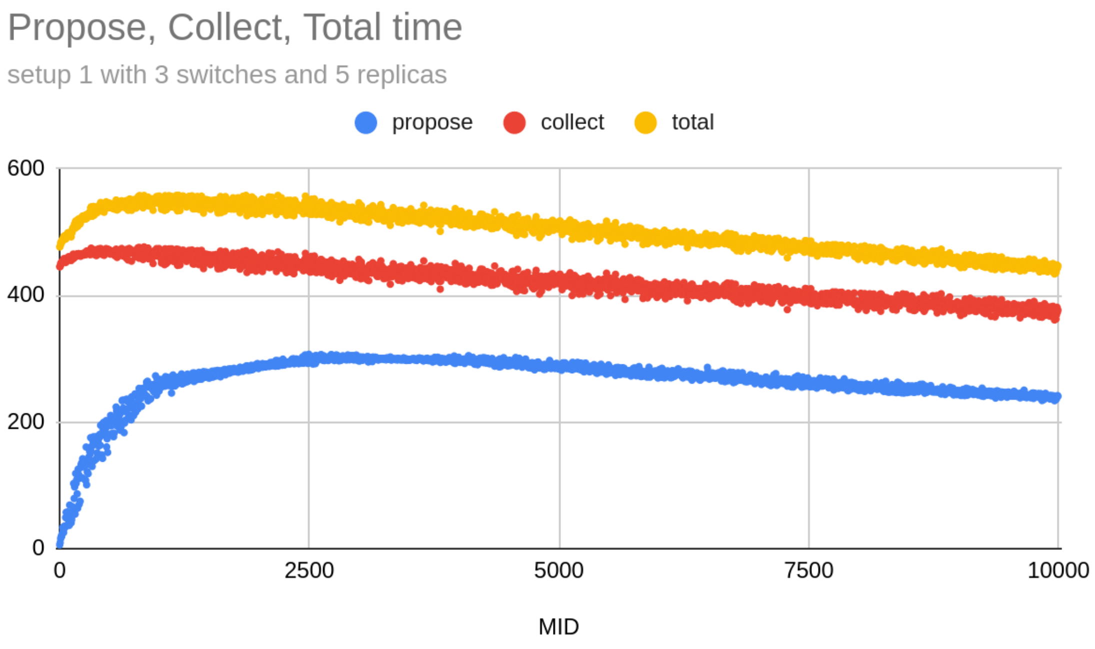
[3] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC,* 2014

[4] Rachid Guerraoui, Jad Hamza, Dragos-Adrian Seredinschi, and Marko Vukolic. Can 100 machines agree? *arXiv preprint arXiv: 1911.07966*, 2019.

Appendix

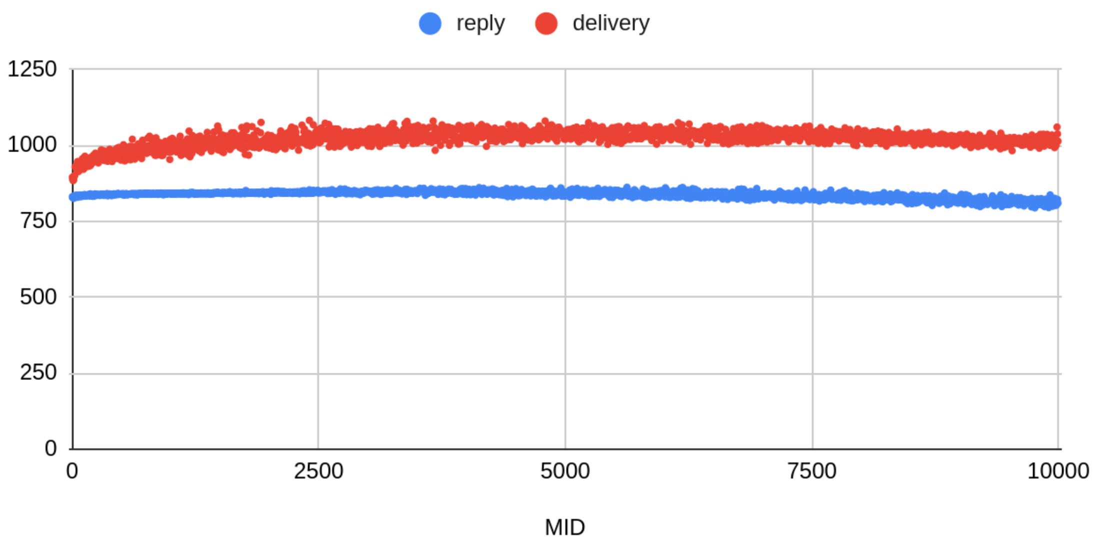
Other Graphs





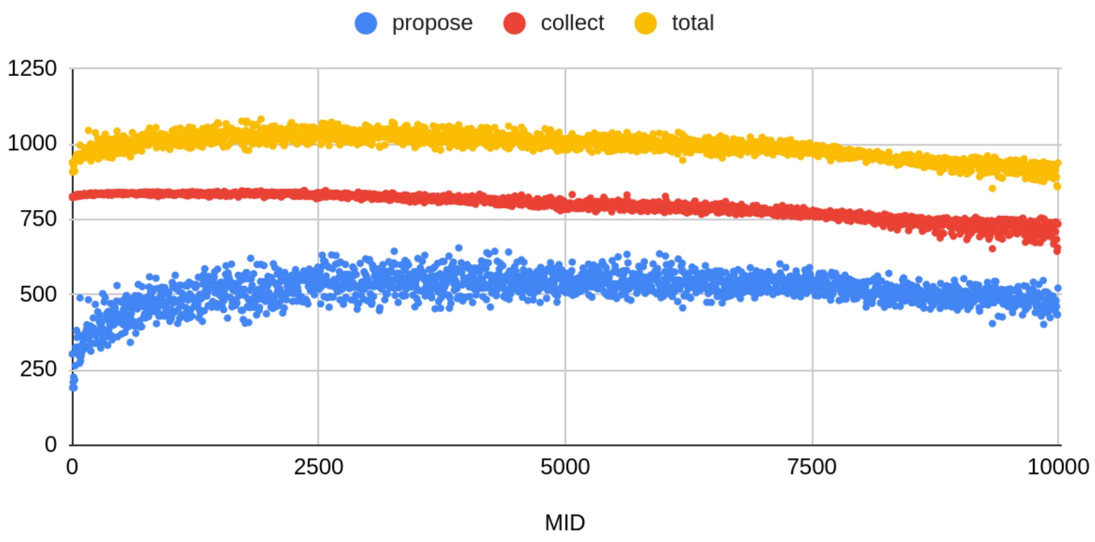
Reply and Delivery time

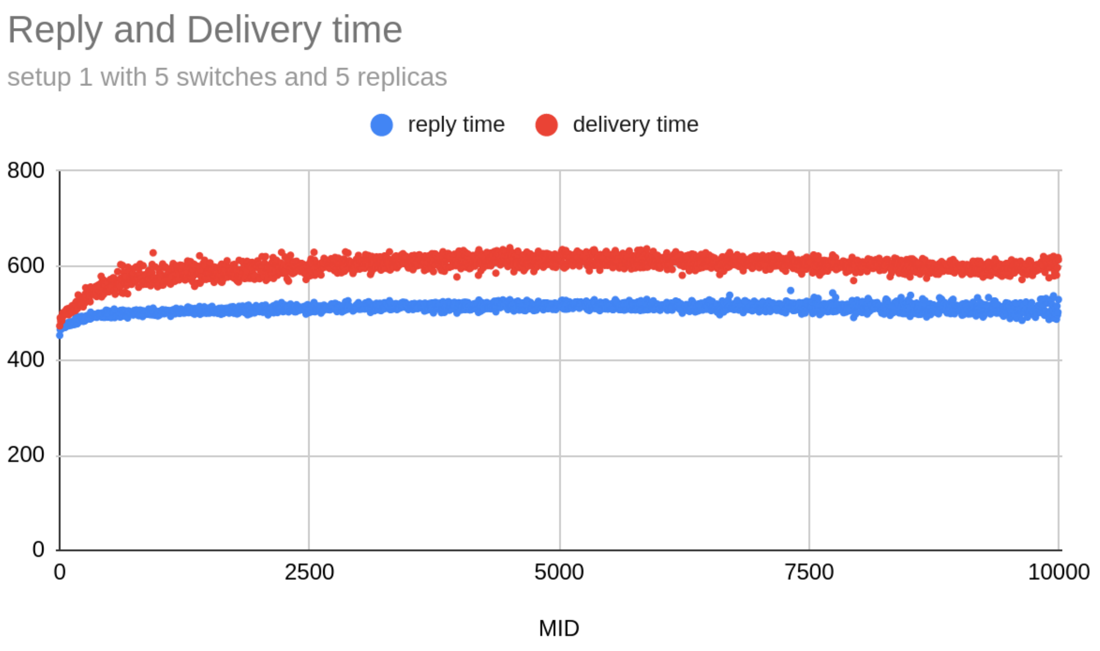
setup 1 with 9 switches and 5 replicas

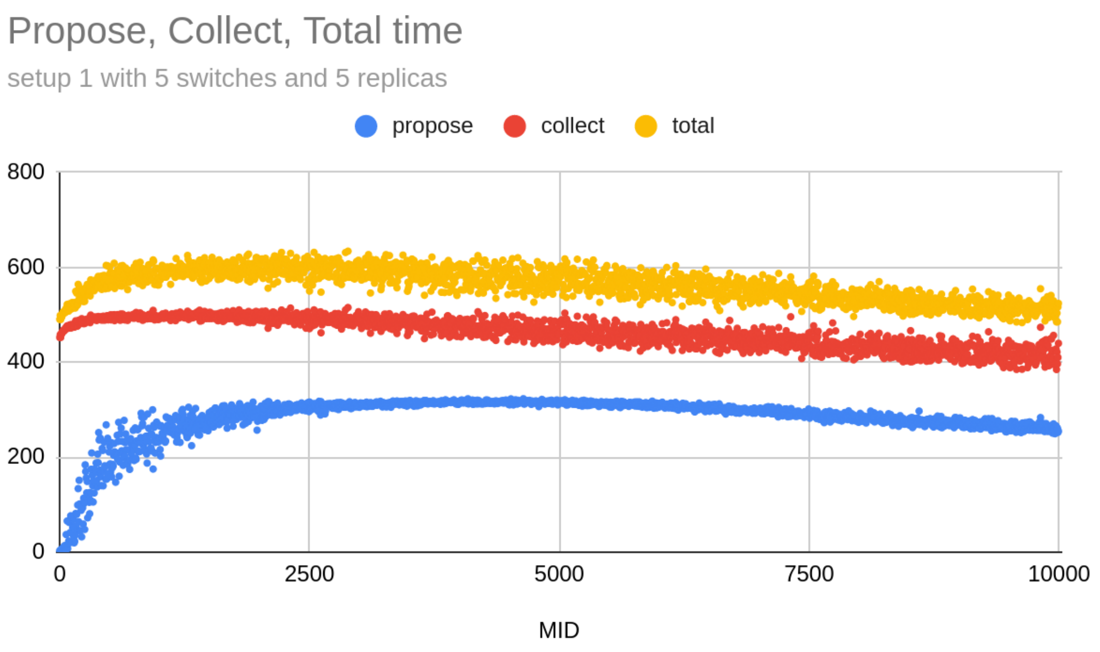


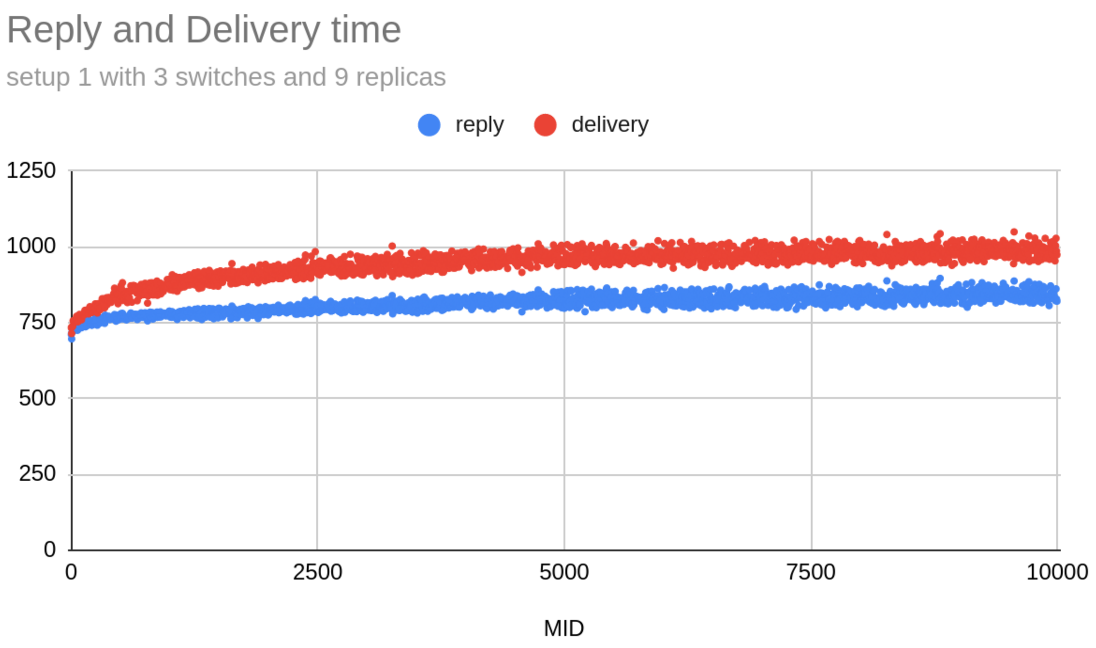
Propose, Collect, Total time

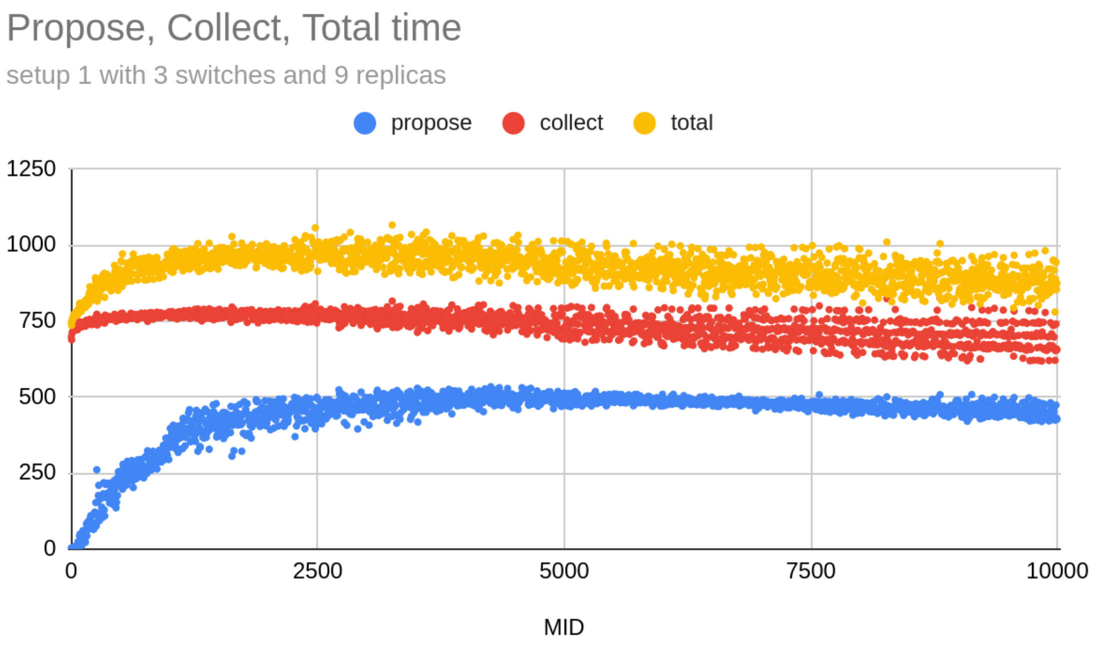
setup 1 with 9 switches and 5 replicas

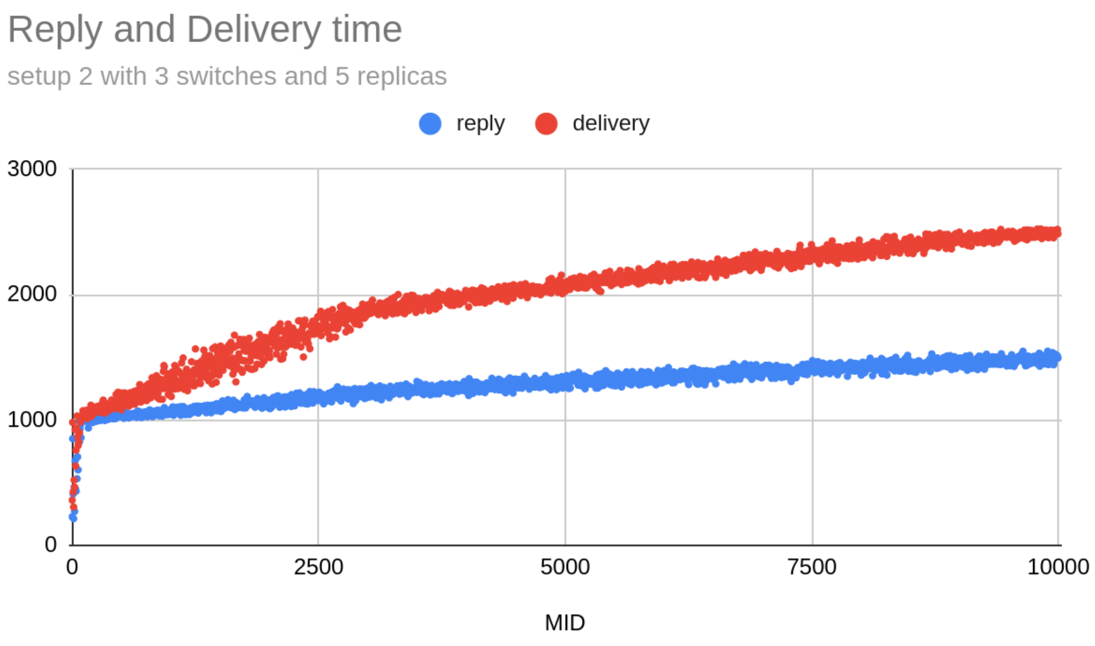


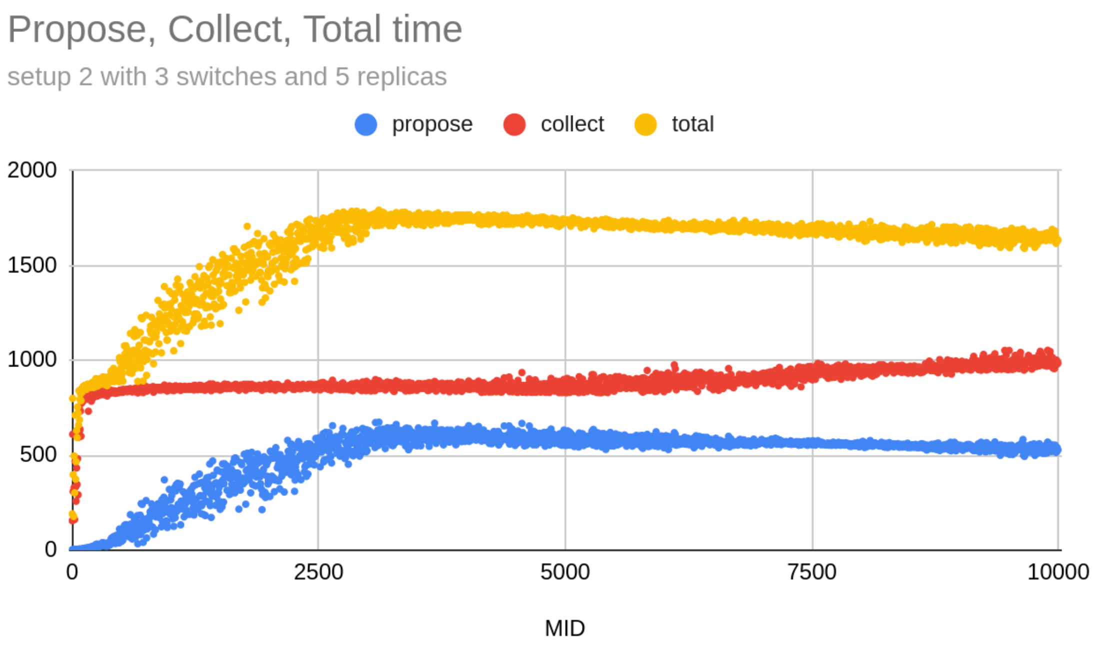

Reply and Delivery time
setup 1 with 5 switches and 5 replicas
reply time
delivery time
800
600
400
200
0
0
2500
5000
7500
10000
MID

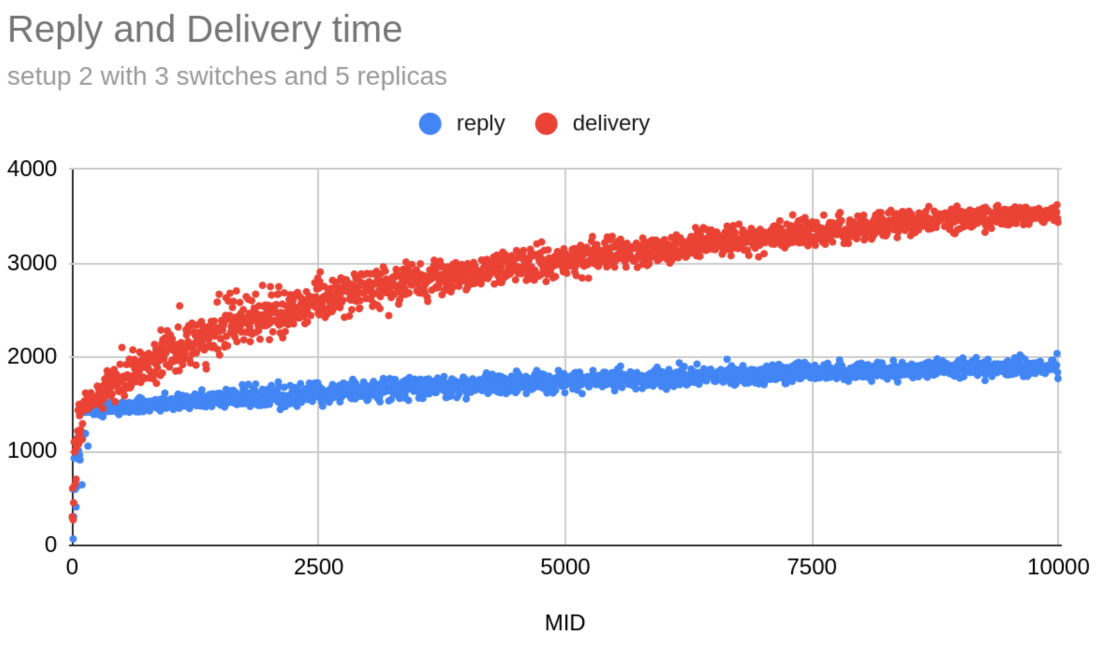

Propose, Collect, Total time
setup 1 with 5 switches and 5 replicas
propose
collect
total
800
600
400
200
0
0
2500
5000
7500
10000
MID

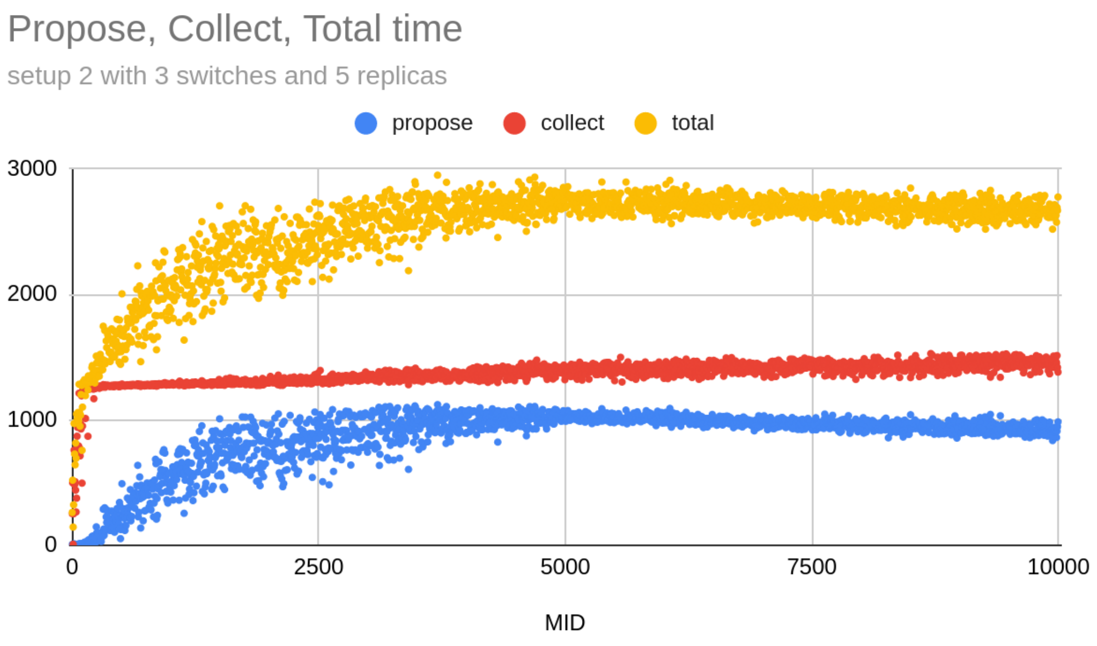

Reply and Delivery time
setup 1 with 3 switches and 9 replicas
reply
delivery
1250
1000
750
500
250
0
0
2500
5000
7500
10000
MID

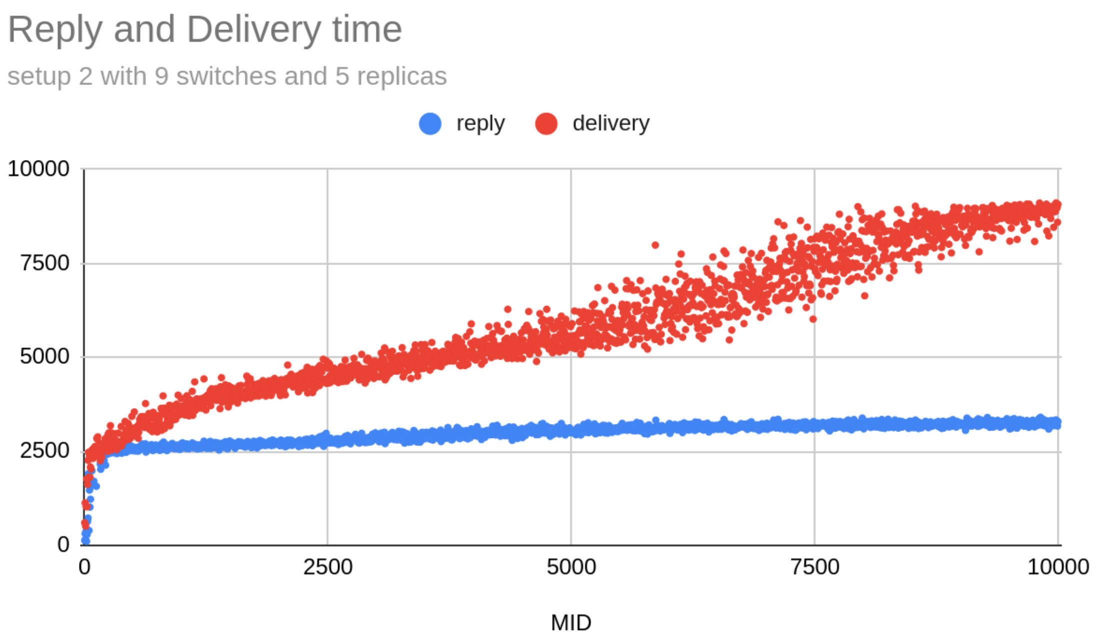

Propose, Collect, Total time
setup 1 with 3 switches and 9 replicas
propose
collect
total
1250
1000
750
500
250
0
0
2500
5000
7500
10000
MID

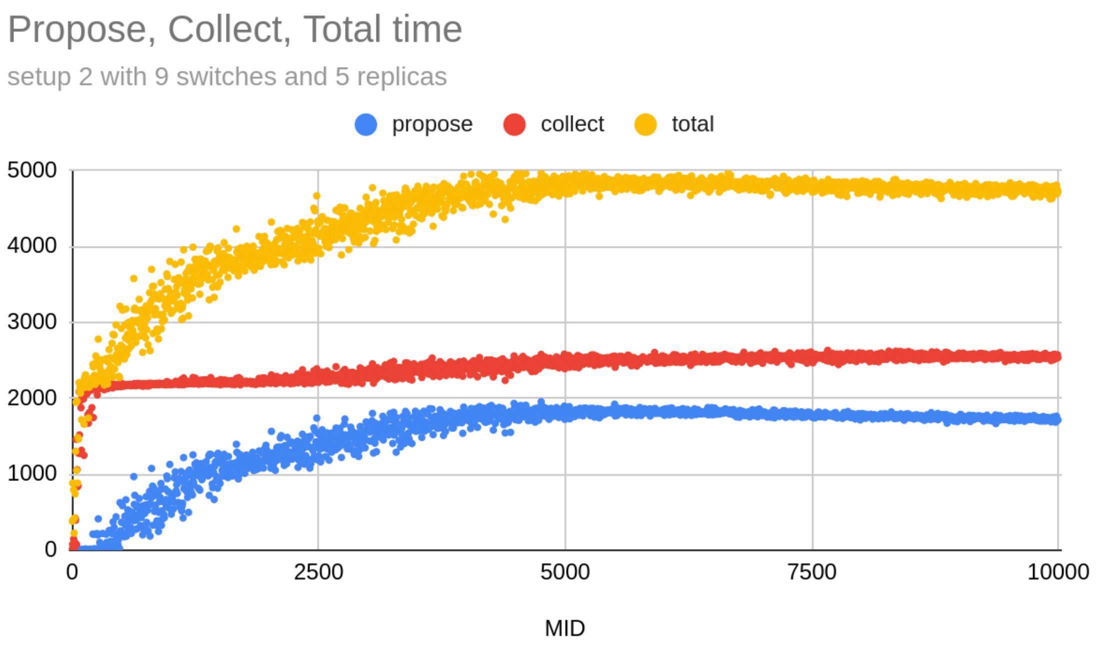

Reply and Delivery time
setup 2 with 3 switches and 5 replicas
reply
delivery
3000
2000
1000
0
0
2500
5000
7500
10000
MID

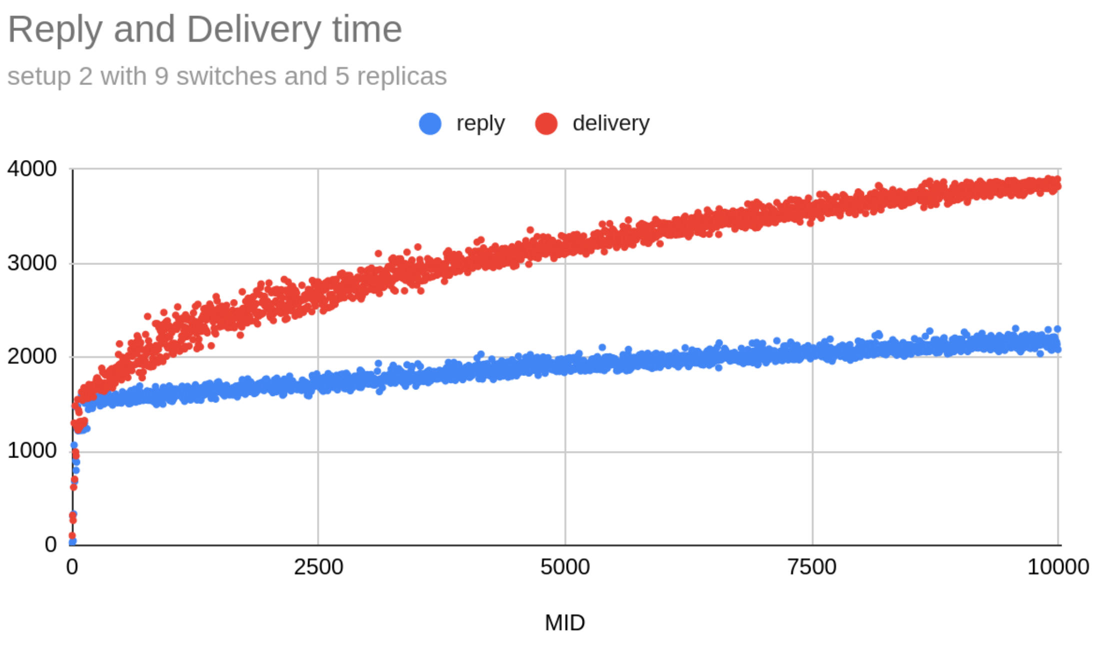

Propose, Collect, Total time
setup 2 with 3 switches and 5 replicas
propose
collect
total
2000
1500
1000
500
0
0
2500
5000
7500
10000
MID

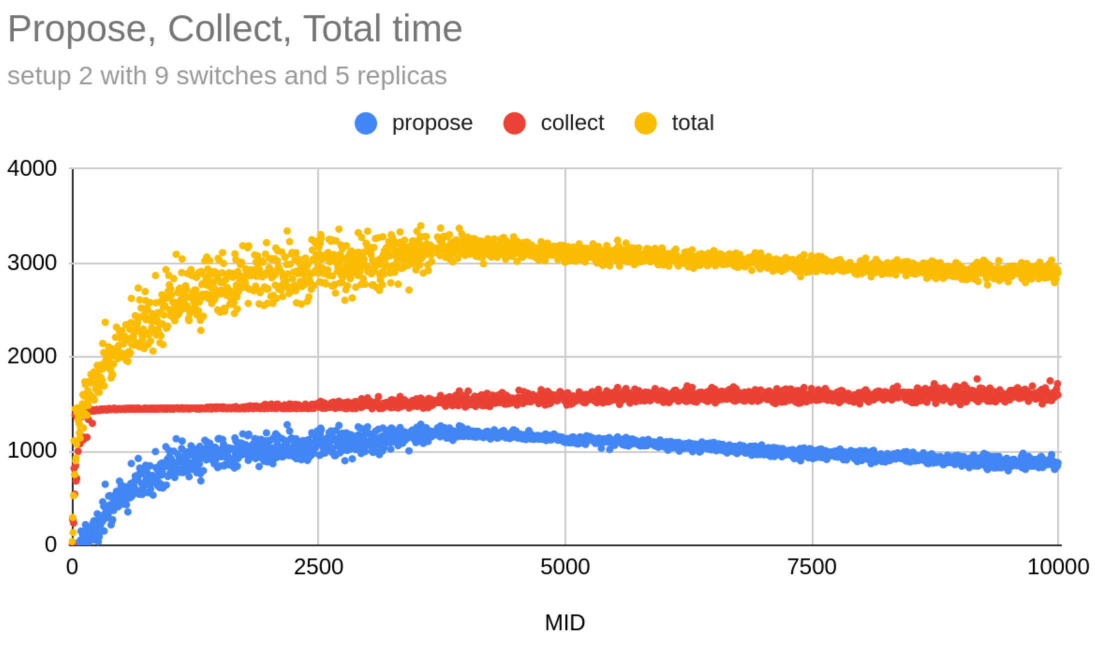

Reply and Delivery time
setup 2 with 3 switches and 5 replicas
reply
delivery
4000
3000
2000
1000
0
0
2500
5000
7500
10000
MID


Propose, Collect, Total time
setup 2 with 3 switches and 5 replicas
propose
collect
total
3000
2000
1000
0
0
2500
5000
7500
10000
MID


Reply and Delivery time
setup 2 with 9 switches and 5 replicas
reply
delivery
10000
7500
5000
2500
0
0
2500
5000
7500
10000
MID


Propose, Collect, Total time
setup 2 with 9 switches and 5 replicas
propose
collect
total
5000
4000
3000
2000
1000
0
0
2500
5000
7500
10000
MID


Reply and Delivery time
setup 2 with 9 switches and 5 replicas
reply
delivery
4000
3000
2000
1000
0
0
2500
5000
7500
10000
MID


Propose, Collect, Total time
setup 2 with 9 switches and 5 replicas
propose
collect
total
4000
3000
2000
1000
0
0
2500
5000
7500
10000
MID