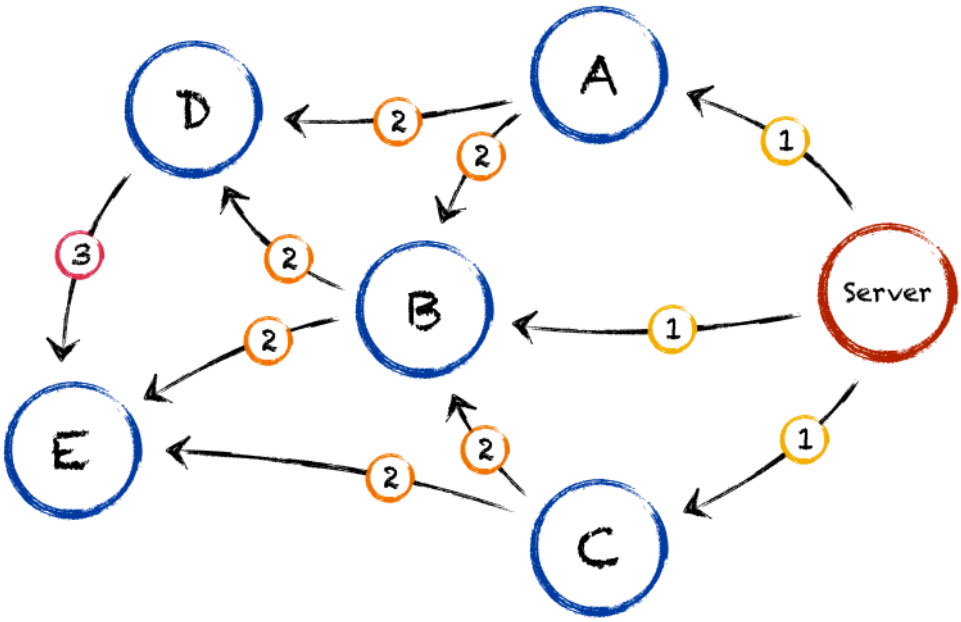
LOGICAL TIME SYNCHRONIZATION IN DISTRIBUTED NETWORKS WITH VOLATILE LATENCY

Boston College Computer Science Senior Thesis



Matthew Ricketson
Class of 2013

TABLE OF CONTENTS

LOGICAL TIME SYNCHRONIZATION IN DISTRIBUTED NETWORKS WITH VOLATILE LATENCY

Boston College Computer Science Senior Thesis

Matthew Ricketson
Class of 2013

Abstract

The ability to accurately synchronize logical time between nodes in a network is important for various applications such as data collection, distributed robotics, and gaming. Most existing algorithms for time synchronization, however, are either only applicable in networks with low, consistent latency, or that are not distributed in nature. This is suitable for computers perpetually connected to WiFi or wired networks, but for mobile, embedded devices connected through volatile-latency networks such as cellular networks and that often communicate in a distributed manner, these algorithms are less than ideal.

This thesis explores a new method for time synchronization in these types of networks, called the "Follower" algorithm, which is shown to be effective at adapting to rapid, inconsistent changes in latency and to be ideal for distributed networks where nodes are disconnected and reconnected constantly. The algorithm is evaluated through extensive virtual network simulations that directly compare its effectiveness against Cristian's time synchronization algorithm, and is also implemented in a physical network of minimalist Arduino-based devices to show its potential benefit for reducing cost and complexity in distributed sensor networks.

The Problem

Logical time synchronization has many applications and has been researched extensibly for the past several decades. However, new technological developments are changing the nature and make-up of modern networks, chiefly mobile computing and the “internet of things”.

First, this paper focuses exclusively on logical time, not real time. Logical time is measured as an amount of ticks that has passed since a certain start time. Ticks can be any unit of time, such as seconds, microseconds, or even days (this paper uses seconds in most cases). What is most important to systems that use logical time is comparison and ordering. If a timestamp has a value of 357 ticks, then it is earlier than a timestamp of 432 ticks. When analyzing a set of timestamped data, logical time can help organize the data

into an ordering without needing to know what historical date that data corresponds to. In many applications, the ordering is more important than the absolute date, such as transaction processing and relative trend detection in arbitrary datasets. Not needing to care about the absolute date of an event is valuable, since it removes the need for a mechanism to convert "time elapsed", which can be measured by a processor's clock cycles, into a date like "August 3rd, 1985", which requires either pre-calibration or an external reference source.

In mobile networks, devices are constantly passing through different types of networks, such as WiFi and cellular, which have different characteristics. WiFi is fast and consistent, while cellular networks like 3G are often slow and inconsistent, experiencing higher latency. While many mobile devices have the ability to accurately measure real time using special hardware, such as GPS, sometimes the real time cannot be dependently accessed by applications, and so custom time synchronization is necessary.

For example, in 2011 I created an iOS game that had time-dependent features. Players quickly learned that by changing their phone's time in settings, they could cheat the game and get ahead of other players by effectively jumping into the future. Unfortunately, there was no way for our app to get access to a universal time value using built-in APIs that users could not affect. So, we had to implement our own logical time synchronization algorithm, and it had to be able to handle mobile phones moving in and out of WiFi and constantly disconnecting and reconnecting. That effort was a motivation for this more in-depth study.

Another trend is the rise in cheap, networkable electronics using platforms such as Arduino and Raspberry Pi combined with Bluetooth and local WiFi. When building custom hardware, every extra component adds to the size and cost of the device. Thus, it is expensive to add real-time clocks or other components capable of getting accurate time data, such as GPS. Thus, implementing a custom logical time synchronization method can allow different custom devices to accurately sync using only their processor clock cycles.

An example of an application for this would be a distributed network of sensors. Each sensor device could be minimally built with only a processor, sensor, battery, and networking component such as Bluetooth. This could be built cheaply and in a tiny form factor, allowing nodes to be mass produced and spread over a large area. The nodes could connect to any detected nearby nodes, forming a distributed network, and could sync their times together. When gathering sensor data from the network, data from different nodes could be reliably compared via their timestamps, allowing for time-dependent features to be uncovered in the data, such as object movement in a network of motion detectors or temperature waves in a network of thermometers.

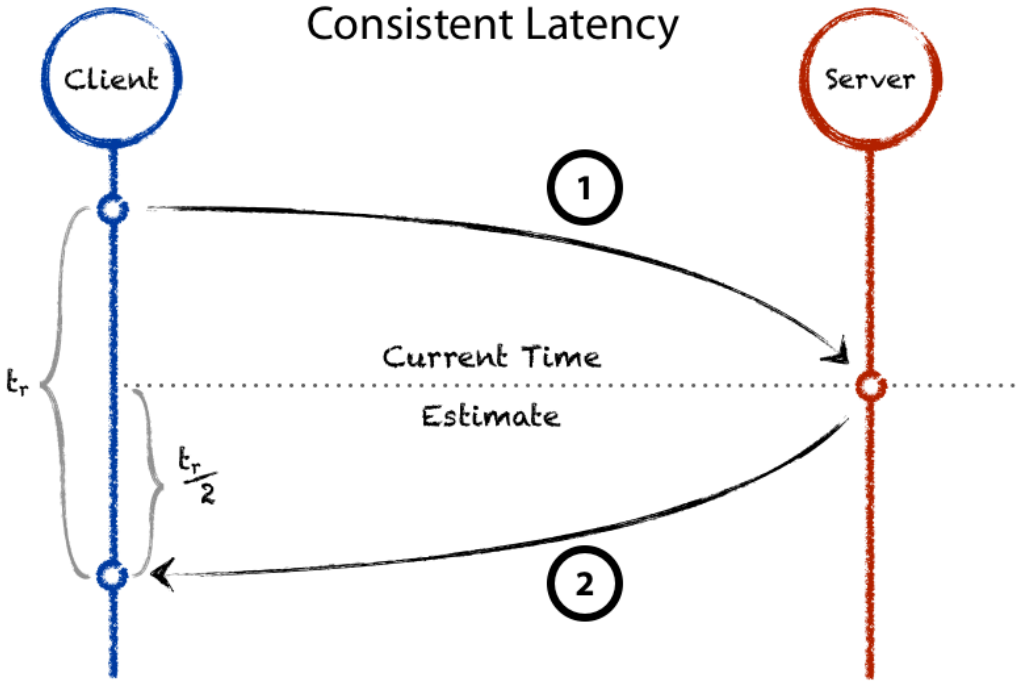
This paper explores different approaches for syncing logical time in network conditions similar to these types of applications, chiefly those with volatile latency and distributed configurations.

Previous Work

CRISTIAN'S ALGORITHM

Cristian's algorithm takes a straightforward approach: send a round-trip message from client to server, and adjust the received time by adding half of the round-trip duration in order to account for latency.

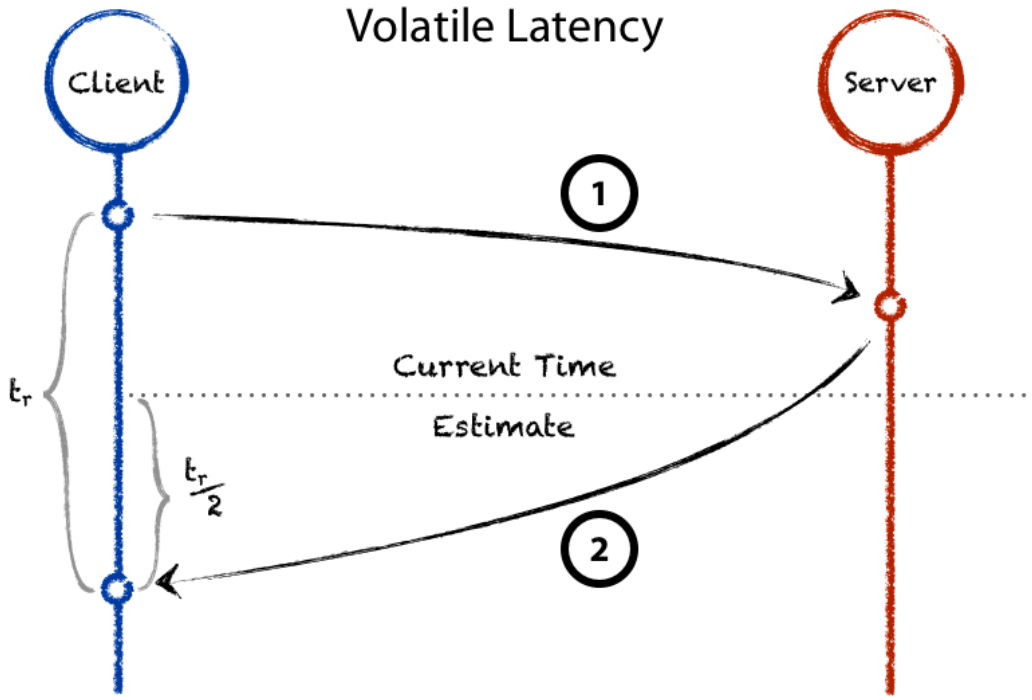
The core assumption of this algorithm is that latency is relatively low and consistent in the network. However, this estimate becomes more inaccurate as the consistency of network latency decreases, such as in a mobile network. This destabilizing effect can be partially compensated for if you perform the algorithm multiple times and use the mean value of the received times. However, higher volatility within the network inevitably leads to greater instability and less accuracy in time estimates.



Cristian's Algorithm assumes latency is relatively consistent, so that dividing the total round-trip message time, t_r, in half will result in an accurate estimate of network latency.

In Cristian's 1989 paper describing the algorithm, significant time is spent discussing a method for implementing a "distributed" time service. However, Cristian's approach assumes a continuously available master time server connected to a hierarchical system of slave servers and slave clients. To counteract master server failures or disconnects, it proposes systems of multiple redundant servers. A truly distributable algorithm, however, would not depend on a set of master servers but rather allow any node to be a server or a

client, or both, making it possible to form a synced network with any arbitrary collection of nodes.



Higher latency volatility leads to less accuracy in Cristian's algorithm, since if message travel times are no longer consistent, dividing the round-trip time in half is no longer as effective. This can be compensated for by averaging several updates together over time.

In summary, Cristian's algorithm has the following advantages:

1. **Very accurate in certain conditions:** In networks with low, consistent latency, Cristian's algorithm works exceedingly well.
2. **Simple to implement:** The protocol can be implemented in very view lines of code, which is important when it comes to embedded devices.
3. **Fast:** There is little processing involved to compute an estimate of the server time.

However, Cristian's algorithm also has these downsides:

1. **It's a two-way protocol:** A client must send a message to the server and get a response for an update to take place.[1]

1 A one-way protocol (where the server sends the client a time update without a prior request) would be much more preferable to a two-way protocol because then the time update can be embedded within other messages. For example, whenever the server sends other data to a client, it can include a time update as well. This reduces bandwidth costs by making it unnecessary to make extra network calls just to update the time. This is especially useful for small, low-power devices, for which every extra call infringes on battery life.

2. **It does not scale well:** Every new client requires a direct connection to the time server.

3. **Accuracy declines with increased latency volatility:** As previously described, increased volatility in the network has a negative effect of estimate accuracy.

4. **It is not naturally distributed:** All clients are directly connected to the server, and not to each other. If the time server fails or disconnects, there is no mechanism for choosing a new server.

An Alternative Solution

The challenge is to create a new synchronization algorithm that retains most of the advantages of Cristian's algorithm but is not plagued by its disadvantages.

This paper describes two related approaches. First, the "Follower" algorithm takes a naive approach to time synchronization that is not ideal, but that is fundamentally different than Cristian's algorithm and has different strengths and weaknesses.

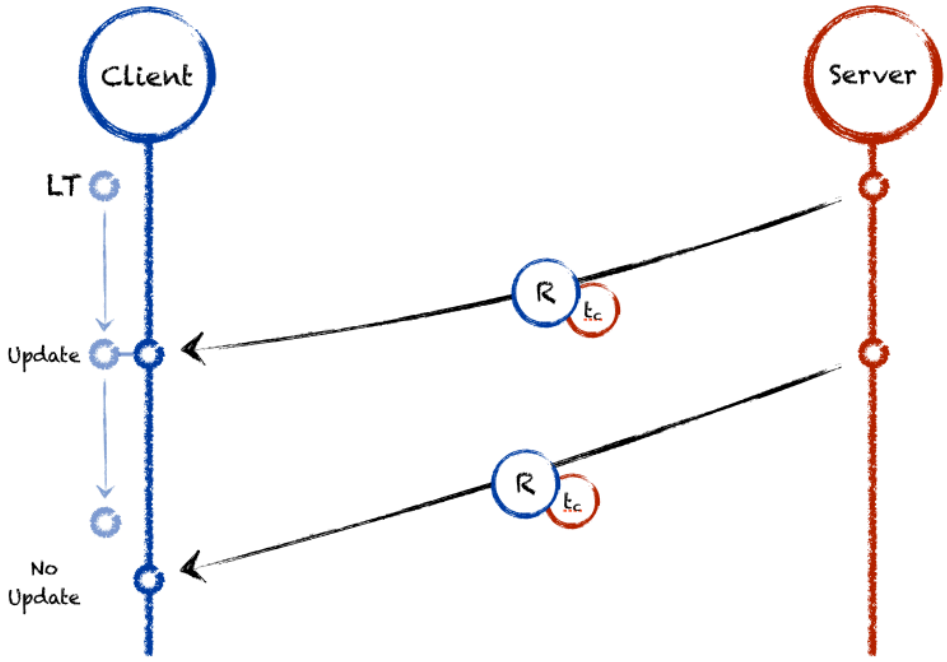
Second, the "Distributed Follower" algorithm attempts to combine the best aspects of Cristian's algorithm and the Follower algorithm into a single method, producing a technique that includes most of the advantages of each without many of their weaknesses.

FOLLOWER ALGORITHM

The basic Follower algorithm takes a simplistic approach to estimating time. A time server sends messages containing the current time to all connected clients. When a time update is received, a client either rejects the update or replaces its local time with the new value. An update is rejected if its time value is behind the local time, such that accepted time updates only ever cause the local time to jump ahead.

The core assumption of this algorithm is that any time update received from "the future" must be due to a message traveling faster than all previous messages have, or in other words, that experienced the least amount of latency during travel from server to client. This mechanism ensures that local time only ever improves, and is always equal to the server time plus the least amount of latency ever experienced along that edge in the network.

The result of this algorithm is that clients' perceptions of the server time tend to "stabilize" over time, because as more updates are received, it becomes increasingly less likely that a message will arrive faster than all previous messages. Having a mechanism for rejecting time updates known to be inaccurate also reduces the negative influences of volatile network latency: if there is a large increase in latency, clients simply ignore updates received during that time and retain previous gains in accuracy.



In this diagram, the first time update sent by the server is accepted by the client, which replaces its local time with the received value. The second update by the server, however, takes longer to arrive than the first (higher latency), and so is rejected.

The main advantages of the Follower algorithm are:

1. **It's a one-way protocol:** A server can send a time update to clients without needing a prior request message.

2. **Very accurate in certain conditions:** In networks that sometimes experience low latency, the Follower algorithm works well. It only takes one fast message to drastically improve a client's time perception.

3. **Simple to implement:** The protocol can be implemented in very view lines of code, which is important when it comes to embedded devices.

4. **Fast:** There is little processing involved to compute an estimate of the server time.

5. **Clients can never jump ahead of the server:** No client will estimate a time further into the future than the server, since each client time is equal to the sum of the server time and update latency, which is never zero or negative.

6. **Client times will only ever improve with time:** A client will never fall further behind the server, because there is no estimating involved.[2]

2 Unlike Cristian's algorithm, which does not have its own "stabilizing" behavior because it involves calculating an estimate of the latency in the network, which can cause either an overestimation or an underestimation of the server time.

7. **Strong resistance to network volatility:** Erratic increases in network latency are easily ignored.

The main disadvantages of this algorithm are:

1. **No client parity:** Clients may experience widely different latencies when receiving time updates, and so might have widely different estimates of the server time.

2. **No client-server parity:** Client times will never be close to equal to the server time, and this gets worse in networks with higher latency.

3. **It does not scale well:** Every client must get updates directly from the server.[3]

4. **It is not naturally distributed:** All clients receive updates directly from the server, and not from each other. If the time server disconnects, there is no mechanism for choosing a new server.

DISTRIBUTED FOLLOWER ALGORITHM

The Distributed Follower algorithm attempts to combine the best aspects of both Cristian's algorithm and the basic Follower algorithm in order to produce a better solution for logical time synchronization in networks with volatile latency.

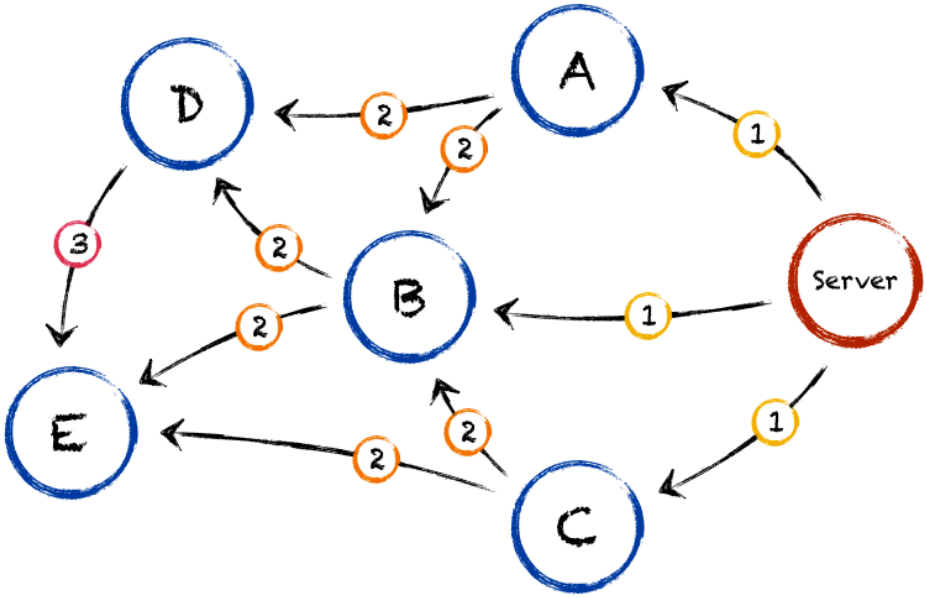
What is missing from the basic Follower algorithm but that Cristian's algorithm provides is a mechanism for estimating latency in the network. If a trustworthy latency estimate can be calculated, then the client can add this estimate to their own raw estimate (received from server updates) to attempt to predict the server's actual time. However, Cristian's algorithm requires a two-way protocol to accomplish this task, while the basic Follower algorithm benefits from a one-way protocol. The goal, then, is to find an algorithm that is both one-way and capable of making a trustworthy estimate of network latency, and this is what the Distributed Follower algorithm attempts. In addition, the new algorithm, as its name would imply, is designed to be easily distributable, resolving an issue with both the basic Follower and Cristian's algorithms.

To understand the Distributed Follower approach, first consider a hypothetical network of connected nodes. In this network, the latency for each connection is constant and equal to the same value across the network. We will refer to this universal latency value as L. When a time server updates its clients in this network, all clients will receive time values that are delayed by L.

Now suppose that each client, upon receiving a time update, resends this update to a certain set of other nodes (its listeners). When those nodes receive the message, the time

3 If the Follower algorithm implemented a forwarding scheme in which updates from the server were resent to other nodes, then the increased latency that comes from making multiple hops in the network would drastically reduce the effectiveness of the algorithm.

value it contains will have experienced two “hops” in the network, and will thus be delayed by 2*L relative to the original time server. If the update were re-broadcasted again in similar fashion, the next set of nodes would receive a time value delayed by 3*L, and so on.



In the Distributed Follower algorithm, time updates from the server do not die when they reach their destination, but are forwarded to other nodes. Each time a message is sent, its hop count increases. The above diagram shows message paths and hop counts. Notice that nodes D and E are not directly connected to the server, but are still able to participate in the network.

If connections between nodes are not arranged hierarchically, such that a node could receive messages from other nodes that have gone through different numbers of hops, then we see an easy way of calculating the latency in the network. Simply find the difference between the time values of two updates that have experienced different numbers of hops, and divide the result by the difference in hop count:

$$L = (t_a - t_b) / (h_a - h_b)$$

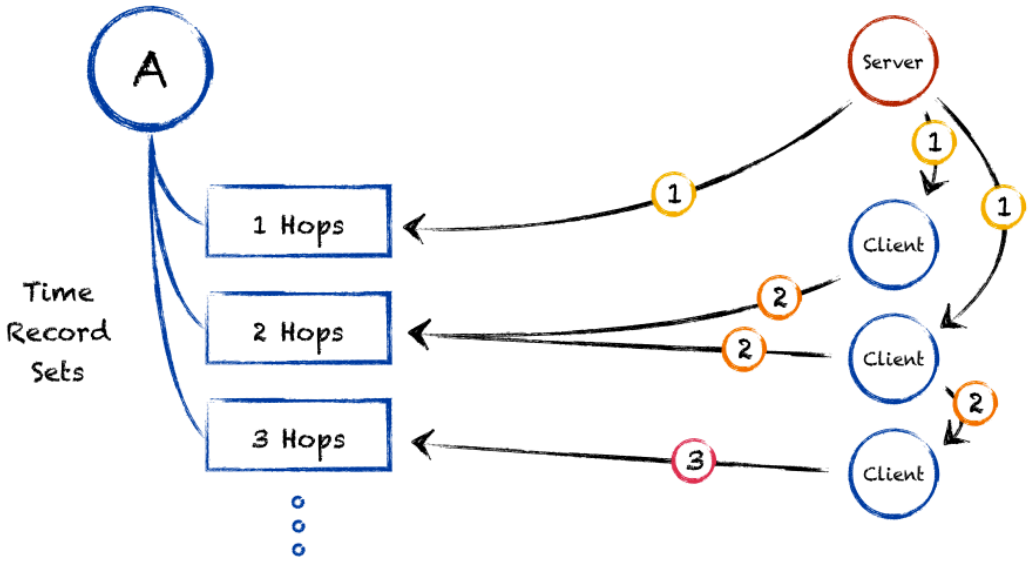
Where t_a, t_b are the adjusted time values in messages “a” and “b”, respectively, and h_a, h_b are the hop counts for each message. “Adjusted” time value means that the time value is equal to the sum of the value that was contained in the message and the time that has passed since the message was received. By using this calculated latency estimate, each node can effectively compute the exact time on the server, even if that node is not directly connected to the server.

This method breaks down when latency values vary across connections, however, and especially if latency values are constantly changing in a volatile way. So, to compensate for this, the Distributed Follower algorithm employs a modified version of this strategy.

Each node in the Distributed Follower algorithm keeps track of time updates received by sorting them into “bins”, called time record sets, that are separated by hop count. When estimating latency, each node calculates the difference between the mean time values of each consecutive time record set, and then computes the mean of all the differences. This use of averaging smooths out the data set to produce a latency estimate that is less susceptible to fluctuations in the network. The node then comutes an estimate of the server time by using the formula

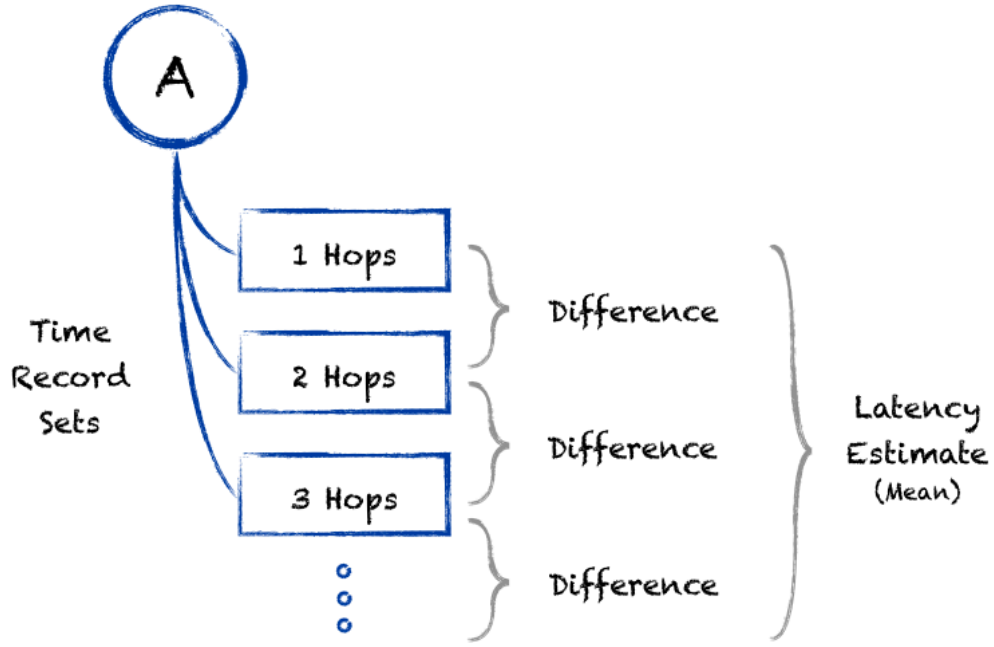
Server Time = (Time Record Set X Mean Time Value) + (X+1)*(Latency Estimate)

where X is the index (or hop count) of a time record set. (As such, once a latency estimate is obtained, there are as many ways to estimate server time as there are time record sets. Simulation Experiment #1 in the Results section explores the problem of what time record should be chosen as the “base” to be used in the above formula.)



In the Distributed Follower algorithm, when a node receives a time update, whether from the server or another client, it is stored in a bin of other updates, or a time record set, according to its hop count.

Various additional rules can be employed in order to try and improve the latency estimate, such as removing time values from time record sets after a certain duration (to compensate for changes in network behavior over time), or not recording extreme values that are most likely resulting from network anomalies. Fully measuring the effect of such rules is a subject for future work.



In the Distributed Follower algorithm, latency estimates are calculated by taking the mean of the differences of the means of the time record sets for a node. Yes, that's a mouthful! This process helps smooth out differences in latency that comes with volatile network communication.

This algorithm is considered to be distributed because of the interconnectedness of the nodes. Clients are not solely connected to a server, but are also in contact with each other, such that if a server becomes disconnected that the network can detect this and easily coordinate the choosing of a new server. Also, just like in the previous algorithms, clients can disconnect or reconnect arbitrarily, but unlike those algorithms, clients do not need to be directly connected to the server in order to estimate an accurate time.

The main advantages of the Distributed Follower algorithm are:

1. **It's a one-way protocol:** Same benefit as the basic Follower algorithm.

2. **Better client-server parity:** Adding a latency estimate, like in Cristian's algorithm, gets closer results than the basic Follower algorithm.

3. **Better client-client parity:** Since clients share information with each other, there is inevitably a smoothing effect across the network. This is an improvement above both Cristian's and the basic Follower algorithms.

4. **It improves, not worsens, with scale:** The more nodes in the network, the more sharing of messages. The more sharing, the more data to help improve the latency estimate. This is an improvement above both Cristian's and the basic Follower algorithms.

5. **Adaptable to distributed networks:** No lasting damage or side-effects are introduced if a server is disconnected from the network. Any client can turn into a replacement server at any time and the network will adapt. This is an improvement above both Cristian's and the basic Follower algorithms.

6. **Adaptable to changes in overall network latency behavior:** If for some reason the mean latency in the network changes over time, clients can smoothly adapt by discarding old time values from their bins.

7. **Works well in both low and high latency volatility:** the lower and more consistent the latency in the network, the better the estimate is for the same reasons as Cristian's algorithm. However, this algorithm deteriorates in high-volatility at a reduced rate than Cristian's.

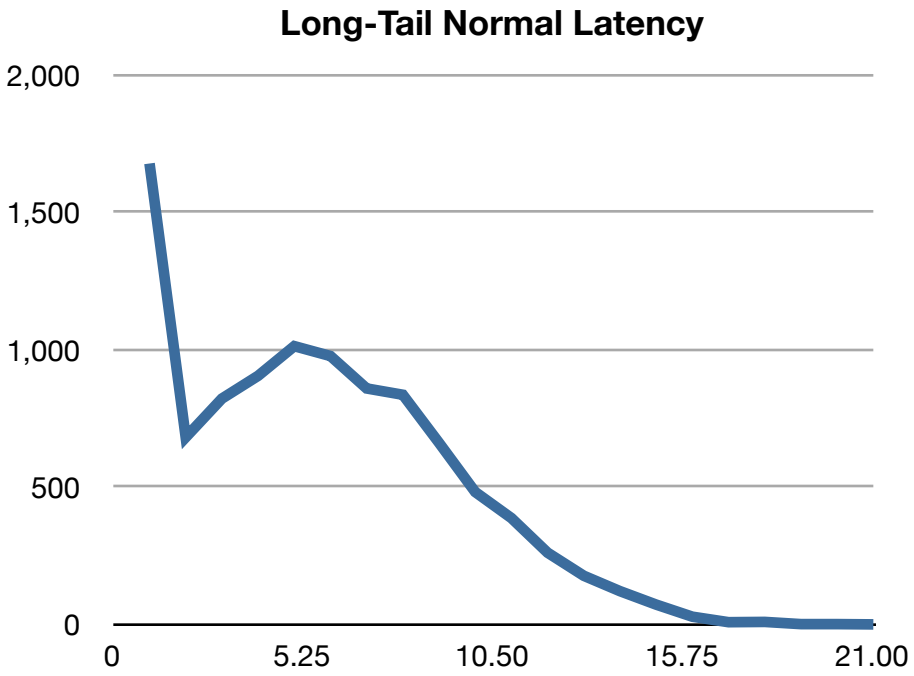
Simulation Results

IMPLEMENTATION

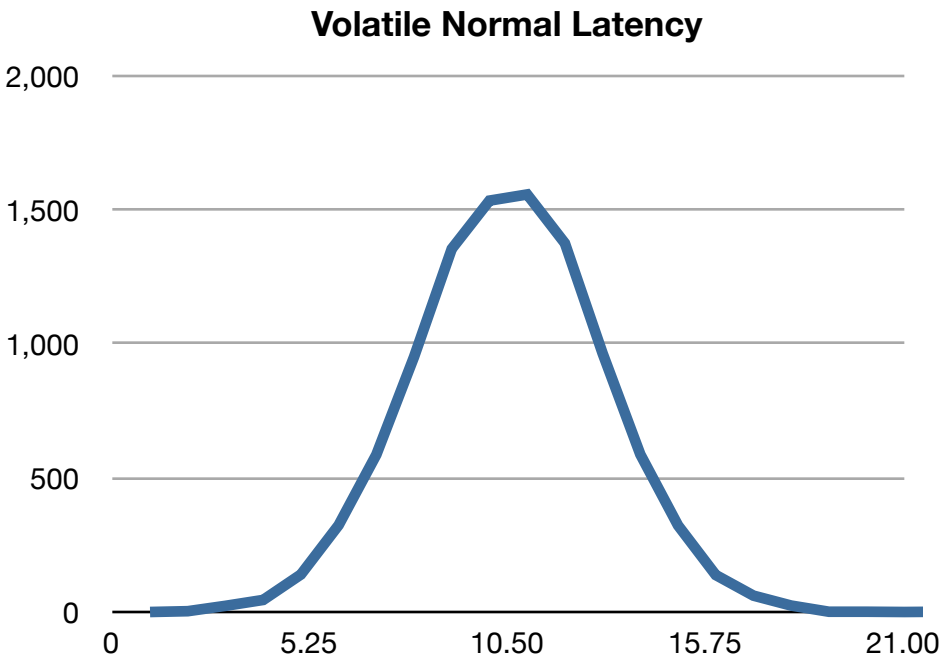
To test the three different algorithms described previously (Cristian's, Follower, Distributed Follower), a program was created that could simulate network communication among a group of nodes.

A key aspect of the program is its ability to simulate different models of latency. In our tests, three latency models were used (charts show distribution of latency values across 10,000 messages, according to each model):

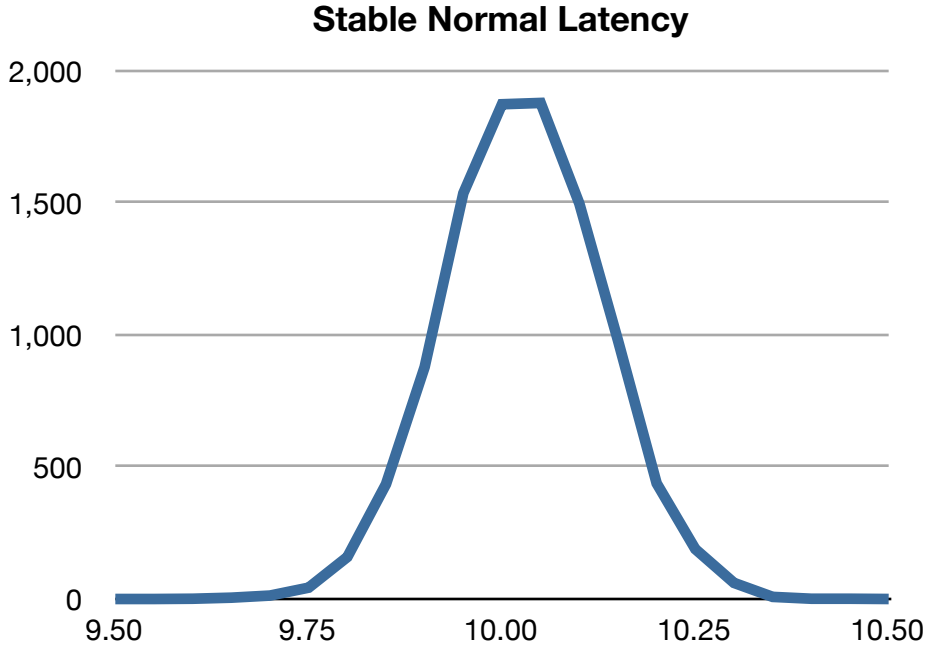
1. **"Long-Tail" Normal Latency:** Most values are consistently low, near zero, but there is a "long-tail" of higher values. This model is meant to resemble a mobile network, where devices may be moving from fast, stable networks like WiFi into slower, volatile networks like cellular and vice versa constantly.



2. **Volatile Normal Latency:** A typical normal distribution, but with a large standard deviation. This type of network has an average latency of 10 seconds, but has large swings in value.



3. **Stable Normal Latency:** A typical normal distribution, but with a very small standard deviation for consistency. This model also has an average of 10 seconds, but varies much less compared to the volatile model. This model is designed to be easy for nodes to make estimations in, and could represent a fast WiFi or wired network connection.



The simulation program implements three main classes for nodes in the network: a client running Cristian's algorithm, a server running Cristian's algorithm, and a node capable of being both a client and server for both the Follower and Distributed Follower algorithms.

During each simulation, whichever node is chosen to act as the server sends periodic time updates to all of the other nodes, which act as clients.

The program is coded in Objective-C as a Mac application that is run in a terminal window. The main portions of the code can be found in Appendix B.

After the completion of a successful simulation, text output is generated that can be easily imported into a spreadsheet for analysis. Multiple simulations can be run serially, making it simple to test multiple different conditions. Simulation output is generated by polling all the nodes in the network at constant intervals and recording their current time values. Node classes are implemented as thread-safe where necessary so that polling and recording of data can be done on a separate thread. Once the simulation finishes, all the collected data is outputted at once.

All graphs of results, unless otherwise indicated, measure the difference between client time and server time on the Y-axis (with a value of 0 indicating perfect sync between client and server), and simulation time on the X-axis.

(Note: for all experiments, the shorthand CR, FO, and EF refer to nodes using Cristian's algorithm, the basic Follower algorithm, and the Distributed Follower algorithm, respectively. These algorithms are also normally represented by the colors blue, green, and red, respectively.)

RESULTS

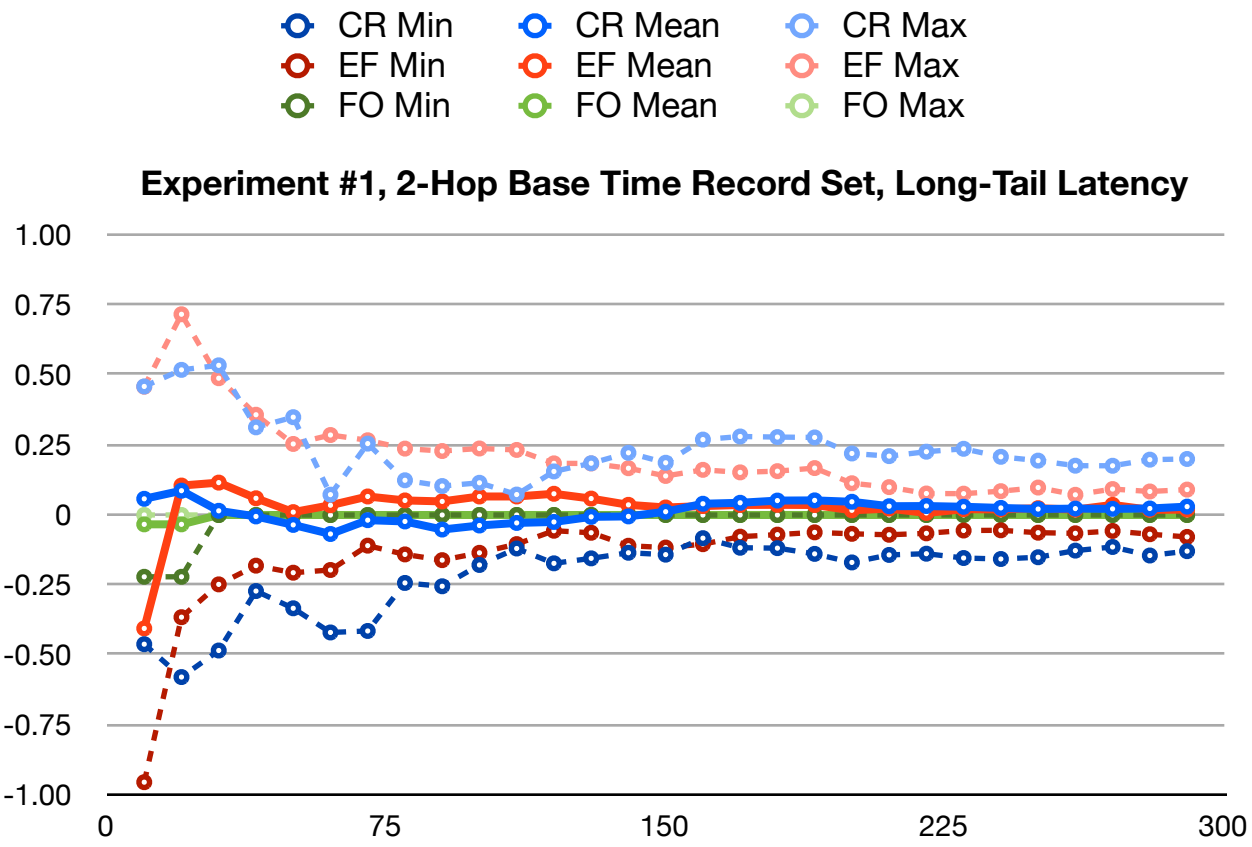
Three main experiments were performed, each compromised of many simulations targeting different variations of a specific condition. These sections describe each experiment, including its purpose, conditions, and important highlights from the results. The complete results for each experiment can be found in the Appendix.

EXPERIMENT #1: BASE TIME RECORD SET

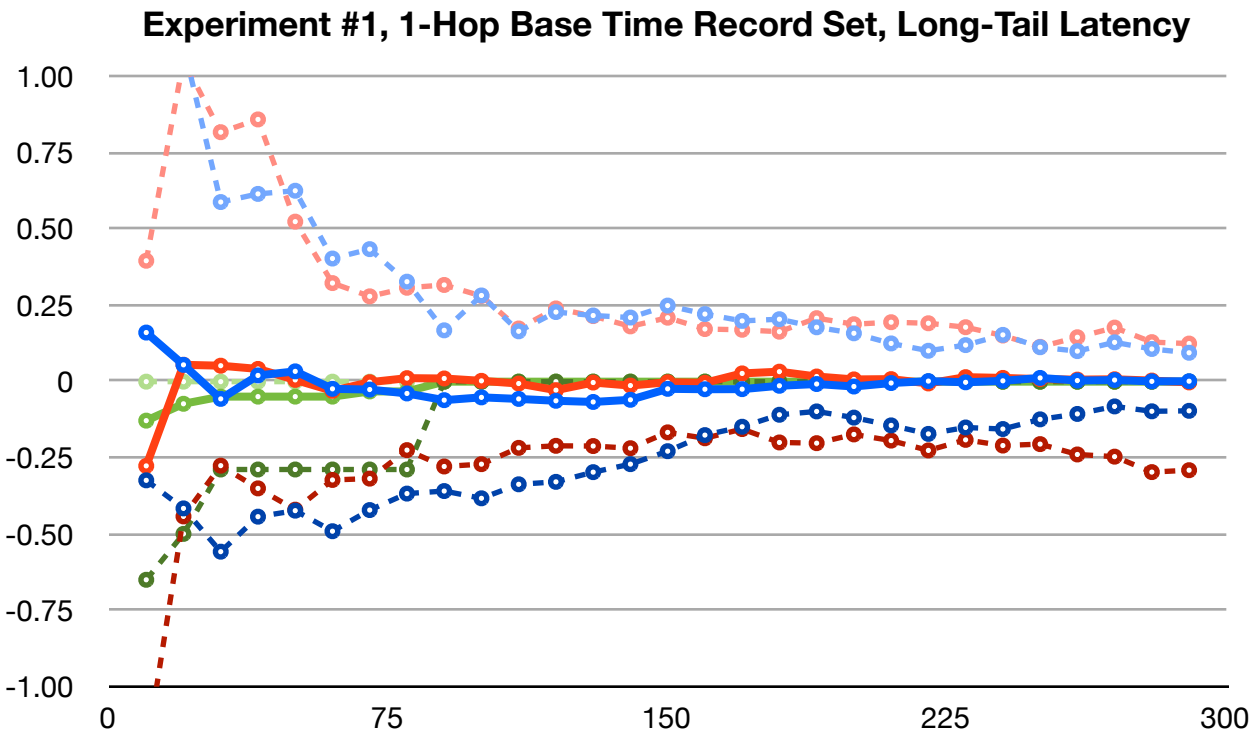
The purpose of this experiment was twofold. First, to server as a general comparison between the three algorithms across the different latency model conditions. Second, to identify the correct time record set to use in the Distributed Follower algorithm to get the best estimate of the server time.

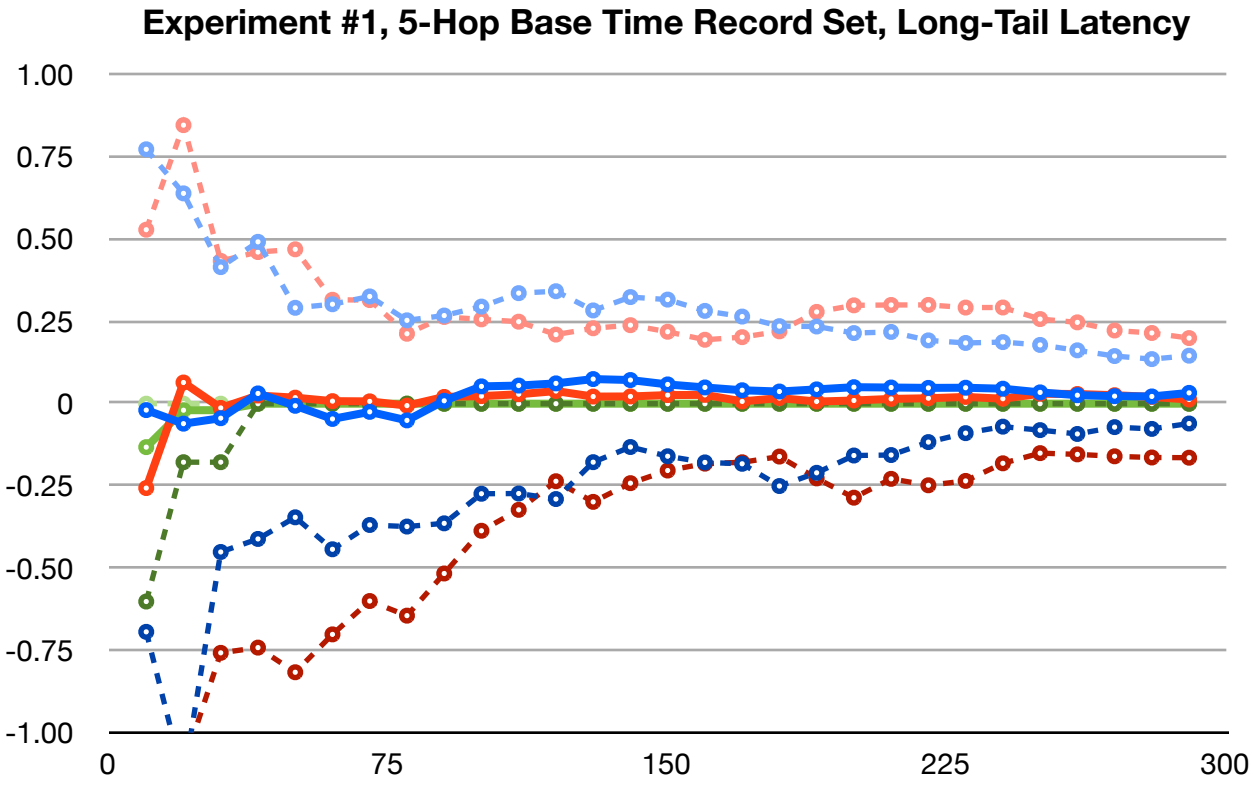
In this experiment, fifteen simulations were run. Each node that ran the Distributed Follower algorithm had five time record sets, and so a different simulation was run for each set chosen, across all three latency models.

There are a a number of key highlights from the results of the experiment. First, it is clear that using the 2-hop time record set produced the best results. Here is the graph of the results using the 2-hop time record set using a long-tail latency model:



Compare this with the graphs of the simulations that had the same conditions, but used 1-hop and 5-hop time record sets, respectively:


Experiment #1, 1-Hop Base Time Record Set, Long-Tail Latency
1.00
0.75
0.50
0.25
0
-0.25
-0.50
-0.75
-1.00
0
75
150
225
300


Experiment #1, 5-Hop Base Time Record Set, Long-Tail Latency
1.00
0.75
0.50
0.25
0
-0.25
-0.50
-0.75
-1.00
0
75
150
225
300

As you can see, using the 2-hop base produced much better results for the Distributed Follower algorithm when compared to any other base.

It makes sense that a 2-hop base would perform better than a 1-hop base. If a message arrives after only one hop, then that means it came directly from the time server, and so that time record set contains updates that only traveled across one specific edge in the network. The next time record set, however, contains many more updates from multiple different sources, since there is a level of indirection introduced in messages with two hops. Calculating the server time estimate from data collected from multiple sources helps protect against volatile latency by averaging the data together, whereas no such protections are in place when only considering direct updates.

This effect also explains why the Distributed Follower algorithm performs better using a two-hop base than Cristian's algorithm, since Cristian's algorithm only takes into account direct updates from the server.

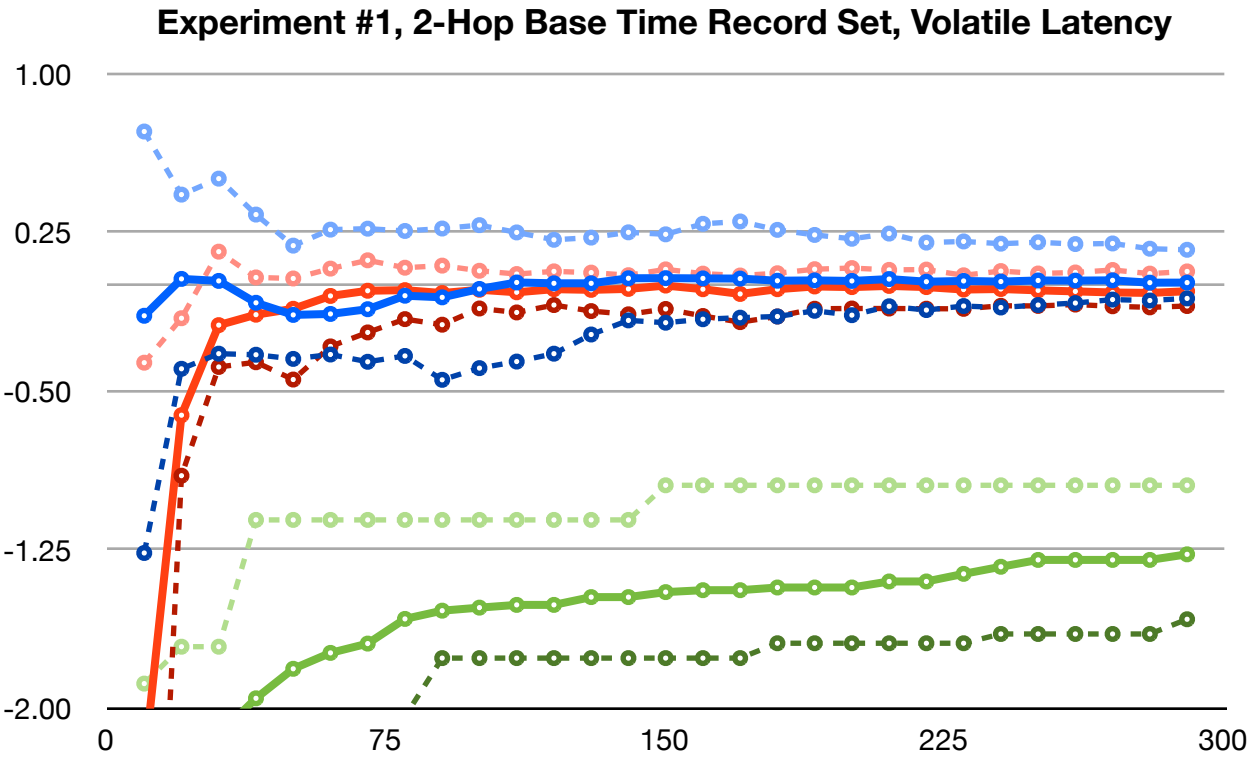
What about using time record sets with higher hop counts as the base? While these other time record sets also collect data from multiple sources, each additional hop increases the error in the server time calculation. If you recall the formula:

Server Time = (Time Record Set X Mean Time Value) + (X+1)*(Latency Estimate)

As *X* increases, any error present in the latency estimate is magnified. The Appendix shows graphs for all base records used. With each higher base time record set past 2, more error is introduced and the Distributed Follower algorithm performs worse, culminating in the poor performance shown above for the 5-hop base.

It is important to point out the basic Follower algorithm in the graphs above, since it performs far better than both the Distributed Follower algorithm and Cristian's algorithm. This is a result of the specific conditions of the long-tail latency model. In conditions where network communication can occasional reach high speeds and latency drops down to near zero, as is the case in this latency model, the basic Follower algorithm is extremely accurate. Instead of relying on averages an estimates, the Follower algorithm simply jumps close to zero as soon as a fast connection is established. Since the Follower algorithm never gets worse over time, this gain in accuracy is permanent, even if the speed increase happens for only one update.

Unfortunately, the Follower algorithm breaks down when the average latency is high, such as in the other two latency models that average at 10 seconds. Here are the results for volatile latency using a 2-hop base for the Distributed Follower algorithm:



Results for the stable latency model do not even show the Follower algorithm, since all of its nodes are so far below the other two algorithms (down near 10 seconds), that it would be impossible to compare all three on a single graph.

Due to the success of using a 2-hop base for the Distributed Follower algorithm in this experiment, all other experiments use a 2-hop base for Distributed Follower nodes.

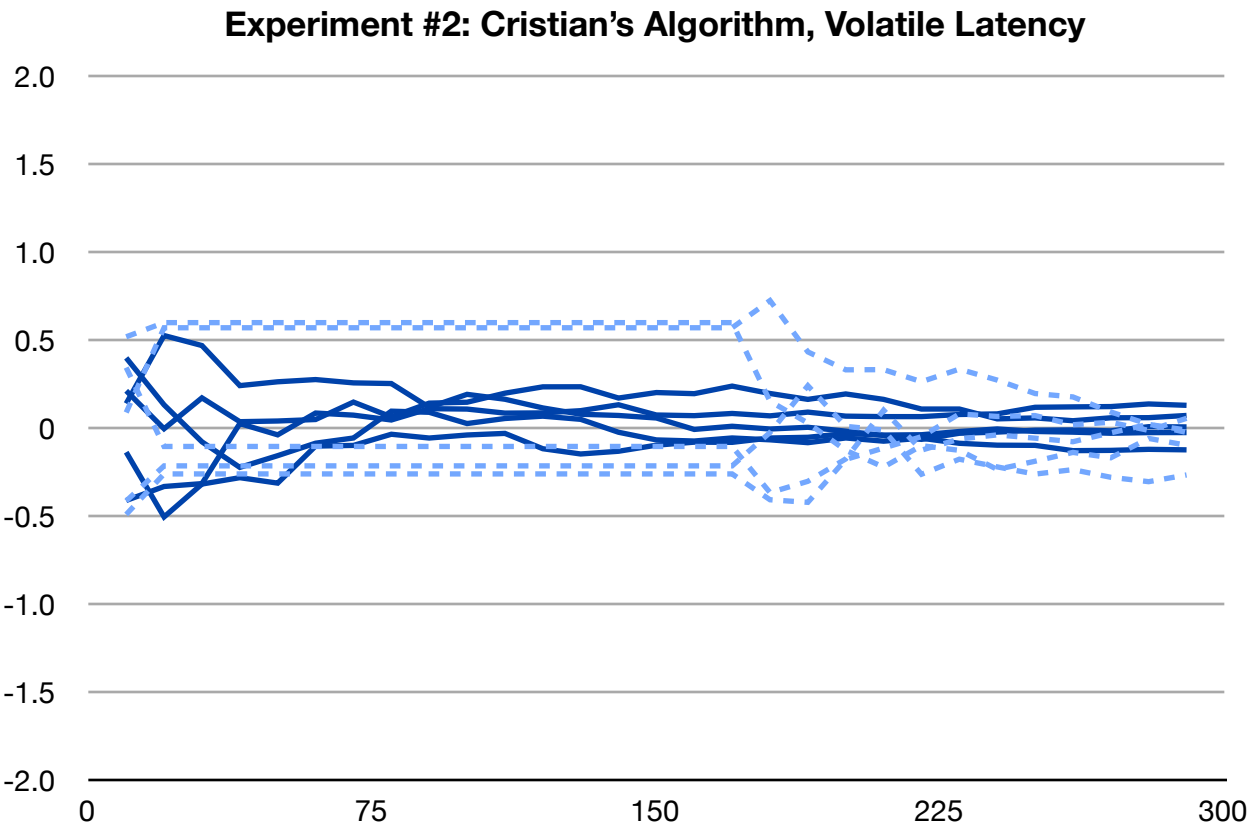
EXPERIMENT #2: CLIENT DISCONNECT

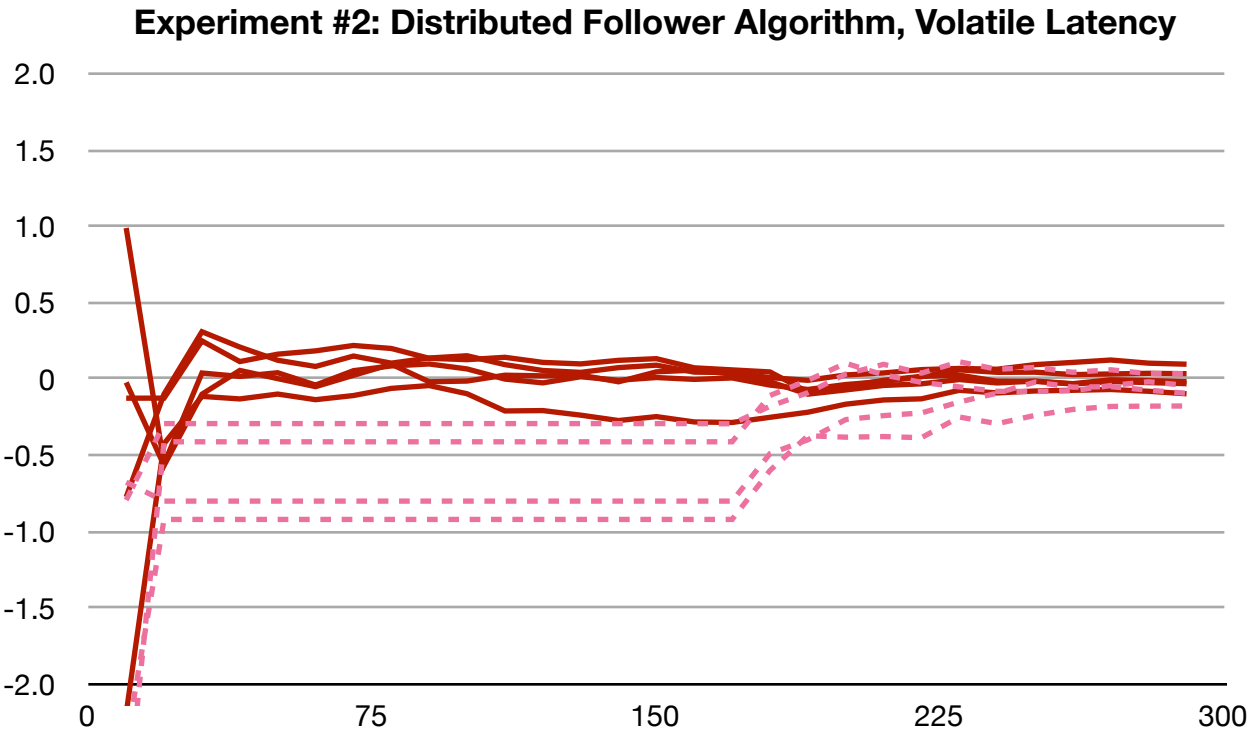
The purpose of the second experiment was to test whether clients can dynamically disconnect and reconnect to a network without significant or prolonged loss of synchronization. If clients cannot arbitrarily disconnect and reconnect without the risk of inhibiting their ability to accurately estimate time, then that algorithm is not suitable for the types of networks this paper focuses on, since mobile devices are constantly changing their connection status.

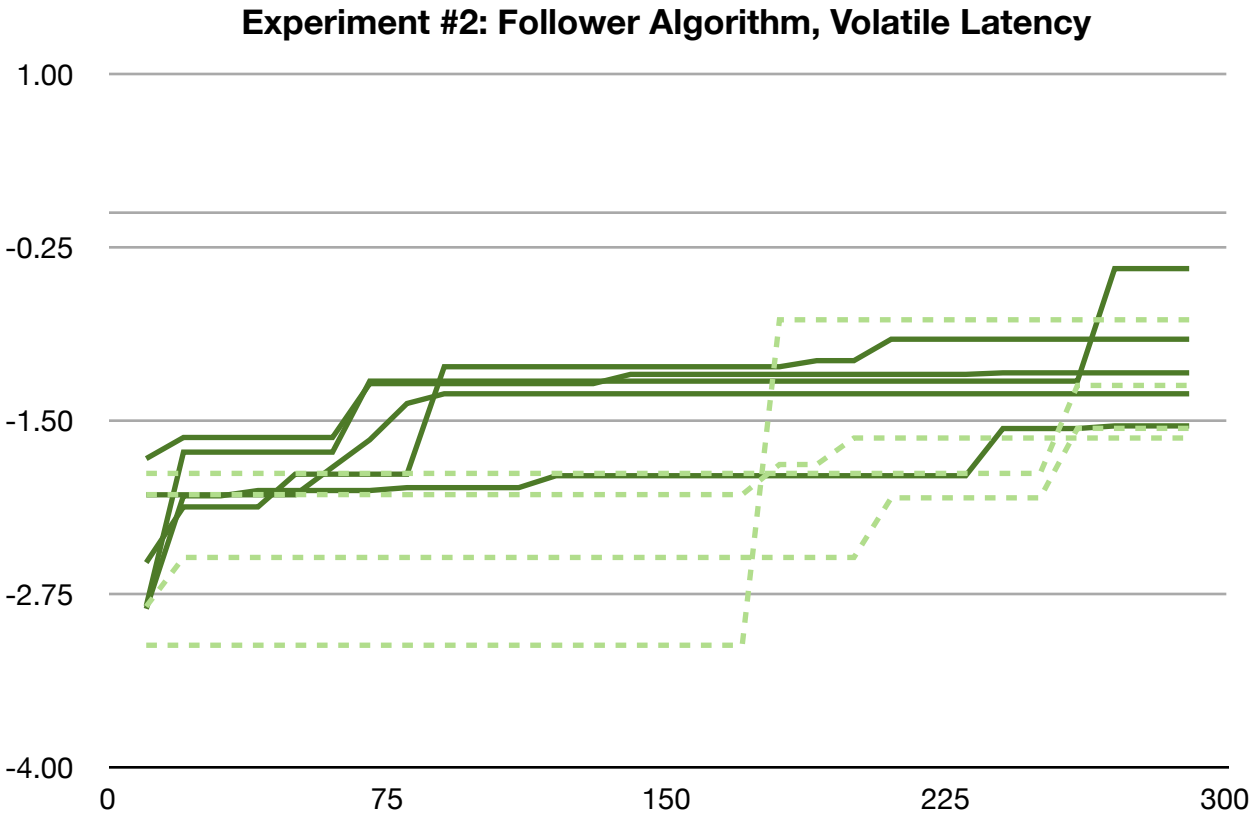
All client nodes were initialized in a connected state, but shortly after the simulation began half of the nodes were disconnected from the server, ceasing all time updates. Halfway through the simulation all disconnected nodes reconnected to the network and began syncing with the server as normal.

All three algorithms exhibited the ability to appropriately cope with a client disconnect and clients recovered quickly when reconnected. For example, compare these three graphs,

showing the individual client lag times for each other algorithms during the same simulation using volatile latency:

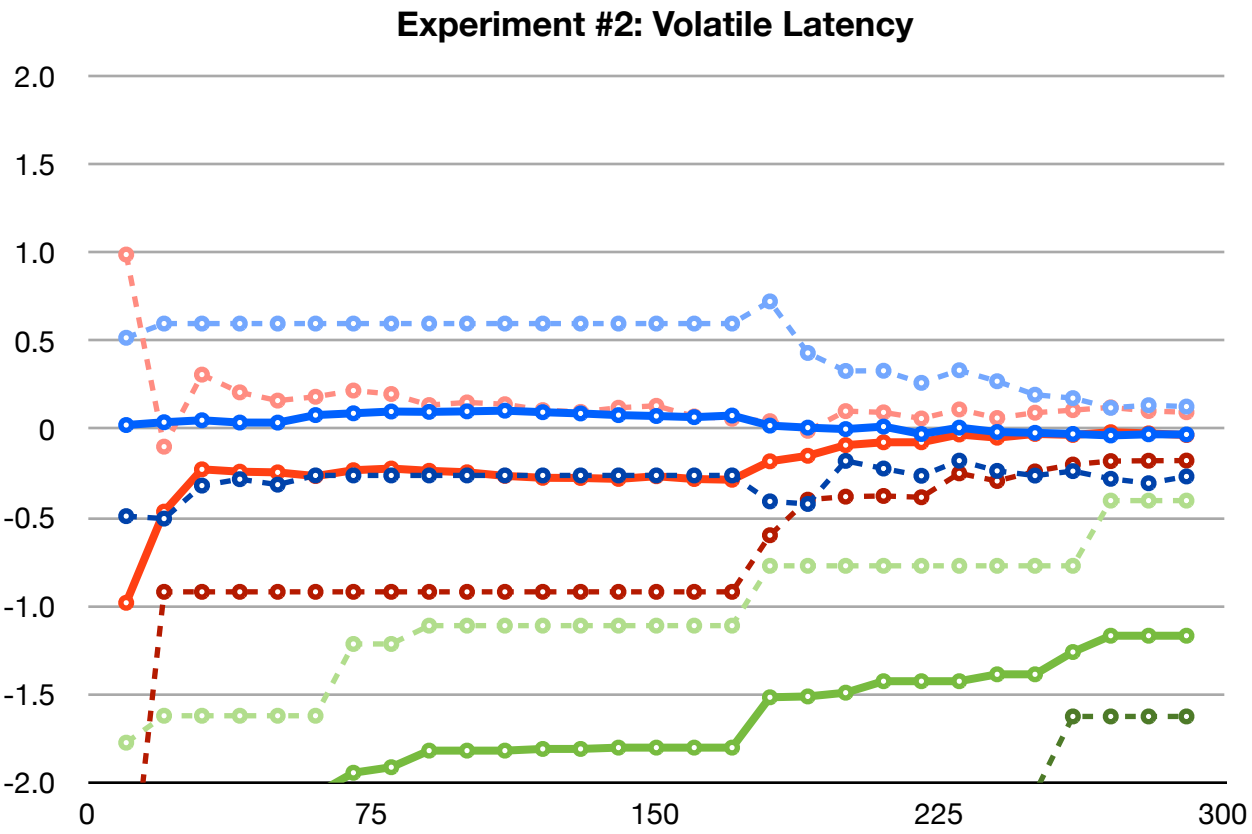




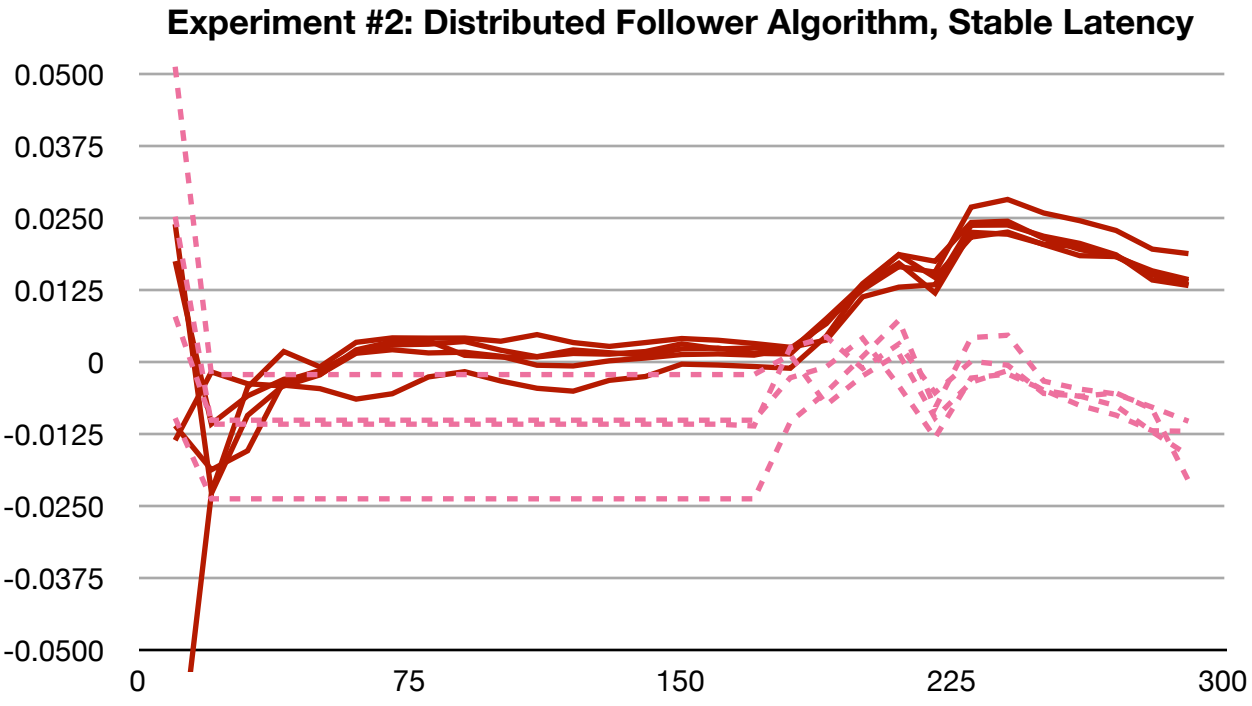


In each instance, half of the client nodes disconnected early on and then reconnected a little after 150 seconds into the experiment. In all three instances, reconnected clients recovered quickly. However, there are some noticeable differences. For example, nodes running Cristian's algorithm experienced severe oscillations in server time estimates after reconnecting, before eventually smoothing out. However, nodes running the Distributed Follower algorithm seem to immediately and smoothly glide back into place among the other previously connected nodes.

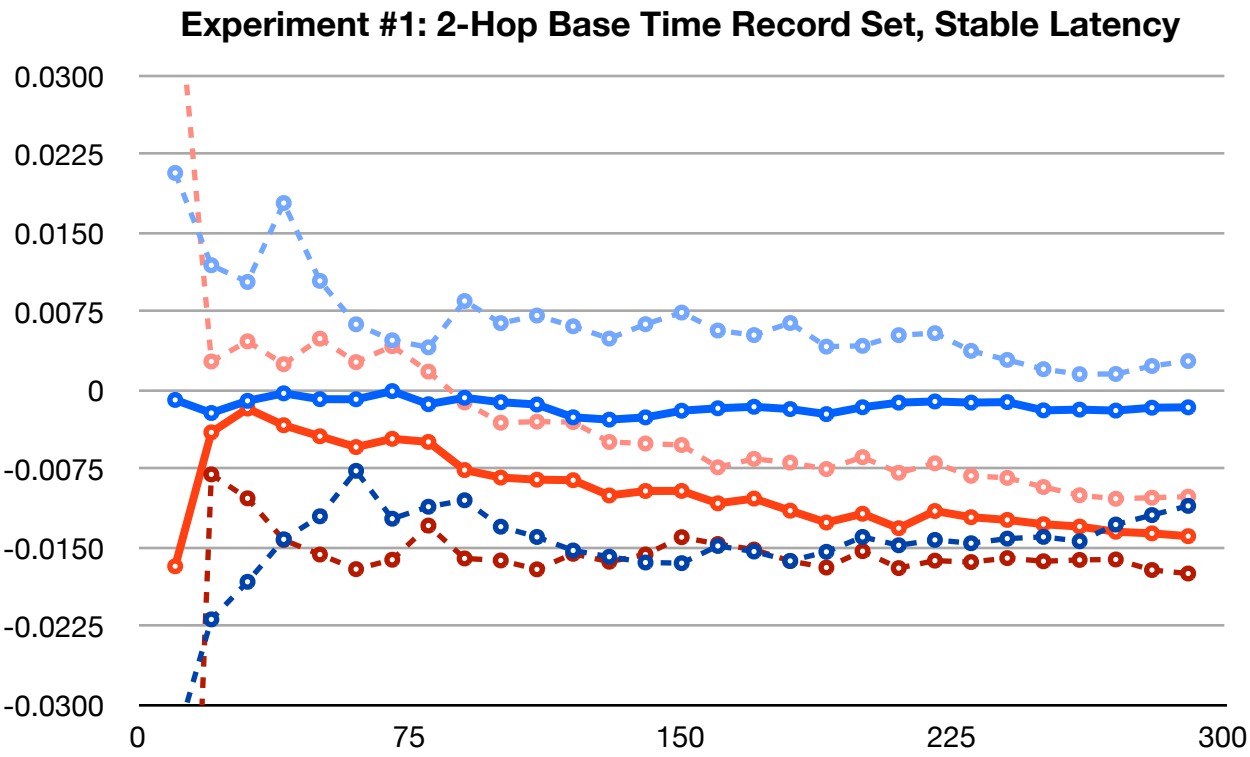
When comparing the algorithms directly, the Distributed Follower algorithm is shown to be slightly more accurate than Cristian's, complementing the results from experiment #1:



This experiment also revealed the existence of an error in the implementation of the Distributed Follower algorithm. Consider this chart, showing the performance of nodes running the Distributed Follower algorithm under conditions of stable latency:



As you can see, something is definitely wrong. Unfortunately the source of this error was not found, and is an opportunity for future work. However, this anomaly only occurs under stable latency conditions and has a minor effect on the results (compare the Y-axis distribution of the stable latency graph above to that of long-tail or volatile latency; the time fluctuation is minuscule). It is, however, responsible for causing some drift in results over time. For example, this drift can be seen in the stable latency results from experiment #1:

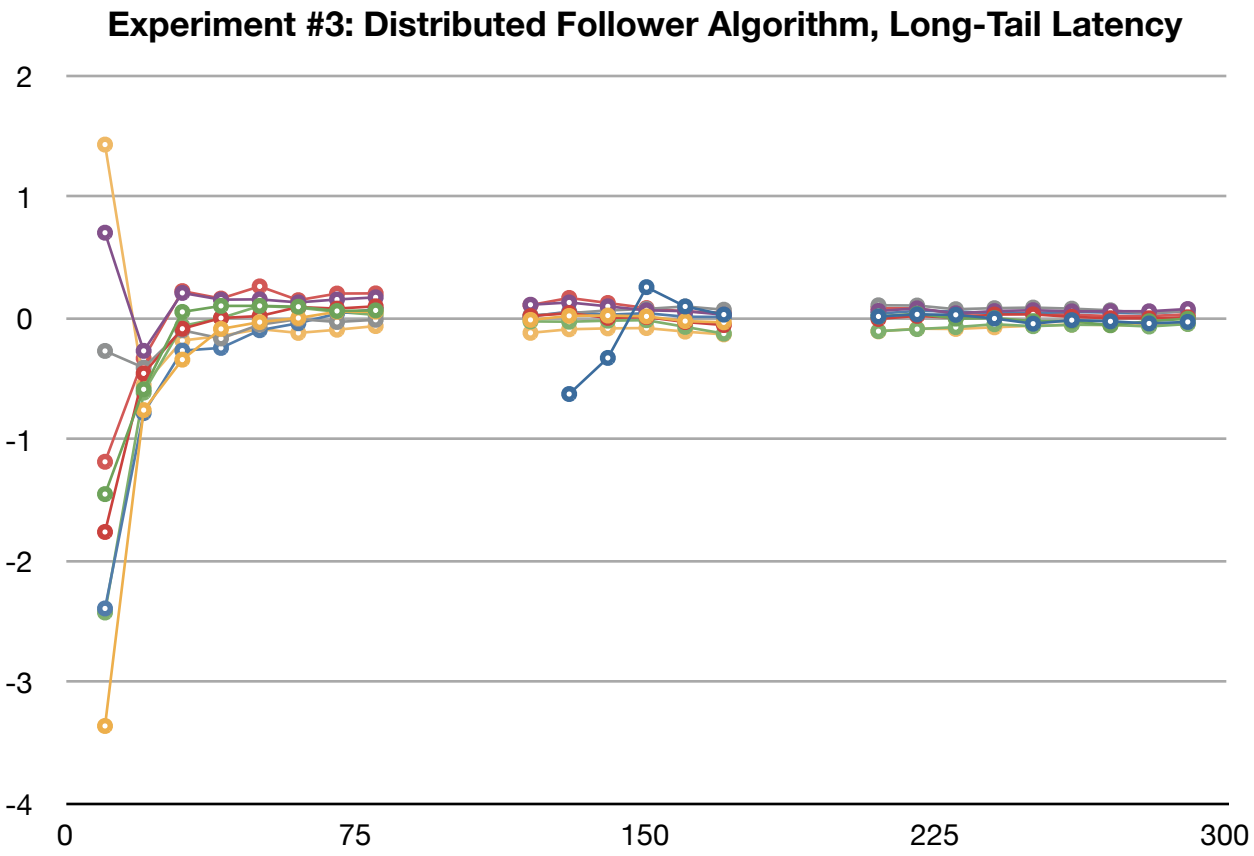


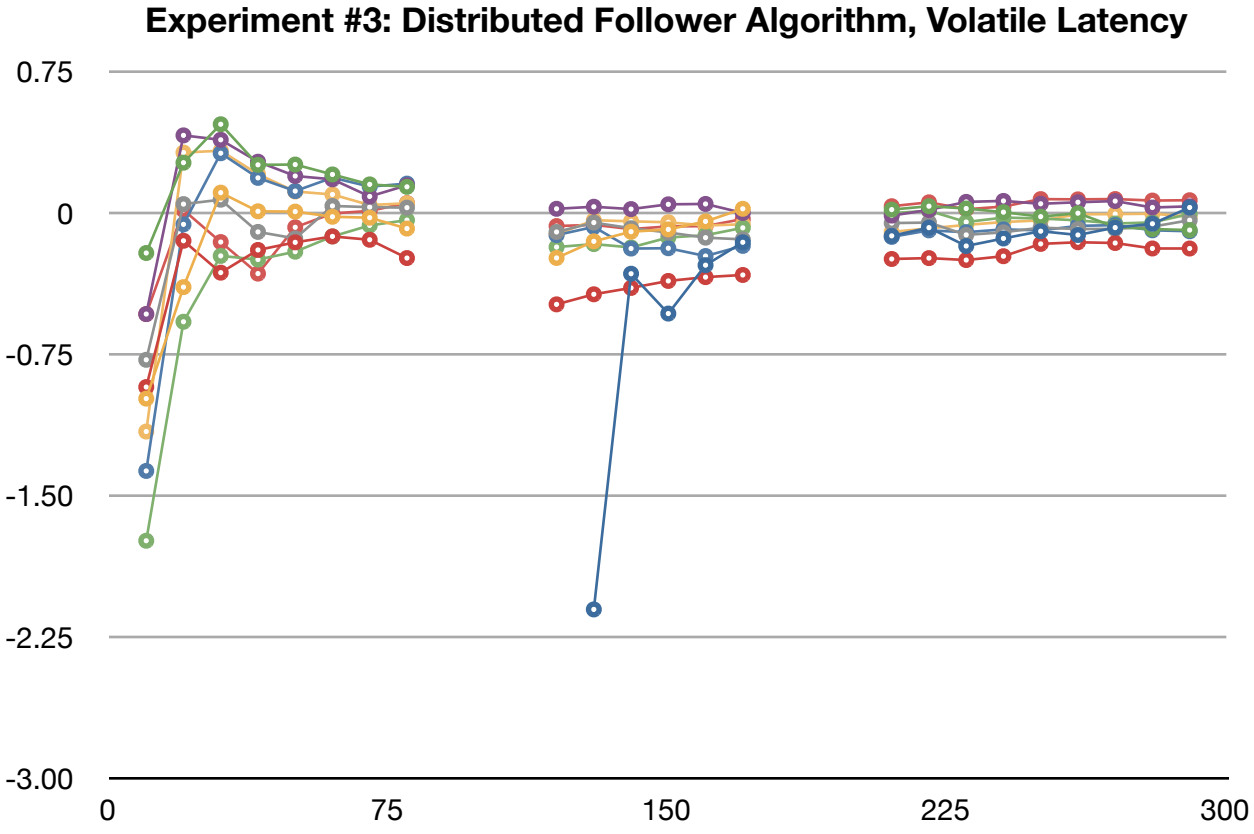
In order to properly account for this drift, it is best to focus on the distribution of the values for the Distributed Follower algorithm, rather than their absolute position. In this instance, it is clear that the Distributed Follower nodes have a much tighter distribution than the Cristian nodes.

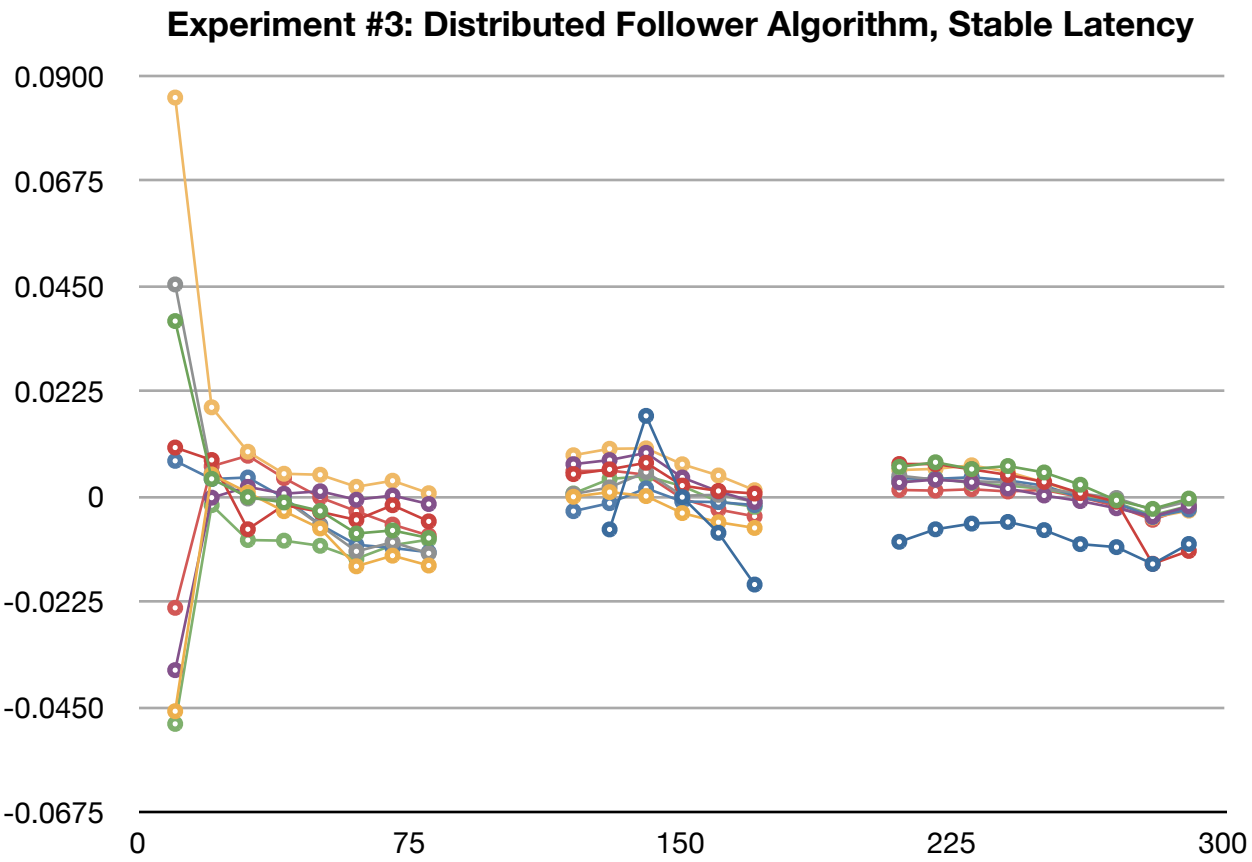
EXPERIMENT #3: SERVER DISCONNECT

This experiment tested whether a networks running the Distributed and Non-Distributed Follower Algorithm can survive the loss of a server node. All client nodes were initialized in a connected state with one node acting as a time server. After a brief period the time server disconnected form the network and all time updates stopped. After another brief period of down-time, a new node would be picked as the server and would begin issuing time updates, and the old time server would reconnect as a normal client. This process was repeated twice, simulating two server disconnects.

This experiment has impressive results. Here are the graphs for the Distributed Follower algorithm in all three latency conditions:


Experiment #3: Distributed Follower Algorithm, Long-Tail Latency
2
1
0
-1
-2
-3
-4
0
75
150
225
300


Experiment #3: Distributed Follower Algorithm, Volatile Latency
0.75
0
-0.75
-1.50
-2.25
-3.00
0
75
150
225
300



In all three graphs, there is one node that is out of place after the first down-time period. This is the original server node reconnecting as a new client, and so must begin syncing its time from scratch. However, in all three instances, it recovers quickly, in line with the results from experiment #2.

Not only does the Distributed Follower algorithm perform superbly in all three conditions, but the distribution of client values seems to tighten and improve after every server down-time period. This proves the algorithm's distributable quality, and shows its ability to continue to improve results even after experiencing repeated server failures.

There is a natural explanation for this ability. As previously described, one of the advantages of the Distributed Follower algorithm is its strong client-to-client parity, as proven by the tight distribution of Distributed Follower nodes across all three experiments. When a client is chosen to become a new server, it adds its latency estimate to its current server time estimate, and uses that as the time to send out to connected clients. This extra unit of latency is meant to account for the latency of the first hop, so that once the update message reaches its destination, it has been delayed enough to be relatively close to the original time.

Since clients share information to build parity, when a new server is picked from among that client group, they have relatively similar server time estimates and relatively similar

latency estimates, and so the new server's time is even closer to the clients' time than the old server's was, increasing client-to-server parity. With each new server, this effect increases, resulting in ever tighter distributions of times within clients.

Hardware Results

IMPLEMENTATION

After completion of the simulated experiments the Follower algorithm was implemented in a small network of five nodes made out of custom hardware.

The hardware consisted of an Arduino Uno connected to an Arduino WiFi shield, with a breadboard for wiring in sensors and LEDs for input and output. Arduino is an open source hardware prototyping platform, allowing for simple and fast electronics prototyping. The main advantage of this platform is that it is cheap and highly configuring with lots of physical pin connections.

However, the main disadvantage is that the Arduino Uno processor board is single-threaded, making distributed networking difficult. After unsuccessfully attempting to satisfactorily implement the Distributed Follower algorithm on the Arduino, the simpler Follower algorithm was used instead. This highlights the additional constraints that hardware can impose on decision-making.

One of the advantages of the Follower and Distributed Follower algorithms is that they can be implemented in several dozen lines of code, which is important for a platform such as Arduino. The Arduino Uno has only 32 kilobytes of flash memory, meaning that every line of code matters, and the smaller the program, the better.

In our specific implementation, any node can act as a server or a client, but must be initialized as one or the other. The server node sends out time updates to any other nodes connected to its network. Time values are generated from the system clock, which measures the number of milliseconds since the Arduino was powered on. This value resets approximately every 50 days due to long integer overflow, however for the purposes of our project this was an acceptable constraint.

When clients receive this time value, it is rejected or accepted according to the Follower algorithm, and the system keeps track of current time by adding the time elapsed since the last update to the stored. Time elapsed is determined from the client's own system clock.

All code for the Arduino implementation can be found in Appendix C.

RESULTS

Unlike the virtual network simulations, it is difficult to compare the data across different nodes. However, several there were a few observations.

First, the network displayed a latency model most similar to the "long-tail" model of the virtual simulations. Since the nodes connect via WiFi, latency was usually extremely low, in

fact almost instant, resulting in close to equal time values between servers and clients, which is in line with the results from the simulations. Occasionally, latency would jump considerably, however the Follower algorithm properly ignores these delayed values.

The occasional latency jump is likely due to the networking overhead in the implementation. As mentioned, the Arduino is only single-threaded, and so the server can only send a message to one client at a time. Our server opens and closes connections to clients as needed to make room for other connections. As such, sometimes update messages get delayed while waiting to be sent.

The Distributed Follower algorithm would not be able to properly operate in this kind of environment, since both servers and clients need to be able to connect to multiple other nodes. If only one connection is possible at a time, and nodes cannot act as both servers and clients simultaneously, then the resulting delays cause the algorithm to lose its benefits. Implementing the Distributed Follower algorithm on custom multi-threaded hardware, such as Raspberry Pi's, is instead left for future work.

Finally, it is important to note that this implementation, although limited, succeeded in accomplishing one of the original goals of proving that time can be synchronized without required specialized hardware. No real time clock module or GPS shield was needed to correctly tell time. Instead, only a processor board and a communication shield (in this case WiFi) was necessary. This reduces the cost of building these devices significantly, and also reduces their size and weight.

Future Work

There are many opportunities for future work and improvements upon the work described in this paper.

ALGORITHM IMPROVEMENTS

The Distributed Follower algorithm contains many areas of potential improvement.

First, time record sets store all updates that are received, so future research could explore new rules to govern how time record sets function. For example, a record set could remove old values over time, allowing nodes to adapt to slow changes in overall network behavior. Also, nodes could be pickier about what updates to accept, employing strategies to weed out potentially corrupting values.

Second, the Distributed Follower algorithm requires nodes to send lots of messages in order for it to have any effect. The number of messages could be reduced by bundling together updates that pass through the node, instead of immediately re-broadcasting every update that is received.

Third, find a solution to the slight "drift" error that occurs in very small ranges in stable latency conditions. This error was highlighted in the discussion of experiment #2.

Fourth, more exploration should be done on the observation that client parity improves when new servers are chosen in a Distributed Follower network. This behavior was highlighted in the discussion of experiment #3.

HYBRID ALGORITHM

As noted in the discussion of experiment #1, the basic Follower algorithm performs much better than both Cristian's and the Distributed Follower algorithms in certain conditions, chiefly those where latency ever decreases to very low levels, even for short periods of time (as in the long-tail latency model). However the Follower algorithm performs much worse compared to the other two algorithms in other types of conditions.

It would be interesting if there were a way of combining the basic Follower algorithm and the Distributed Follower algorithm into a single, hybrid algorithm that can switch between the two depending on what network currently conditions exist. As shown in the simulation implementation, it is easy to combine the code for both algorithms into a single class, so all that is left is a mechanism for determining which algorithm to use for the server time estimate at which time.

MORE EXTENSIVE HARDWARE IMPLEMENTATION

As mentioned in the hardware implementation section, Arduino imposed limitations on what could be implemented on the hardware. For example, being only single-threaded, it

was not possible to easily implement the Distributed Follower algorithm, which requires nodes to be constantly sending and receiving messages to and from multiple different other nodes.

There are multiple other opportunities for further testing of the algorithms on custom hardware as well. Future experiments can compare performance when using different connection methods other than WiFi (such as Bluetooth, or Ethernet). Also, since our implementation only involved five nodes due to restrictions in cost, future experiments could test larger, distributed networks.

References

1. "Arduino - HomePage." Arduino - HomePage. N.p., n.d. Web. 12 May 2013. <http://arduino.cc/>.

2. Coulouris, George F., and Jean Dollimore. "Time and Global States." Distributed Systems: Concepts and Design. Wokingham, England: Addison-Wesley, 2012. N. pag. Print.

3. Cristian, Flaviu. "Probabilistic Clock Synchronization." Distributed Computing 3 (1989): 146-58. Print.

Appendix A: Simulation Results

EXPERIMENT #1: BASE TIME RECORD SET

Under the Distributed Follower Algorithm, client nodes estimate server time by taking a mean value of one of their time record sets, which are bins of time values separated by hop count, and adding a latency estimate. Here is the formula:

```
Server Time Estimate = (Base Time Record Set X Mean) + (X+1)*(Latency Estimate)
```

where X is the index of the record set to use as the "base record", and is equal to the hop count of the time values in the set.

This experiment tested using different base records for calculating the server time estimate. Test groups are broken down first by the latency model used, and then the hop count for the base record.

Tests:

1. Long-Tail Volatile Latency Model:

    - **Base Record = 0:** Chart LTL-1
    - **Base Record = 1:** Chart LTL-2
    - **Base Record = 2:** Chart LTL-3
    - **Base Record = 3:** Chart LTL-4
    - **Base Record = 4:** Chart LTL-5

2. Volatile Latency Model:

    - **Base Record = 0:** Chart VL-1
    - **Base Record = 1:** Chart VL-2
    - **Base Record = 2:** Chart VL-3
    - **Base Record = 3:** Chart VL-4
    - **Base Record = 4:** Chart VL-5

3. Stable Latency Model:

    - **Base Record = 0:** Chart SL-1
    - **Base Record = 1:** Chart SL-2
    - **Base Record = 2:** Chart SL-3
    - **Base Record = 3:** Chart SL-4
    - **Base Record = 4:** Chart SL-5

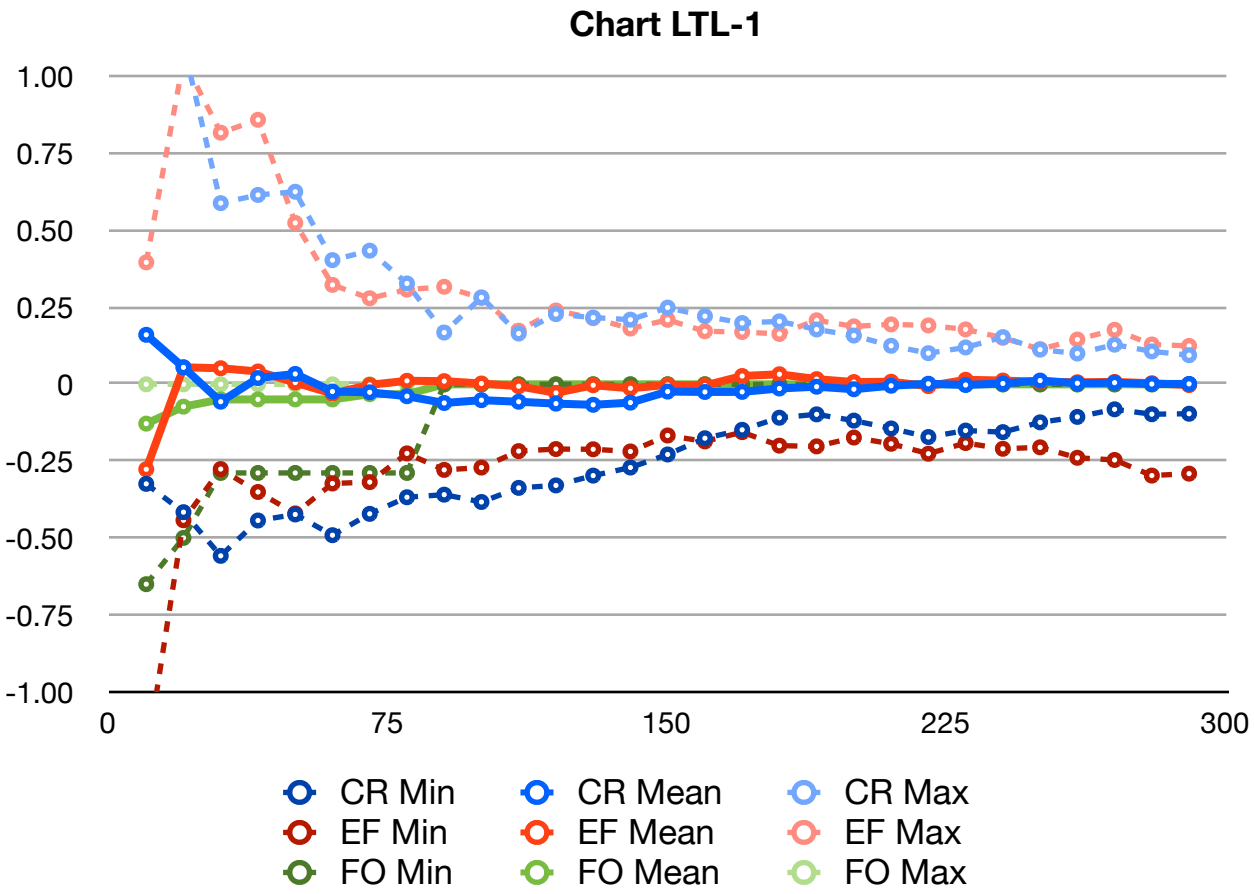


Chart LTL-1
1.00
0.75
0.50
0.25
0
-0.25
-0.50
-0.75
-1.00
0
75
150
225
300
CR Min
CR Mean
CR Max
EF Min
EF Mean
EF Max
FO Min
FO Mean
FO Max

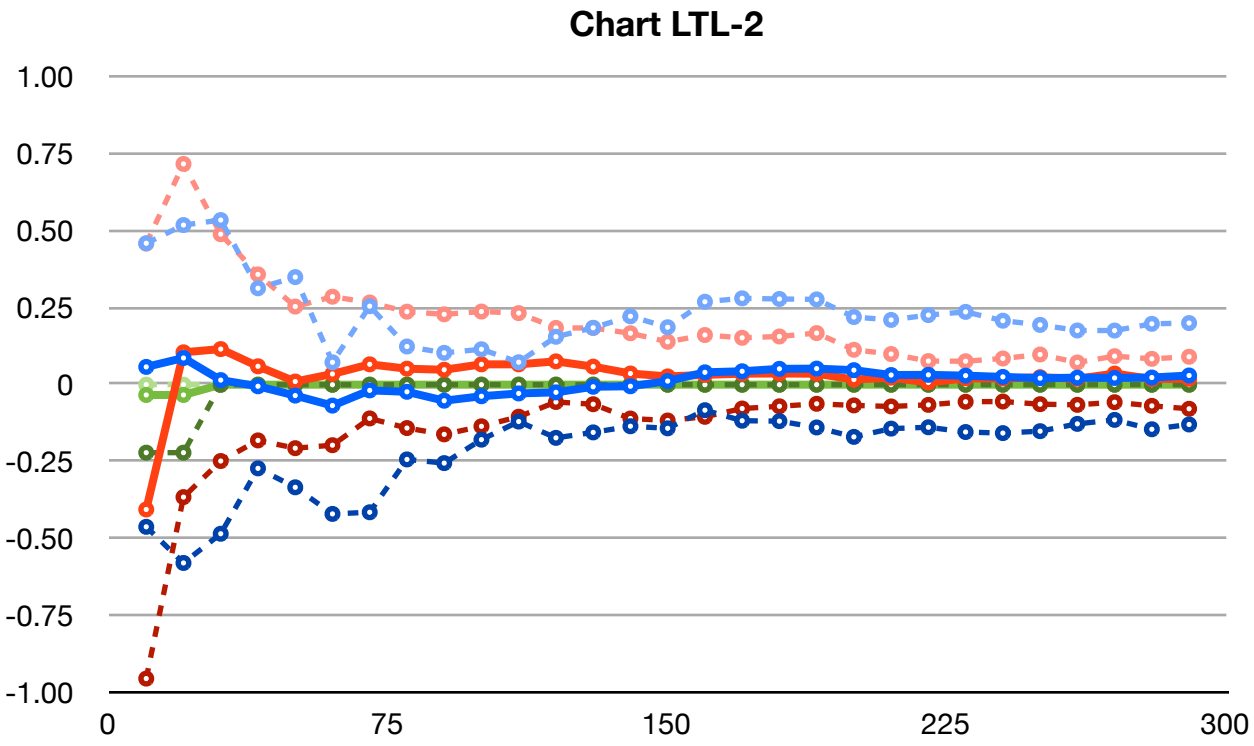


Chart LTL-2
1.00
0.75
0.50
0.25
0
-0.25
-0.50
-0.75
-1.00
0
75
150
225
300

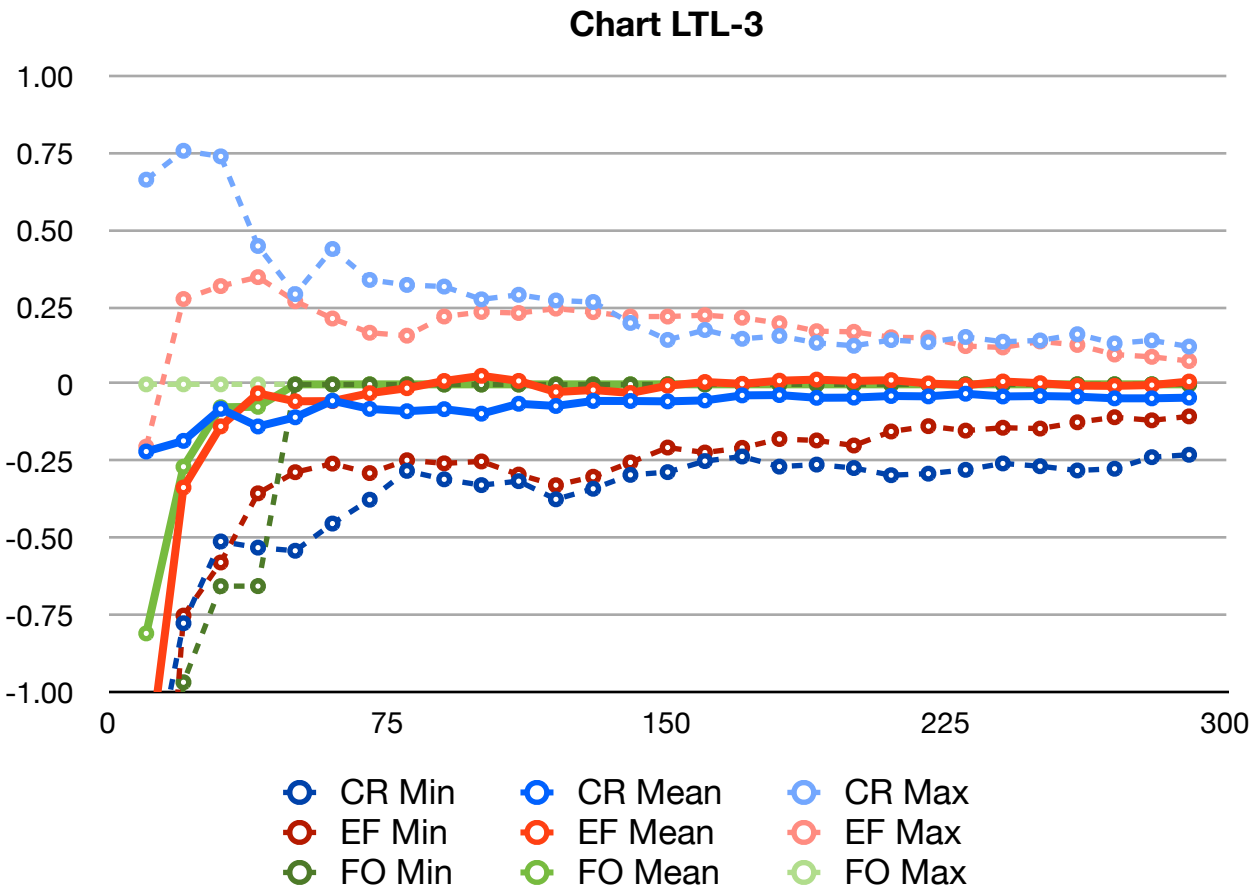


Chart LTL-3
1.00
0.75
0.50
0.25
0
-0.25
-0.50
-0.75
-1.00
0
75
150
225
300
CR Min
CR Mean
CR Max
EF Min
EF Mean
EF Max
FO Min
FO Mean
FO Max

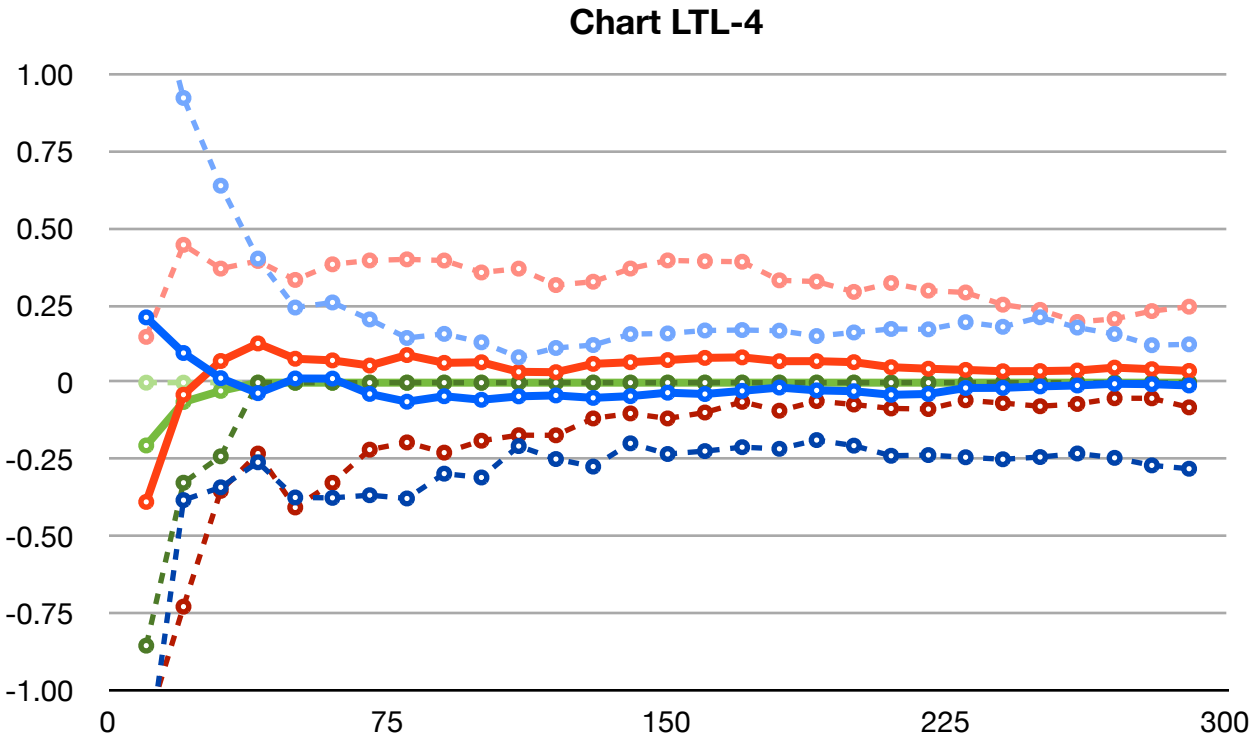


Chart LTL-4
1.00
0.75
0.50
0.25
0
-0.25
-0.50
-0.75
-1.00
0
75
150
225
300

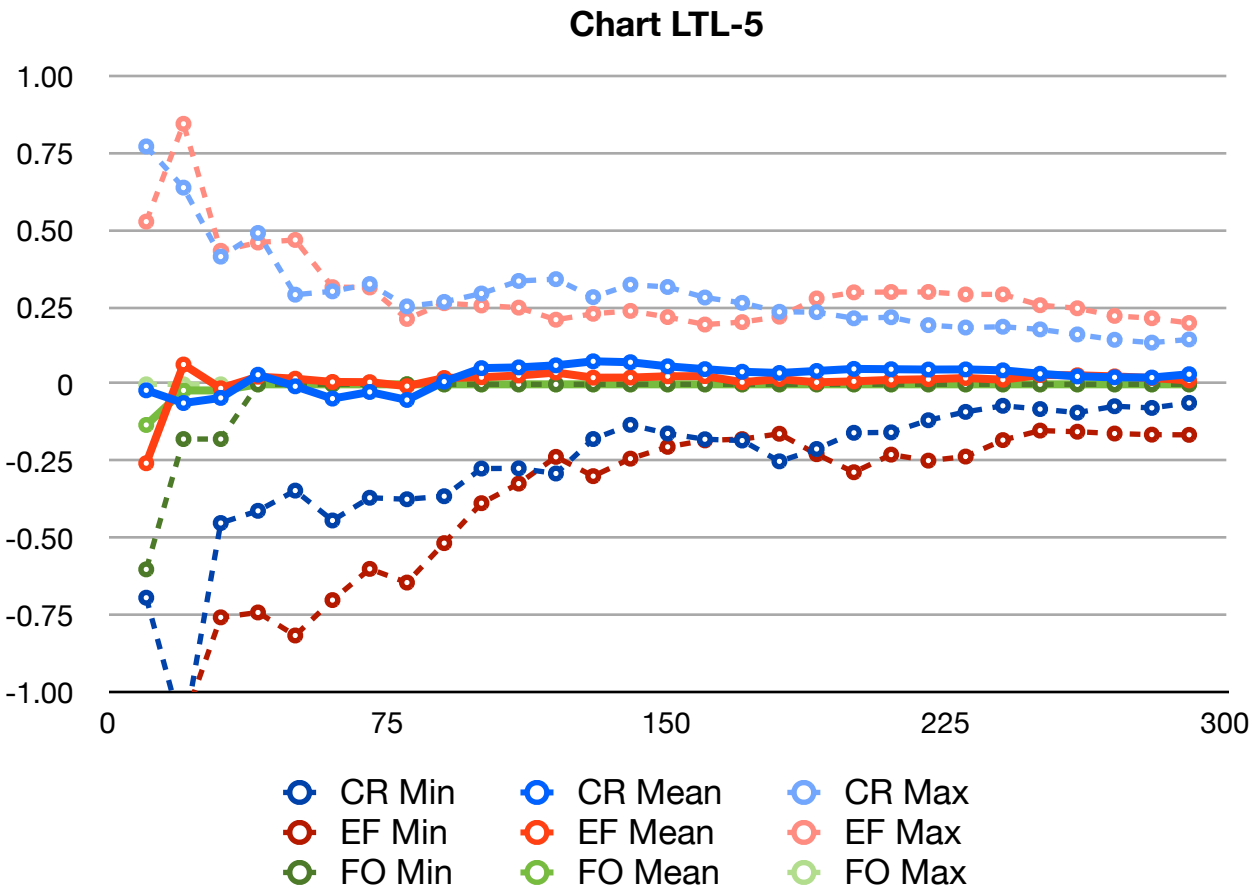


Chart LTL-5
1.00
0.75
0.50
0.25
0
-0.25
-0.50
-0.75
-1.00
0
75
150
225
300
CR Min
CR Mean
CR Max
EF Min
EF Mean
EF Max
FO Min
FO Mean
FO Max

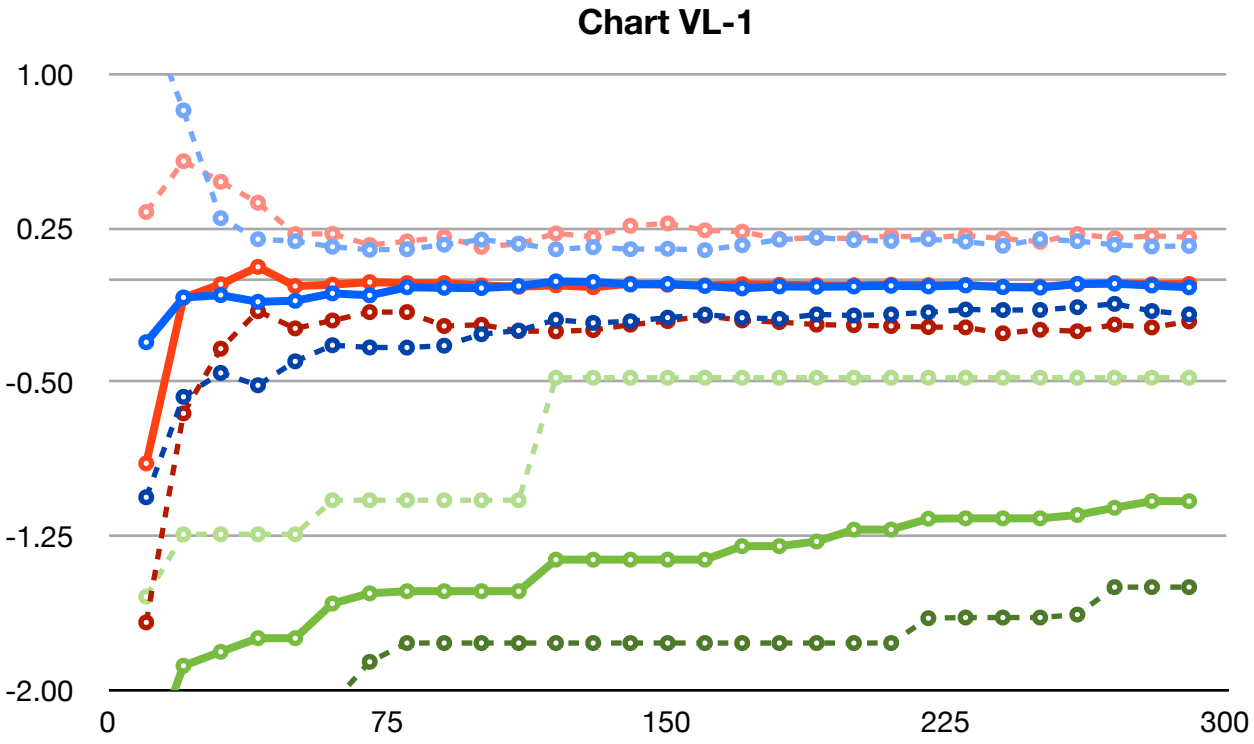


Chart VL-1
1.00
0.25
-0.50
-1.25
-2.00
0
75
150
225
300

Chart VL-2

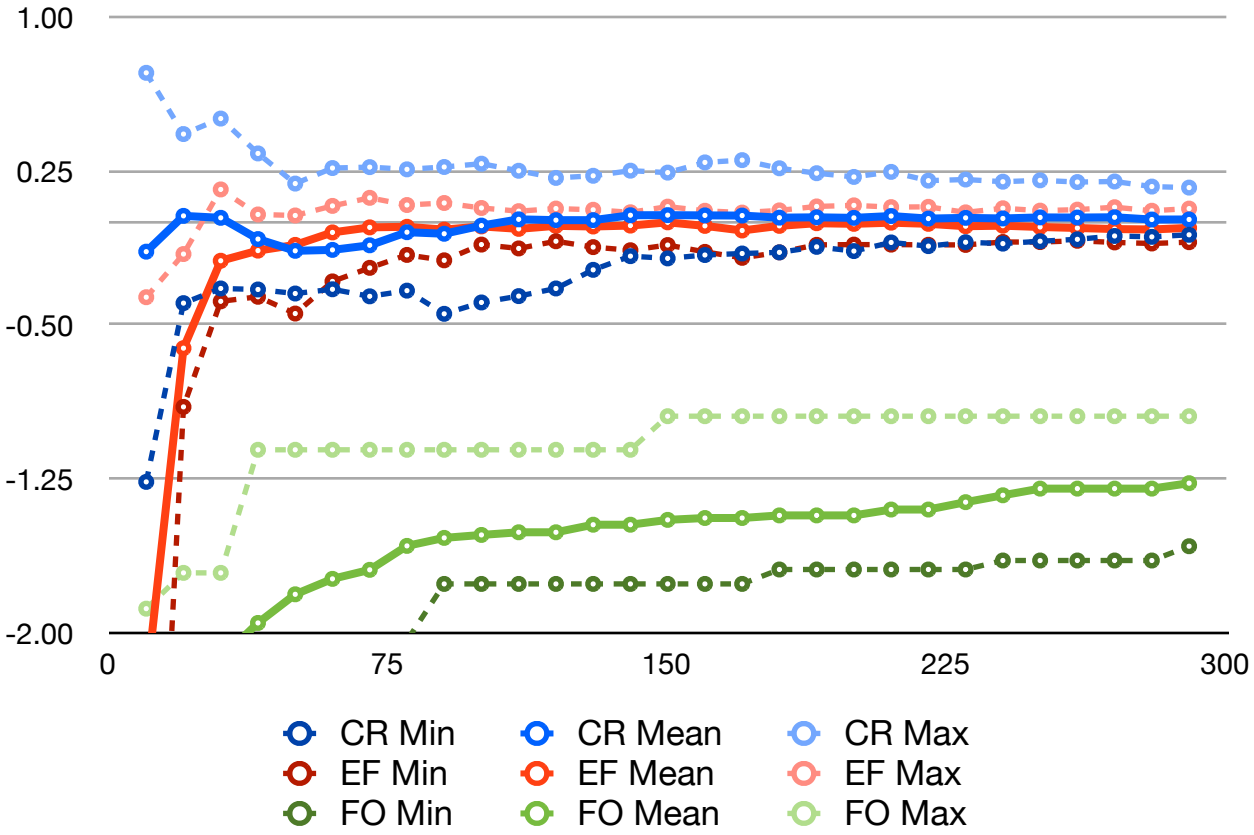
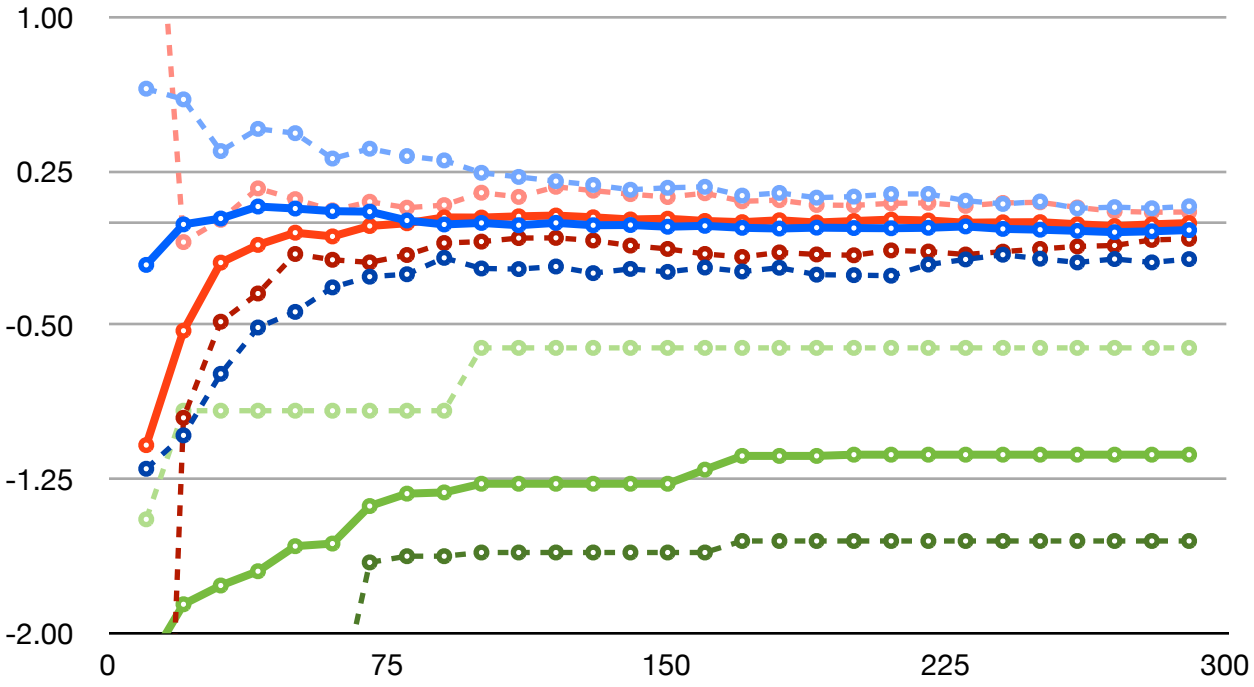


Chart VL-3



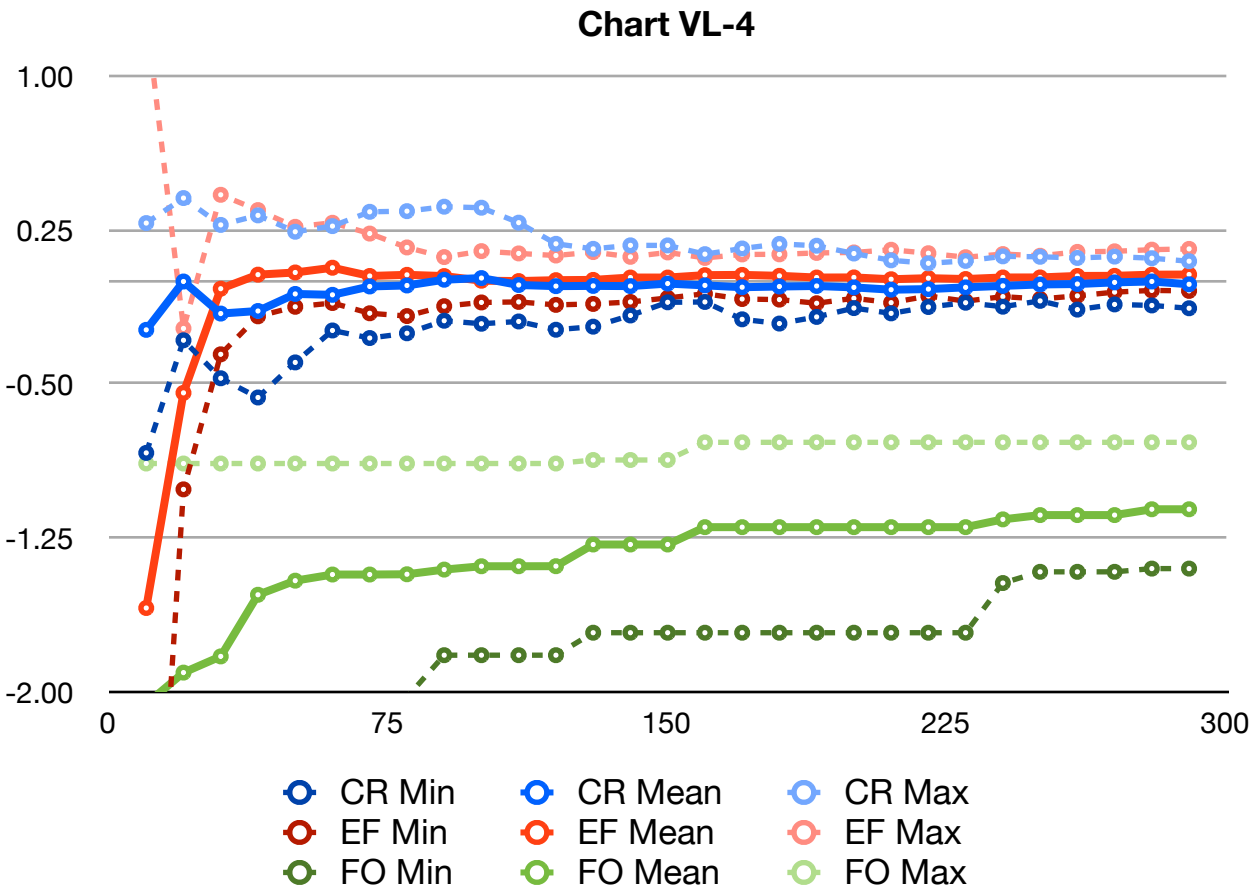


Chart VL-4
1.00
0.25
-0.50
-1.25
-2.00
0
75
150
225
300
CR Min
CR Mean
CR Max
EF Min
EF Mean
EF Max
FO Min
FO Mean
FO Max

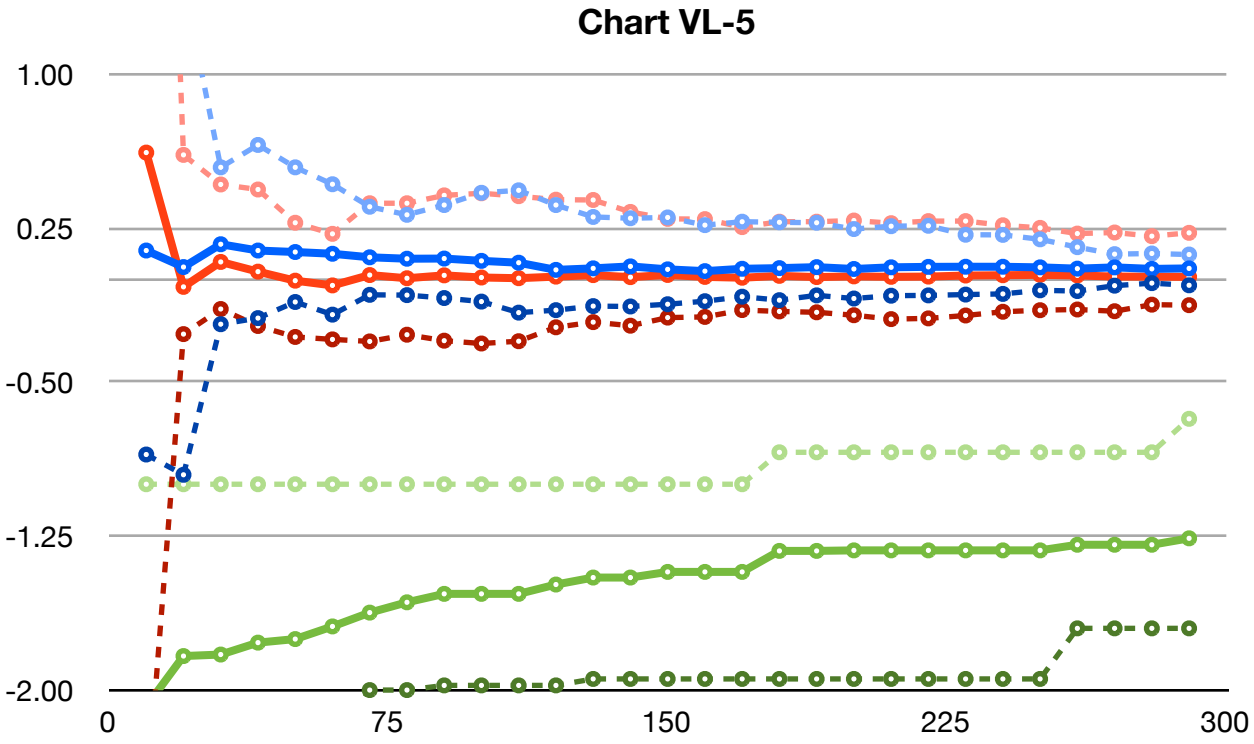


Chart VL-5
1.00
0.25
-0.50
-1.25
-2.00
0
75
150
225
300

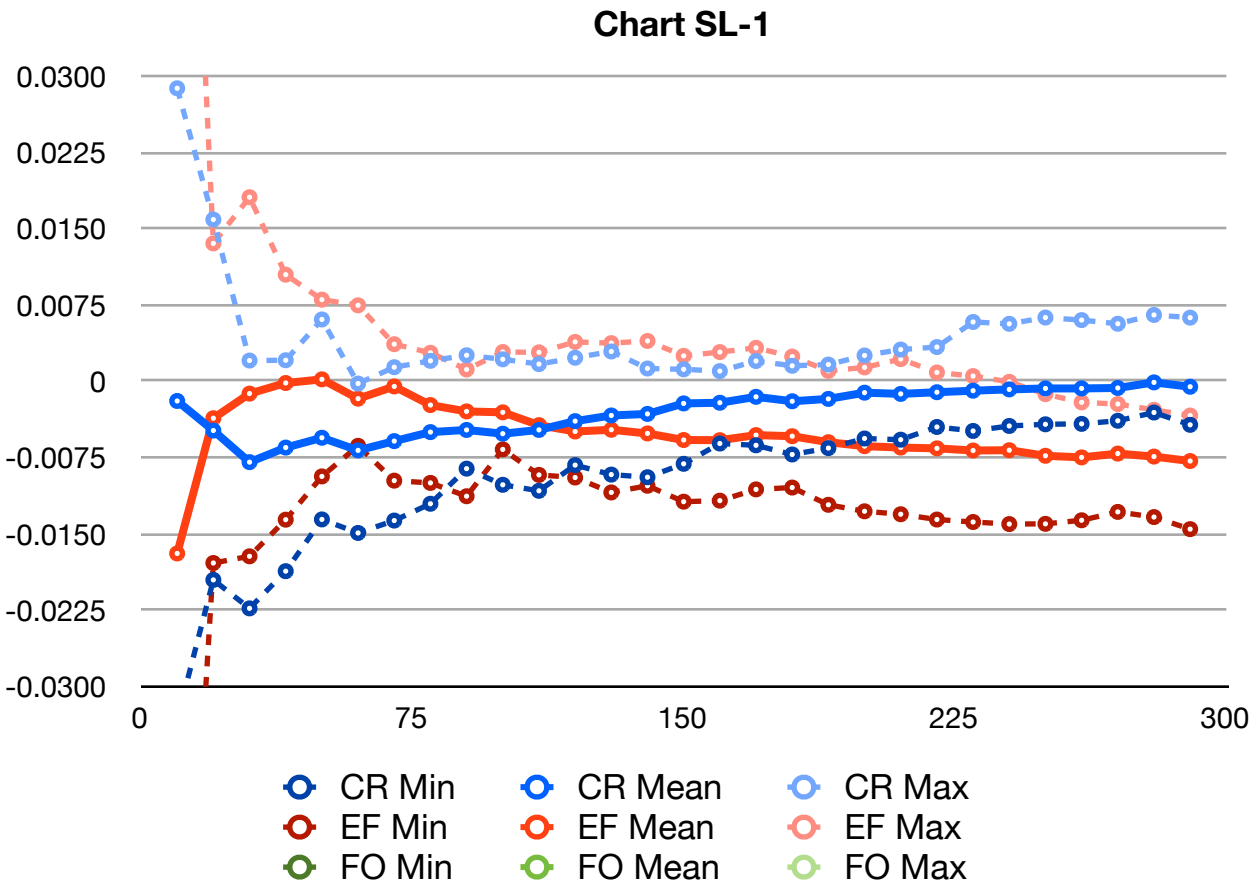


Chart SL-1
0.0300
0.0225
0.0150
0.0075
0
-0.0075
-0.0150
-0.0225
-0.0300
0
75
150
225
300
CR Min
CR Mean
CR Max
EF Min
EF Mean
EF Max
FO Min
FO Mean
FO Max

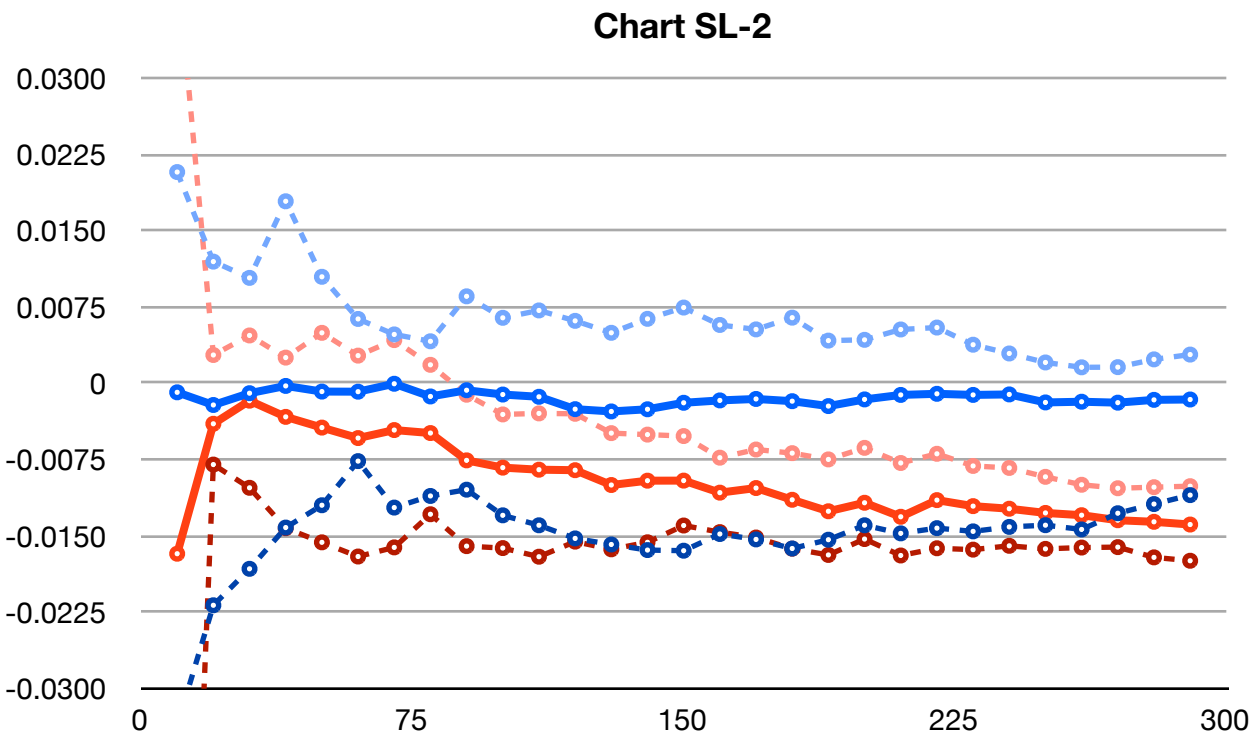


Chart SL-2
0.0300
0.0225
0.0150
0.0075
0
-0.0075
-0.0150
-0.0225
-0.0300
0
75
150
225
300

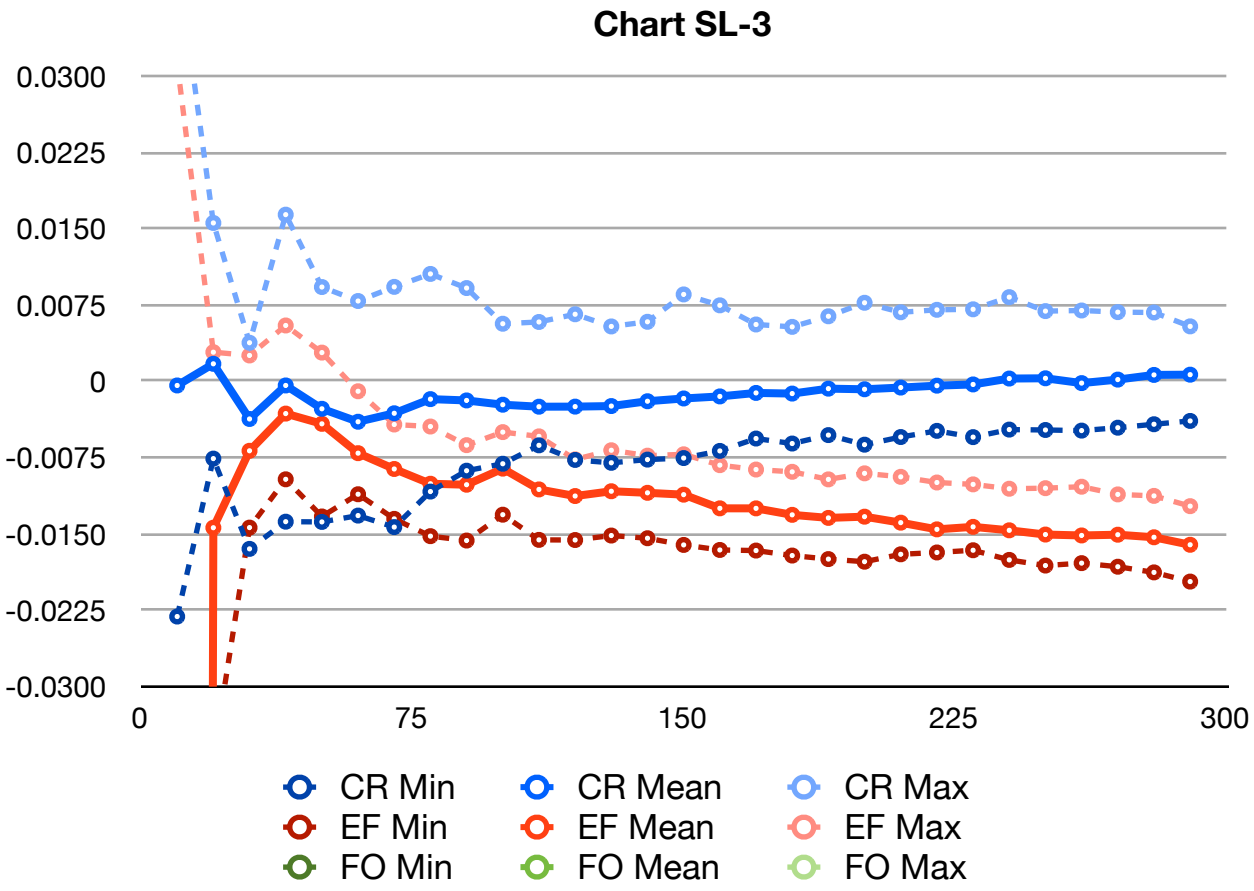


Chart SL-3
0.0300
0.0225
0.0150
0.0075
0
-0.0075
-0.0150
-0.0225
-0.0300
0
75
150
225
300
CR Min
CR Mean
CR Max
EF Min
EF Mean
EF Max
FO Min
FO Mean
FO Max

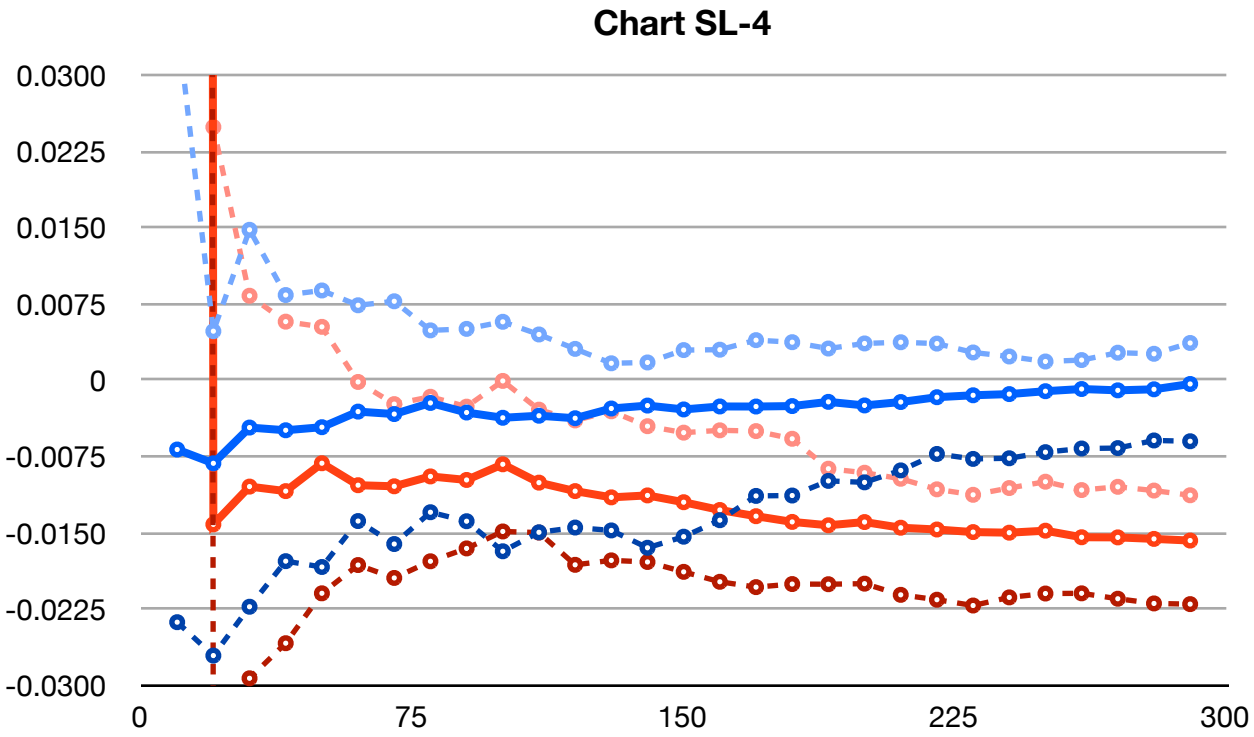


Chart SL-4
0.0300
0.0225
0.0150
0.0075
0
-0.0075
-0.0150
-0.0225
-0.0300
0
75
150
225
300

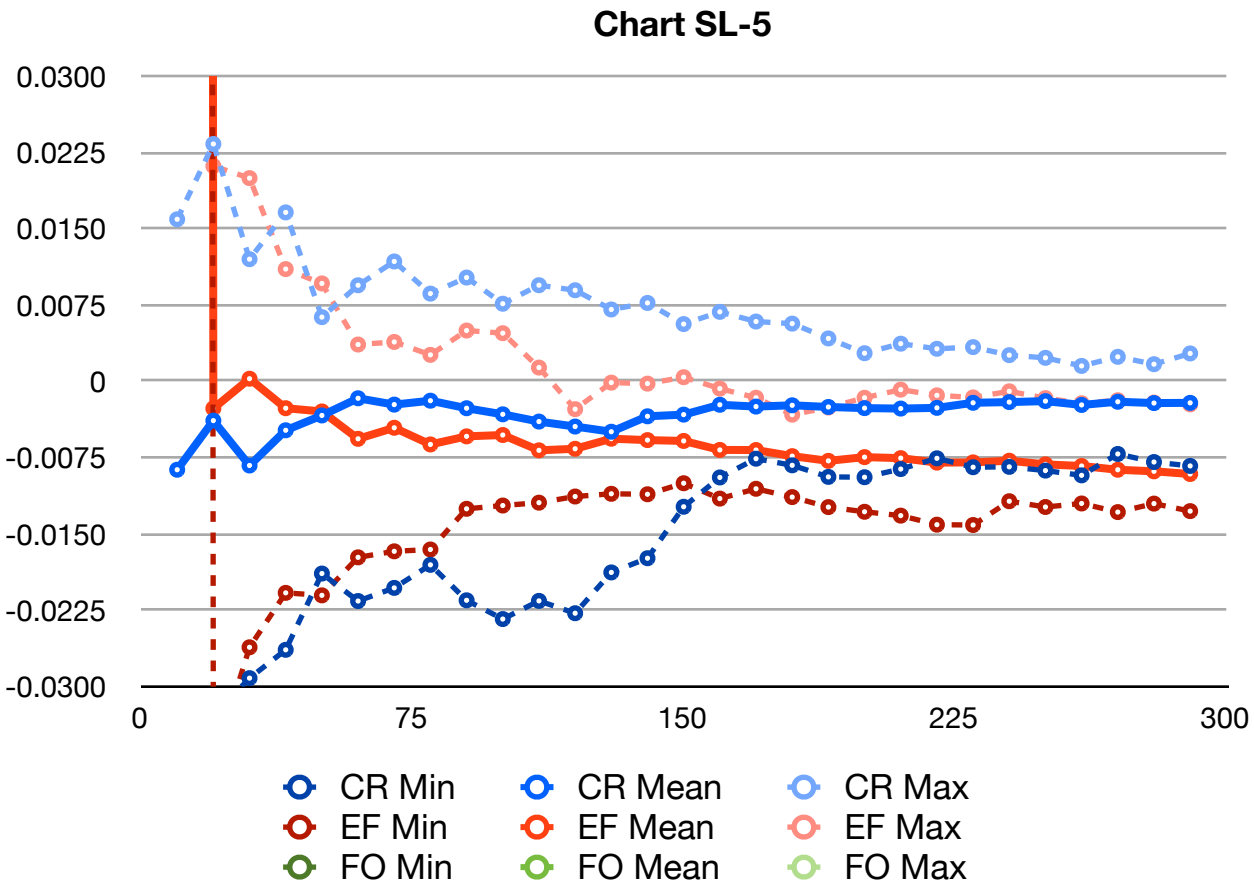


Chart SL-5
0.0300
0.0225
0.0150
0.0075
0
-0.0075
-0.0150
-0.0225
-0.0300
0
75
150
225
300
CR Min
CR Mean
CR Max
EF Min
EF Mean
EF Max
FO Min
FO Mean
FO Max

EXPERIMENT #2: CLIENT DISCONNECT

This experiment tested whether clients can dynamically disconnect and reconnect to a network without significant or prolonged loss of synchronization. All client nodes were initialized in a connected state, but shortly after the simulation began half of the nodes were disconnected from the server, ceasing all time updates. Halfway through the simulation all disconnected nodes reconnected to the network and began syncing with the server as normal.

Test groups are broken down first by the latency model used. For each model, four charts are shown: one for the min/mean/max values for all nodes (divided by algorithm), then one chart for individual lag times for nodes of each algorithm.

Tests:

1. Long-Tail Volatile Latency Model:

    - **LTL Overview:** Chart LTL
    - **Cristian's Algorithm:** Chart LTL-CR
    - **Distributed Follower Algorithm:** Chart LTL-EF
    - **Follower Algorithm:** Chart LTL-FO

2. Volatile Latency Model:

    - **VL Overview:** Chart VL
    - **Cristian's Algorithm:** Chart VL-CR
    - **Distributed Follower Algorithm:** Chart VL-EF
    - **Follower Algorithm:** Chart VL-FO

3. Stable Latency Model:

    - **SL Overview:** Chart SL
    - **Cristian's Algorithm:** Chart SL-CR
    - **Distributed Follower Algorithm:** Chart SL-EF
    - **Follower Algorithm:** Chart SL-FO

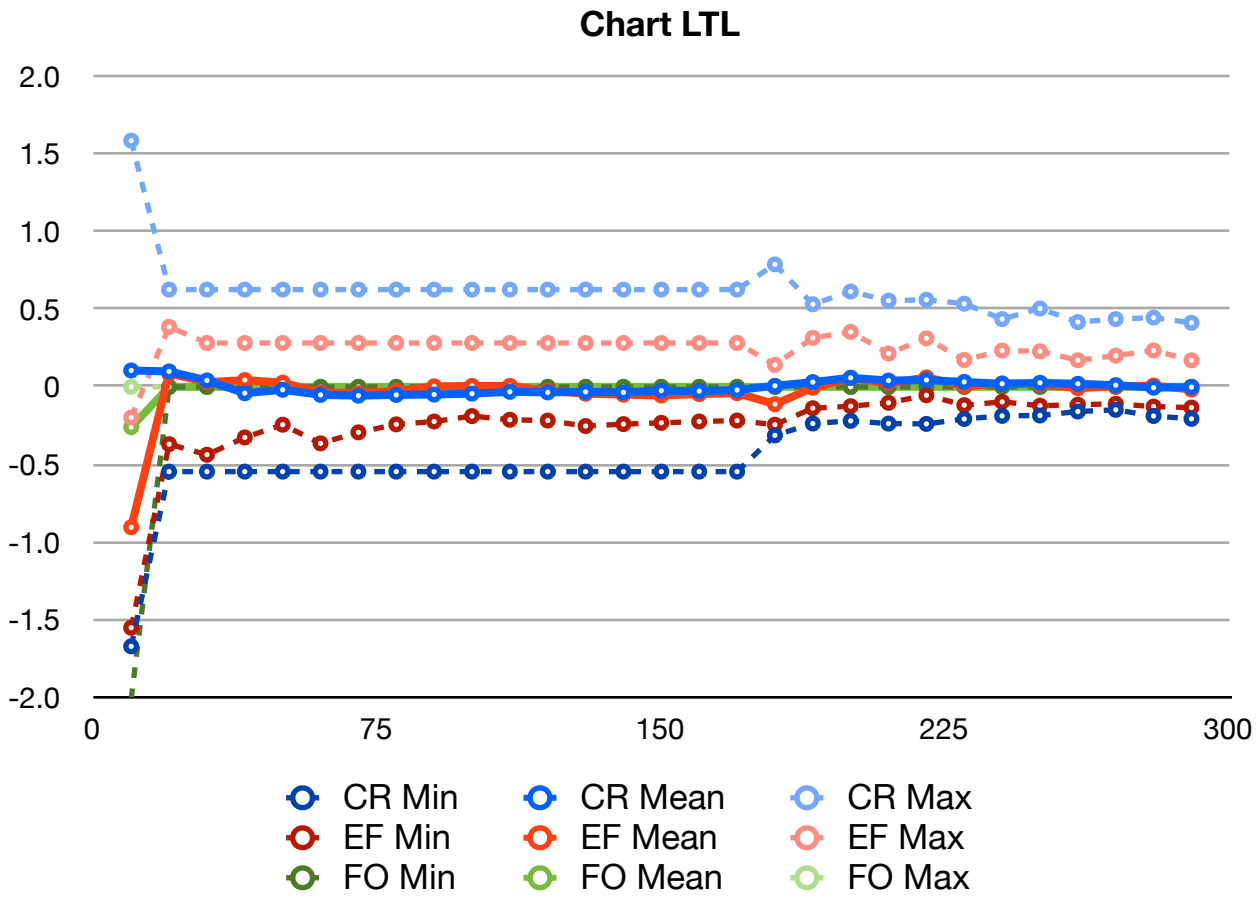


Chart LTL
2.0
1.5
1.0
0.5
0
-0.5
-1.0
-1.5
-2.0
0
75
150
225
300
CR Min
CR Mean
CR Max
EF Min
EF Mean
EF Max
FO Min
FO Mean
FO Max

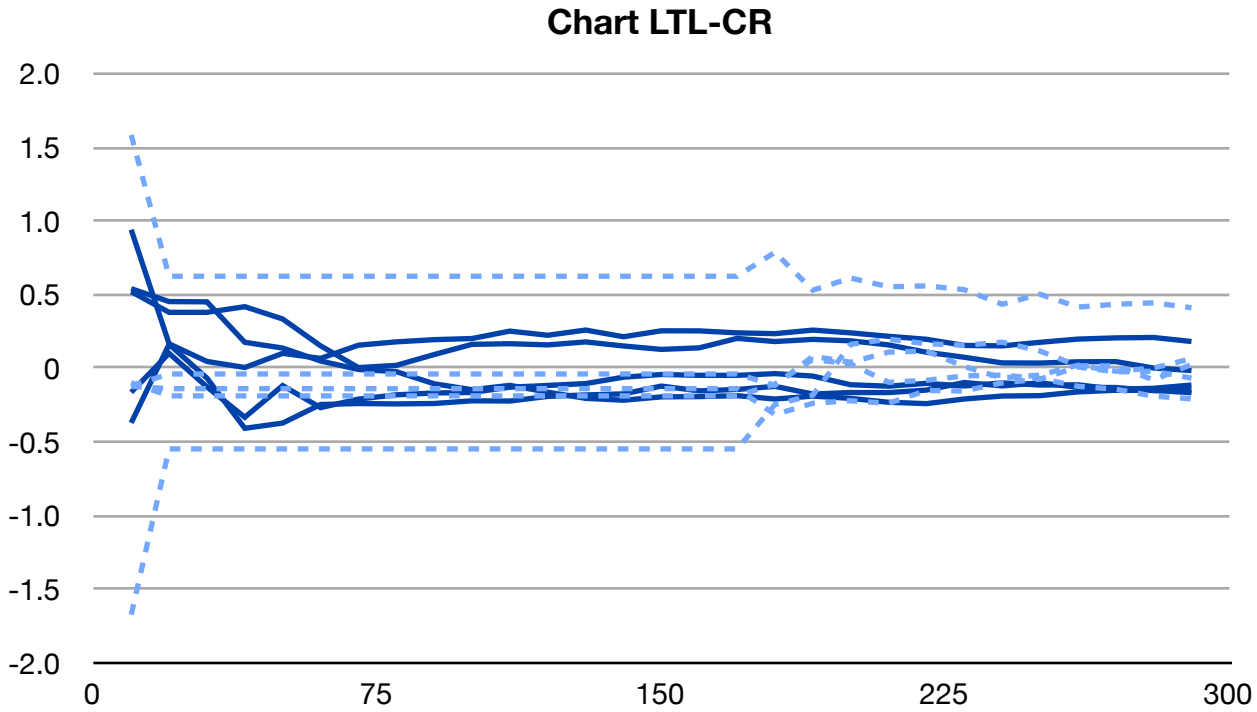


Chart LTL-CR
2.0
1.5
1.0
0.5
0
-0.5
-1.0
-1.5
-2.0
0
75
150
225
300

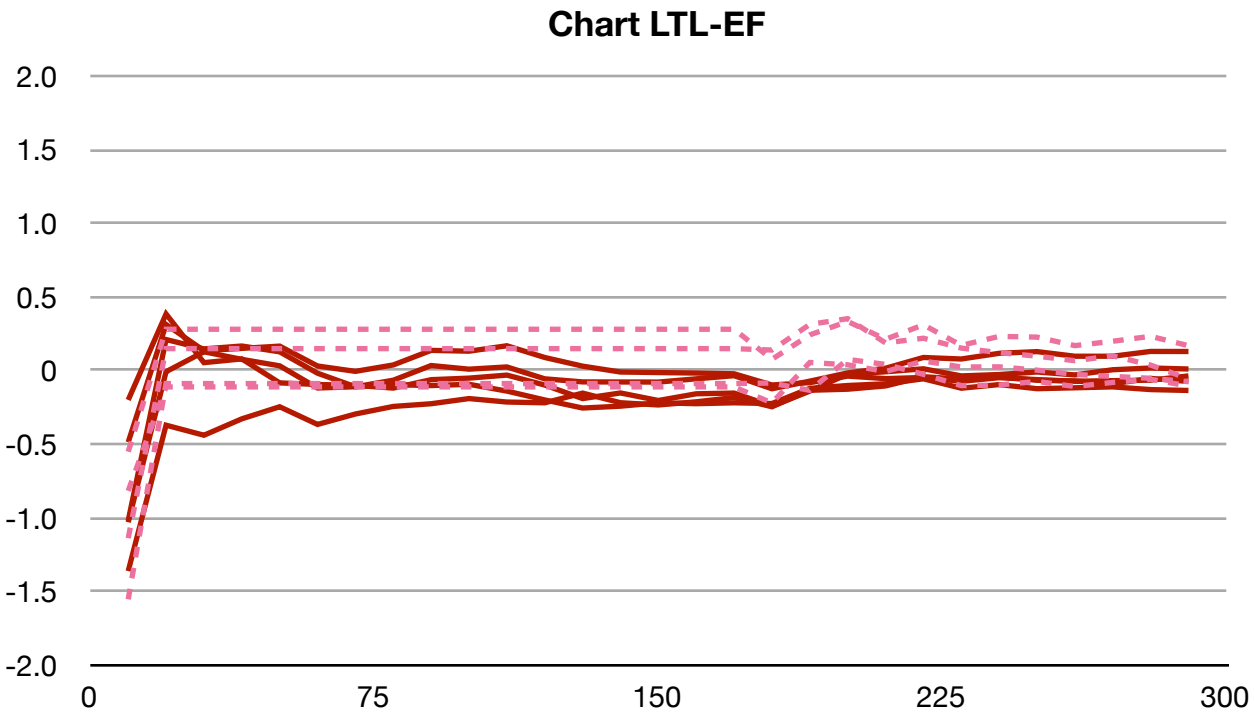


Chart LTL-EF
2.0
1.5
1.0
0.5
0
-0.5
-1.0
-1.5
-2.0
0
75
150
225
300

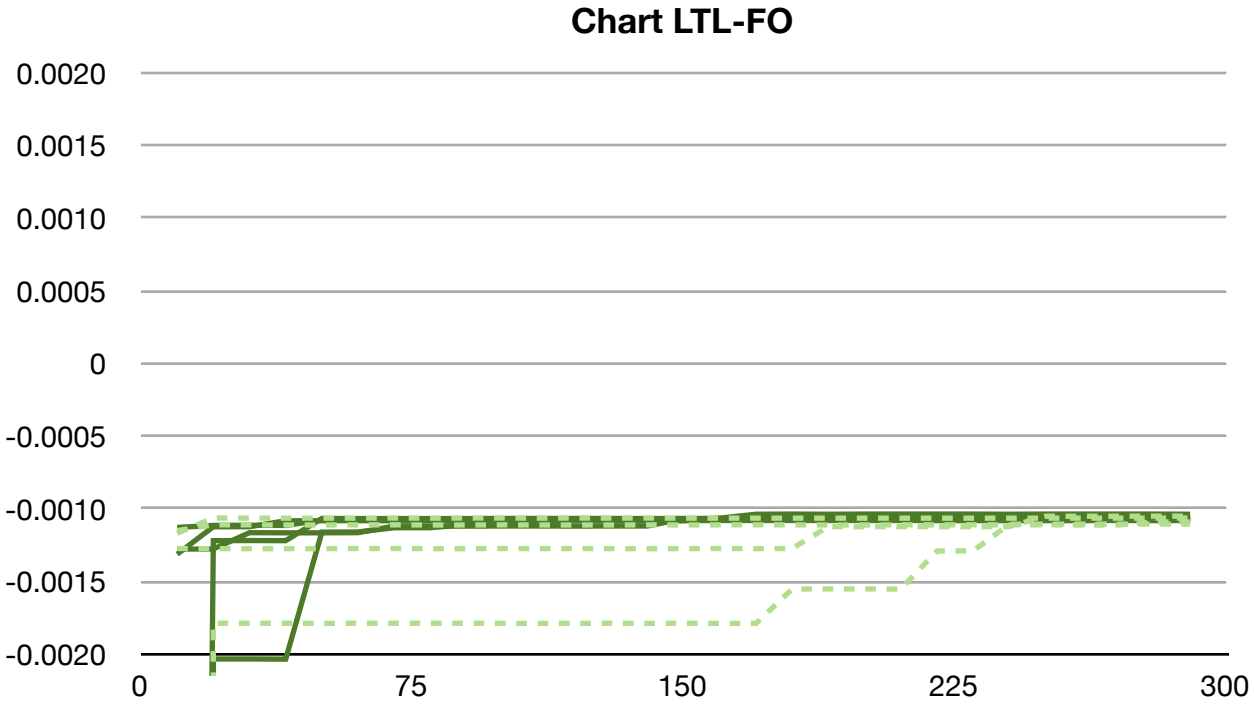


Chart LTL-FO
0.0020
0.0015
0.0010
0.0005
0
-0.0005
-0.0010
-0.0015
-0.0020
0
75
150
225
300

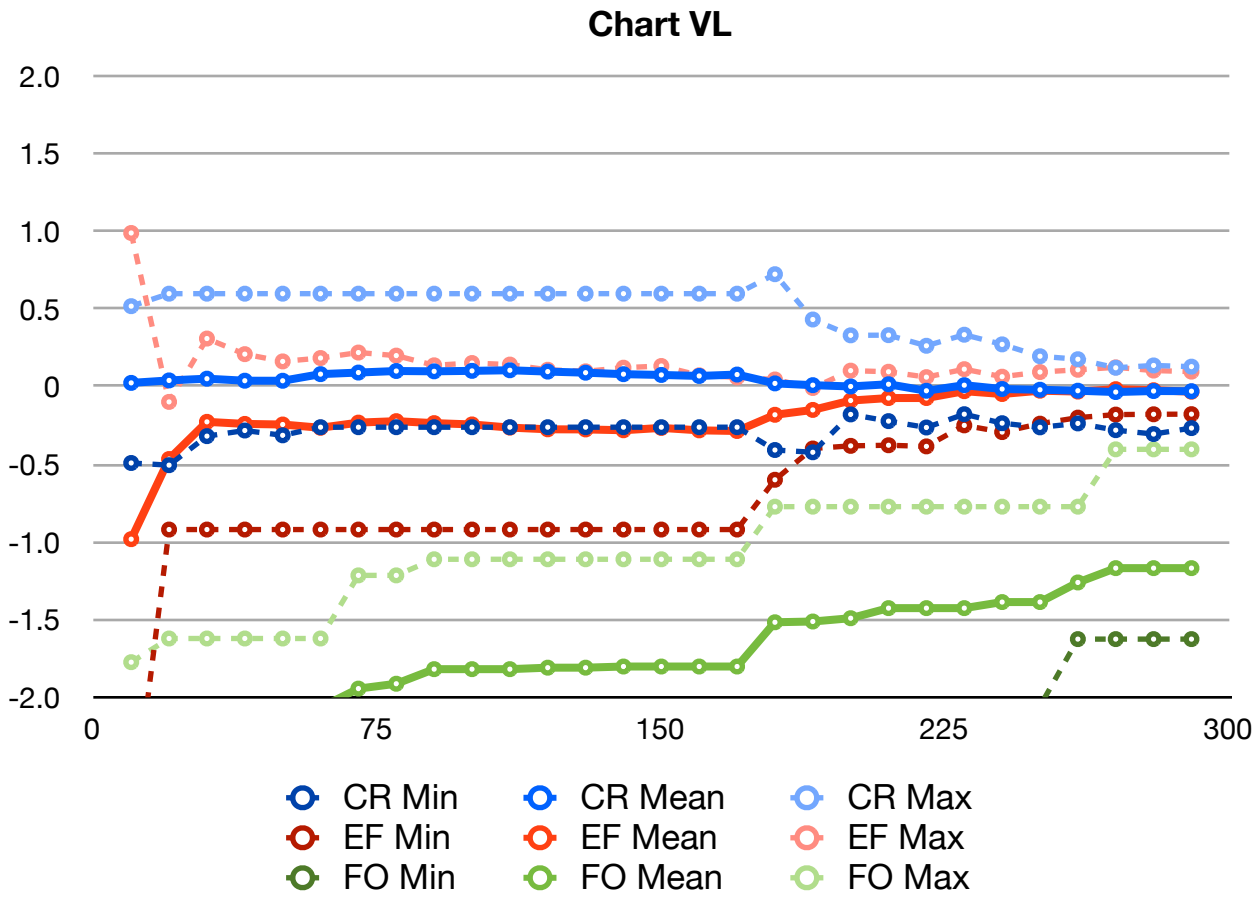


Chart VL
2.0
1.5
1.0
0.5
0
-0.5
-1.0
-1.5
-2.0
0
75
150
225
300
CR Min
CR Mean
CR Max
EF Min
EF Mean
EF Max
FO Min
FO Mean
FO Max

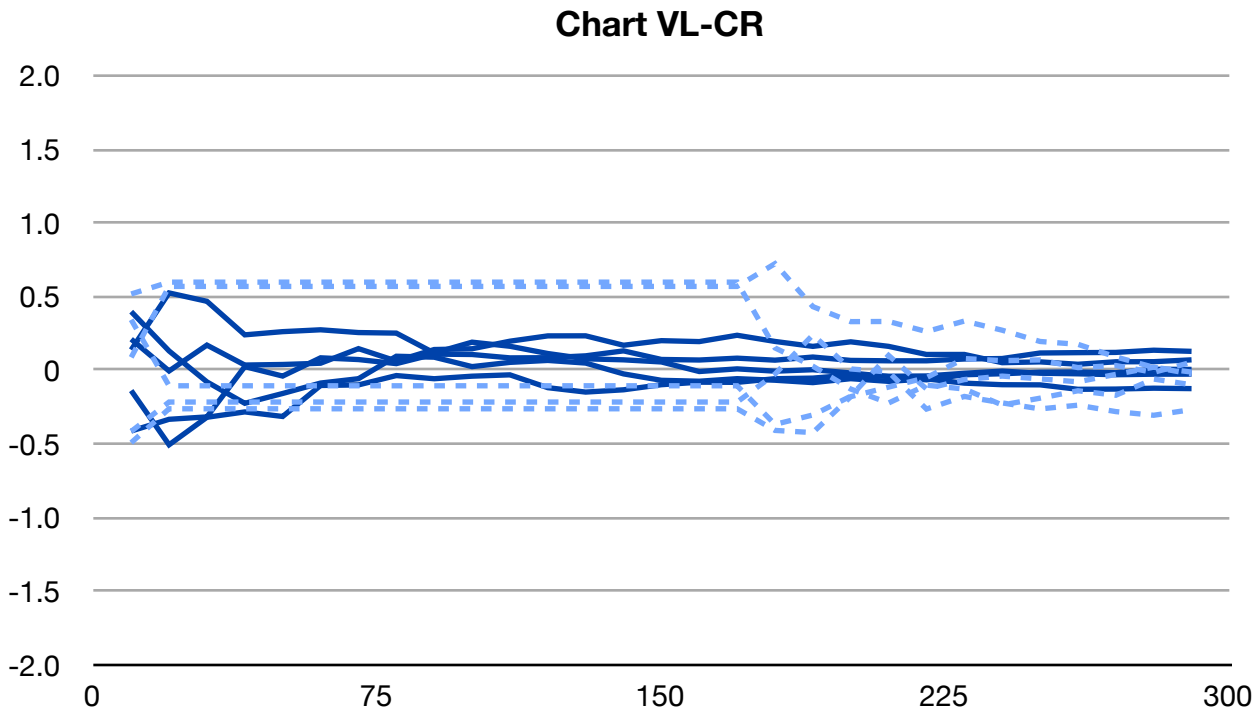


Chart VL-CR
2.0
1.5
1.0
0.5
0
-0.5
-1.0
-1.5
-2.0
0
75
150
225
300

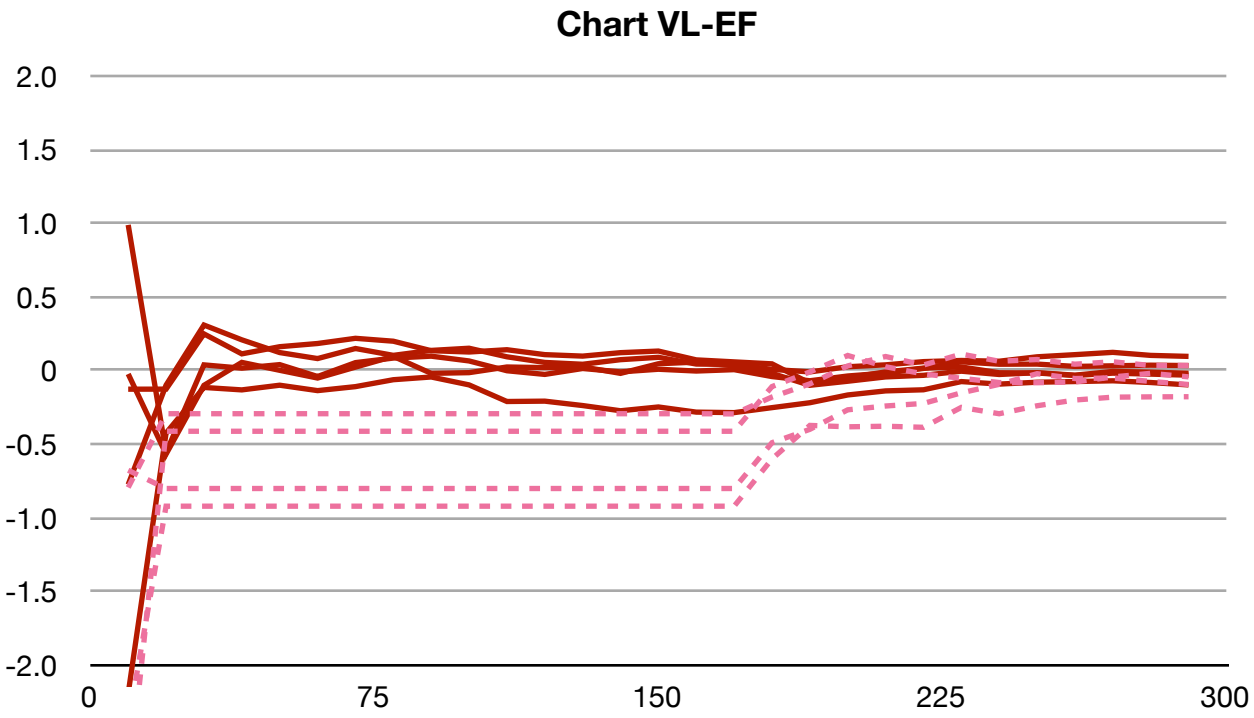


Chart VL-EF
2.0
1.5
1.0
0.5
0
-0.5
-1.0
-1.5
-2.0
0
75
150
225
300

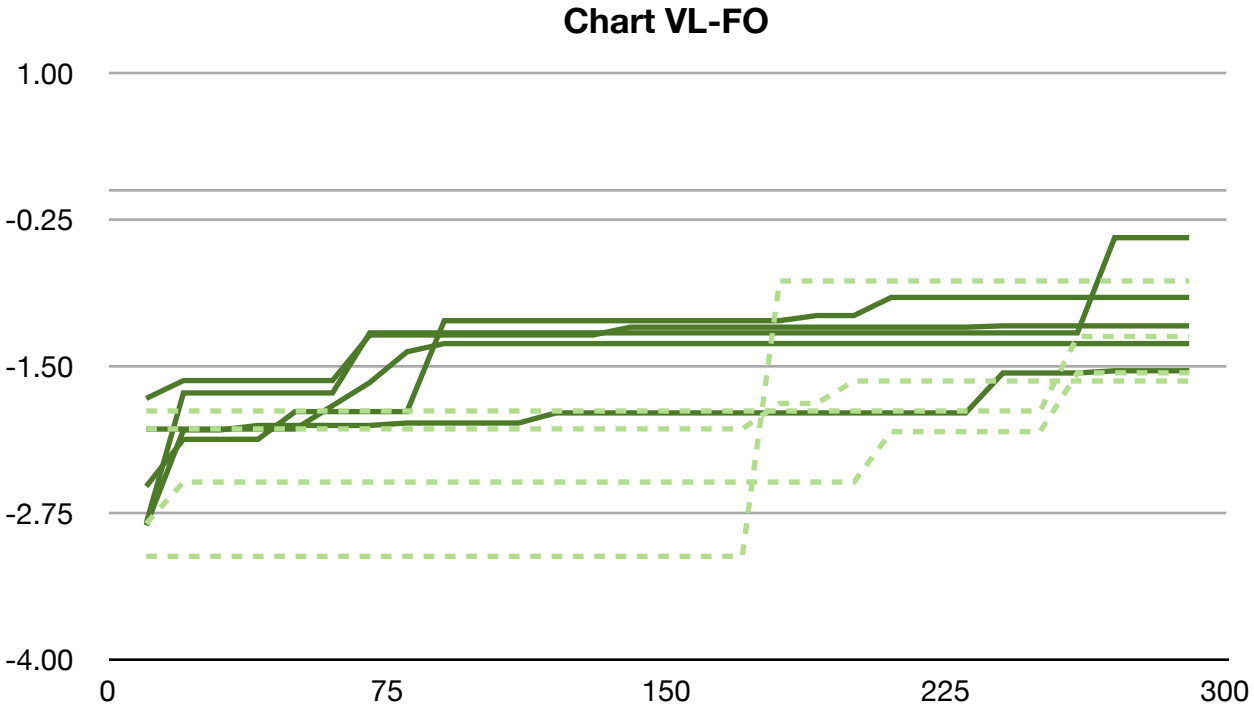


Chart VL-FO
1.00
-0.25
-1.50
-2.75
-4.00
0
75
150
225
300

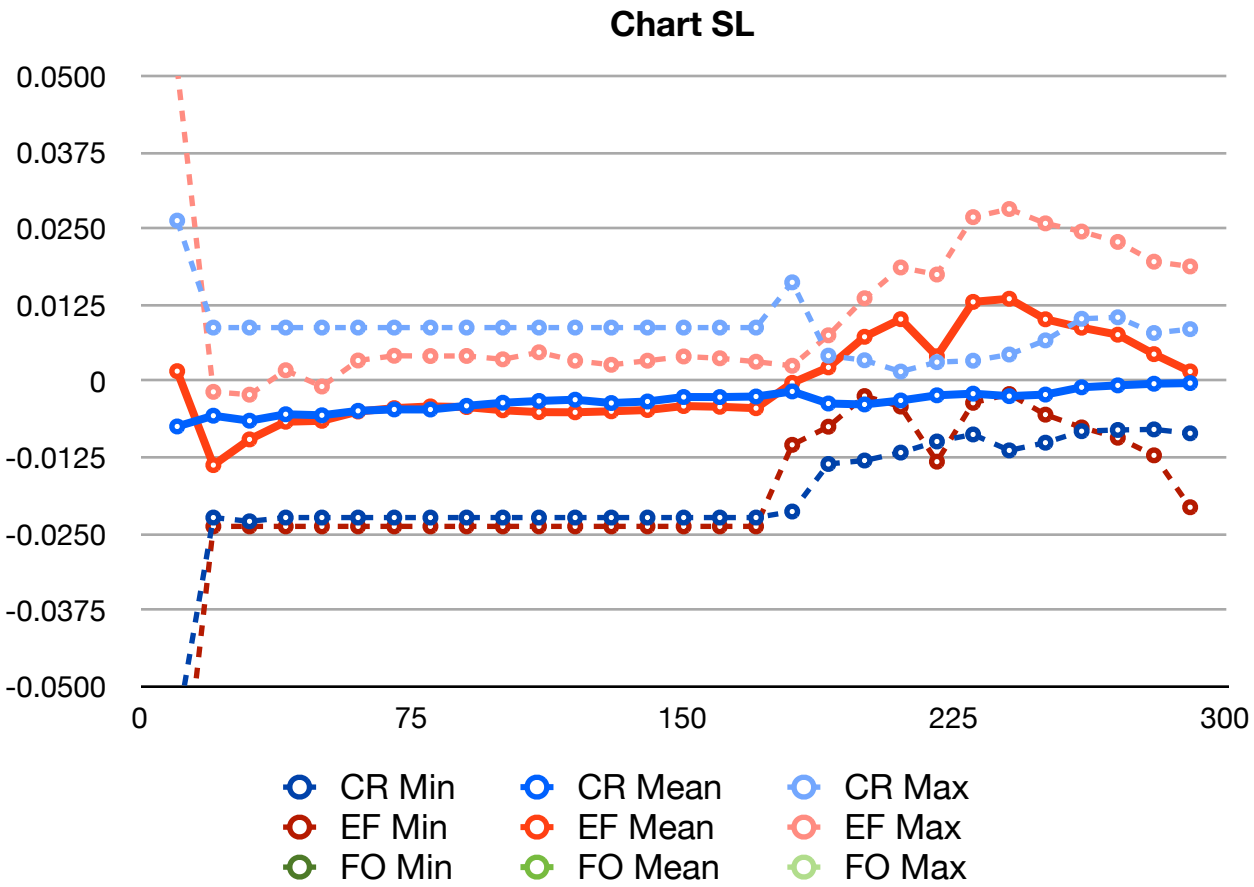


Chart SL
0.0500
0.0375
0.0250
0.0125
0
-0.0125
-0.0250
-0.0375
-0.0500
0
75
150
225
300
CR Min
CR Mean
CR Max
EF Min
EF Mean
EF Max
FO Min
FO Mean
FO Max

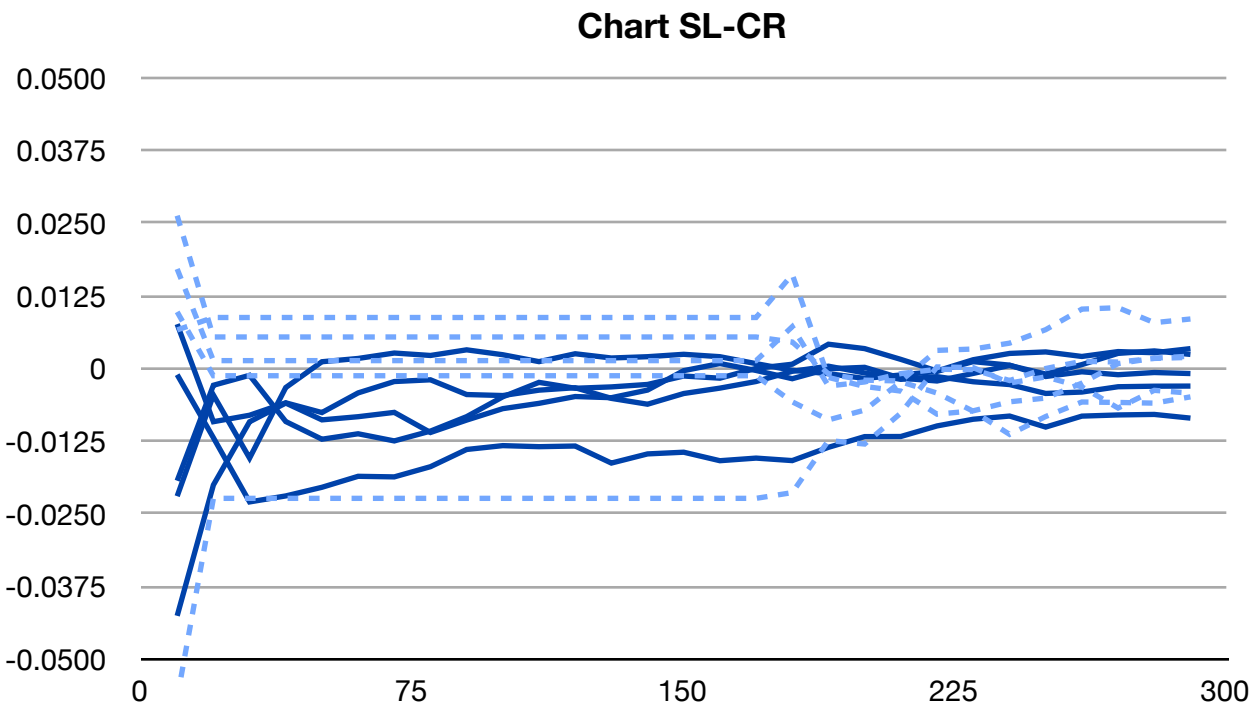


Chart SL-CR
0.0500
0.0375
0.0250
0.0125
0
-0.0125
-0.0250
-0.0375
-0.0500
0
75
150
225
300

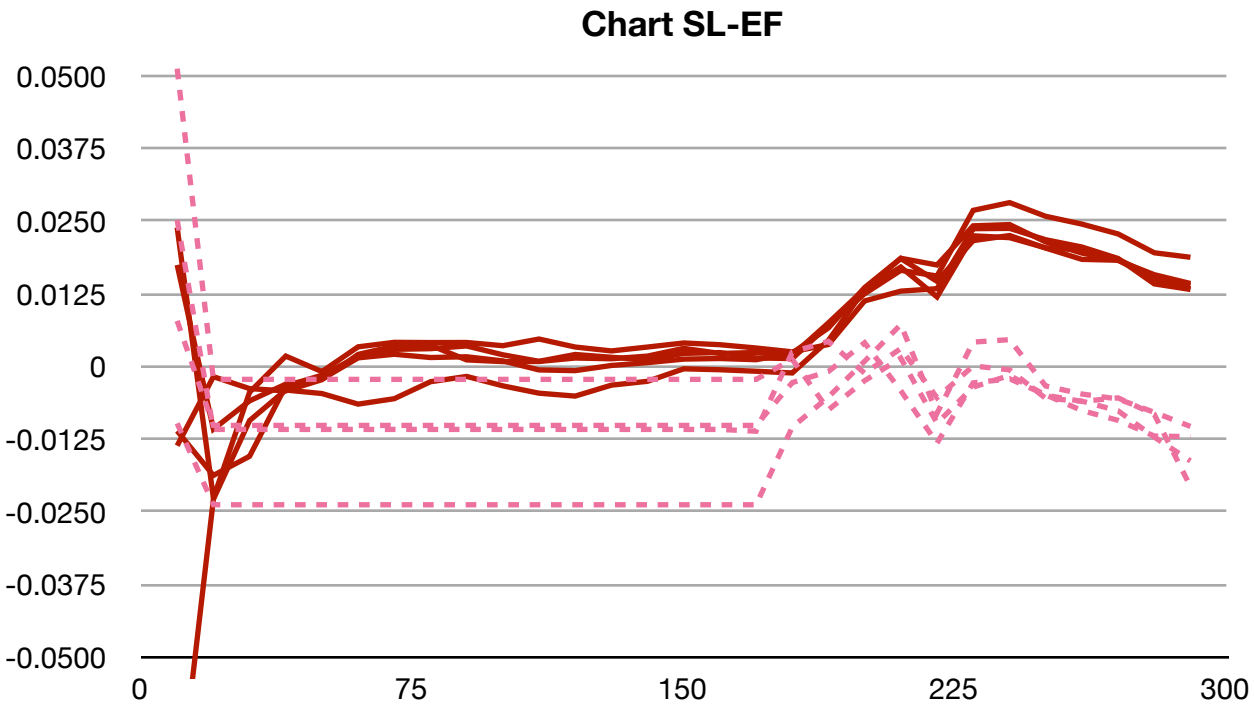


Chart SL-EF
0.0500
0.0375
0.0250
0.0125
0
-0.0125
-0.0250
-0.0375
-0.0500
0
75
150
225
300

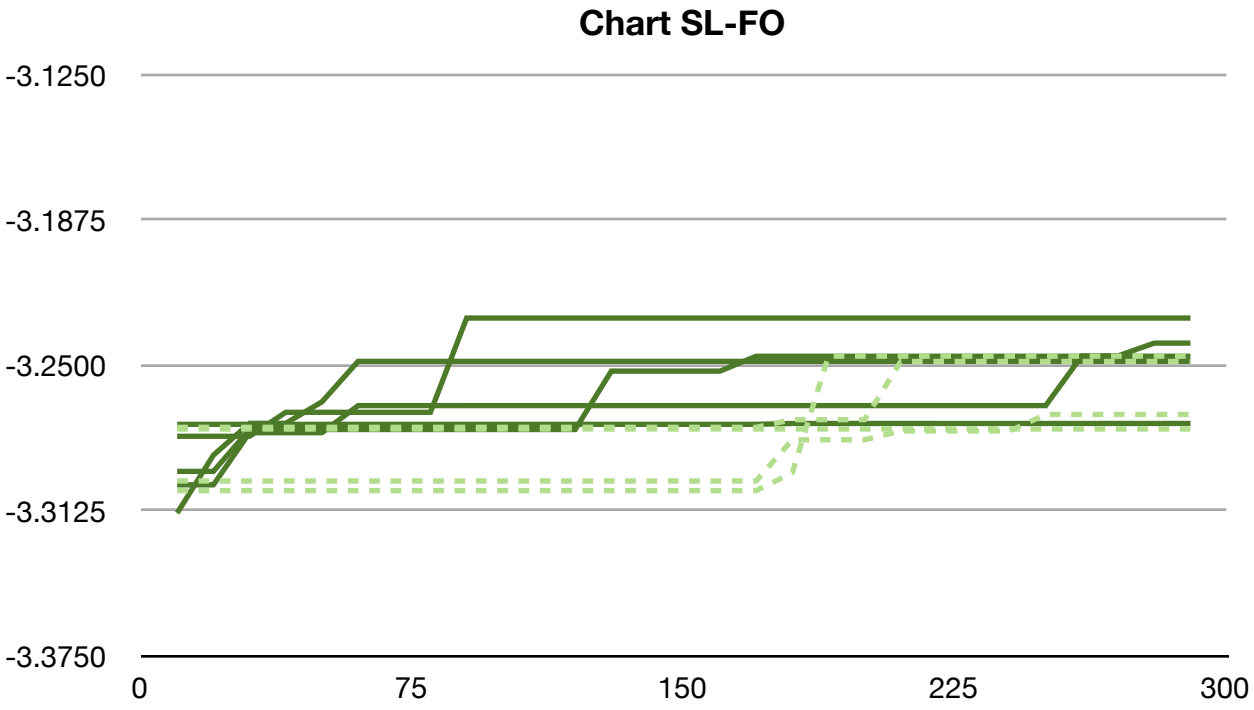


Chart SL-FO
-3.1250
-3.1875
-3.2500
-3.3125
-3.3750
0
75
150
225
300

EXPERIMENT #3: SERVER DISCONNECT

This experiment tested whether a networks running the Distributed and Non-Distributed Follower Algorithm can survive the loss of a server node. All client nodes were initialized in a connected state with one node acting as a time server. After a brief period the time server disconnected form the network and all time updates stopped. After another brief period of down-time, a new node would be picked as the server and would begin issuing time updates, and the old time server would reconnect as a normal client. This process was repeated twice, simulating two server disconnects.

Test groups are broken down first by the latency model used. For each model, two charts are shown, one for the Distributed Follower Algorithm and one for the Non-Distributed Follower Algorithm.

Tests:

1. Long-Tail Volatile Latency Model:

    - **Distributed Follower Algorithm:** Chart LTL-EF
    - **Follower Algorithm:** Chart LTL-FO

2. Volatile Latency Model:

    - **Distributed Follower Algorithm:** Chart VL-EF
    - **Follower Algorithm:** Chart VL-FO

3. Stable Latency Model:

    - **Distributed Follower Algorithm:** Chart SL-EF
    - **Follower Algorithm:** Chart SL-FO

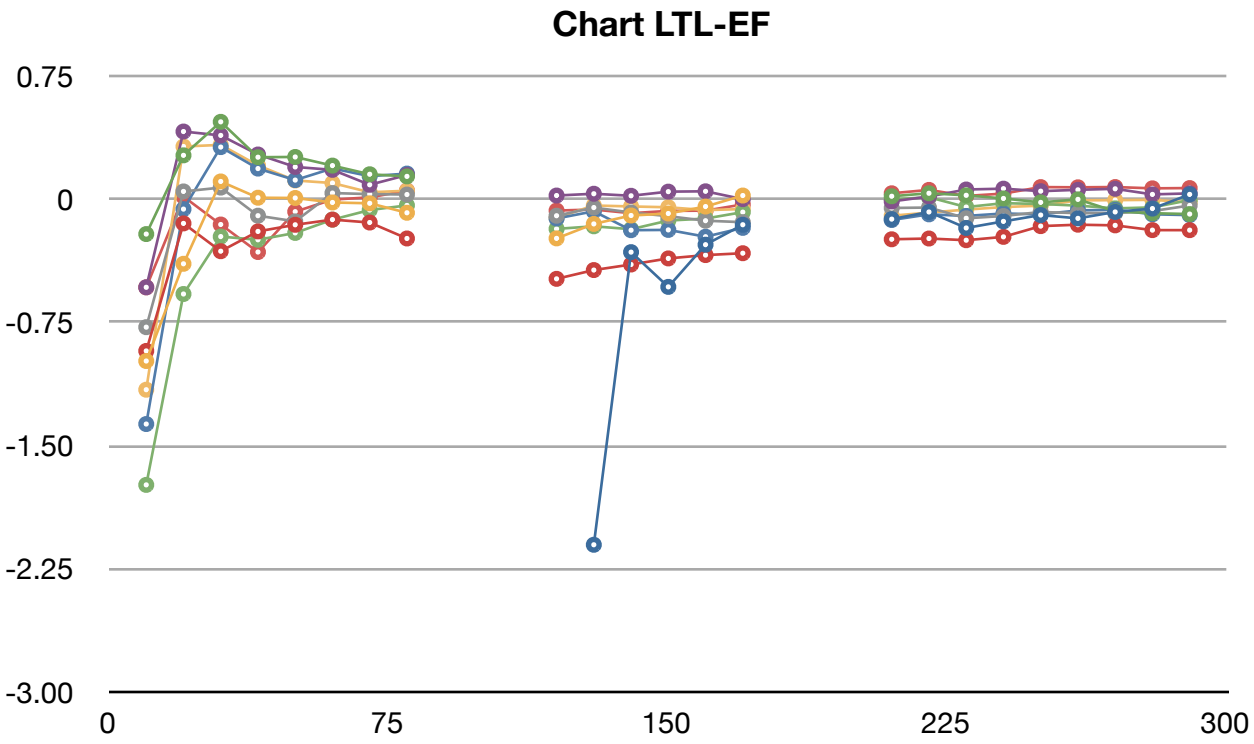


Chart LTL-EF
0.75
0
-0.75
-1.50
-2.25
-3.00
0
75
150
225
300

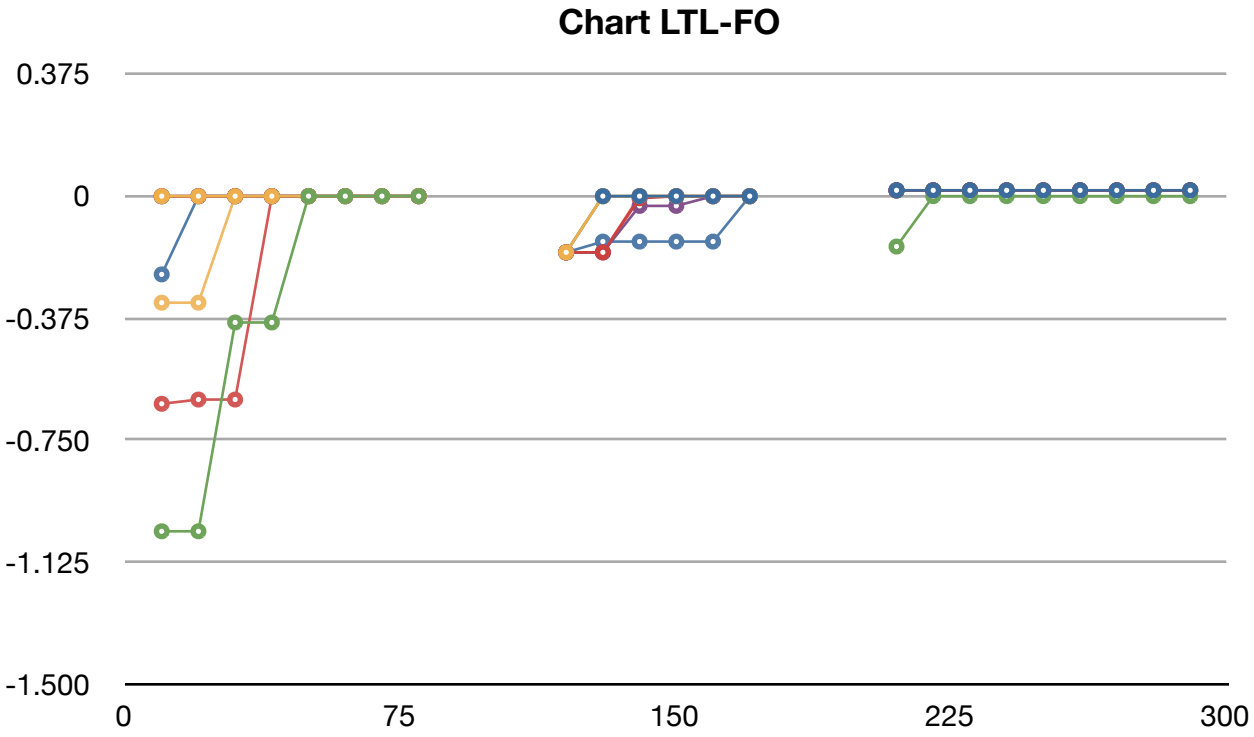


Chart LTL-FO
0.375
0
-0.375
-0.750
-1.125
-1.500
0
75
150
225
300

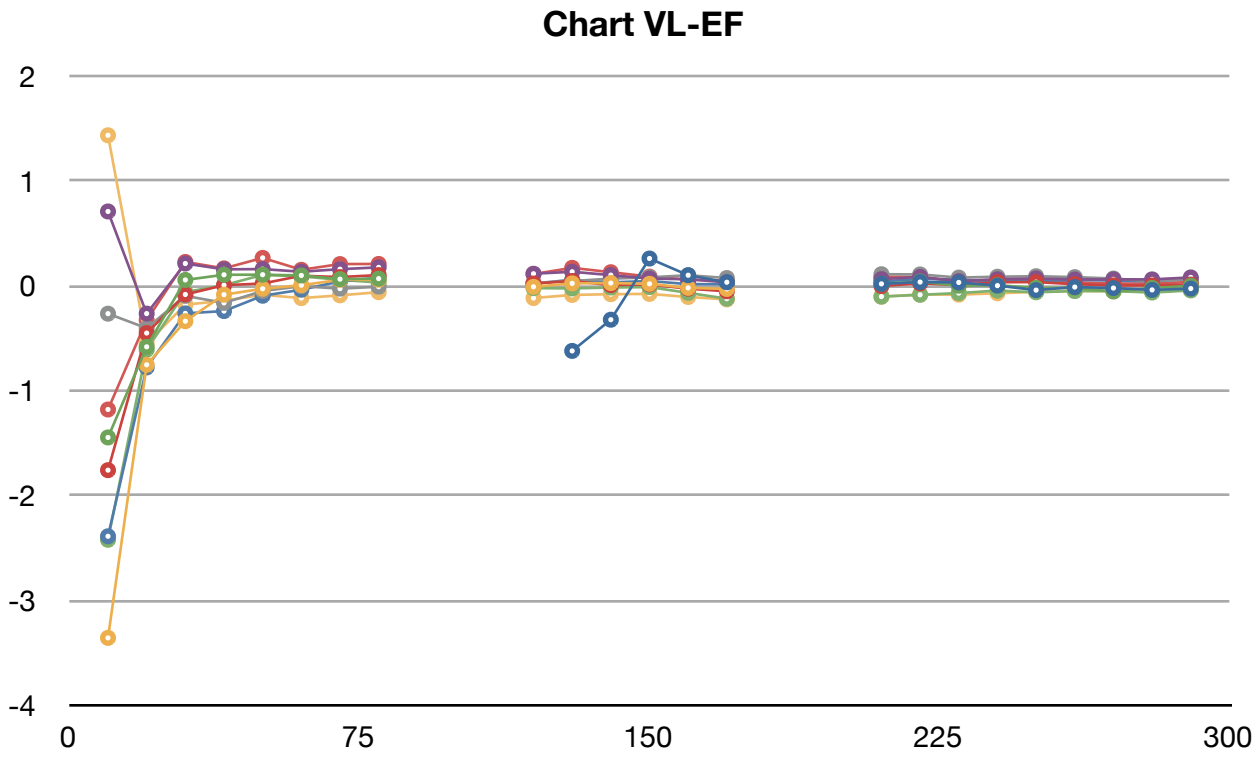


Chart VL-EF
2
1
0
-1
-2
-3
-4
0
75
150
225
300

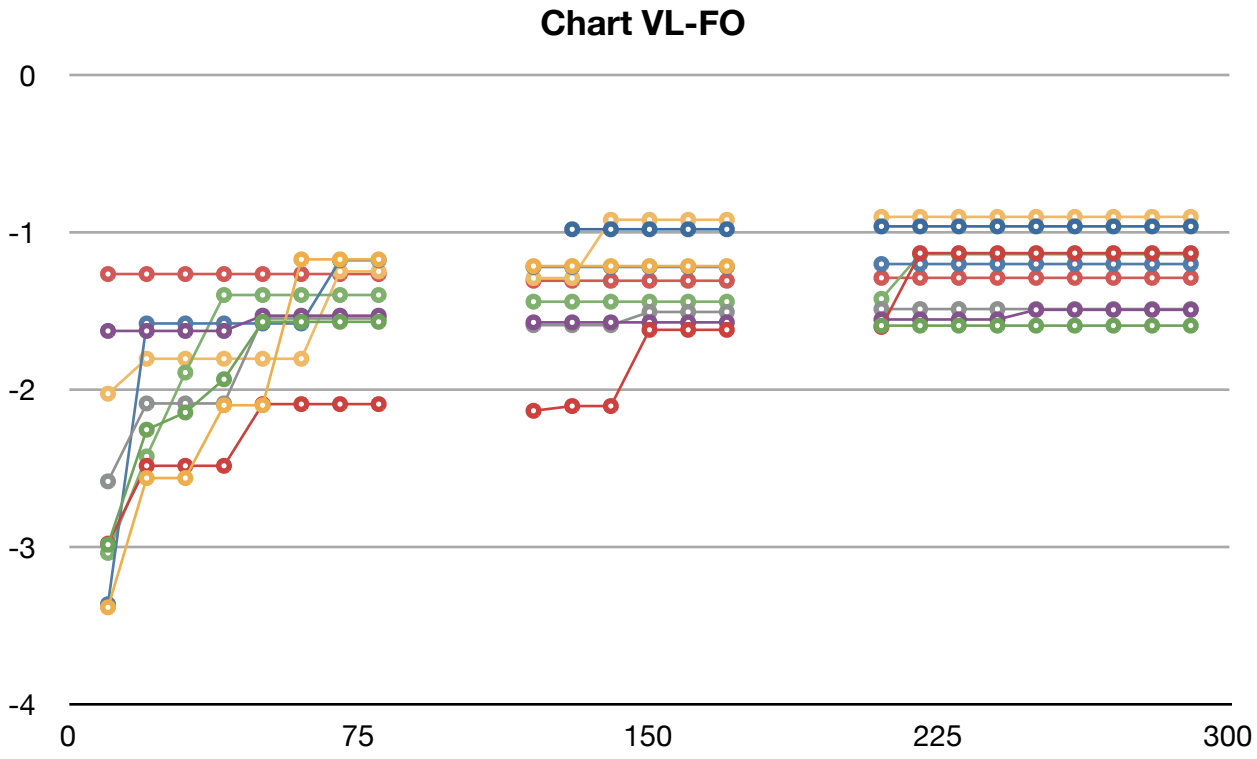


Chart VL-FO
0
-1
-2
-3
-4
0
75
150
225
300

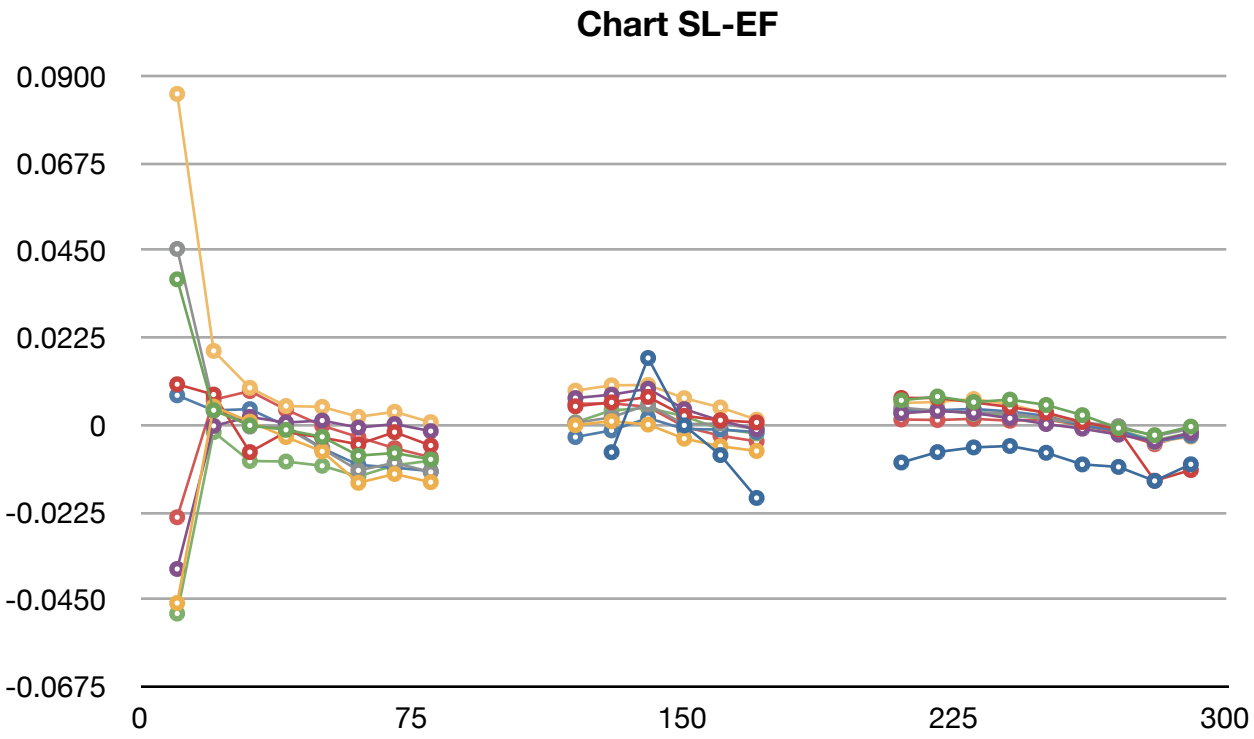


Chart SL-EF
0.0900
0.0675
0.0450
0.0225
0
-0.0225
-0.0450
-0.0675
0
75
150
225
300

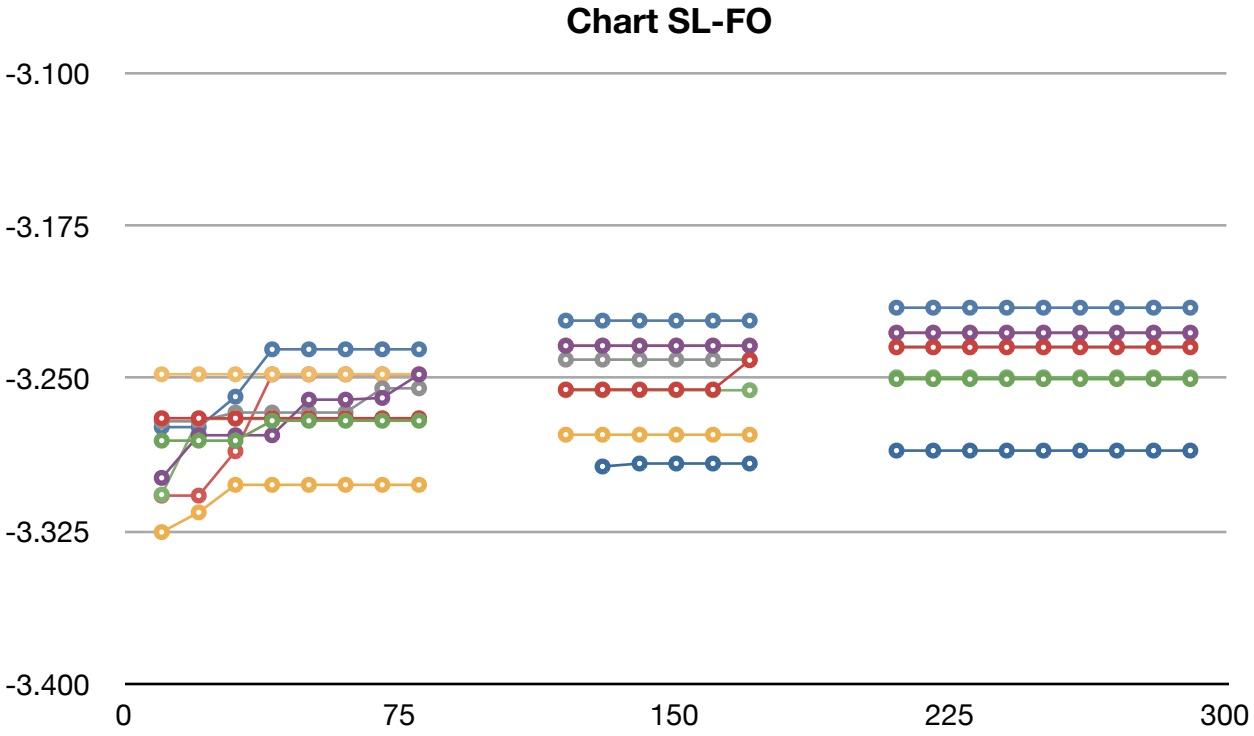


Chart SL-FO
-3.100
-3.175
-3.250
-3.325
-3.400
0
75
150
225
300

Appendix B: Simulation Implementation

MRFollowerNode Class

MRFollowerNode.h

```
//
//  MRFollowerNode.h
//  Thesis Simulations
//
//  Created by Matt Ricketson on 2/21/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRNetworkNode.h"
#import "MRTimeRecord.h"
#import "MRTimeUpdateMessage.h"

@class MRTimeRecordSet;


@interface MRFollowerNode : MRNetworkNode

#pragma mark - Server Mode
@property (atomic, assign, readonly) BOOL isServer;
@property (atomic, strong, readonly) MRTimeRecord *serverTime;

#pragma mark - Client Mode
@property (atomic, assign) NSUInteger clientMeanTimeBaseRecord;  // layer that
estimates are calculated from
@property (atomic, strong, readonly) MRTimeRecord *clientMeanTime;
@property (atomic, strong, readonly) MRTimeRecord *clientFollowerTime;
@property (atomic, assign, readonly) NSTimeInterval clientServerTimeEstimate;
@property (atomic, assign, getter = isOffline, readonly) BOOL offline;

#pragma mark - Shared
@property (atomic, assign, readonly) NSTimeInterval latencyEstimate;
@property (atomic, strong, readonly) NSMutableArray *listeners;
@property (atomic, strong, readonly) MRTimeRecordSet *recordSet;
@property (atomic, assign) NSUInteger maxBroadcastCount;

#pragma mark - Initialization
+ (id)nodeWithLayerCount:(NSUInteger)layerCount updateThreshold:
(NSTimeInterval)updateTheshold;
+ (id)nodeWithName:(NSString *)name layerCount:(NSUInteger)layerCount updateThreshold:
(NSTimeInterval)updateTheshold;

- (id)initWithLayerCount:(NSUInteger)layerCount updateThreshold:
(NSTimeInterval)updateTheshold;
- (id)initWithName:(NSString *)name layerCount:(NSUInteger)layerCount updateThreshold:
(NSTimeInterval)updateTheshold;

#pragma mark - Server Mode
- (void)becomeServerForDuration:(NSTimeInterval)duration updatePeriod:
(NSTimeInterval)period;
- (void)resignServer;

#pragma mark - Offline Mode
- (void)disconnectForDuration:(NSTimeInterval)duration;

#pragma mark - Managing Listeners
- (void)addListener:(MRNetworkNode *)listener;
```

```
@end



#pragma mark - MRTimeRecordSet

@class MRTimeRecordSet;
typedef id(^MRTimeRecordAllocator)(MRTimeRecordSet *recordSet, NSInteger index);


@interface MRTimeRecordSet : NSObject

@property (atomic, copy, readonly) NSArray *records;
@property (atomic, assign, readonly) NSTimeInterval meanRecordDifferenceEstimate;

#pragma mark - Initialization
+ (id)timeRecordSet;
+ (id)timeRecordSetWithRecordAllocationBlock:(MRTimeRecordAllocator)allocator count:
(NSInteger)count;
- (id)initWithRecordAllocationBlock:(MRTimeRecordAllocator)allocator count:
(NSInteger)count;

#pragma mark - Processing Time Updates
- (void)updateRecordsWithContentsOfMessage:(MRTimeUpdateMessage *)message;
- (BOOL)isMessageExpired:(MRTimeUpdateMessage *)message;

#pragma mark - Managing Records
- (MRStatisticalTimeRecord *)firstRecord;
- (MRStatisticalTimeRecord *)recordAtIndex:(NSUInteger)index;

#pragma mark - Analyzing Records
- (NSTimeInterval)differenceBetweenRecordsAtIndex:(NSInteger)index1 andIndex:
(NSInteger)index2;
- (NSTimeInterval)refreshMeanRecordDifferenceEstimate;

@end



#pragma mark - NSMutableArray (Shuffling)

@interface NSMutableArray (Shuffling)
- (void)shuffle;
@end
```

MRFollowerNode.m

```
//
//  MRFollowerNode.m
//  Thesis Simulations
//
//  Created by Matt Ricketson on 2/21/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRFollowerNode.h"


@interface MRFollowerNode () {
     NSDate *_messageProcessingTimeStart;
```

```
}

#pragma mark – Server Mode
@property (atomic, assign, readwrite) BOOL isServer;
@property (atomic, strong, readwrite) MRTimeRecord *serverTime;

#pragma mark – Client Mode
@property (atomic, assign, getter = isOffline, readwrite) BOOL offline;

#pragma mark – Shared
@property (atomic, strong, readwrite) MRTimeRecordSet *recordSet;
@property (atomic, strong, readwrite) NSMutableArray *listeners;

#pragma mark – Managing Listeners
- (void)broadcastMessage:(MRTimeUpdateMessage *)message;

#pragma mark – Error Reduction
- (void)beginProcessingMessage;
- (NSTimeInterval)messageProcessingTime;

@end


@implementation MRFollowerNode

@synthesize isServer = _isServer;
@dynamic clientMeanTime;
@dynamic clientFollowerTime;
@dynamic clientServerTimeEstimate;
@dynamic latencyEstimate;


#pragma mark – Initialization

+ (id)nodeWithLayerCount:(NSUInteger)layerCount updateThreshold:
(NSTimeInterval)updateTheshold
{
     return [[self alloc] initWithLayerCount:layerCount
updateThreshold:updateTheshold];
}

+ (id)nodeWithName:(NSString *)name layerCount:(NSUInteger)layerCount updateThreshold:
(NSTimeInterval)updateTheshold
{
     return [[self alloc] initWithName:name layerCount:layerCount
updateThreshold:updateTheshold];
}


- (id)initWithLayerCount:(NSUInteger)layerCount updateThreshold:
(NSTimeInterval)updateTheshold
{
     return [self initWithName:[[self class] defaultNodeName] layerCount:layerCount
updateThreshold:updateTheshold];
}

- (id)initWithName:(NSString *)name layerCount:(NSUInteger)layerCount updateThreshold:
(NSTimeInterval)updateTheshold
{
     self = [super initWithName:name];
     if (self) {
              _isServer = NO;
              _offline = NO;
```

```
            _recordSet = [MRTimeRecordSet
timeRecordSetWithRecordAllocationBlock:^id(MRTimeRecordSet *recordSet, NSInteger
index) {
                  return [MRStatisticalTimeRecord
timeRecordWithReturnType:MRStatisticalTimeRecordReturnTypeMean
updateThreshold:updateTheshold tag:index];
            } count:MAX(layerCount, 1)];

            _clientMeanTimeBaseRecord = 0;
            _maxBroadcastCount = NSIntegerMax;
    }
    return self;
}


#pragma mark - NSObject

- (NSString *)description
{
    NSMutableString *desc = [NSMutableString stringWithFormat:@"%@%s: %f + %f = %f
(MEAN + l = EST), Layers:",
                                                self.name,
                                                (self.isServer ? " (SERVER)" : ""),
                                                self.clientMeanTime.currentTime,
                                                self.latencyEstimate,
                                                self.clientServerTimeEstimate];
    NSUInteger count = self.recordSet.records.count;
    for (NSUInteger i=1; i<count; i++) {
            MRStatisticalTimeRecord *record = [self.recordSet.records
objectAtIndex:i-1];
            [desc appendFormat:@" [%f]", record.meanTime.currentTime];
            [desc appendFormat:@" <%f>", [self.recordSet
differenceBetweenRecordsAtIndex:i andIndex:i-1]];
    }
    [desc appendFormat:@" [%f]", [[self.recordSet.records objectAtIndex:count-1]
meanTime].currentTime];
    return desc;
}


#pragma mark - MRNetworkNode

- (void)receiveMessage:(MRTimeUpdateMessage *)message
{
    if (![self isOffline]) {
            [self beginProcessingMessage];

            if (!message.backtracking) {
                  if (self.isServer && message.isCristianTimeUpdate) {
                        message.timeValue = self.serverTime.currentTime;
                        [message backtrack];
                  } else {
                        [self.recordSet
updateRecordsWithContentsOfMessage:message];

                        if (![self.recordSet isMessageExpired:message])
                              [self broadcastMessage:message];
                  }
            }
    }
}


#pragma mark - Properties
```

```
- (MRTimeRecord *)clientMeanTime
{
     return ((MRStatisticalTimeRecord *)[self.recordSet
recordAtIndex:self.clientMeanTimeBaseRecord]).meanTime;
}

- (MRTimeRecord *)clientFollowerTime
{
     return ((MRStatisticalTimeRecord *)[self.recordSet firstRecord]).minTime;
}

- (NSTimeInterval)clientServerTimeEstimate
{
     return self.clientMeanTime.currentTime + self.latencyEstimate *
(self.clientMeanTimeBaseRecord + 1);
}

- (NSTimeInterval)latencyEstimate
{
     return self.recordSet.meanRecordDifferenceEstimate;
}


#pragma mark - Server Mode

- (BOOL)isServer
{
     return _isServer;
}

- (void)setIsServer:(BOOL)isServer
{
     @synchronized(self) {
               if (_isServer != isServer) {
                         _isServer = isServer;

                         if (_isServer) {
                                   self.serverTime = [MRTimeRecord timeRecord];
                                   [self.serverTime updateTime:
(self.clientServerTimeEstimate)];// + self.latencyEstimate)];
                         } else {
                                   self.serverTime = nil;
                         }
               }
     }
}

- (void)becomeServerForDuration:(NSTimeInterval)duration updatePeriod:
(NSTimeInterval)period
{
     self.isServer = YES;

     NSInteger cycles = (NSInteger)(duration/period);
     dispatch_block_t block = ^{
               printf("(%s) sending server time (%f) to listeners.\n", [self.name
UTF8String], self.serverTime.currentTime);
               for (MRNetworkNode *listener in self.listeners) {
                         [self sendMessage:[MRTimeUpdateMessage
messageWithTimeValue:self.serverTime.currentTime] toNodeNamed:listener.name];
               }
     };

//   mr_delay(period, ^{
```

```
//           block();
//           if (cycles > 0 && self.isServer) mr_delay_loop((cycles-1), period,
block);
//   });

     mr_delay_conditional_loop(cycles, period, block, ^BOOL{
               return self.isServer;
     });
}

- (void)resignServer
{
     self.isServer = NO;
}


#pragma mark - Offline Mode

- (void)disconnectForDuration:(NSTimeInterval)duration
{
     self.offline = YES;

     mr_delay(duration, ^{
               self.offline = NO;
     });
}


#pragma mark - Managing Listeners

- (void)addListener:(MRNetworkNode *)listener
{
     if (!_listeners) _listeners = [NSMutableArray array];

     if (![self.listeners containsObject:listener]) {
               [self.listeners addObject:listener];
     }
}

- (void)broadcastMessage:(MRTimeUpdateMessage *)message
{
     NSInteger count = MIN(self.maxBroadcastCount, self.listeners.count);
     for (NSUInteger i=0; i<count; i++) {
               MRNetworkNode *listener = [self.listeners objectAtIndex:i];
               if (![message hasVisitedNode:listener.name]) {
                         MRTimeUpdateMessage *msg = [message copy];
                         msg.timeValue += [self messageProcessingTime];
                         [self sendMessage:msg toNodeNamed:listener.name];
               }
     }
     [self.listeners shuffle];
}


#pragma mark - Error Reduction

- (void)beginProcessingMessage
{
     _messageProcessingTimeStart = [NSDate date];
}

- (NSTimeInterval)messageProcessingTime
{
     return -[_messageProcessingTimeStart timeIntervalSinceNow];
```

```
}


@end




#pragma mark - MRTimeRecordSet

@interface MRTimeRecordSet ()

@property (atomic, copy, readwrite) NSArray *records;
@property (atomic, assign, readwrite) NSTimeInterval meanRecordDifferenceEstimate;

@end


@implementation MRTimeRecordSet

#pragma mark - Initialization

+ (id)timeRecordSet
{
     return [self new];
}

+ (id)timeRecordSetWithRecordAllocationBlock:(MRTimeRecordAllocator)allocator count:
(NSInteger)count
{
     return [[self alloc] initWithRecordAllocationBlock:allocator count:count];
}

- (id)init
{
    return [self initWithRecordAllocationBlock:nil count:0];
}

- (id)initWithRecordAllocationBlock:(MRTimeRecordAllocator)allocator count:
(NSInteger)count
{
     self = [super init];
     if (self) {
              NSMutableArray *records = [NSMutableArray arrayWithCapacity:count];
              for (NSUInteger i=0; i<count; i++) {
                       id record = (allocator ? allocator(self, i) :
[MRStatisticalTimeRecord timeRecord]);
                       [records addObject:record];
              }
              _records = records;
     }
     return self;
}


#pragma mark - Processing Time Updates

- (void)updateRecordsWithContentsOfMessage:(MRTimeUpdateMessage *)message
{
     MRStatisticalTimeRecord *record = [self.records objectAtIndex:message.hops-1];
     [record updateTime:message.timeValue];
     [self refreshMeanRecordDifferenceEstimate];
}
```

```
- (BOOL)isMessageExpired:(MRTimeUpdateMessage *)message
{
    return (message.hops >= self.records.count);
}


#pragma mark - Managing Records

- (MRStatisticalTimeRecord *)firstRecord
{
    return [self recordAtIndex:0];
}

- (MRStatisticalTimeRecord *)recordAtIndex:(NSUInteger)index
{
    return [self.records objectAtIndex:index];
}


#pragma mark - Analyzing Records

- (NSTimeInterval)differenceBetweenRecordsAtIndex:(NSInteger)index1 andIndex:
(NSInteger)index2
{
    NSAssert(index1 < self.records.count, @"Record #1 index out of bounds");
    NSAssert(index2 < self.records.count, @"Record #2 index out of bounds");

    MRStatisticalTimeRecord *record1 = [self.records objectAtIndex:index1];
    MRStatisticalTimeRecord *record2 = [self.records objectAtIndex:index2];
    if (record1.tag < record2.tag)
            return (record1.currentTime - record2.currentTime);
    else
            return (record2.currentTime - record1.currentTime);
}

- (NSTimeInterval)refreshMeanRecordDifferenceEstimate
{
    NSInteger ri1, ri2;
    NSTimeInterval total, meanCount;
    NSInteger count = self.records.count;

    if (count < 2)
            return 0.0;

    ri1 = ri2 = -1;
    total = meanCount = 0.0;
    for (NSInteger i=0; i<count; i++) {
            MRStatisticalTimeRecord *record = [self.records objectAtIndex:i];

            if (!record.isTouched)
                    continue;

            if (ri1 == -1)
                    ri1 = i;
            else
                    ri2 = i;

            if (ri1 > -1 && ri2 > -1) {
                    total += ([self differenceBetweenRecordsAtIndex:ri1
andIndex:ri2] / (ri2 - ri1));
                    meanCount += 1.0;
                    ri1 = ri2;
                    ri2 = -1;
```

```
            }
    }

    if (total == 0.0)
            return 0.0;

    _meanRecordDifferenceEstimate = (total/meanCount);
    return _meanRecordDifferenceEstimate;
}

@end



#pragma mark - NSMutableArray (Shuffling)

@implementation NSMutableArray (Shuffling)

- (void)shuffle
{
    u_int32_t count = (u_int32_t)self.count;
    for (u_int32_t i=0; i<count; i++) {
       u_int32_t nElements = count - i;
       u_int32_t n = arc4random_uniform(nElements) + i;
       [self exchangeObjectAtIndex:i withObjectAtIndex:n];
    }
}

@end
```

MRTimeRecord Class

MRTimeRecord.h

```
//
//  MRTimeRecord.h
//  Thesis Simulations
//
//  Created by Matt Ricketson on 2/21/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import <Foundation/Foundation.h>

@class MRTimeRecord;


typedef BOOL(^MRTimeRecordUpdateCondition)(MRTimeRecord *record, NSTimeInterval
newTime);
typedef NSTimeInterval(^MRTimeRecordUpdateAction)(MRTimeRecord *record, NSTimeInterval
newTime);


#pragma mark - MRTimeRecording

@protocol MRTimeRecording <NSObject>

@property (nonatomic, assign, readonly) NSTimeInterval currentTime;  // lastUpdateTime
+ timeSinceLastUpdate
@property (nonatomic, assign, readonly) NSTimeInterval lastUpdateTime;  // the time
value recorded when last updated
@property (nonatomic, assign, readonly) BOOL isTouched;  // has been updated at least
once
@property (nonatomic, copy, readonly) NSArray *allTimeValues;  // value history
@property (nonatomic, assign, readonly) NSInteger updateCount;  // total accepted
updates
@property (nonatomic, assign, readonly) NSInteger updateAttemptCount;  // total
attempted updates
@property (nonatomic, assign, readonly) NSInteger updateAttemptsSinceLastUpdate;
@property (nonatomic, assign, readonly) NSTimeInterval timeSinceLastUpdate;
@property (nonatomic, assign, readonly) NSTimeInterval timeSinceLastUpdateAttempt;

#pragma mark - Updating Time
- (void)resetTime;  // sresets time properties
- (BOOL)updateTime:(NSTimeInterval)time;  // returns yes if updated

@end


#pragma mark - MRTimeRecord

@interface MRTimeRecord : NSObject <MRTimeRecording>

@property (nonatomic, copy) MRTimeRecordUpdateCondition updateCondition;  // decides
whether an update should be accepted
@property (nonatomic, copy) MRTimeRecordUpdateAction updateAction;  // implements the
update

#pragma mark - Initialization
+ (id)timeRecord;
+ (id)timeRecordWithUpdateCondition:(MRTimeRecordUpdateCondition)updateCondition
updateAction:(MRTimeRecordUpdateAction)updateAction;
- (id)initWithUpdateCondition:(MRTimeRecordUpdateCondition)updateCondition
updateAction:(MRTimeRecordUpdateAction)updateAction;
```

```
#pragma mark - Common Time Records
+ (id)minTimeRecordUsingCurrentTime:(BOOL)useCurrentTime updateThreshold:
(NSTimeInterval)updateThreshold;
+ (id)maxTimeRecordUsingCurrentTime:(BOOL)useCurrentTime updateThreshold:
(NSTimeInterval)updateThreshold;
+ (id)meanTimeRecordUsingCurrentTime:(BOOL)useCurrentTime;

#pragma mark - Output
- (NSString *)stringFromCurrentTime;
- (NSString *)stringFromLastUpdateTime;

@end


#pragma mark - MRStatisticalTimeRecord

typedef NS_ENUM(NSUInteger, MRStatisticalTimeRecordReturnType) {
    MRStatisticalTimeRecordReturnTypeMostRecent,
    MRStatisticalTimeRecordReturnTypeMinimum,
    MRStatisticalTimeRecordReturnTypeMaximum,
    MRStatisticalTimeRecordReturnTypeMean
};

@interface MRStatisticalTimeRecord : NSObject <MRTimeRecording>

@property (nonatomic, strong, readonly) MRTimeRecord *mostRecentTime;
@property (nonatomic, strong, readonly) MRTimeRecord *minTime;
@property (nonatomic, strong, readonly) MRTimeRecord *maxTime;
@property (nonatomic, strong, readonly) MRTimeRecord *meanTime;
@property (nonatomic, assign) MRStatisticalTimeRecordReturnType returnType;
@property (nonatomic, assign) NSInteger tag;

#pragma mark - Initialization
+ (id)timeRecord;
+ (id)timeRecordWithReturnType:(MRStatisticalTimeRecordReturnType)returnType
updateThreshold:(NSTimeInterval)updateTheshold tag:(NSInteger)tag;
- (id)initWithReturnType:(MRStatisticalTimeRecordReturnType)returnType
updateThreshold:(NSTimeInterval)updateTheshold tag:(NSInteger)tag;
- (id)initWithReturnType:(MRStatisticalTimeRecordReturnType)returnType useCurrentTime:
(BOOL)useCurrentTime updateThreshold:(NSTimeInterval)updateTheshold tag:
(NSInteger)tag;

#pragma mark - Output
- (NSString *)stringFromCurrentTime;
- (NSString *)stringFromLastUpdateTime;

@end
```

MRTimeRecord.m

```
//
//  MRTimeRecord.m
//  Thesis Simulations
//
//  Created by Matt Ricketson on 2/21/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRTimeRecord.h"


@interface MRTimeRecord () {
```

```
    NSDate *_dateLastUpdated;
    NSDate *_dateLastUpdateAttempted;
    NSMutableArray *_allTimeValues;
}

@property (nonatomic, assign, readwrite) NSTimeInterval lastUpdateTime;
@property (nonatomic, assign, readwrite) NSInteger updateCount;
@property (nonatomic, assign, readwrite) NSInteger updateAttemptCount;
@property (nonatomic, assign, readwrite) NSInteger updateAttemptsSinceLastUpdate;

@end


@implementation MRTimeRecord

@dynamic currentTime;
@dynamic isTouched;
@dynamic allTimeValues;
@dynamic timeSinceLastUpdate;
@dynamic timeSinceLastUpdateAttempt;


#pragma mark - Initialization

+ (id)timeRecord
{
    return [[self alloc] init];
}

+ (id)timeRecordWithUpdateCondition:(MRTimeRecordUpdateCondition)updateCondition
updateAction:(MRTimeRecordUpdateAction)updateAction
{
    return [[self alloc] initWithUpdateCondition:updateCondition
updateAction:updateAction];
}

- (id)init
{
    self = [super init];
    if (self) {
              _updateCondition = nil;
              _updateAction = nil;

              [self resetTime];
    }
    return self;
}

- (id)initWithUpdateCondition:(MRTimeRecordUpdateCondition)updateCondition
updateAction:(MRTimeRecordUpdateAction)updateAction
{
    self = [self init];
    if (self) {
              _updateCondition = updateCondition;
              _updateAction = updateAction;
    }
    return self;
}


#pragma mark - Common Time Records

+ (id)minTimeRecordUsingCurrentTime:(BOOL)useCurrentTime updateThreshold:
(NSTimeInterval)updateThreshold
```

```
{
    return [MRTimeRecord timeRecordWithUpdateCondition:^BOOL(MRTimeRecord *record,
NSTimeInterval newTime) {
             if (!record.isTouched) return YES;
             NSTimeInterval time = (useCurrentTime ? record.currentTime :
record.lastUpdateTime);
             NSTimeInterval diff = newTime - time;
             return (fabs(diff) > updateThreshold && diff > 0.0);
    } updateAction:nil];
}

+ (id)maxTimeRecordUsingCurrentTime:(BOOL)useCurrentTime updateThreshold:
(NSTimeInterval)updateThreshold
{
    return [MRTimeRecord timeRecordWithUpdateCondition:^BOOL(MRTimeRecord *record,
NSTimeInterval newTime) {
             if (!record.isTouched) return YES;
             NSTimeInterval time = (useCurrentTime ? record.currentTime :
record.lastUpdateTime);
             NSTimeInterval diff = newTime - time;
             return (fabs(diff) > updateThreshold && diff > 0.0);
    } updateAction:nil];
}

+ (id)meanTimeRecordUsingCurrentTime:(BOOL)useCurrentTime
{
    return [MRTimeRecord timeRecordWithUpdateCondition:nil
updateAction:^NSTimeInterval(MRTimeRecord *record, NSTimeInterval newTime) {
             NSTimeInterval time = (useCurrentTime ? record.currentTime :
record.lastUpdateTime);
             return (time*record.updateCount + newTime) / (record.updateCount + 1);
    }];
}


#pragma mark - Output

- (NSString *)description
{
    return [self stringFromCurrentTime];
}

- (NSString *)stringFromCurrentTime
{
    return [NSString stringWithFormat:@"%f", self.currentTime];
}

- (NSString *)stringFromLastUpdateTime
{
    return [NSString stringWithFormat:@"%f", self.lastUpdateTime];
}


#pragma mark - Dynamic Properties

- (NSTimeInterval)currentTime
{
    if (_dateLastUpdated) {
             return (_lastUpdateTime + self.timeSinceLastUpdate);
    }
    return 0.0;
}

- (BOOL)isTouched
```

```
{
    return (_dateLastUpdated != nil);
}

- (NSArray *)allTimeValues
{
    return [_allTimeValues copy];
}

- (NSTimeInterval)timeSinceLastUpdate
{
    if (_dateLastUpdated)
            return -[_dateLastUpdated timeIntervalSinceNow];
    return 0.0;
}

- (NSTimeInterval)timeSinceLastUpdateAttempt
{
    if (_dateLastUpdateAttempted)
            return -[_dateLastUpdateAttempted timeIntervalSinceNow];
    return 0.0f;
}


#pragma mark - Updating Time

- (void)resetTime
{
    _dateLastUpdated = nil;
    _dateLastUpdateAttempted = nil;
    _allTimeValues = [NSMutableArray array];

    _lastUpdateTime = 0.0;
    _updateCount = 0;
    _updateAttemptCount = 0;
    _updateAttemptsSinceLastUpdate = 0;
}

- (BOOL)updateTime:(NSTimeInterval)time
{
    _dateLastUpdateAttempted = [NSDate date];
    self.updateAttemptCount++;
    self.updateAttemptsSinceLastUpdate++;

    BOOL shouldUpdate = YES;
    if (self.updateCondition) {
            shouldUpdate = self.updateCondition(self, time);
    }

    if (shouldUpdate) {
            NSTimeInterval updateValue = time;
            if (self.updateAction) {
                    updateValue = self.updateAction(self, time);
            }

            _lastUpdateTime = updateValue;
            _dateLastUpdated = _dateLastUpdateAttempted;

            [_allTimeValues insertObject:@(_lastUpdateTime) atIndex:0];
            self.updateCount++;
            self.updateAttemptsSinceLastUpdate = 0;
    }

    return shouldUpdate;
```

```
}

@end




#pragma mark - Time Record Group

#define MRStatisticalTimeRecordReturn(method) \
(self.returnType == MRStatisticalTimeRecordReturnTypeMostRecent ? [self.mostRecentTime
method] : \
(self.returnType == MRStatisticalTimeRecordReturnTypeMinimum ? [self.minTime method] :
\
(self.returnType == MRStatisticalTimeRecordReturnTypeMaximum ? [self.maxTime method] :
\
(self.returnType == MRStatisticalTimeRecordReturnTypeMean ? [self.meanTime method] :
0))))

@interface MRStatisticalTimeRecord ()

@property (nonatomic, strong, readwrite) MRTimeRecord *mostRecentTime;
@property (nonatomic, strong, readwrite) MRTimeRecord *minTime;
@property (nonatomic, strong, readwrite) MRTimeRecord *maxTime;
@property (nonatomic, strong, readwrite) MRTimeRecord *meanTime;

@end


@implementation MRStatisticalTimeRecord

#pragma mark - Initialization

+ (id)timeRecord
{
    return [self new];
}

+ (id)timeRecordWithReturnType:(MRStatisticalTimeRecordReturnType)returnType
updateThreshold:(NSTimeInterval)updateTheshold tag:(NSInteger)tag
{
    return [[self alloc] initWithReturnType:returnType updateThreshold:updateTheshold
tag:tag];
}

- (id)init
{
    return [self initWithReturnType:MRStatisticalTimeRecordReturnTypeMean
updateThreshold:0.0 tag:0];
}

- (id)initWithReturnType:(MRStatisticalTimeRecordReturnType)returnType
updateThreshold:(NSTimeInterval)updateTheshold tag:(NSInteger)tag
{
    return [self initWithReturnType:returnType useCurrentTime:YES
updateThreshold:updateTheshold tag:tag];
}

- (id)initWithReturnType:(MRStatisticalTimeRecordReturnType)returnType useCurrentTime:
(BOOL)useCurrentTime updateThreshold:(NSTimeInterval)updateTheshold tag:(NSInteger)tag
{
    self = [super init];
```

```
    if (self) {
            _mostRecentTime = [MRTimeRecord timeRecord];
        _minTime = [MRTimeRecord minTimeRecordUsingCurrentTime:useCurrentTime
updateThreshold:updateTheshold];
            _maxTime = [MRTimeRecord maxTimeRecordUsingCurrentTime:useCurrentTime
updateThreshold:updateTheshold];
            _meanTime = [MRTimeRecord meanTimeRecordUsingCurrentTime:useCurrentTime];
            _returnType = returnType;
            _tag = tag;
    }
    return self;
}


#pragma mark - Output

- (NSString *)description
{
    return [self stringFromCurrentTime];
}

- (NSString *)stringFromCurrentTime
{
    return [NSString stringWithFormat:@"%f", self.currentTime];
}

- (NSString *)stringFromLastUpdateTime
{
    return [NSString stringWithFormat:@"%f", self.lastUpdateTime];
}


#pragma mark - NSTimeRecording

- (NSTimeInterval)currentTime
    { return MRStatisticalTimeRecordReturn(currentTime); }
- (NSTimeInterval)lastUpdateTime                                        { return
MRStatisticalTimeRecordReturn(lastUpdateTime); }
-
(BOOL)isTouched
{ return MRStatisticalTimeRecordReturn(isTouched); }
- (NSArray *)allTimeValues
    { return MRStatisticalTimeRecordReturn(allTimeValues); }
- (NSInteger)updateCount
    { return MRStatisticalTimeRecordReturn(updateCount); }
- (NSInteger)updateAttemptCount
    { return MRStatisticalTimeRecordReturn(updateAttemptCount); }
- (NSInteger)updateAttemptsSinceLastUpdate                              { return
MRStatisticalTimeRecordReturn(updateAttemptsSinceLastUpdate); }
- (NSTimeInterval)timeSinceLastUpdate                                   { return
MRStatisticalTimeRecordReturn(timeSinceLastUpdate); }
- (NSTimeInterval)timeSinceLastUpdateAttempt                        { return
MRStatisticalTimeRecordReturn(timeSinceLastUpdateAttempt); }

- (void)resetTime
{
    [self.mostRecentTime resetTime];
    [self.minTime resetTime];
    [self.maxTime resetTime];
    [self.meanTime resetTime];
}

- (BOOL)updateTime:(NSTimeInterval)time
{
```

```
    BOOL mostRecentTimeUpdated = [self.mostRecentTime updateTime:time];
    BOOL minTimeUpdated = [self.minTime updateTime:time];
    BOOL maxTimeUpdated = [self.maxTime updateTime:time];
    BOOL meanTimeUpdated = [self.meanTime updateTime:time];

    switch (self.returnType) {
            case MRStatisticalTimeRecordReturnTypeMostRecent: return
mostRecentTimeUpdated;
            case MRStatisticalTimeRecordReturnTypeMinimum: return minTimeUpdated;
            case MRStatisticalTimeRecordReturnTypeMaximum: return maxTimeUpdated;
            case MRStatisticalTimeRecordReturnTypeMean: return meanTimeUpdated;
    }
}

@end
```

MRTimeUpdateMessage Class

MRTimeUpdateMessage.h

```
//
//  MRUpdateTimeMessage.h
//  Thesis Simulations
//
//  Created by Matt Ricketson on 2/25/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRNetworkMessage.h"

@interface MRTimeUpdateMessage : MRNetworkMessage

@property (nonatomic, assign, readwrite) NSTimeInterval timeValue;
@property (nonatomic, assign, readonly) BOOL latencyEstimateIsValid;
@property (nonatomic, assign, readonly) NSTimeInterval latencyEstimate;
@property (nonatomic, assign) BOOL isCristianTimeUpdate;

+ (id)messageWithTimeValue:(NSTimeInterval)timeValue;
- (id)initWithTimeValue:(NSTimeInterval)timeValue;

- (void)backtrackWithLatencyEstimate:(NSTimeInterval)latencyEstimate;

@end
```

MRTimeUpdateMessage.m

```
//
//  MRUpdateTimeMessage.m
//  Thesis Simulations
//
//  Created by Matt Ricketson on 2/25/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRTimeUpdateMessage.h"
#import "MRNetworkDirectory.h"


@interface MRTimeUpdateMessage ()

@property (nonatomic, assign, readwrite) BOOL latencyEstimateIsValid;
@property (nonatomic, assign, readwrite) NSTimeInterval latencyEstimate;

@end


@implementation MRTimeUpdateMessage

- (id)init
{
    self = [super init];
    if (self) {
            _timeValue = 0.0f;
            _latencyEstimate = 0.0f;
            _isCristianTimeUpdate = NO;
    }
    return self;
}
```

```
+ (id)messageWithTimeValue:(NSTimeInterval)timeValue
{
    return [[self alloc] initWithTimeValue:timeValue];
}

- (id)initWithTimeValue:(NSTimeInterval)timeValue
{
    self = [self init];
    if (self) {
             _timeValue = timeValue;
    }
    return self;
}

- (id)copyWithZone:(NSZone *)zone
{
    MRTimeUpdateMessage *message = [super copyWithZone:zone];
    message.timeValue = self.timeValue;
    message.latencyEstimate = self.latencyEstimate;
    message.isCristianTimeUpdate = self.isCristianTimeUpdate;
    return message;
}

- (void)backtrack
{
    self.latencyEstimateIsValid = NO;
    [super backtrack];
}
- (void)backtrackWithLatencyEstimate:(NSTimeInterval)latencyEstimate
{
    self.latencyEstimate = latencyEstimate;
    self.latencyEstimateIsValid = YES;
    [super backtrack];
}

@end
```

MRNetworkMessage Class

MRNetworkMessage.h

```
//
//  MRNetworkMessage.h
//  Thesis Simulations
//
//  Created by Matt Ricketson on 11/17/12.
//  Copyright (c) 2012 Matt Ricketson. All rights reserved.
//

#import <Foundation/Foundation.h>

@class MRNetworkNode;

@interface MRNetworkMessage : NSObject <NSCopying>

@property (nonatomic, copy, readonly) NSString *uuid;
@property (nonatomic, copy, readonly) NSString *sender;
@property (nonatomic, copy, readonly) NSString *recipient;
@property (nonatomic, strong, readonly) NSDate *sent;
@property (nonatomic, strong, readonly) NSDate *received;
@property (nonatomic, assign, readonly) NSUInteger hops;
@property (nonatomic, assign, readonly) NSTimeInterval latency;
@property (nonatomic, assign, readonly) BOOL backtracking;

+ (id)message;

- (void)willBeSentToNode:(MRNetworkNode *)recipient fromNode:(MRNetworkNode *)sender;
- (void)wasReceived;

- (void)backtrack;
- (BOOL)hasVisitedNode:(NSString *)node;

@end
```

MRNetworkMessage.m

```
//
//  MRNetworkMessage.m
//  Thesis Simulations
//
//  Created by Matt Ricketson on 11/17/12.
//  Copyright (c) 2012 Matt Ricketson. All rights reserved.
//

#import "MRNetworkMessage.h"
#import "MRNetworkDirectory.h"
#import "MRNetworkNode.h"


@interface MRNetworkMessage () {
    NSString *_uuid;
}

@property (nonatomic, copy, readwrite) NSString *sender;
@property (nonatomic, copy, readwrite) NSString *recipient;
@property (nonatomic, strong, readwrite) NSDate *sent;
@property (nonatomic, strong, readwrite) NSDate *received;
@property (nonatomic, strong, readwrite) NSMutableArray *backtrackPath;
@property (nonatomic, assign, readwrite) BOOL backtracking;
```

```
@end


@implementation MRNetworkMessage

+ (id)message
{
    return [[self alloc] init];
}

- (id)init
{
    self = [super init];
    if (self) {
              _sender = nil;
              _recipient = nil;
              _sent = nil;
              _received = nil;
              _backtrackPath = [NSMutableArray array];
              _backtracking = NO;
    }
    return self;
}

- (id)copyWithZone:(NSZone *)zone
{
    MRNetworkMessage *message = [[[self class] allocWithZone:zone] init];
    message.sender = [self.sender copyWithZone:zone];
    message.recipient = [self.recipient copyWithZone:zone];
    message.sent = [self.sent copyWithZone:zone];
    message.received = [self.received copyWithZone:zone];
    message.backtrackPath = [self.backtrackPath mutableCopyWithZone:zone];
    message.backtracking = self.backtracking;
    return message;
}

- (NSString *)uuid
{
    if (!_uuid) {
        CFUUIDRef uuid = CFUUIDCreate(NULL);
        CFStringRef str = CFUUIDCreateString(NULL, uuid);
        CFRelease(uuid);
              _uuid = CFBridgingRelease(str);
    }
    return _uuid;
}

- (NSTimeInterval)latency
{
    if (self.sent && self.received) {
              return [self.received timeIntervalSinceDate:self.sent];
    }
    return 0.0f;
}

- (NSUInteger)hops
{
    return [self.backtrackPath count];
}

- (void)willBeSentToNode:(MRNetworkNode *)recipient fromNode:(MRNetworkNode *)sender
{
```

```
    if (!recipient || !sender) return;
    self.sender = sender.name;
    self.recipient = recipient.name;
    self.sent = [NSDate date];

    if (self.backtracking) {
            if ([self.recipient isEqualToString:[self.backtrackPath lastObject]])
                  [_backtrackPath removeLastObject];
            else
                  self.backtracking = NO;
    }
}

- (void)wasReceived
{
    if (!self.recipient) return;
    self.received = [NSDate date];

    if (!self.backtracking)
            [self.backtrackPath addObject:self.sender];
}

- (void)backtrack
{
    if (self.hops == 0) return;
    self.backtracking = YES;
    NSString *backtrackNode = [_backtrackPath lastObject]; // go to the new last
object, the one before self.recipient
    [[MRNetworkDirectory sharedDirectory] sendMessage:self toNodeNamed:backtrackNode
fromNodeNamed:self.recipient];
}

- (BOOL)hasVisitedNode:(NSString *)node
{
    return ([self.recipient isEqualToString:node] || [self.backtrackPath
containsObject:node]);
}

@end
```

MRCristianTimeServerNode Class

MRCristianTimeServerNode.h

```
//
//  MRCristianTimeServerNode.h
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 3/16/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRNetworkNode.h"
#import "MRTimeUpdateMessage.h"

@interface MRCristianTimeServerNode : MRNetworkNode

@property (nonatomic, assign, readonly) NSTimeInterval serverTime;

- (void)startTiming;

@end
```

MRCristianTimeServerNode.m

```
//
//  MRCristianTimeServerNode.m
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 3/16/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRCristianTimeServerNode.h"


@interface MRCristianTimeServerNode ()

@property (nonatomic, strong, readwrite) NSDate *startDate;

@end



@implementation MRCristianTimeServerNode

@dynamic serverTime;

- (NSTimeInterval)serverTime
{
    if (self.startDate) {
            return -[self.startDate timeIntervalSinceNow];
    }
    return 0.0f;
}

- (void)startTiming
{
    self.startDate = [NSDate date];
}

- (void)receiveMessage:(MRTimeUpdateMessage *)message
{
```

```
    if (message.isCristianTimeUpdate && !message.backtracking) {
            message.timeValue = self.serverTime;
            [message backtrack];
    }
}

@end
```

MRCristianTimeClientNode Class

MRCristianTimeClientNode.h

```
//
//  MRCristianTimeClientNode.h
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 3/16/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRNetworkNode.h"
#import "MRCristianTimeServerNode.h"
#import "MRTimeRecord.h"

@interface MRCristianTimeClientNode : MRNetworkNode

@property (atomic, strong, readonly) MRTimeRecord *minTime;
@property (atomic, strong, readonly) MRTimeRecord *meanTime;
@property (atomic, assign, readonly) NSTimeInterval serverTime;
@property (atomic, assign, readonly) NSTimeInterval timeSinceLastUpdate;
@property (atomic, assign, readonly) NSInteger cyclesSinceLastUpdate;
@property (atomic, assign, readonly) NSInteger updateCount;
@property (atomic, assign, readonly) NSTimeInterval initialUpdateJumpTime;
@property (atomic, assign, readonly) NSTimeInterval totalUpdateJumpTime;
@property (atomic, assign, readwrite) NSTimeInterval updateJumpThreshold;
@property (atomic, assign, getter = isOffline, readonly) BOOL offline;

+ (id)nodeWithUpdateThreshold:(NSTimeInterval)updateTheshold;
- (id)initWithUpdateThreshold:(NSTimeInterval)updateTheshold;

+ (id)nodeWithName:(NSString *)name updateThreshold:(NSTimeInterval)updateTheshold;
- (id)initWithName:(NSString *)name updateThreshold:(NSTimeInterval)updateTheshold;

- (void)startRequestingTimeFromServer:(NSString *)serverName;
- (void)switchToServer:(NSString *)serverName;

#pragma mark - Offline Mode
- (void)disconnectForDuration:(NSTimeInterval)duration;

@end
```

MRCristianTimeClientNode.m

```
//
//  MRCristianTimeClientNode.m
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 3/16/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRCristianTimeClientNode.h"
#import "MRTimeUpdateMessage.h"


@interface MRCristianTimeClientNode ()

@property (atomic, strong, readwrite) MRTimeRecord *minTime;
@property (atomic, strong, readwrite) MRTimeRecord *meanTime;
@property (atomic, assign, readwrite) NSTimeInterval lastServerTime;
```

```
@property (atomic, strong, readwrite) NSDate *dateUpdated;
@property (atomic, assign, readwrite) NSInteger cyclesSinceLastUpdate;
@property (atomic, assign, readwrite) NSInteger updateCount;
@property (atomic, assign, readwrite) NSTimeInterval initialUpdateJumpTime;
@property (atomic, assign, readwrite) NSTimeInterval totalUpdateJumpTime;
@property (atomic, assign, getter = isOffline, readwrite) BOOL offline;

@property (atomic, copy) NSString *serverName;
@property (atomic, strong) NSDate *dateLastRequested;

- (void)requestTimeUpdate;

@end


@implementation MRCristianTimeClientNode

@dynamic serverTime;
@dynamic timeSinceLastUpdate;


+ (id)nodeWithUpdateThreshold:(NSTimeInterval)updateTheshold
{
    return [[self alloc] initWithUpdateThreshold:updateTheshold];
}

- (id)initWithUpdateThreshold:(NSTimeInterval)updateTheshold
{
    return [self initWithName:[[self class] defaultNodeName]
updateThreshold:updateTheshold];
}

+ (id)nodeWithName:(NSString *)name updateThreshold:(NSTimeInterval)updateTheshold
{
    return [[self alloc] initWithName:name updateThreshold:updateTheshold];
}

- (id)initWithName:(NSString *)name updateThreshold:(NSTimeInterval)updateTheshold
{
    self = [super initWithName:name];
    if (self) {
              _serverName = nil;
              _dateLastRequested = nil;
              _updateJumpThreshold = updateTheshold;
              _minTime = [MRTimeRecord minTimeRecordUsingCurrentTime:YES
updateThreshold:updateTheshold];
              _meanTime = [MRTimeRecord meanTimeRecordUsingCurrentTime:YES];
    }
    return self;
}


- (NSString *)description
{
    NSMutableString *desc = [NSMutableString stringWithFormat:@"%@: TE %f (%ld)",
self.name, self.serverTime, self.meanTime.updateCount];
    return desc;
}


- (NSTimeInterval)serverTime
{
//  if (self.dateUpdated) {
//            return (self.lastServerTime + [self timeSinceLastUpdate]);
```

```
//  }
//  return 0.0f;
    return self.meanTime.currentTime;
}

- (NSTimeInterval)timeSinceLastUpdate
{
    if (self.dateUpdated) {
            return -[self.dateUpdated timeIntervalSinceNow];
    }
    return 0.0f;
}

- (void)receiveMessage:(MRTimeUpdateMessage *)message
{
    if (self.offline) return;

    if (message.isCristianTimeUpdate && message.backtracking &&
self.dateLastRequested) {
            NSTimeInterval estimatedServerTime = message.timeValue -
[self.dateLastRequested timeIntervalSinceNow]/2.0f;
//            NSTimeInterval updateJumpTime = estimatedServerTime - self.serverTime;

//            [self.minTime updateTime:estimatedServerTime];
            if ([self.meanTime updateTime:estimatedServerTime])
                  [self requestTimeUpdate];

//            BOOL updated = NO;
//            if (self.dateUpdated == nil || updateJumpTime > self.updateJumpThreshold)
{
//                  if (self.initialUpdateJumpTime == 0.0f) self.initialUpdateJumpTime
= updateJumpTime;
//                  self.totalUpdateJumpTime += updateJumpTime;
//                  self.lastServerTime = estimatedServerTime;
//                  self.dateUpdated = [NSDate date];
//                  updated = YES;
//                  self.cyclesSinceLastUpdate = 0;
//                  self.updateCount++;
//                  [self requestTimeUpdate];
//            } else {
//                  self.cyclesSinceLastUpdate++;
//            }
    }
}

- (void)startRequestingTimeFromServer:(NSString *)serverName
{
    self.serverName = serverName;
    [self requestTimeUpdate];
}

- (void)switchToServer:(NSString *)serverName
{
    self.serverName = serverName;
}

- (void)requestTimeUpdate
{
    self.dateLastRequested = [NSDate date];

    MRTimeUpdateMessage *msg = [MRTimeUpdateMessage message];
    msg.isCristianTimeUpdate = YES;

    [self sendMessage:msg toNodeNamed:self.serverName];
```

```
}


#pragma mark - Offline Mode

- (void)disconnectForDuration:(NSTimeInterval)duration
{
    self.offline = YES;

    mr_delay(duration, ^{
              self.offline = NO;
              [self requestTimeUpdate];
    });
}


@end
```

MRNetworkNode Class

MRNetworkNode.h

```
//
//  MRNetworkNode.h
//  Thesis Simulations
//
//  Created by Matt Ricketson on 11/17/12.
//  Copyright (c) 2012 Matt Ricketson. All rights reserved.
//

#import <Foundation/Foundation.h>
#import "MRNetworkMessage.h"

@interface MRNetworkNode : NSObject

@property (atomic, copy, readonly) NSString *name;
@property (atomic, strong, readwrite) dispatch_queue_t dispatchQueue;

+ (NSString *) defaultNodeName;

+ (id) nodeWithName:(NSString *)name;
- (id) initWithName:(NSString *)name;

- (void) sendMessage:(MRNetworkMessage *)message toNodeNamed:(NSString *)recipient;
- (void) receiveMessage:(MRNetworkMessage *)message;

@end
```

MRNetworkNode.m

```
//
//  MRNetworkNode.m
//  Thesis Simulations
//
//  Created by Matt Ricketson on 11/17/12.
//  Copyright (c) 2012 Matt Ricketson. All rights reserved.
//

#import "MRNetworkNode.h"
#import "MRNetworkDirectory.h"


@interface MRNetworkNode ()



@end


@implementation MRNetworkNode

static NSInteger __nodeCount = -1;

+ (NSString *) defaultNodeName
{
    return [NSString stringWithFormat:@"%@-%ld", NSStringFromClass([self class]),
(__nodeCount+1)];
}

- (id)init
```

```
{
    return [self initWithName:[[self class] defaultNodeName]];
}

+ (id)nodeWithName:(NSString *)name
{
    return [[self alloc] initWithName:name];
}

- (id)initWithName:(NSString *)name
{
    self = [super init];
    if (self) {
              __nodeCount++;
              _name = name;

              _dispatchQueue = dispatch_queue_create([[NSString
stringWithFormat:@"com.node.%@",_name] UTF8String], DISPATCH_QUEUE_CONCURRENT);
    }
    return self;
}

- (NSString *)description
{
    return [NSString stringWithFormat:@"<%@; name=\"%@\">",[super
description],self.name];
}



- (void) sendMessage:(MRNetworkMessage *)message toNodeNamed:(NSString *)recipient
{
//  dispatch_async(self.dispatchQueue, ^{
              [[MRNetworkDirectory sharedDirectory] sendMessage:message
toNodeNamed:recipient fromNodeNamed:self.name];
//  });
}

- (void) receiveMessage:(MRNetworkMessage *)message
{
//  dispatch_async(self.dispatchQueue, ^{
//            MRLog(@"(%@) message received: %@",self.name,message);
//  });
}

@end
```

MRNetworkDirectory Class

MRNetworkDirectory.h

```
//
//  MRNetworkDirectory.h
//  Thesis Simulations
//
//  Created by Matt Ricketson on 11/17/12.
//  Copyright (c) 2012 Matt Ricketson. All rights reserved.
//

#import <Foundation/Foundation.h>
#import "MRNetworkNode.h"
#import "MRNetworkMessage.h"
#import "MRNetworkLatencyModel.h"

@interface MRNetworkDirectory : NSObject

+ (MRNetworkDirectory *) sharedDirectory;

@property (nonatomic, assign, readonly) NSTimeInterval
currentTheoreticalNetworkLatency;
@property (nonatomic, assign, readonly) NSTimeInterval currentAverageNetworkLatency;
@property (nonatomic, assign, readwrite) NSInteger latencyAverageSampleSize;
@property (nonatomic, strong, readwrite) id<MRNetworkLatencyModel> latencyModel;

- (void)reset;

- (id)registerNode:(MRNetworkNode *)node;

- (void)sendMessage:(MRNetworkMessage *)message toNodeNamed:(NSString *)recipient
fromNodeNamed:(NSString *)sender;
- (void)sendMessage:(MRNetworkMessage *)message toNode:(MRNetworkNode *)recipient
fromNode:(MRNetworkNode *)sender;

@end
```

MRNetworkDirectory.m

```
//
//  MRNetworkDirectory.m
//  Thesis Simulations
//
//  Created by Matt Ricketson on 11/17/12.
//  Copyright (c) 2012 Matt Ricketson. All rights reserved.
//

#import "MRNetworkDirectory.h"
#import "MRConstantLatencyModel.h"


@interface MRNetworkDirectory () {
    NSMutableArray *_messageQueue;
    BOOL _messageLoopRunning;
}

@property (nonatomic, assign, readwrite) NSTimeInterval currentAverageNetworkLatency;
@property (nonatomic, strong, readwrite) NSMutableArray *latencyMeasurements;
@property (nonatomic, strong, readwrite) NSMutableDictionary *directory;

- (void)recordLatency:(NSTimeInterval)latency;
```

```
- (void)addMessageToQueue:(MRNetworkMessage *)message;

@end


@implementation MRNetworkDirectory

static MRNetworkDirectory *__sharedDirectory = nil;

+ (MRNetworkDirectory *)sharedDirectory
{
    if (!__sharedDirectory)
            __sharedDirectory = [[MRNetworkDirectory alloc] init];
    return __sharedDirectory;
}

@dynamic currentTheoreticalNetworkLatency;

- (id)init
{
    self = [super init];
    if (self) {
        [self reset];
    }
    return self;
}

- (void)reset
{
    _directory = [NSMutableDictionary dictionary];
    _latencyMeasurements = [NSMutableArray array];
    _latencyAverageSampleSize = 10;
    _latencyModel = [MRConstantLatencyModel modelWithLatency:1.0f];
    _messageQueue = [NSMutableArray array];
    _messageLoopRunning = NO;
}


- (id) registerNode:(MRNetworkNode *)node
{
    if (![self.directory objectForKey:node.name]) {
            [self.directory setObject:node forKey:node.name];
            return node;
    }
    return nil;
}

- (void) sendMessage:(MRNetworkMessage *)message toNodeNamed:(NSString *)recipient
fromNodeNamed:(NSString *)sender
{
    MRNetworkNode *senderNode = [self.directory objectForKey:sender];
    MRNetworkNode *recipientNode = [self.directory objectForKey:recipient];
    [self sendMessage:message toNode:recipientNode fromNode:senderNode];
}
- (void) sendMessage:(MRNetworkMessage *)message toNode:(MRNetworkNode *)recipient
fromNode:(MRNetworkNode *)sender
{
    if (sender && recipient) {
        [message willBeSentToNode:recipient fromNode:sender];

             mr_delay([self currentTheoreticalNetworkLatency], ^{
            [message wasReceived];
                   [recipient receiveMessage:message];
                   [self recordLatency:message.latency];
```

```
            });
    }
}

- (void)addMessageToQueue:(MRNetworkMessage *)message
{
    // TODO: implement message queue to replace mr_delay
}

- (void) recordLatency:(NSTimeInterval)latency
{
    [self.latencyMeasurements insertObject:@(latency) atIndex:0];
    while (self.latencyMeasurements.count > self.latencyAverageSampleSize) {
            [self.latencyMeasurements removeLastObject];
    }

    NSTimeInterval totalLatency = 0.0f;
    for (NSNumber *latency in self.latencyMeasurements) {
            totalLatency += (NSTimeInterval)[latency floatValue];
    }
    self.currentAverageNetworkLatency = (NSTimeInterval)(totalLatency/
(NSTimeInterval)self.latencyMeasurements.count);
}

- (NSTimeInterval)currentTheoreticalNetworkLatency
{
    if (self.latencyModel) {
            return [self.latencyModel currentLatency];
    }
    return 1.0f;
}


@end
```

MRNetworkLatencyModel Protocol

MRNetworkLatencyModel.h

```
//
//  MRNetworkLatencyModel.h
//  Thesis Simulations
//
//  Created by Matt Ricketson on 11/17/12.
//  Copyright (c) 2012 Matt Ricketson. All rights reserved.
//

#import <Foundation/Foundation.h>

@protocol MRNetworkLatencyModel <NSObject>

- (NSTimeInterval) currentLatency;

@end
```

MRNormalDistrLatencyModel Class

MRNormalDistrLatencyModel.h

```
//
//  MRNormalDistrLatencyModel.h
//  Thesis Simulations
//
//  Created by Matt Ricketson on 11/19/12.
//  Copyright (c) 2012 Matt Ricketson. All rights reserved.
//

#import <Foundation/Foundation.h>

@interface MRNormalDistrLatencyModel : NSObject

@property (nonatomic, assign, readwrite) NSTimeInterval mean;
@property (nonatomic, assign, readwrite) NSTimeInterval standardDeviation;

+ (id) modelWithMean:(NSTimeInterval)mean standardDeviation:
(NSTimeInterval)standardDeviation;
- (id) initWithMean:(NSTimeInterval)mean standardDeviation:
(NSTimeInterval)standardDeviation;

@end
```

MRNormalDistrLatencyModel.m

```
//
//  MRNormalDistrLatencyModel.m
//  Thesis Simulations
//
//  Created by Matt Ricketson on 11/19/12.
//  Copyright (c) 2012 Matt Ricketson. All rights reserved.
//

#import "MRNormalDistrLatencyModel.h"


@implementation MRNormalDistrLatencyModel

+ (id) modelWithMean:(NSTimeInterval)mean standardDeviation:
(NSTimeInterval)standardDeviation
{
    return [[self alloc] initWithMean:mean standardDeviation:standardDeviation];
}

- (id) initWithMean:(NSTimeInterval)mean standardDeviation:
(NSTimeInterval)standardDeviation
{
    self = [super init];
    if (self) {
            _mean = mean;
            _standardDeviation = standardDeviation;
    }
    return self;
}

- (id) init
{
    return [self initWithMean:1.0f standardDeviation:0.2f];
}
```

```
- (NSString *)description
{
    return [NSString stringWithFormat:@"Normal Distribution (mean = %f, stdDev = %f)",
_mean, _standardDeviation];
}


- (NSTimeInterval) currentLatency
{
    return (NSTimeInterval)box_muller_bounded(self.mean, self.standardDeviation,
0.001f);
}

@end
```

MRSimulation Class

MRSimulation.h

```
//
//  MRSimulation.h
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 3/10/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import <Foundation/Foundation.h>

extern NSTimeInterval MRSimulationDurationForever;
extern NSTimeInterval MRSimulationUpdateLogPeriodNever;


@interface MRSimulation : NSObject

@property (nonatomic, copy) NSString *title;
@property (nonatomic, assign) NSTimeInterval duration;
@property (nonatomic, assign) NSTimeInterval updateLogPeriod;
@property (nonatomic, strong, readonly) NSMutableString *outputLog;

#pragma mark - Initialization
+ (id)simulation;
+ (id)simulationWithTitle:(NSString *)title duration:(NSTimeInterval)duration;
- (id)initWithTitle:(NSString *)title duration:(NSTimeInterval)duration;

#pragma mark - Running
- (void)run;
- (void)stop;
- (void)setUp;
- (void)simulate;
- (void)tearDown;

#pragma mark - Output
- (void)printHeader;
- (void)printUpdateLog;

#pragma mark - Helpers
- (void)runBlockCycleWithBlock:(void(^)(void))block delay:(NSTimeInterval)delay
cycles:(NSInteger)cycles;
- (BOOL)waitForCompletion:(NSTimeInterval)timeoutSecs;

@end
```

MRSimulation.m

```
//
//  MRSimulation.m
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 3/10/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRSimulation.h"

NSTimeInterval MRSimulationDurationForever = DBL_MAX;
NSTimeInterval MRSimulationUpdateLogPeriodNever = -1;
```

```
@interface MRSimulation () {
    BOOL _asyncDone;
}

@property (nonatomic, strong, readwrite) NSMutableString *outputLog;

#pragma mark - Output
- (void)startUpdateLog;

@end


@implementation MRSimulation

#pragma mark - Initialization

+ (id)simulation
{
    return [[self alloc] init];
}

+ (id)simulationWithTitle:(NSString *)title duration:(NSTimeInterval)duration
{
    return [[self alloc] initWithTitle:title duration:duration];
}


- (id)init
{
    return [self initWithTitle:NSStringFromClass([self class])
duration:MRSimulationDurationForever];
}

- (id)initWithTitle:(NSString *)title duration:(NSTimeInterval)duration
{
    self = [super init];
    if (self) {
              _title = title;
              _duration = duration;
              _updateLogPeriod = MRSimulationUpdateLogPeriodNever;
              _outputLog = [NSMutableString string];
    }
    return self;
}


#pragma mark - Running

- (void)run
{
    printf("%sBegin Simulation: %s\n%s",
                LOG_DIVIDER,
                [self.title UTF8String],
                LOG_DIVIDER);

    [self setUp];
    [self printHeader];
    printf("%s\n", LOG_DIVIDER);

    [self startUpdateLog];
    [self simulate];
```

```
    [self waitForCompletion:self.duration];
    [self tearDown];

    [self waitForCompletion:5.0];

    printf("\n%sEnd Simulation: %s\n%s\n\n\n\n\n\n\n\n\n",
                LOG_DIVIDER,
                [self.title UTF8String],
                LOG_DIVIDER);

    [self waitForCompletion:2.0];
}

- (void)stop
{
    _asyncDone = YES;
}

- (void)setUp
{
    _asyncDone = NO;
}

- (void)simulate
{
    printf("simulate\n");
}

- (void)tearDown
{
    self.outputLog = [NSMutableString string];
}


#pragma mark - Output

- (void)printHeader
{
    printf("Update Log Period = ");
    if (self.updateLogPeriod == MRSimulationUpdateLogPeriodNever) {
              printf("Never\n");
    } else {
              printf("%s\n", [[NSString stringWithFormat:@"%ld seconds",
(NSInteger)self.updateLogPeriod] UTF8String]);
    }
}

- (void)startUpdateLog
{
    if (self.updateLogPeriod != MRSimulationUpdateLogPeriodNever) {
              NSInteger cycles = (NSInteger)(self.duration/self.updateLogPeriod)-2;
              mr_delay_loop(cycles, self.updateLogPeriod, ^{
                    [self printUpdateLog];
              });
    }
}

- (void)printUpdateLog
{
    // abstract
}


#pragma mark - Helpers
```

```
- (void)runBlockCycleWithBlock:(void(^)(void))block delay:(NSTimeInterval)delay
cycles:(NSInteger)cycles
{
    mr_delay(delay, ^{
              block();
              if (cycles > 0) [self runBlockCycleWithBlock:block delay:delay cycles:
(cycles-1)];
    });
}

- (BOOL)waitForCompletion:(NSTimeInterval)timeoutSecs
{
    NSDate *timeoutDate = [NSDate dateWithTimeIntervalSinceNow:timeoutSecs];

    do {
        [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode
beforeDate:timeoutDate];
        if([timeoutDate timeIntervalSinceNow] < 0.0f)
            break;
    } while (!_asyncDone);

    return _asyncDone;
}

@end
```

MRNetworkSimulation Class

MRNetworkSimulation.h

```
//
//  MRNetworkSimulation.h
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 3/10/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRSimulation.h"
#import "MRNetworkDirectory.h"
#import "MRConstantLatencyModel.h"
#import "MRRandomLatencyModel.h"
#import "MRNormalDistrLatencyModel.h"


@interface MRNetworkSimulation : MRSimulation

@property (nonatomic, strong) id<MRNetworkLatencyModel> latencyModel;

@end
```

MRNetworkSimulation.m

```
//
//  MRNetworkSimulation.m
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 3/10/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRNetworkSimulation.h"


@interface MRNetworkSimulation ()

@end



@implementation MRNetworkSimulation

#pragma mark - Initialization

- (id)initWithTitle:(NSString *)title duration:(NSTimeInterval)duration
{
    self = [super initWithTitle:title duration:duration];
    if (self) {
              NSTimeInterval mean = 4.0f;
              NSTimeInterval stdDev = 0.5f;
              _latencyModel = [MRNormalDistrLatencyModel modelWithMean:mean
standardDeviation:stdDev];
    }
    return self;
}


#pragma mark - Simulation
```

```
- (void)setUp
{
    [super setUp];
    [[MRNetworkDirectory sharedDirectory] reset];
    [MRNetworkDirectory sharedDirectory].latencyAverageSampleSize = 100;
    [MRNetworkDirectory sharedDirectory].latencyModel = self.latencyModel;
}

- (void)printHeader
{
    [super printHeader];
    printf("Latency Model = %s\n\n", [[[MRNetworkDirectory
sharedDirectory].latencyModel description] UTF8String]);
}

- (void)tearDown
{
    [[MRNetworkDirectory sharedDirectory] reset];
    [super tearDown];
}

@end
```

MRUnifiedAnomalySimulation Class

MRUnifiedAnomalySimulation.h

```
//
//  MRUnifiedAnomalySimulation.h
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 4/4/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRNetworkSimulation.h"
#import "MRUnifiedServerNode.h"
#import "MRFollowerNode.h"
#import "MRCristianTimeClientNode.h"

typedef NS_ENUM(NSUInteger, MRUnifiedSimulationMode) {
    MRUnifiedSimulationModeNormal,
    MRUnifiedSimulationModeClientDisconnect,
    MRUnifiedSimulationModeServerDisconnect
};


@interface MRUnifiedAnomalySimulation : MRNetworkSimulation

@property (nonatomic, assign) MRUnifiedSimulationMode simulationMode;
@property (nonatomic, assign) NSTimeInterval serverCyclePeriod;
@property (nonatomic, assign) NSUInteger nodeCount;
@property (nonatomic, assign) NSUInteger nodeLayerCount;
@property (nonatomic, assign) NSTimeInterval nodeUpdateThreshold;
@property (nonatomic, assign) NSUInteger nodeMeanTimeBaseRecord;
@property (nonatomic, assign) NSUInteger nodeMaxBroadcastCount;

@property (nonatomic, assign) NSTimeInterval clientDisconnectStartTime;
@property (nonatomic, assign) NSTimeInterval clientDisconnectDuration;

@property (nonatomic, copy) NSArray *serverDisconnectStartTimes;
@property (nonatomic, copy) NSArray *serverDisconnectDurations;

@end
```

MRUnifiedAnomalySimulation.m

```
//
//  MRUnifiedAnomalySimulation.m
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 4/4/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import "MRUnifiedAnomalySimulation.h"


@interface MRUnifiedAnomalySimulation () {
    NSUInteger _serverIndex;
}

@property (nonatomic, weak) MRFollowerNode *server;
@property (nonatomic, strong) NSMutableArray *cristianNodes;
@property (nonatomic, strong) NSMutableArray *followerNodes;
```

```
@end


@implementation MRUnifiedAnomalySimulation

#pragma mark - Simulation

- (void)setUp
{
    [super setUp];

    //// Create the nodes

    if (self.simulationMode != MRUnifiedSimulationModeServerDisconnect) {
              // Cristian
              self.cristianNodes = [NSMutableArray arrayWithCapacity:self.nodeCount];
              for (NSUInteger i=0; i<self.nodeCount; i++) {
                    MRCristianTimeClientNode *node = [[MRNetworkDirectory
sharedDirectory] registerNode:

[MRCristianTimeClientNode nodeWithUpdateThreshold:self.nodeUpdateThreshold]];
                    [self.cristianNodes addObject:node];
              }
    }

    // Follower
    self.followerNodes = [NSMutableArray arrayWithCapacity:self.nodeCount];
    for (NSUInteger i=0; i<self.nodeCount; i++) {
              MRFollowerNode *node = [[MRNetworkDirectory sharedDirectory]
registerNode:
                                                        [MRFollowerNode
nodeWithLayerCount:self.nodeLayerCount updateThreshold:self.nodeUpdateThreshold]];
              node.clientMeanTimeBaseRecord = self.nodeMeanTimeBaseRecord;
              node.maxBroadcastCount = self.nodeMaxBroadcastCount;
              [self.followerNodes addObject:node];
    }

    // Build the followe network edges
    for (MRFollowerNode *node in self.followerNodes) {
              for (MRFollowerNode *node2 in self.followerNodes) {
                    if (node != node2) [node addListener:node2];
              }
    }

    // Set the server as the first node
    _serverIndex = 0;
    self.server = [self.followerNodes objectAtIndex:_serverIndex];


    //// Output
    // Time column
    [self.outputLog appendString:@"Time"];

    if (self.simulationMode != MRUnifiedSimulationModeServerDisconnect) {
              // CR node columns
              [self.outputLog appendString:@"\t"];
              for (NSUInteger i=0; i<self.nodeCount; i++)
                    [self.outputLog appendFormat:@"\tCR-%ld", i];
//            [self.outputLog appendString:@"\tMin (CR)"];
//            [self.outputLog appendString:@"\tMean (CR)"];
//            [self.outputLog appendString:@"\tMax (CR)"];
    }
```

```
    // EF node columns
    [self.outputLog appendString:@"\t"];
    for (NSUInteger i=0; i<self.nodeCount; i++)
            [self.outputLog appendFormat:@"\tEF-%ld", i];
//  [self.outputLog appendString:@"\tMin (EF)"];
//  [self.outputLog appendString:@"\tMean (EF)"];
//  [self.outputLog appendString:@"\tMax (EF)"];

    // FO node columns
    [self.outputLog appendString:@"\t"];
    for (NSUInteger i=0; i<self.nodeCount; i++)
            [self.outputLog appendFormat:@"\tFO-%ld", i];
//  [self.outputLog appendString:@"\tMin (FO)"];
//  [self.outputLog appendString:@"\tMean (FO)"];
//  [self.outputLog appendString:@"\tMax (EFO)"];
}

- (void)printHeader
{
    [super printHeader];
    printf("Time Server Duration = %ld minutes\n", (NSInteger)(self.duration/60));
    printf("Time Server Cycle Period = %ld seconds\n",
(NSInteger)self.serverCyclePeriod);

    // Follower
    printf("Node Count = %ld\n", self.nodeCount);
    printf("Node Update Threshold = %f seconds\n", self.nodeUpdateThreshold);
    printf("Follower Node Layer Count = %ld\n", self.nodeLayerCount);
    printf("Follower Node Mean Time Base Record = %ld\n",
self.nodeMeanTimeBaseRecord);
    printf("Follower Node Maximum Broadcast Count = %ld nodes\n",
self.nodeMaxBroadcastCount);
}

- (void)simulate
{
    [self.server becomeServerForDuration:self.duration
updatePeriod:self.serverCyclePeriod];

    if (self.simulationMode != MRUnifiedSimulationModeServerDisconnect) {
            // Cristian
            for (MRCristianTimeClientNode *node in self.cristianNodes) {
                  [node startRequestingTimeFromServer:self.server.name];
            }
    }

    if (self.simulationMode == MRUnifiedSimulationModeClientDisconnect) {
            mr_delay(self.clientDisconnectStartTime, ^{
                  for (NSUInteger i=0; i<(self.nodeCount/2.0); i++) {
                        MRFollowerNode *followerNode = [self.followerNodes
objectAtIndex:i];
                        if (followerNode != self.server) [followerNode
disconnectForDuration:self.clientDisconnectDuration];
                        [[self.cristianNodes objectAtIndex:i]
disconnectForDuration:self.clientDisconnectDuration];
                  }
            });
    } else if (self.simulationMode == MRUnifiedSimulationModeServerDisconnect) {
            for (NSUInteger i=0; i<self.serverDisconnectStartTimes.count; i++) {
                  mr_delay([self.serverDisconnectStartTimes[i] doubleValue], ^{
                        [self.server resignServer];

                        mr_delay([self.serverDisconnectDurations[i] doubleValue],
^{
```

```
                                NSTimeInterval duration = self.duration-
self.server.serverTime.currentTime;
                                self.server = [self.followerNodes objectAtIndex:+
+_serverIndex];
                                [self.server becomeServerForDuration:duration
updatePeriod:self.serverCyclePeriod];
                          });
                    });
              }


//            mr_delay(self.serverDisconnectDuration, ^{
//                  NSInteger iterations = (NSInteger)(self.duration/
self.serverDisconnectPeriod);
//                  mr_delay_loop(iterations, self.serverDisconnectPeriod, ^{
//                        [self.server resignServer];
//
//                        mr_delay(self.serverDisconnectDuration, ^{
//                              MRFollowerNode *newServer = nil;
//                              NSTimeInterval serverTime = 0.0;
//
//                              for (MRFollowerNode *node in self.followerNodes) {
//                                    if (node != self.server) {
//                                          NSTimeInterval serverTimeEstimate =
node.clientServerTimeEstimate;
//                                          if (serverTimeEstimate > serverTime) {
//                                                newServer = node;
//                                                serverTime = serverTimeEstimate;
//                                          }
//                                    }
//                              }
//
//                              self.server = newServer;
//                              [self.server becomeServerForDuration:(self.duration-
self.server.serverTime.currentTime-self.serverDisconnectDuration)
updatePeriod:self.serverCyclePeriod];
//                          });
//                    });
//            });
    }
}

- (void)printUpdateLog
{
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0),
^{

              NSMutableString *updateLogOutput = [NSMutableString
stringWithFormat:@"%@Current Time Perceptions: %@\n\t%@: %f\n\t%@",
                                                                        @LOG_DIVIDER,
                                                                        self.title,

    self.server.name, self.server.serverTime.currentTime,

    @LOG_DIVIDER];

              if (self.server.serverTime.currentTime >= 0.0f) {
//                  NSTimeInterval minDiff[3], meanDiff[3], maxDiff[3];
                    [self.outputLog appendFormat:@"\n%f",
self.server.serverTime.currentTime];

                    // Cristian
//                  [updateLogOutput appendFormat:@"\n\tCristian's Algorithm:"];
```

```
                    if (self.simulationMode !=
MRUnifiedSimulationModeServerDisconnect)
                          [self.outputLog appendString:@"\t"];

//                  minDiff[0] = minDiff[1]= minDiff[2] = DBL_MAX;
//                  meanDiff[0] = meanDiff[1] = meanDiff[2] = 0;
//                  maxDiff[0] = maxDiff[1] = maxDiff[2] = -DBL_MAX;

                    for (MRCristianTimeClientNode *node in self.cristianNodes) {
                          NSTimeInterval diff = (node.serverTime -
self.server.serverTime.currentTime);
//                        [updateLogOutput appendFormat:@"\n\t\t%f  %@", diff, node];
                          [self.outputLog appendFormat:@"\t%f", diff];

//                        if (diff < minDiff[0])
//                              minDiff[0] = diff;
//                        if (diff > maxDiff[0])
//                              maxDiff[0] = diff;
//                        meanDiff[0] += diff;
                    }

//                  meanDiff[0] = meanDiff[0]/(NSTimeInterval)self.nodeCount;
//                  [updateLogOutput appendFormat:@"\n\tSummary: MIN = %f, MEAN = %f,
MAX = %f", minDiff[0], meanDiff[0], maxDiff[0]];
//
//                  [self.outputLog appendFormat:@"\t%f", minDiff[0]];
//                  [self.outputLog appendFormat:@"\t%f", meanDiff[0]];
//                  [self.outputLog appendFormat:@"\t%f", maxDiff[0]];

                    // Follower
                    NSMutableString *followerHeader = [NSMutableString
stringWithString:@"\t"];

                    // Enhanced Follower
//                  [updateLogOutput appendFormat:@"\n\n\tEnhanced Follower
Algorithm:"];
                    [self.outputLog appendString:@"\t"];

//                  minDiff[0] = minDiff[1]= minDiff[2] = DBL_MAX;
//                  meanDiff[0] = meanDiff[1] = meanDiff[2] = 0;
//                  maxDiff[0] = maxDiff[1] = maxDiff[2] = -DBL_MAX;

                    for (MRFollowerNode *node in self.followerNodes) {
                          if (node == self.server) {
                                [self.outputLog appendFormat:@"\t%f", 0.0];
                                [followerHeader appendFormat:@"\t%f", 0.0];
                                continue;
                          }

                          NSTimeInterval diff = (node.clientMeanTime.currentTime -
self.server.serverTime.currentTime);
                          NSTimeInterval ediff = (node.clientServerTimeEstimate -
self.server.serverTime.currentTime);
                          [updateLogOutput appendFormat:@"\n\t\t%f / %f  %@", diff,
ediff, node];

                          [self.outputLog appendFormat:@"\t%f", ediff];

//                        if (diff < minDiff[0])
//                              minDiff[0] = diff;
//                        if (diff > maxDiff[0])
//                              maxDiff[0] = diff;
//                        meanDiff[0] += diff;
```

```
//                          if (ediff < minDiff[1])
//                                minDiff[1] = ediff;
//                          if (ediff > maxDiff[1])
//                                maxDiff[1] = ediff;
//                          meanDiff[1] += ediff;


                            NSTimeInterval fdiff = (node.clientFollowerTime.currentTime
- self.server.serverTime.currentTime);

                            [followerHeader appendFormat:@"\t%f", fdiff];

//                          if (fdiff < minDiff[2])
//                                minDiff[2] = fdiff;
//                          if (fdiff > maxDiff[2])
//                                maxDiff[2] = fdiff;
//                          meanDiff[2] += fdiff;
                    }

                    //                  meanDiff[0] = meanDiff[0]/
(NSTimeInterval)_ef_nodeCount;
                    //                  [updateLogOutput appendFormat:@"\n\tSummary
STD: MIN = %f, MEAN = %f, MAX = %f", minDiff[0], meanDiff[0], maxDiff[0]];

//                  meanDiff[1] = meanDiff[1]/(NSTimeInterval)(self.nodeCount - 1);
//                  [updateLogOutput appendFormat:@"\n\tSummary: MIN = %f, MEAN = %f,
MAX = %f", minDiff[1], meanDiff[1], maxDiff[1]];
//
//                  [self.outputLog appendFormat:@"\t%f", minDiff[1]];
//                  [self.outputLog appendFormat:@"\t%f", meanDiff[1]];
//                  [self.outputLog appendFormat:@"\t%f", maxDiff[1]];
//
//
//                  meanDiff[2] = meanDiff[2]/(NSTimeInterval)self.nodeCount;
//                  [updateLogOutput appendFormat:@"\n\n\tFollower Algorithm:\n
\tSummary: MIN = %f, MEAN = %f, MAX = %f", minDiff[2], meanDiff[2], maxDiff[2]];
//
//                  [followerHeader appendFormat:@"\t%f", minDiff[2]];
//                  [followerHeader appendFormat:@"\t%f", meanDiff[2]];
//                  [followerHeader appendFormat:@"\t%f", maxDiff[2]];

                    [self.outputLog appendString:followerHeader];
                }

                printf("%s\n%s", [updateLogOutput UTF8String], LOG_DIVIDER);
    });
}
- (void)tearDown
{
    NSLog(@"Final Output:\n\n\n%@\n\n\n", self.outputLog);

    [self.server resignServer];
    self.server = nil;

    self.cristianNodes = nil;

    // Follower
    for (MRFollowerNode *node in self.followerNodes)
            [node.listeners removeAllObjects];
    self.followerNodes = nil;

    [super tearDown];
}
@end
```

Main

main.m

```
//
//  main.m
//  SeniorThesisSimulations-Mac
//
//  Created by Matt Ricketson on 3/10/13.
//  Copyright (c) 2013 Matt Ricketson. All rights reserved.
//

#import <Foundation/Foundation.h>
#import "MRTestSimulation.h"
#import "MRUnifiedSimulation.h"
#import "MRUnifiedAnomalySimulation.h"
#import "MRLeanFollowerNodeSimulation.h"

#pragma mark - Simulations

//#define SIM_TEST                                                  @"Test
Simulation"

//#define SIM_LONG_TAIL_LATENCY_TEST                              @"Long-Tail Latency
Model Test Simulation"
//#define SIM_VOLATILE_LATENCY_TEST                               @"Volatile Latency
Model Test Simulation"
//#define SIM_STABLE_LATENCY_TEST                                 @"Stable Latency
Model Test Simulation"

//#define SIM_SET_A_CLIENT_TIME_BASE_RECORD               @"Sim Set A (Client Time
Base Record Tests)"
//#define SIM_SET_B_CLIENT_DISCONNECT                             @"Sim Set B (Client
Disconnect Tests)"
//#define SIM_SET_C_SERVER_DISCONNECT                             @"Sim Set C (Server
Disconnect Tests)"


#pragma mark - Simulation Make Macros


#pragma mark - Default Simulation Properties

#define LONG_TAIL_NORMAL_LATENCY              [MRNormalDistrLatencyModel
modelWithMean:5.0 standardDeviation:4.0]
#define VOLATILE_NORMAL_LATENCY                     [MRNormalDistrLatencyModel
modelWithMean:10.0 standardDeviation:2.5]
#define STABLE_NORMAL_LATENCY                       [MRNormalDistrLatencyModel
modelWithMean:10.0 standardDeviation:0.1]
#define DEFAULT_RANDOM_LATENCY                      [MRRandomLatencyModel
modelWithLowerBound:2.0 upperBound:6.0]

#define DEFAULT_LATENCY_MODEL_TEST_COUNT      20000

#define DEFAULT_DURATION_MINUTES              15
#define DEFAULT_DURATION                            (DEFAULT_DURATION_MINUTES * 60.0)
#define MINUTES_TO_SECONDS(minutes)                 ((minutes) * 60.0)
#define DEFAULT_CYCLE_PERIOD                        5.0

#define DEFAULT_NODE_COUNT                          20
#define DEFAULT_UPDATE_THRESHOLD              0.0

#define DEFAULT_LAYER_COUNT                               4
#define DEFAULT_MAX_BROADCAST_COUNT                 5
```

```
MRUnifiedSimulation * MRUnifiedSimulationMake(NSString *title,
id<MRNetworkLatencyModel> model);
MRUnifiedAnomalySimulation * MRUnifiedAnomalySimulationMake(NSString *title,
id<MRNetworkLatencyModel> model);
void latencyModelTest(NSString *testTitle, id<MRNetworkLatencyModel> model, NSUInteger
count);


int main(int argc, const char * argv[])
{
    @autoreleasepool {

#pragma mark - Tests

#ifdef SIM_TEST
            [[MRTestSimulation simulationWithTitle:SIM_TEST] run];
#endif



#pragma mark - Latency Model Tests

#ifdef SIM_LONG_TAIL_LATENCY_TEST
            latencyModelTest(SIM_LONG_TAIL_LATENCY_TEST,
DEFAULT_NORMAL_LATENCY_MODEL, DEFAULT_LATENCY_MODEL_TEST_COUNT);
#endif

#ifdef SIM_VOLATILE_LATENCY_TEST
            latencyModelTest(SIM_VOLATILE_LATENCY_TEST,
VOLATILE_NORMAL_LATENCY_MODEL, DEFAULT_LATENCY_MODEL_TEST_COUNT);
#endif

#ifdef SIM_STABLE_LATENCY_TEST
            latencyModelTest(SIM_STABLE_LATENCY_TEST, STABLE_NORMAL_LATENCY_MODEL,
DEFAULT_LATENCY_MODEL_TEST_COUNT);
#endif



#pragma mark - Sim Sets

#define SIM_SET_LONG_TAIL_NORMAL_LATENCY                    [MRNormalDistrLatencyModel
modelWithMean:(5.0/3.0) standardDeviation:(4.0/3.0)]
#define SIM_SET_VOLATILE_NORMAL_LATENCY                     [MRNormalDistrLatencyModel
modelWithMean:(10.0/3.0) standardDeviation:(2.5/3.0)]
#define SIM_SET_STABLE_NORMAL_LATENCY                       [MRNormalDistrLatencyModel
modelWithMean:(10.0/3.0) standardDeviation:(0.1/3.0)]


            NSArray *latencyModels = @[SIM_SET_LONG_TAIL_NORMAL_LATENCY,
SIM_SET_VOLATILE_NORMAL_LATENCY, SIM_SET_STABLE_NORMAL_LATENCY];
            NSArray *latencyModelLabels = @[@"Long-Tail Volatile Latency", @"Volatile
Latency", @"Stable Latency"];


#define SIM_SET_DEFAULT_MEAN_TIME_BASE_RECORD 1


#ifdef SIM_SET_A_CLIENT_TIME_BASE_RECORD
            for (NSUInteger i=0; i<latencyModels.count; i++) {
                  id<MRNetworkLatencyModel> latencyModel = latencyModels[i];
                  NSString *latencyModelLabel = latencyModelLabels[i];
```

```
                    NSUInteger layerCount = DEFAULT_LAYER_COUNT + 1;
                    for (NSUInteger j=0; j<layerCount; j++) {
                          NSString *title = [NSString stringWithFormat:@"%@ - Test
%ld (%@)", SIM_SET_A_CLIENT_TIME_BASE_RECORD, (layerCount*i+j), latencyModelLabel];
                          MRUnifiedAnomalySimulation *sim =
[MRUnifiedAnomalySimulation simulationWithTitle:title duration:(DEFAULT_DURATION/
3.0)];

                          sim.latencyModel = latencyModel;
                          sim.serverCyclePeriod = (DEFAULT_CYCLE_PERIOD/3.0);
                          sim.updateLogPeriod = sim.serverCyclePeriod * 6;

                          sim.nodeCount = DEFAULT_NODE_COUNT/2;
                          sim.nodeUpdateThreshold = DEFAULT_UPDATE_THRESHOLD;

                          sim.nodeLayerCount = layerCount;
                          sim.nodeMeanTimeBaseRecord = j;
                          sim.nodeMaxBroadcastCount = DEFAULT_MAX_BROADCAST_COUNT;

                          sim.simulationMode = MRUnifiedSimulationModeNormal;

                          [sim run];
                    }
              }
#endif



#ifdef SIM_SET_B_CLIENT_DISCONNECT
              for (NSUInteger i=0; i<latencyModels.count; i++) {
                    id<MRNetworkLatencyModel> latencyModel = latencyModels[i];
                    NSString *latencyModelLabel = latencyModelLabels[i];

                    NSString *title = [NSString stringWithFormat:@"%@ - Test %ld
(%@)", SIM_SET_B_CLIENT_DISCONNECT, i, latencyModelLabel];
                    MRUnifiedAnomalySimulation *sim = [MRUnifiedAnomalySimulation
simulationWithTitle:title duration:(DEFAULT_DURATION/3.0)];

                    sim.latencyModel = latencyModel;
                    sim.serverCyclePeriod = (DEFAULT_CYCLE_PERIOD/3.0);
                    sim.updateLogPeriod = sim.serverCyclePeriod * 6;

                    sim.nodeCount = DEFAULT_NODE_COUNT/2;
                    sim.nodeUpdateThreshold = DEFAULT_UPDATE_THRESHOLD;

                    sim.nodeLayerCount = DEFAULT_LAYER_COUNT + 1;
                    sim.nodeMeanTimeBaseRecord =
SIM_SET_DEFAULT_MEAN_TIME_BASE_RECORD;
                    sim.nodeMaxBroadcastCount = DEFAULT_MAX_BROADCAST_COUNT;

                    sim.simulationMode = MRUnifiedSimulationModeClientDisconnect;
                    sim.clientDisconnectTime = 20.0;
                    sim.clientDisconnectDuration = sim.duration / 2.0;

                    [sim run];
              }
#endif



#ifdef SIM_SET_C_SERVER_DISCONNECT
              for (NSUInteger i=0; i<latencyModels.count; i++) {
                    id<MRNetworkLatencyModel> latencyModel = latencyModels[i];
                    NSString *latencyModelLabel = latencyModelLabels[i];
```

```
                    NSString *title = [NSString stringWithFormat:@"%@ - Test %ld
(%@)", SIM_SET_C_SERVER_DISCONNECT, i, latencyModelLabel];
                    MRUnifiedAnomalySimulation *sim = [MRUnifiedAnomalySimulation
simulationWithTitle:title duration:(DEFAULT_DURATION/3.0)];

                    sim.latencyModel = latencyModel;
                    sim.serverCyclePeriod = (DEFAULT_CYCLE_PERIOD/3.0);
                    sim.updateLogPeriod = sim.serverCyclePeriod * 6;

                    sim.nodeCount = DEFAULT_NODE_COUNT/2;
                    sim.nodeUpdateThreshold = DEFAULT_UPDATE_THRESHOLD;

                    sim.nodeLayerCount = DEFAULT_LAYER_COUNT + 1;
                    sim.nodeMeanTimeBaseRecord =
SIM_SET_DEFAULT_MEAN_TIME_BASE_RECORD;
                    sim.nodeMaxBroadcastCount = DEFAULT_MAX_BROADCAST_COUNT;

                    sim.simulationMode = MRUnifiedSimulationModeServerDisconnect;
                    sim.serverDisconnectStartTimes = @[ @(sim.duration*0.3),
@(sim.duration*0.6) ];
                    sim.serverDisconnectDurations =        @[ @(sim.duration*0.1),
@(sim.duration*0.1) ];

                    [sim run];
            }
#endif


//            printf("done");
    }
    return 0;
}



#pragma mark - Helpers

MRUnifiedSimulation * MRUnifiedSimulationMake(NSString *title,
id<MRNetworkLatencyModel> model)
{
    MRUnifiedSimulation *sim = [MRUnifiedSimulation simulationWithTitle:title
duration:DEFAULT_DURATION];

    sim.latencyModel = model;
    sim.serverCyclePeriod = DEFAULT_CYCLE_PERIOD;
    sim.updateLogPeriod = DEFAULT_CYCLE_PERIOD * 6;

    sim.cr_nodeCount = DEFAULT_NODE_COUNT/2;
    sim.cr_nodeUpdateThreshold = DEFAULT_UPDATE_THRESHOLD;

    sim.ef_nodeCount = DEFAULT_NODE_COUNT/2;
    sim.ef_nodeUpdateThreshold = DEFAULT_UPDATE_THRESHOLD;
    sim.ef_nodeLayerCount = DEFAULT_LAYER_COUNT;
    sim.ef_nodeMaxBroadcastCount = DEFAULT_MAX_BROADCAST_COUNT;

    return sim;
}

MRUnifiedAnomalySimulation * MRUnifiedAnomalySimulationMake(NSString *title,
id<MRNetworkLatencyModel> model)
{
```

```
    MRUnifiedAnomalySimulation *sim = [MRUnifiedAnomalySimulation
simulationWithTitle:title duration:DEFAULT_DURATION];

    sim.latencyModel = model;
    sim.serverCyclePeriod = DEFAULT_CYCLE_PERIOD;
    sim.updateLogPeriod = DEFAULT_CYCLE_PERIOD * 6;

    sim.nodeCount = DEFAULT_NODE_COUNT/2;
    sim.nodeUpdateThreshold = DEFAULT_UPDATE_THRESHOLD;

    sim.nodeLayerCount = DEFAULT_LAYER_COUNT;
    sim.nodeMeanTimeBaseRecord = 0;
    sim.nodeMaxBroadcastCount = DEFAULT_MAX_BROADCAST_COUNT;

    return sim;
}

void latencyModelTest(NSString *testTitle, id<MRNetworkLatencyModel> model, NSUInteger
count)
{
    MRStatisticalTimeRecord *record = [[MRStatisticalTimeRecord alloc]
initWithReturnType:MRStatisticalTimeRecordReturnTypeMostRecent useCurrentTime:NO
updateThreshold:0.0 tag:0];

    printf("%s:\n", [testTitle UTF8String]);

    for (NSInteger i=0; i<count; i++) {
            [record updateTime:[model currentLatency]];
            printf("%f\n", record.lastUpdateTime);
    }

//  printf("\n\n\tMIN = %f\n\tMEAN = %f\n\tMAX = %f\n\n\n\n",
record.minTime.lastUpdateTime, record.meanTime.lastUpdateTime,
record.maxTime.lastUpdateTime);
}
```

Appendix C: Hardware Implementation

Main

Main.ino

```
/*

 Created 20 Jan 2013
 By Matthew D Ricketson

 */

#include <SPI.h>
#include <WiFi.h>
#include "MRWiFi.h"
#include "MRLeanTimeRecord.h"


////////////////////////////////////////////////////////////////////////////////
//////////////////////////
#pragma mark - Definitions

typedef struct {
    int hops;
    MRTimeInterval timeValue;
} MRTimeUpdateMessage;

String StringFromTimeUpdateMessage(MRTimeUpdateMessage message);


////////////////////////////////////////////////////////////////////////////////
//////////////////////////

#pragma mark - Initialization
void initNodes();
void initPins();

#pragma mark - Lean Follower Node
MRTimeInterval followerTime();
void sendTimeUpdateMessage(WiFiClient stream, MRTimeUpdateMessage message);
void receiveMessage(MRTimeUpdateMessage message);
void updateClient(WiFiClient serverClient);

#pragma mark - Physical Actions
void blinkLED(int LED);

#pragma mark - Server Actions
void serverBegin();
void serverLoop();
void serveClient(WiFiClient serverClient);
void listenToClient(WiFiClient serverClient);
void closeClient(WiFiClient serverClient);

#pragma mark - Client Actions
void clientBegin();
void clientLoop();
void listenToServer();
void connectToServer(char *serverName);

#pragma mark - Stream Actions
String readMessage(WiFiClient stream);
void sendMessage(WiFiClient stream, String message);
```

```
////////////////////////////////////////////////////////////////////////////////
//////////////////////////
#pragma mark - Globals

char *nodes[4];
#define NODE_A1           0
#define NODE_A2           1
#define NODE_A3           2
#define NODE_A4           3

#define NODE_SERVER       NODE_A1
#define NODE_SELF         NODE_A2

#define IS_SERVER         (NODE_SELF == NODE_SERVER)
#define IS_CLIENT         (NODE_SELF != NODE_SERVER)

int messageLED = 9;

WiFiServer server(23);
boolean clientConnected = false; // whether or not the client was connected previously

WiFiClient client;
String readBuffer = "";

#pragma mark - Lean Follower Node
MRLeanTimeRecord serverTime;
MRLeanTimeRecord clientTime;


////////////////////////////////////////////////////////////////////////////////
//////////////////////////
#pragma mark - Lean Follower Node

MRTimeInterval followerTime()
{
    return clientTime.currentTime();
}

void sendTimeUpdateMessage(WiFiClient stream, MRTimeUpdateMessage message)
{
    String strMessage = "";
    strMessage += message.hops;
    strMessage += ',';
    strMessage += message.timeValue;
    sendMessage(stream, strMessage);
}

void receiveMessage(MRTimeUpdateMessage message)
{
    clientTime.updateTime(message.timeValue, MRTimeRecordUpdateMethodMin);
}

void updateClient(WiFiClient serverClient)
{
    MRTimeUpdateMessage message;
    message.hops = 0;
    message.timeValue = serverTime.currentTime();

    sendTimeUpdateMessage(serverClient, message);
}
```

```
////////////////////////////////////////////////////////////////////////////////
//////////////////////////
#pragma mark - Arduino Actions

void setup()
{
    Serial.begin(9600);

    initPins();
    initNodes();

    MRWiFi.connect();

#if IS_SERVER
    serverBegin();
#else
    clientBegin();
#endif
}

void loop()
{
#if IS_SERVER
    serverLoop();
#else
    clientLoop();
#endif
}


////////////////////////////////////////////////////////////////////////////////
//////////////////////////
#pragma mark - Server Actions

void serverBegin()
{
    serverTime.updateTime(0, MRTimeRecordUpdateMethodAlways);

    server.begin();
    Serial.println("SERVER: waiting for clients...");
}

void serverLoop()
{
    WiFiClient serverClient = server.available();

    if (serverClient) {
              Serial.println("SERVER: new client connected.");
              updateClient(serverClient);
              serverClient.stop();
              Serial.println("SERVER: client disconnected.");
    }
}

void serveClient(WiFiClient serverClient)
{
    if (!clientConnected) {
              serverClient.flush();
              Serial.println("SERVER: new client connected.");
              clientConnected = true;
    }

    updateClient(serverClient);
```

```
//  if (serverClient.available() > 0)
//            listenToClient(serverClient);
}

//void listenToClient(WiFiClient serverClient)
//{
//  String message = readMessage(serverClient);
//
//  if (message == "pong") {
//            blinkLED(messageLED);
//            sendMessage(serverClient, "ping");
//  } else if (message == "stop") {
//            closeClient();
//  }
//}

void closeClient(WiFiClient serverClient)
{
    if (clientConnected) {
              Serial.println("SERVER: client disconnected.");
              clientConnected = false;
              readBuffer = "";
              serverClient.stop();
    }
}


////////////////////////////////////////////////////////////////////////////////
//////////////////////////
#pragma mark - Client Actions

void clientBegin()
{
    clientTime.updateTime(0, MRTimeRecordUpdateMethodAlways);
}

void clientLoop()
{
    if (client.connected()) {
              listenToServer();
    } else {
              delay(3000);
              connectToServer(nodes[NODE_SERVER]);
    }
}

void listenToServer()
{
    String message = readMessage(client);

    if (message != "") {
              MRTimeUpdateMessage timeUpdate;
              int delim;

              message.trim();
              delim = message.indexOf(',');

              timeUpdate.hops = (int)((message.substring(0, delim)).toInt());
              timeUpdate.timeValue = (MRTimeInterval)((message.substring(delim+1,
message.length())).toInt());

              receiveMessage(timeUpdate);
              blinkLED(messageLED);
    }
```

```
}

void connectToServer(char *serverName)
{
    Serial.println("CLIENT: attempting to connect to server...");
    if (client.connect(serverName, 23)) {
            Serial.println("CLIENT: successfully connected to server.");
    } else {
            Serial.println("CLIENT: failed to connect to server.");
    }
}


////////////////////////////////////////////////////////////////////////////////
//////////////////////////
#pragma mark - Stream Actions

String readMessage(WiFiClient stream)
{
    String message = "";

    char c = stream.read();
    if (c != -1) {
            if (c == '\n') {
                    message = readBuffer;
                    readBuffer = "";
                    Serial.print("New message from stream: ");
                    Serial.println(message);
            } else {
                    readBuffer += c;
            }
    }

    return message;
}

void sendMessage(WiFiClient stream, String message)
{
    stream.println(message);
    Serial.print("Message sent through stream: ");
    Serial.println(message);
    blinkLED(messageLED);
}


////////////////////////////////////////////////////////////////////////////////
//////////////////////////
#pragma mark - Initialization

void initNodes()
{
    nodes[NODE_A1] = "136.167.243.114";
    nodes[NODE_A2] = "10.6.194.22";
    nodes[NODE_A3] = "10.6.212.14";
    nodes[NODE_A4] = "10.6.214.14";
}

void initPins()
{
    pinMode(messageLED, OUTPUT);
}
```

```
////////////////////////////////////////////////////////////////////////////////
//////////////////////////
#pragma mark – Physical Actions

void blinkLED(int LED)
{
    digitalWrite(LED, HIGH);
    delay(500);
    digitalWrite(LED, LOW);
}
```

MRLeanTimeRecord Class

MRLeanTimeRecord.h

```
#ifndef MRLEANTIMERECORD_h
#define MRLEANTIMERECORD_h

#include "Arduino.h"

typedef unsigned long MRTimeInterval;

typedef enum {
    MRTimeRecordUpdateMethodAlways,
    MRTimeRecordUpdateMethodMean,
    MRTimeRecordUpdateMethodMin
} MRTimeRecordUpdateMethod;

class MRLeanTimeRecord
{
    private:
             MRTimeInterval _localTimeLastUpdated;
             MRTimeInterval _timeLastUpdated;
             int _updateCount;

    public:
             MRLeanTimeRecord();
             MRTimeInterval currentTime();
             void updateTime(MRTimeInterval time, MRTimeRecordUpdateMethod
updateMethod);
};

#endif // MRLEANTIMERECORD_h
```

MRLeanTimeRecord.cpp

```
#include "Arduino.h"
#include "MRLeanTimeRecord.h"


#pragma mark - Initialization

MRLeanTimeRecord::MRLeanTimeRecord()
{
    _localTimeLastUpdated = 0;
    _timeLastUpdated = 0;
    _updateCount = 0;
}


#pragma mark - Dynamic Properties

MRTimeInterval MRLeanTimeRecord::currentTime()
{
    if (_localTimeLastUpdated > 0) {
             MRTimeInterval localTimeElapsed = millis() - _localTimeLastUpdated;
             return (_timeLastUpdated + localTimeElapsed);
    }
    return 0;
}
```

```
#pragma mark - Updating Time

void MRLeanTimeRecord::updateTime(MRTimeInterval time, MRTimeRecordUpdateMethod
updateMethod)
{
    if (updateMethod == MRTimeRecordUpdateMethodMean) {
            _timeLastUpdated = (this->currentTime()*_updateCount + time) / (+
+_updateCount);
    } else {
            if (this->currentTime() > time)
                  _timeLastUpdated = time;
    }
    _localTimeLastUpdated = millis();
}
```

MRWiFi Class

MRWiFi.h

```
#ifndef MRWIFI_h
#define MRWIFI_h

#include "Arduino.h"
#include <WiFi.h>

typedef enum {
    BCWiFiSSIDBostonCollege,
    BCWiFiSSIDBCSecure
} BCWiFiSSID;


class MRWiFiClass
{
    public:
              BCWiFiSSID ssid;

              MRWiFiClass();
              void connect(BCWiFiSSID ssid = BCWiFiSSIDBCSecure);
              void printStatus();

    private:
              int _status;
              char *_ssid;
              char *_password;

              void setWiFiCredentials(BCWiFiSSID ssid);
};

extern MRWiFiClass MRWiFi;

#endif
```

MRWiFi.cpp

```
#include "Arduino.h"
#include "MRWiFi.h"

char BostonCollege_ssid[] = "BostonCollege";
char bcsecure_ssid[] = "bcsecure";
char bcsecure_pass[] = "bc$3cur3n3twork$";

MRWiFiClass::MRWiFiClass()
{
    _status = WL_IDLE_STATUS;
    _ssid = NULL;
    _password = NULL;
}

void MRWiFiClass::setWiFiCredentials(BCWiFiSSID ssid)
{
    this->ssid = ssid;
    if (ssid == BCWiFiSSIDBostonCollege) {
              _ssid = BostonCollege_ssid;
              _password = NULL;
    } else {
              _ssid = bcsecure_ssid;
              _password = bcsecure_pass;
```

```
    }
}

void MRWiFiClass::connect(BCWiFiSSID ssid)
{
    this->setWiFiCredentials(ssid);

    // check for the presence of the shield:
    if (WiFi.status() == WL_NO_SHIELD) {
             Serial.println("WiFi shield not present");
             while(true);  // Pause indefinitely
    }

    // attempt to connect to Wifi network:
    while (_status != WL_CONNECTED) {
             Serial.print("Attempting to connect to SSID: ");
             Serial.println(_ssid);

             if (_password == NULL) {
                    _status = WiFi.begin(_ssid);
             } else {
                    _status = WiFi.begin(_ssid, _password);
             }

             delay(1000);
    }

    this->printStatus();
}

void MRWiFiClass::printStatus()
{
    // print the SSID of the network you're attached to:
    Serial.print("SSID: ");
    Serial.println(WiFi.SSID());

    // print your WiFi shield's IP address:
    IPAddress ip = WiFi.localIP();
    Serial.print("IP Address: ");
    Serial.println(ip);

    // print your WiFi shield's MAC address:
    byte mac[6];
    WiFi.macAddress(mac);
    Serial.print("MAC: ");
    Serial.print(mac[5],HEX);
    Serial.print(":");
    Serial.print(mac[4],HEX);
    Serial.print(":");
    Serial.print(mac[3],HEX);
    Serial.print(":");
    Serial.print(mac[2],HEX);
    Serial.print(":");
    Serial.print(mac[1],HEX);
    Serial.print(":");
    Serial.println(mac[0],HEX);

    // print the received signal strength:
    long rssi = WiFi.RSSI();
    Serial.print("signal strength (RSSI):");
    Serial.print(rssi);
    Serial.println(" dBm");
}
MRWiFiClass MRWiFi;
```