VISUALIZATION OF CONSISTENCY IN A DISTRIBUTED KEY-VALUE STORE

By

David Lauschke

Senior Thesis in Computer Engineering

University of Illinois at Urbana-Champaign

Advisor: Professor Nitin Vaidya

May 2015

Abstract

A wide variety of consistency models are available to designers of key-value storage systems, such as Apache's Cassandra or Amazon's Dynamo. Each consistency model has been extensively analyzed for advantages and disadvantages as they relate to a system's availability and consistency properties. Our purpose is to create a modular visualization tool, described herein as the visualization authority (VA), supported by unique and customizable communication handlers. We have created a prototype of this tool for the purpose of easily comparing and contrasting consistency models available to a key-value store system such that its designers and administrators can fine-tune the trade-offs between availability and consistency to fit their specific needs.

Subject Keywords: Distributed computing, Distributed databases, Availability, Consistency, Visualization

Acknowledgments

I would like to thank Professor Vaidya and Lewis Tseng for providing me both support and resources as we completed our work.

Contents

1. Introduction

In designing and constructing new distributed systems, there are many trade-offs to consider between system performance, availability of data, and data consistency guarantees. When we began our research for this thesis, we first reviewed some of the fundamental aspects of distributed systems and a few popular implementations that perform in-depth analysis on the trade-offs listed above. Initially, our work focused more on the security considerations of a distributed peer-to-peer social network, so we then refined our literature review to the security considerations of such a system, specifically pertaining to request routing, data validity, and unknowingly or purposefully faulty nodes. As our work continued, the subject became more focused on the prospect of visualizing consistency in the core element of the peer-to-peer social network, the distributed key-store system. Many system operators use theoretical statistics as well as network analysis to observe their system's behavior. However, there are not many visual tools to assist in real-time human analysis of how a network's data evolves over time. So, we began creating a visualization scheme for consistency in the system.

In this thesis, we will first discuss the literature that we reviewed during our work. Next, we will discuss the main objectives of the visualization authority (VA) and its communication handlers. We will also demonstrate the current state of the work. Finally, we will discuss future work to be carried out by current and future members of Professor Vaidya's research team.

2. Literature Review

As the focus of this thesis has changed throughout the literature review, we analyze the contents of each related work with a few frames of reference – one directed towards the security considerations of the distributed peer-to-peer social network, and another towards the visualization of consistency in the distributed key-store. To keep the review thorough, included are both frames of reference.

2.1 Distributed Systems Review

We first revisited the basics of different consistency schemes with the help of Doug Terry of the Microsoft Research team in Silicon Valley. In his paper [1], he uses the game of baseball to describe different consistency needs of various users in a distributed system. The research team was able to gain a better understanding of the trade-offs when deciding between an existing simple service such as a key-value store across Windows Azure, Amazon's Simple Storage Service, Google's App Engine or Yahoo's PNUTS. Terry deemed consistency models weaker than eventual consistency as being "less-than-useful." Thus, offering stronger consistency usually comes at the sacrifice of high-performance and availability. His work underscores the benefit to be gained from an intermediate consistency scheme.

Terry's examples seemed to implore developers to understand the consequences of relaxed consistency. I realized it was incorrect to assume that eventual consistency was as strong as his consistent prefix schema, but this is flawed as with eventual consistency a user can receive an out-of-order sequence of writes. As this thesis turned more towards visualization, his statement of "less-than-useful" consistency models became more of a hypothesis to prove or disprove.

2.2 Example Implementations of Distributed Systems

The first detailed review of an implementation of a key-value store was the study of Amazon's Dynamo by DeCandia *et al.* [2]. Dynamo was originally created because of the Amazon services that had very high reliability requirements and required extreme customization of trade-offs between consistency, availability, cost, and performance. Amazon's platform has strict latency requirements for its services, measured at the 99.9th percentile of the distribution. With these tight bounds on their service level agreements, Dynamo was a necessity for service developers. In contrast to some of its related peers, Dynamo does not consider hostile or untrusted environments. Dynamo was designed to appeal to applications that require only key-value access with a high uptime, where not even network partitions and server failures can delay reads and writes.

Dynamo has only two main operations - get() and put(), both of which deal with contexts; and the system uses this context to verify the validity of the object supplied to put() requests. Dynamo instance managers can customize their deployment of Dynamo by specifying values for N (the number of replicas it creates for each data item), R (the number of nodes that must participate in a successful read operation), and W (the number of nodes that must participate in a successful write operation). Consider a traditional quorum system which stores key, *k*, in nodes A, B, and C. Suppose that A is temporarily down during a write with W = 3. Whereas the traditional quorum will fail to write to *k*, Dynamo will not fail. Instead, Dynamo uses the concept of hinted-handoffs, wherein the write meant for A is handed off to node D, who stores the write with metadata hinting that the write for A in a local database that is periodically scanned. As soon as A recovers, the write is sent to A and D deletes it's local replica, ensuring that the number of replicas never falls below N.

Through extensive testing, DeCandia *et al*. have been able to verify their claims as well as their strict performance standards - keeping 99.9th percentile latencies below 300 milliseconds for Read and Writes, with averages below 20 milliseconds [2]. By monitoring the load distribution, Amazon has discovered the most efficient load balancing technique and key partitioning strategy such that they could maximize their load efficiency. Over the past two years, many internal Amazon services have used Dynamo, receiving successful responses (without timing out) 99.9995% of the time, without any data loss.

While this entire case study on such a reliable system was insightful, there were a few key aspects applicable to this work. Dynamo is an "always writeable" data store, which would be appealing in the peer-to-peer distributed social network as it would be ideal to allow users to always be able to create content on the network and have that content eventually be committed to the storage system. However, Dynamo assumes that all nodes in its network are trustworthy, which could not be assumed in the social network implementation. In order to meet its latency requirements, Dynamo cannot route requests through nodes, and accordingly is characterized as a "zero-hop" DHT: one in which every node contains enough routing information to route a request directly to the proper node and replicas. While this would be interesting for the performance of the social network, it is not maintainable as the system scales due to the amount of routing data that nodes would need to store within their systems.

Stoica *et al.* [3] describe Chord as a scalable lookup protocol for distributed peer-to-peer systems with high churn rates (the rate at which nodes enter and leave the system). While this is not actually a key-value store protocol, it accomplishes a large part of that process - given a key, Chord returns a mapping onto a member node. The most interesting part about Chord is that unlike much of its related work that uses consistent hashing to balance the load of keys across nodes, a node only needs to know routing information about *O*(*log N*) other nodes. Other protocols need to know information about every other node in the system, which gets unrealistic when scaled. Chord also resolves lookups in *O(log N)* messages to other nodes, using efficient routing.

An immediate and direct application of Chord to the peer-to-peer social network Stoica *et al.* [3] outline is its “Time-shared storage” application. It describes Chord as being advantageous when a system has intermittently connected nodes that wish to have their data always available. An implementation could have a user offer to store others’ data while they are connected, in return for others to store theirs while they are disconnected from the service. Also worth noting is the efficient routing scheme, which utilizes a finger table at each node to reduce the number of jumps a lookup has to make. Where this system is incompatible with the social network implementation, however, is in its assumptions of “high probability” concerning its consistent hashing protocol. This phrase is invalidated outside of a benign environment. Whenever a random hash has been chosen to select the distribution of the keys across nodes, an adversary party could generate a large amount of keys according to the hash function and only insert into the system those which correspond to a small amount of nodes in order to put the system into an imbalance. Also, nodes who generate a fake outlook of the Chord ring could render some data unavailable by never correctly sending the request to the designated node. A possible detection scheme for this problem would be to randomly poll ring nodes for a key, and if a node returns a different value, then the system could identify some problem with the ring.

Gladenning *et al*. [4] elaborate on an alternative to Chord named Scatter. Scatter is a scalable and linearizable distributed key-value storage system. Their interest in creating a system like Scatter stems from this work’s exact interest - building a “planetary scale” peer-to-peer social network. The implementation of the distributed hash table over sets of groups of nodes arranged in a ring was an intriguing extension of other related work. The technique of “nested consensus,” in which groups execute distributed transactions (such as group splits, merges, migrations of nodes, or repartitions of groups) through a two-phase commit scheme, by first replicating the commit internally through the Paxos distributed consensus algorithm is an interesting group management strategy. After the coordinator (initiating) group receives commits/aborts from each participant group leader, the coordinator broadcasts the result of the action, and if it was committed, each participant group executes the steps of the transaction. Locks on commits of this type are implemented on the relations of the groups - so if adjacent groups are attempting a commit with their opposing groups, these processes can simultaneously complete.

Gladenning *et al*. [4] found that "if average node lifetimes are as short as three minutes…, Scatter is able to maintain overall consistency and data availability, serving its read in an average of 1.3 seconds in a typical wide area setting." This seemed like a very applicable aspect of the protocol to the social network scheme. The group aspect within the distributed hash table also appeared to be a sound idea for having two tiers of operation validation. They describe the node joining protocol, named "latency-join optimization," which optimizes new node placement by having the node send no-operation commands to *k* randomly selected groups, estimating the performance of operating in the group. The node then selects the group which had the lowest latency to join. The node-join scheme also allows the administrator to do a similar check on how many requests the random groups have processed recently. Groups essentially perform all of their own organizing techniques in coordination with neighbor groups to properly and dynamically adjust the ring. While their work covers Scatter as a system resistant to non-malicious network issues, such as churn of member devices, varying computational capabilities of nodes, and unreliable network behavior, such attacks like distributed denial of service attacks or Byzantine faults are not covered. Their work also raises the question of how different visualization schemes may be applied to the visualization authority in order to maximize the operator's understanding of the state of the system. Whereas some systems may rely more heavily on a linear series visualization (to be discussed in Chapter 3), a system like this where a quorum must be satisfied might find the node layout visualization more useful.

2.3 Security Considerations

In this section, we deeply analyze the security vulnerabilities outlined in related works. In most studies, the author proposes an attack strategy, and describes mitigation techniques to defend against the proposed attacks.

Wallach [5] surveys some of the standard vulnerabilities to be considered in modern distributed systems. These security issues that occur in routing protocols as well as file (data) sharing applications, and how incentives, cryptography, and random probing can be used to combat these problems. He considers protocols which can be subject to one or more malicious nodes acting alone or as a coalition to pull more than a usual amount of data from the system, sniff for traffic in usually censored paths, prevent users from accessing data, return false data, or any number of other unknown and wide-ranging goals. He describes a situation involving a routing model based on Pastry, which is like Chord described earlier by Stoica *et al*. [3] (nodes arranged in a ring with routing decided by neighbor information residing in each node). In Wallach's scenario, each node is assigned a nodeId which acts to partition the network into *N* responsible nodes. In the set of nodes, the fraction of nodes that may be faulty is bound by f ($0 \leq f < 1$). He models faults using a constrained-collusion Byzantine failure model, that is, multiple but not all faulty nodes can be acting in concert. His system assumes that each node has a static IP address at which it can be addressed (this excludes dynamically assigned IP addresses, but models can be extended to account for this). Nodes in his system communicate over Internet connections, communicating directly through network-level communication that is assumed to be encrypted between benign nodes. However, malicious parties have control over what is sent on the network-level to and from the nodes it controls. This is a common description of adversaries that will be referred to later in this section.

Wallach declares that secure routing guarantees two things: that replicas of data are initially placed on legitimate replica roots and that a lookup message reaches a replica if one exists. In the context of the model that this paper describes, secure routing requires solving three problems: securely assigning nodeIds to nodes, securely maintaining the routing tables, and securely forwarding messages. The simplest design to secure nodeId assignment is centralizing the nodeId authority, but as this opens up the possibility of single points of failure, this thesis considers a decentralized solution. The problem with decentralization lies in the possibility that an attacker can rejoin the network until it has somehow gained an advantage over the nodeId assignment scheme. Even if secure nodeId assignment is established and malicious parties could only control fN nodes, there is still a probability of f that a legitimate node routes a request through a malicious node which therefore could not reach its proper replica. Wallach describes a relatively high probability solution to prevent locality-based attacks where a malicious party would route requests near a controlled node to get more entries in that node's routing table. Constrained routing sends messages to replicas through all of the neighbors of the source node. Since the nodeIds are random, this should be a geographically diverse set of nodes, and if those nodes all route the message towards its replica, the probability of the message reaching its replica is increased to 99.9%, as long as $f \leq 30\%$.

Wallach goes on to discuss creating mechanisms that actively reduce the proportion of malicious nodes. Every peer-to-peer system must be able to recover a failed node, but this process should be evocable when nodes are identified as running maliciously. He suggests a system should have a way that a benign node can accuse another of malicious acts. It should also be possible, if the node is innocent, that it can prove its authenticity or, if the node is malicious, the other nodes can be convinced to eject the faulty node. He identifies the main issue with devising such a scheme is differentiating normal Internet fabric behavior (dropped packets, etc.) from maliciously ignored messages.

Wallach continued to outline the concept of “fair sharing” for each instance of his system to maintain - both in terms of data storage and bandwidth. This work’s peer-to-peer social network would want to ensure that one node, because of allowances it would have to make on the various computational ability of client systems, is neither lying that its storage is full when it has free space, nor is ignoring requests it has promised to hold. Wallach proposes offering storage incentives as an interesting solution to this problem: require a node to store some amount of data before the rest of the system will store that amount for said node. An alternative is the idea of data quotas to guarantee correct sharing of system resources. Wallach’s random quota auditing scheme is a distributed audit policy that each node has to participate in, in which responsible nodes randomly request data checks to verify that replica nodes are correctly storing their data.

Sit and Morris [6] look at the security of distributed hash tables when their algorithms are executed incorrectly; more specifically in the presence of misbehaving nodes. This introductory paragraph gives one of the more realistic and applicable descriptions of an unsecured distributed system:

> Peer-to-peer systems present an interesting security problem as there is no central system to protect. Instead, the nodes must work together to ensure correct and secure behavior. Unfortunately, deployment on an open network, such as the Internet, implies that there will be malicious nodes in the system. These nodes will try and disrupt the system or subvert it to their advantage. Peer-to-peer systems must be designed to operate correctly even in these situations.

Sit and Morris begin by outlining a general distributed hash table protocol. Routing in this protocol is finding the shortest defined distance between a node and a key and the closest node is classified as the responsible node. By constructing a system of invariants in this protocol as well as the storage system layered on top of it, Sit and Morris describe how innocent nodes can use the framework to identify malicious behavior and broadcast an error in protocol throughout the system in order to fix it.

Next, Sit and Morris model their adversary in terms of its abilities inside their system, helping to form more concrete definitions of the possibilities of attacks and defenses. They stress that detection is the first line of defense, but after that detection happens, they emphasize the need for a careful course of action. Verifying they are only removing malicious and not benign nodes is the main tone of their detection scheme.

Sit and Morris then go on to describe various attack schemes, including incorrect routing routines and updates, partitions, inconsistent behavior, and denial of service. Again, they stress the need verifiability in the protocols a system uses to assign keys to nodes, to update routing tables in the distributed hash table, to add nodes to the system, to distribute data across replica nodes, and to rebalance the system upon node entrances and exits. One verification scheme they propose is the addition of randomness to audit node behavior, similar to Wallach's random quota auditing scheme [5 pp. 52-53].

Sit and Morris provided many different attack scenarios for this thesis to consider, as well as strategies for preventing these attacks and resolving them. In approaching the security of the peer-to-peer social network, the adversary must be solidly modeled such that its behavior in the system can be realistically bound. By building out attack plans and making protocols to detect, prevent, and correct those attacks, the network can actively counter its adversaries.

In the last paper this work reviews on the topic of security, Castro *et al.* [7] delve deeper into the specifics of securing the routing protocol of a system. They examine the issues associated to open peer-to-peer systems without pre-existing trust relationships, because many modern algorithms fail with a small fraction of malicious nodes. They describe and evaluate techniques to ensure secure routing similarly to other authors: secure assignment of node identifiers, secure routing table updates, and secure message forwarding. The techniques presented by Castro *et al.* are not all directly applicable the peer-to-peer social network of this work, but maintain secure routing in situations of 25% maliciously participating nodes. They consider structured overlays as a whole, but fully focus on a Pastry implementation. The overlay model that they describe at a high level is very similar to Wallach's model [5], with nodes assigned nodeIds, responsible nodes declared "roots" of keys, routing tables maintained by each nodes, and replicas of keys assigned randomly across the nodeId space.

After modeling the behavior of their adversary similar to that of Wallach [5], Castro *et al*. go on to define secure routing as a primitive that "ensures that when a non-faulty node sends a message to a key *k*, the message reaches all non-faulty members in the set of replica roots … with very high probability." Once secure routing is achieved, they claim that by adding other existing security techniques such as self-certifying data or a Byzantine fault tolerant algorithm the network can maintain a safely replicated state.

Castro *et al.* name an attack on their secure nodeId assignment protocol a “Sybil” attack, where an adversary builds a large amount of nodes such that they control an unfair proportion of the nodeId space. This thesis disregards their centralized solution to this attack, but considers their solution to increase the cost of such an attack by requiring computational resource consumption for a new assigned nodeId to slow down aggregation of nodeIds. By also setting an expiration date on nodeIds, they argue that an adversary would have a more difficult time building a coalition of rogue nodes. However, this would also put a tax on this work’s social network’s users, as they would also have to update their nodeIds.

Castro *et al.* describe how attackers can utilize routing table maintenance messages to bias routing towards malicious nodes, increasing the probability that messages will get intercepted. In one attack scheme, they consider a node that can intercept probing messages to exploit proximity. However, they describe a defense through ensuring that nodes only receive communication directed toward their IP address. They claim that Chord has an advantage in routing schemes due to the constraints put on nodes’ routing table entries: entries need to be the closest nodeIds to some point in the id space. Therefore, they argue Chord’s disadvantage in exploiting proximity to improve performance is an advantage against adversaries attempting to exploit proximity to mount an attack.

Regarding secure message forwarding, Castro *et al.* begin by defining the probability of a node failing on h hops as $(1 - f)^{h-1}$, where f is the fraction of faulty nodes. Their solution to secure message forwarding is a failure test based on the replica concentration of the destination node during a query. After this relatively inexpensive failure test is performed, if the test returns an answer designating that the node was reached, the query is finished. If it does not, then an expensive redundant routing method is employed to attain a high probability of message delivery. They describe this method as the routing of messages via different neighbors and using nodeIds and random nonces to ensure that messages are correctly received.

Lastly, Castro *et al.* revisit some of the additional security schemes to overlay on secure routing in order to ensure completely secure data delivery. Self-certifying data can be used to check that an efficient routing method has returned the correct result, and if the data does not check out upon delivery, the more expensive redundant routing method outlined above may be employed. Another situation employs a Byzantine-fault-tolerant replication algorithm for each replica group to ensure linearizability for reads and writes while only using the expensive routing technique if the algorithm does not come to an appropriate consensus.

2.4 Visualization Related Works

This thesis explores some of the modern difficulties in measuring consistency and the strategies associated with determining consistency in a distributed system. As much of the visualization work was centered on the implementation and prototyping of the visualization authority, the research and reading was mostly focused to the technical documents associated with the tools utilized to complete the prototype. However, this work reviews some concepts dealing with the measurement of the weakness of a consistency model.

As storage systems across modern distributed systems become more stressed with growing user bases, many developers have shied away from centralized databases to more available distributed databases. As we knew from our initial readings, this comes at a sacrifice to consistency. Golab *et al.* [8] analyze the argument that anomalies caused by relaxing consistency standards are tolerable given the appropriate safeguards paired with their relatively rarity. Services like Amazon's Dynamo have shown, as we discovered, that this relaxation seems to be acceptable when implemented correctly.

Golab *et al.* go on to describe strategies to observe and describe the behaviors of such relaxed consistency models. Here they define the terms *fresh* and *stale* data – *stale* data stemming from weakened consistency standards. Staleness in this case can be defined in two separate ways – as version staleness, in which a piece of data returned from a read is the *k*th oldest value; and as time staleness, where a piece of data returned is time *t* behind the most recent value. Thus from these two staleness concepts, we can work backward from the ACID-like linearizability, and move towards Lamport's definition of atomicity. If a system is *k*-atomic where $k = 1$, then it is linearizable. This work measures this version staleness of weakened consistency with the visualization authority.

Golab *et al.* [9] also created a more advanced model in which they define a new, client-centric measurement for consistency that they call Γ (Gamma). This metric was inspired by the Δ metric, which is another moniker for the aforementioned time staleness measure. This measurement is based off the Lamport definition of "happened-before" and "concurrent." Given operations *op* and *op'*, *op* happened-before *op'* iff end(*op*) < start(*op'*). If neither *op* happened-before *op'* nor *op'* happened-before *op*, then the two operations are concurrent. Given a linearizability infringement, or that an operation produced a stale value, the Δ metric is defined as the difference between start(operation that returned the stale value) - end(operation that wrote the next freshest value), which we will denote Y. If we were to move the operation that returned the stale value Y time units in the past, the operations would be linearizable.

Golab *et al.* note that this does not take into account the possibility that the write operations were out of order due to clock-skew between the machines. This introduces a new possibility that instead of Y, the operations could be out of order such that shifting the end of the stale write forward past the start of the fresh write or, conversely, shifting the start of the fresh write backwards before the end of the stale write. We will denote this value X. They then go on to argue by Occam's razor that any stale value that we encounter in consistency analysis is more likely due to that which moves the given operation least, or by the quantity Min(X, Y). If the minimum distance still yields a stale read, then the linearizability has been compromised. However, if the reorder happens, the authors deem the operations Γ-atomic.

Through this research, it is evident that there are many ways to quantify the linearizability and consistency of models that take liberties to weaken the consistency of data in order to achieve greater availability. The remaining work was developing the visualization authority to understand more.

3. Description of Research Results

Over this chapter, this thesis will discuss our workflow as it pertained to designing and constructing the visualization authority (VA) and a simple communication handler. Next, the two methods used to complete the prototype will be evaluated. Finally, the thesis will discuss future work that remains for Professor Vaidya's research team.

3.1 Initial Design Thoughts

First and foremost the VA needed to be scalable and modular. As many distributed systems have massive amounts of clients or nodes, and each keeps tracks of millions of keys and values, this was extremely important. A minimal interface, something that was easy to understand so that the user could actually benefit from using the tool, was also a major concern. The visualization authority was to be implementation-independent with regard to the observed system. So, the VA was initially completely abstracted from the key-value store. The work in this thesis began by processing communication data from logs (dead analysis), and has left real-time communications (live analysis) for future work.

The idea of a visualization authority was born from our research into the security considerations in a peer-to-peer distributed system. In order to guard against various attacks, some systems may implement a certification authority so that nodes that enter the system can be trusted to adhere to the system's protocol. The drawback of a central authority in these cases is that it presents a single point of failure for the system; if the authority is offline – no new nodes can join the system. However, in the case of the VA, the single point of failure is acceptable since it is not pivotal to the function of the distributed key-store.

In order to process the data that the VA would receive, we devised a communication protocol that the VA would expect to receive. Keeping the modular design in mind, the idea of communication handlers was developed. These handlers would be selected during configuration of the VA, and would be responsible for serving pieces of data to the VA, one piece at a time. A base handler class has been created that has an unimplemented function get_one that is to be overridden by child classes. These child classes should be written by the operator of the VA given her communication needs. This function would be designed to return the latest piece of data seen by the handler, whether that be the next in-order TCP communication, a UDP packet, or a system log line. The VA would utilize the overridden function to receive in-order data for the system.

Once the visualization authority receives this piece of data, the purpose is to display the state of the system given this update. In the linear series example, this includes coloring “get” requests according to how fresh the data is, that is, determining the value of *k* if this is the *k*th value we have seen. A three-color scheme was devised, such that the values denoted with *k* = 1 would be colored green, those with *k* = 2 yellow, and *k* >= 3 red.

3.2 Python-Based Visualization Authority

The initial project language was Python, because of my familiarity with the language. After searching for different graphics libraries, pygtk was selected. Pygtk is an easy-to-use, Python based graphics library. Over the beginning majority of this work, this design choice was tested and issues with this choice were discovered.

3.2.1 Design and Planning

The first VA implementation is the linear series development of a system. An example drawing the design was based upon is shown in Figure 1.

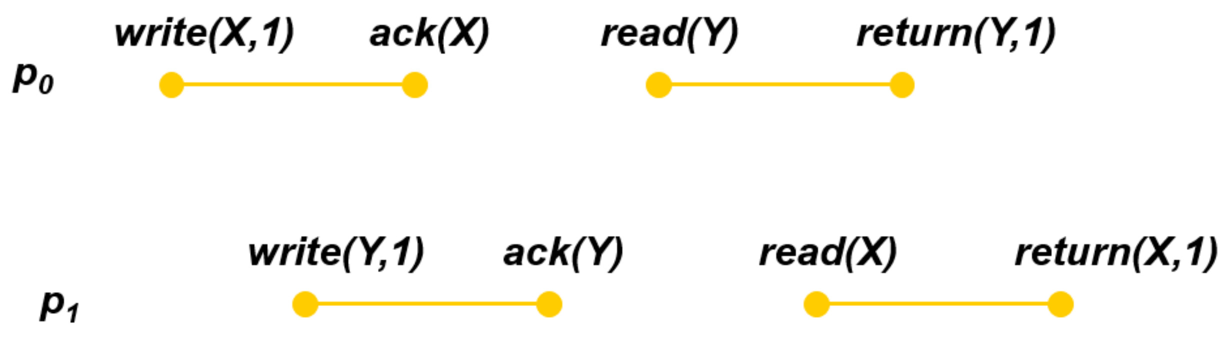

Figure 1 Example of two processes reading and writing variables in a distributed key-store. [10]

This VA has the frame of reference of an observer picking up client requests sent to storage nodes and the respective responses. This work continued by exploring pygtk and determining the data that would be necessary to draw a picture like Figure 1 in real time.

Pygtk has an object called a DrawingArea that was deemed most appropriate for drawing lines and points similar to those pictured in Figure 1. The pygtk library contains many objects which are event-oriented in that each has a set of events whose handlers can be bound to user-defined functions. An example of this functionality utilized in this work is the binding of the “area-expose” event, which is called every time the window is first displayed, to the function that drew the client names on the DrawingArea.

The pygtk coordinate system is setup such that (0, 0) is at the top left corner of the DrawingArea. The Y axis grows vertically downward while the X axis grows horizontally to the right. Each object drawn on the screen needs an (X, Y) coordinate, and that anchor becomes the top left corner of the given object. The area of those objects is then dictated by other properties such as height and width, or radius. Some objects, for example lines, need two sets, (X1, Y1) and (X2, Y2), in order to specify a beginning and ending point.

Text writing in pygtk was another challenge. In order to write text to the DrawingArea, the VA must create a pangolayout object for the target DrawingArea. Each time text was to be written to the screen, the VA would set the text attribute of the pangolayout to the text to be written. Then, it would determine the height and width of the layout by calling the get_size() method provided by the pangolayout object. Dividing by the SCALE constant provided by the pygtk library, the VA could programmatically center the text at the (X, Y) point at which the text was to be drawn.

With the knowledge of the methods to draw the objects to the screen, pseudo code was created to model the window update method. Figure 2 shows the process that we devised to correctly display the visualization.

```
1. Call get_one() to get next piece of data, validate data
2. Calculate the difference between the last timestamp and current, draw difference to screen
3. Identify the request as the initial request or the response to the request
        Initial request:
4.              Mark the process as currently in a request to change line weight
5.              Start the request for the client, adding the appropriate data to the store
        Response to the request:
6.              Mark the process as no longer in a request to change line weight back
7.              End the request for the client, freeing unnecessary data and updating the
                        store to correctly reflect the most recent state of the keys
8. Draw lines and points given the new classification of the clients
9. Repeat from 1. until all data processed
```

Figure 2 Pseudo code for update process

Considering the data necessary to make an accurate visual representation of the data given this process, a data structure to hold the state of the key space was created. The constructed hierarchy to manage the data associated with each key is displayed in Figure 3.

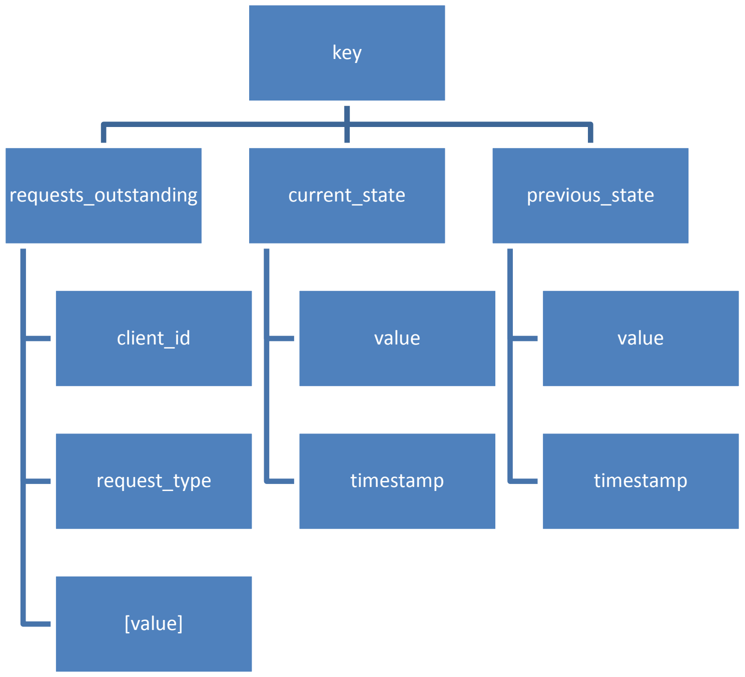

Figure 3 Hierarchy of key data to be used for visualization. "[value]" under "requests_outstanding" is optional depending upon whether the request is a "put" or "get."

After some research on holding dictionaries of dictionaries like this in Python, and what alternatives there might be in order to efficiently use memory, the prototype simply employed the basic dictionary object, nested three times. Initially this seemed taxing on memory usage, but since each class in python is effectively implemented as a dictionary-like object itself, creating a custom class to hold the data is actually more inefficient.

3.2.2 Current State of Implementation

Figure 4 shows the current state of the code with a good example log file created for the purpose of testing the prototype. The GET(x) that returns x=t1 for client 2 is marked as yellow, given that the most recent value for x that the system has seen is “t2” rather than “t1.” All other GETs return the most recent data, and are therefore marked green.

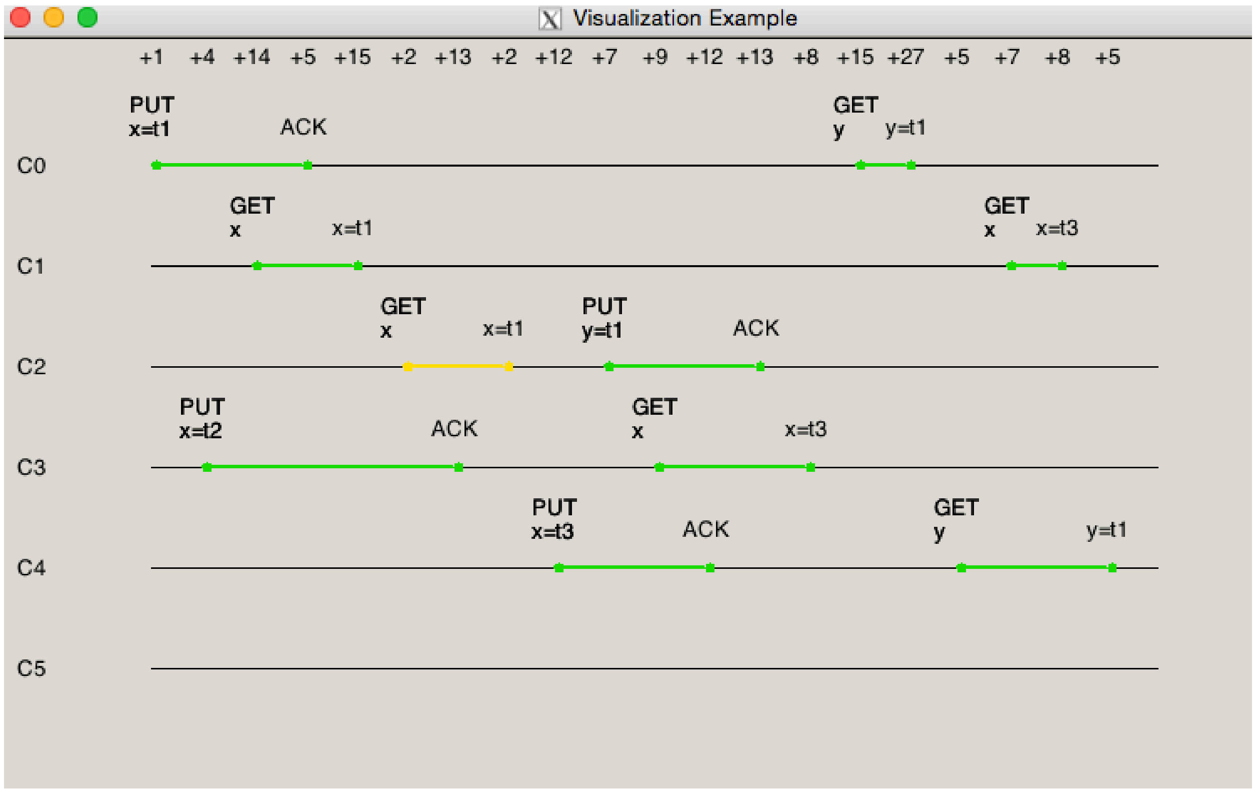

Figure 4 Example of a log with an acceptable amount of fresh data

Figure 5 shows an example of a log with a likely unacceptable amount of stale-ness in data responses. The GET(x) for client 1 returns the second-most-recent value of x, “t1,” rather than “t2,” as does the GET(x) for client 2. Client 5 then makes a GET(x) request after client 4 has updated x and received an acknowledgement again, but the response that client 5 receives is still the “t1” value, which is more than the second-most-recent value, and is therefore marked red. This also happens with the last request from client 0. Clients 1 and 4 then make GET requests on y, which return old values of “t1” after client 0 PUTs y=“t4”.

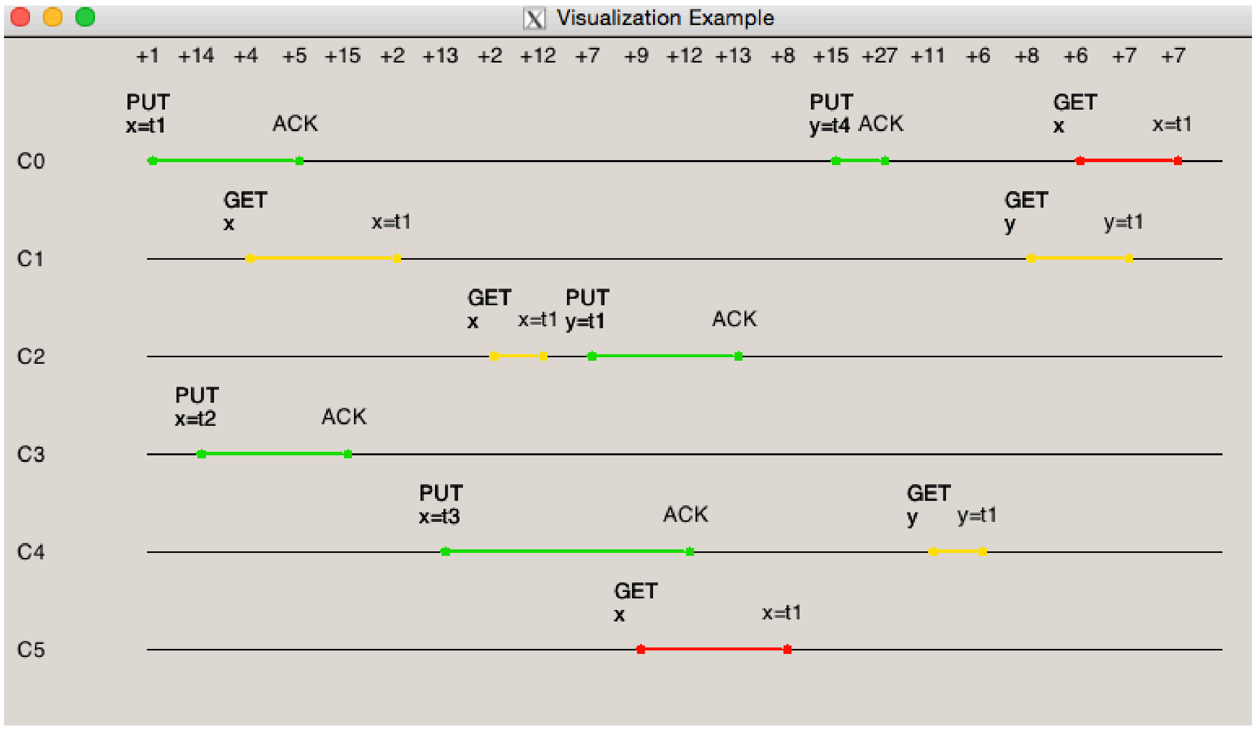

Figure 5 Example of a log with an unacceptable amount of stale data

3.2.3 Concluding Thoughts on Python Implementations

While pygtk was initially a promising candidate for a visualization tool, the interface is clunky and clearly built originally for the C programming language, which gave rise to some complications in Python. It is visible in the pictures that rewriting text created a blurring effect. Upon resizing the screen for more communication, all existing drawings are erased, leading to the requirement that all writing events be cached. This would not be manageable as the system scales. Whenever the VA was demonstrated to the ECE Undergraduate Symposium, the presenter had to constantly keep the mouse moving - otherwise the drawing would not update correctly. Considering all the difficulties in dealing with pygtk, another solution was devised.

3.3 D3-Powered Visualization Authority

The next graphics library the VA utilized was D3, or Data-Driven-Documents, a free JavaScript library whose purpose is focusing on data to drive visualization using HTML, SVG, and CSS. The primary language behind the design of this prototype would be JavaScript, with many elements of HTML and CSS as well.

3.3.1 Design and Planning

The initial design of the D3 line series VA was extremely similar to that of the pygtk line series VA.

However, this was a new graphics library to become familiar with. First, analysis on how the VA should draw items to the screen was performed. Tutorials [11-12] were utilized to help understand the different visualization tools made available by the library.

The main starting point discovered was for the VA to create a scalable vector graphic (SVG) HTML element. The SVG is a graphic that allows the VA to append various common shapes with certain attributes such that it can size, color, align, transform, etc., the shapes as desired. One specific advantage over pygtk immediately identified was the ability to append a unique "id" or group "class" to each element, allowing the VA to easily select and modify or eliminate certain elements by caching only this value where necessary.

One of the library's greatest strengths is its ability to handle and process data, but as of yet the VA does not take advantage of this asset. From work thus far, it appears that D3 works well with static data, not data that is gradually processed as it is currently. With each new communication that the VA receives from its handler, it needs to assess which shapes to draw to the screen and where they should be drawn. This thesis has left a new exploitation of the data capabilities of D3 in the configuration of the VA for future work.

The one change made to the data storage structure for the line series VA in this implementation was to add to the dictionary the x and y values of the end of the stale request point, as seen in Figure 6.

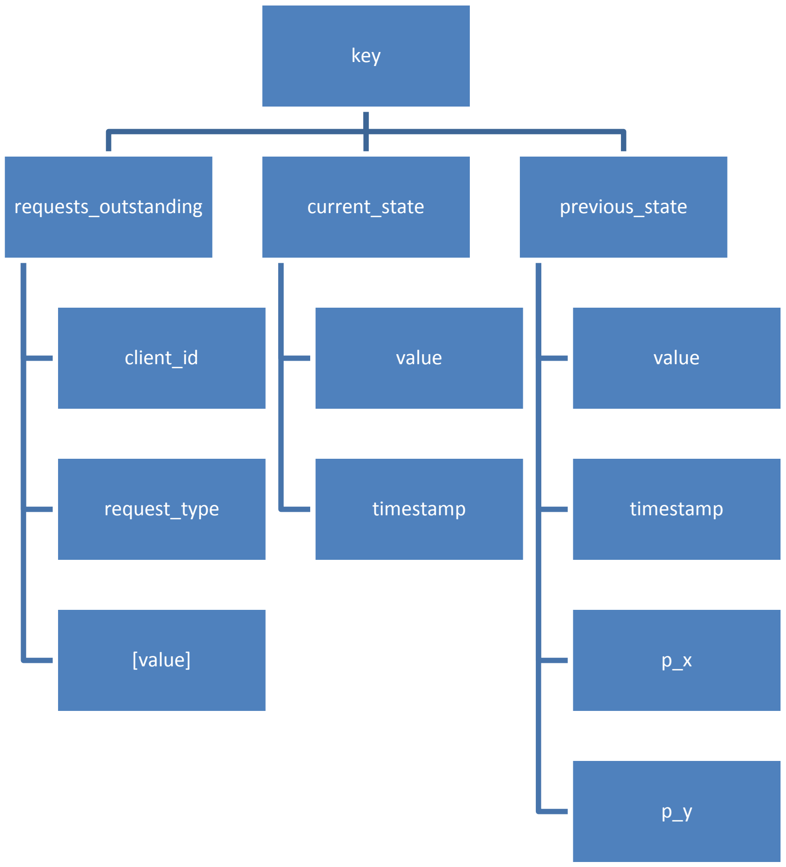

Figure 6 Revised data store

As D3 is a JavaScript library, the VA was executed in the Chrome web browser. D3 supplies a few native data file parsers, including a comma-separated-value parser. This allows a great amount of flexibility in the communication handlers, as this is an easy format to comply too. For the specific “dead-analysis” prototype, logs may be hosted anywhere, as long as the Access-Control-Allow-Origin header is included in the HTTP response to the request for the log. D3 submits an XMLHttpRequest to the target that it is supplied, therefore unless the cross-origin-request policy of the browser rejects the request, the data will be delivered to the VA. This thesis has left the extension of this request functionality as future work with other modes of communicating with the VA.

3.3.2 Line Series Current State

In Figures 7 and 8 the current state of the line series implementation is shown, first with a good (k = 2) log example in Figure 7 and bad (k >= 3) log example in Figure 8, both similar to those in the Python implementation for direct comparison.

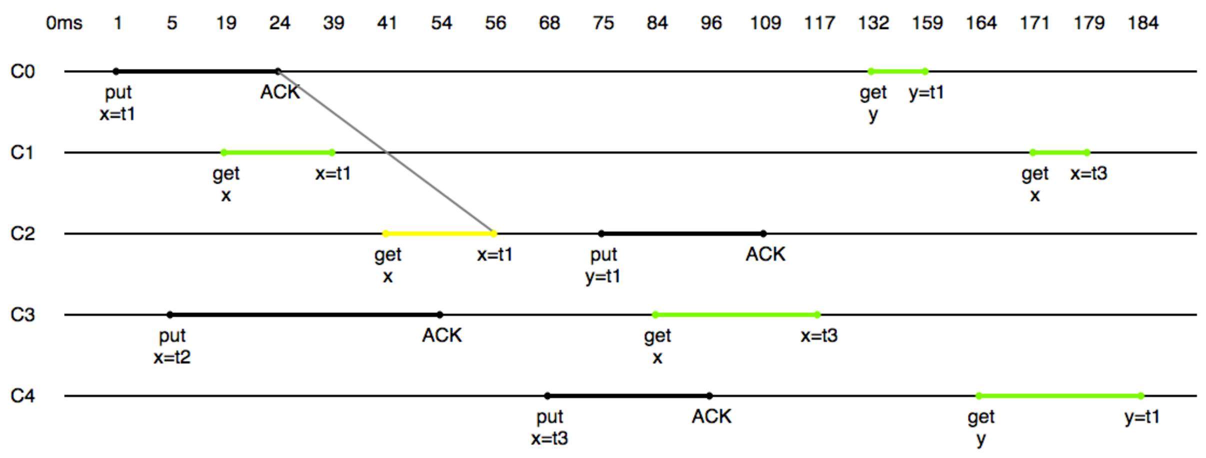

Figure 7 Example of a good Line Series log in the D3 implementation.

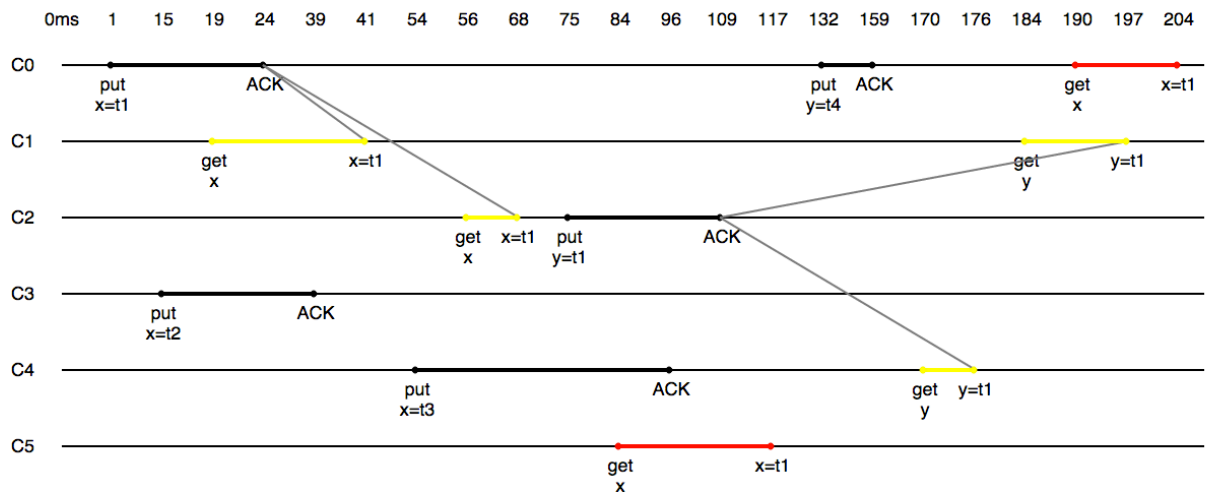

Figure 8 Example of a bad Line Series log in the D3 implementation.

The marked difference here is the addition of the back-trace functionality available with the modification of the data store design. Unlike the original pygtk implementation, it is clear where stale values are being sourced from. This thesis argues the graphics in general look much cleaner and centered, thanks to the advantages from simple but effective CSS attributes such as “text-align.” Another difference pictured in Figure 9 is the ability for the new implementation to dynamically add clients to the window as the communications from the system proceed. As the VA receives messages with new client names, it will automatically add additional space for them.

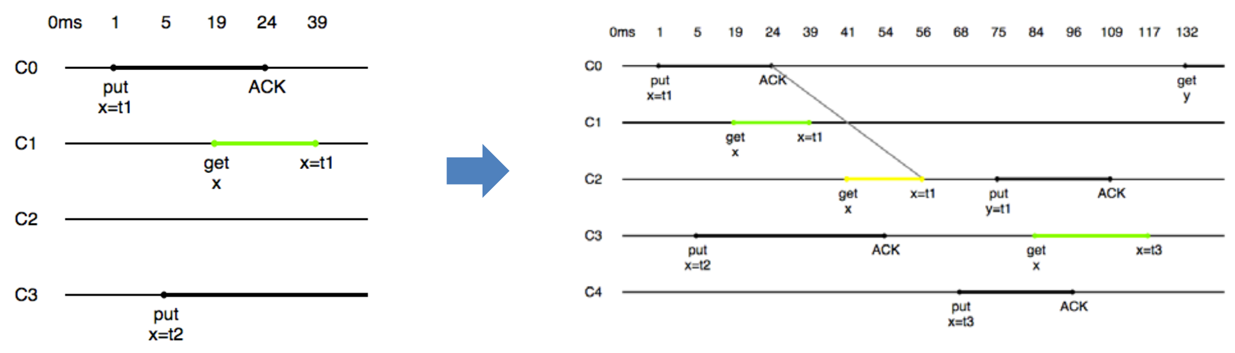

Figure 9 Illustration of the dynamic addition of new clients as communications develop.

A more subtle difference in the two implementations is that due to the ability to select elements on the screen by identifiers or by groups, the VA can remove old and unnecessary data efficiently; freeing up memory to continue accepting communications. Also, none of the existing shapes get lost as a consequence of resizing the SVG element, unlike in pygtk.

3.3.3 Node Layout Visualization

Given the ability to effectively accomplish the goals of the line series visualization, the work to develop the node layout visualization authority that had been infeasible with pygtk began. The most notable difference between the two VAs is the change in perspective: the line series depicts the development of the data store as the clients push and retrieve data to the nodes. The node layout, however, depicts the progression of the data storage nodes themselves: how data propagates through the replicas responsible for a key, and how nodes discover new keys through read or write requests they receive and subsequently send to the responsible group. Since this is an entirely new visualization, this section is broken into two parts: one detailing the design steps considered, and the other detailing the current functionality of this visualization.

3.3.3.1 Node Layout Design Considerations

The initial vision of the node layout VA allowed the user to identify each node and the current state of the keys that node has seen versus the state of the keys in the system as a whole. After seeing the dynamic addition of clients work so well, dynamic client detection was also implemented with this VA. Continuing down this path, as not all the keys in a system would be known before they arrived to the VA in the communications, it must dynamically resize the key display as well. Considering these initial objectives, the data structures the node layout would require were determined.

First, the VA would require another key state store similar to that in the line series VA, so this implementation borrowed some of the same methods from the previous VA. Unlike the line series VA, there was no reason to keep track of requests beginning and ending. The VA also needed a similar additional store for each node to hold their current key dictionary, in order to determine the state of each nodes' keys. The node layout also introduced the need for multiple constants to draw the "node boxes" – the containers for each nodes' data – and the "key boxes" – the boxes to hold the key name and color for each key in each node. These constants include the width, height, and various paddings for the node name and spacing outside the key boxes.

A design scheme for arranging our nodes and keys had to be determined, as well as the VA's behavior as more were detected. The node layout VA implements an approach that attempts to keep the layout as square as possible; then preferring a wide view when not square. It then arranges the node boxes and key boxes to take up the maximum amount of space less the padding discussed in the previous paragraph.

In order to make the dynamic addition work, the VA would need to erase and redraw nodes any time the blocks resized. By utilizing the unique and group identifiers, the VA was able to efficiently accomplish this task. Whenever a resizing is needed, the VA first calculates the new drawing constants given the new number of nodes/keys, then removes all affected blocks and those blocks according to the newly set constants.

3.3.3.2 Node Layout Current State

As the strength of this visualization lies primarily in the ability for the system to develop over time, in Figures 10-17 this thesis provides a storyboard of events in both good and bad log examples.

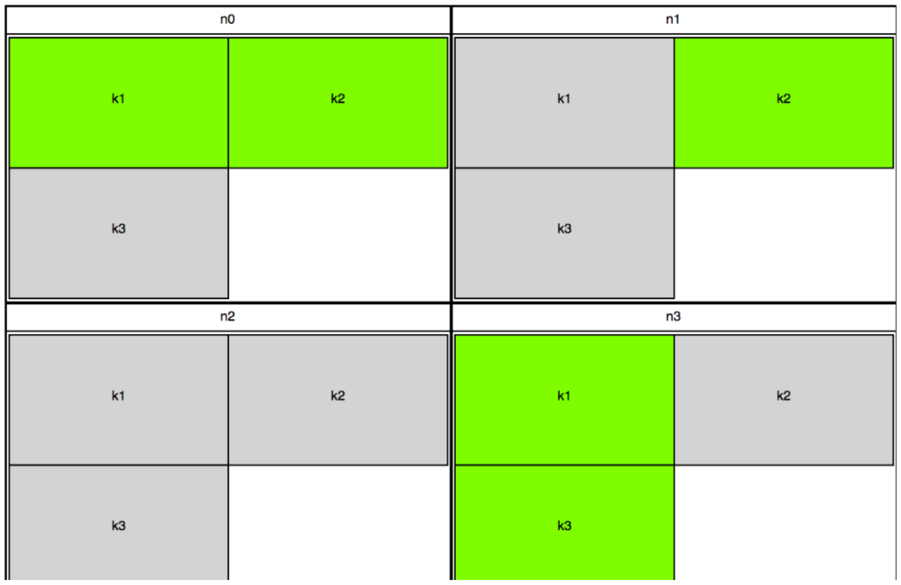

Figure 10 First stage of a good example log after a few messages have arrived.

Figure 10 shows the makings of a good (*k* = 1) log. There are four nodes in the system, three of which have notified the visualization authority that they are storing keys (n0, n1, n3). Those nodes who are aware of keys have the most recently updated values.

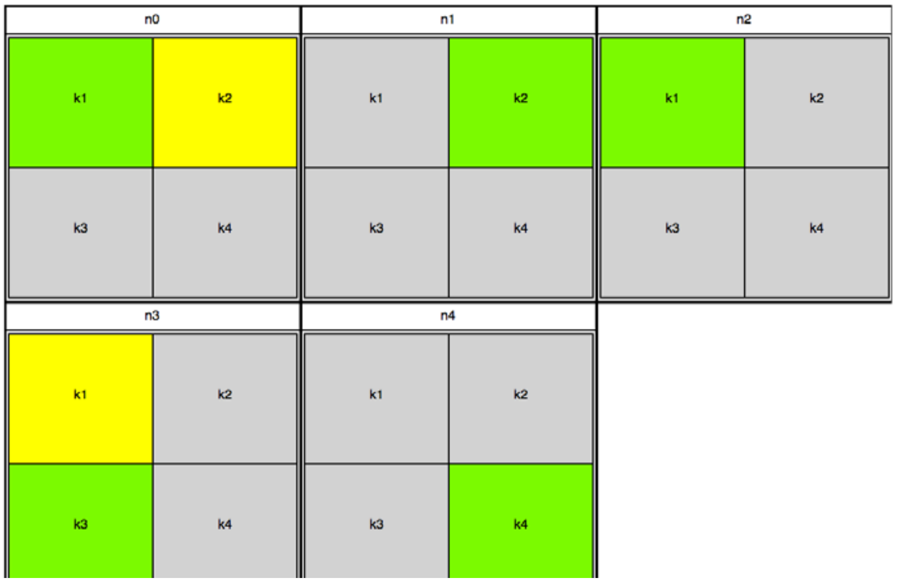

Figure 11 Second stage of a good example log

In Figure 11, the second stage progresses with a few stale values (still *k* = 2) but nothing concerning yet. Notice that n3 has notified the authority of a few keys, and that a new node, n4 has joined the node pool. Also of note is the functionality of the resizing that happened in this stage, moving from a perfect square of four member nodes to five. The visualization authority has resized both the node and key boxes accordingly.

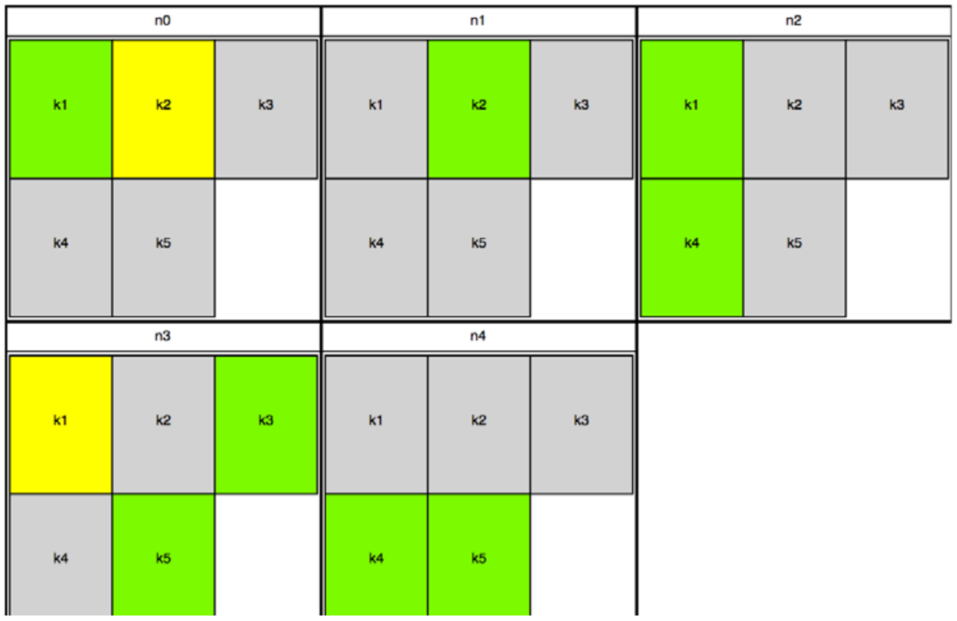

Figure 12 Third stage of a good example log

Figure 12 shows the third node stage in which a k = 2 behavior is still observed. The VA has resized the key boxes to account for the new keys that the system has identified.

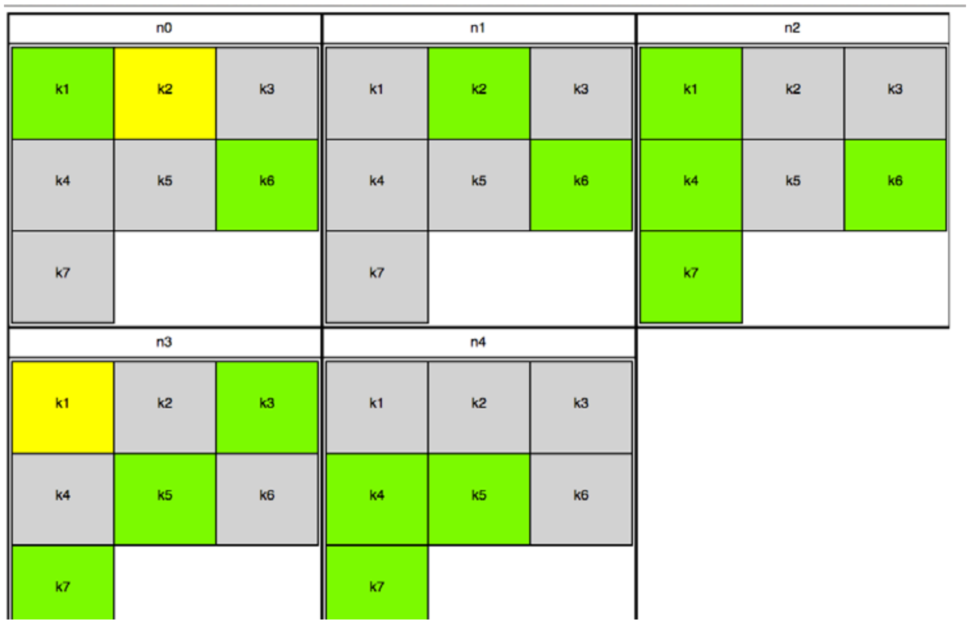

Figure 13 Final stage of a good example log

In Figure 13 the final stage after communication has ended and the resting state of the node visualization is pictured. This situation can be categorized as *k* = 2. There are only two values in n3 and n0 which had not received the correct replicas for (or were not in charge of in the first place).

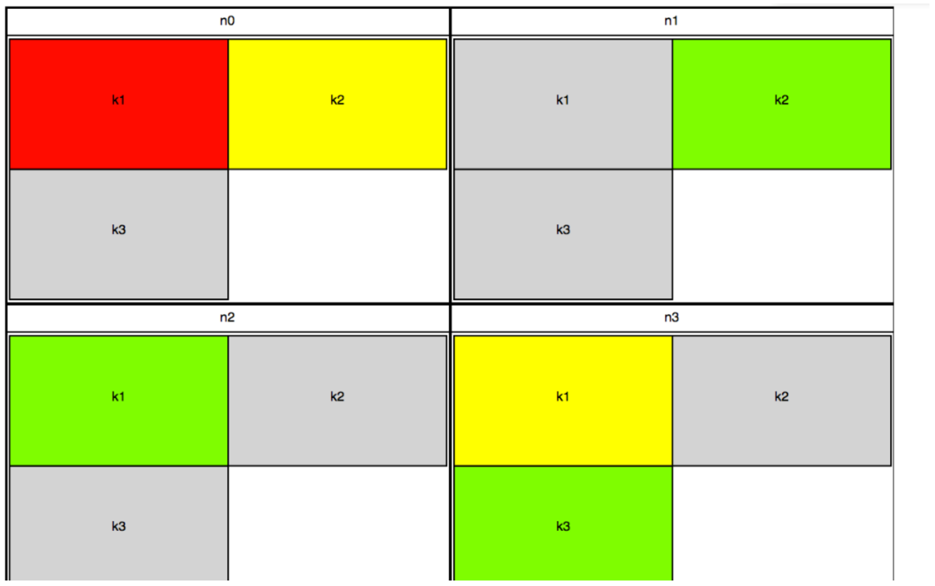

Figure 14 Stage one of an example bad log

A contrast to the good example log is given through Figures 14-17, the first stage of which is pictured in Figure 14. Here n0 has missed at least two key updates on k1, as well as one on k2. Immediately this can be recognized as a *k* >= 3 situation.

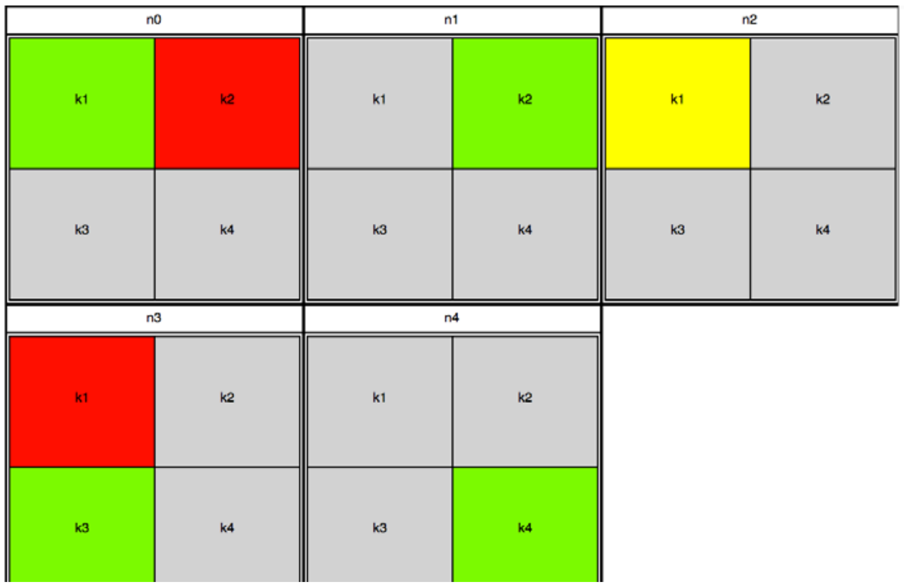

Figure 15 Second stage of an example bad log

Figure 15 shows the second stage and more issues, as both n0 and n3 have missed at least two updates on two keys.

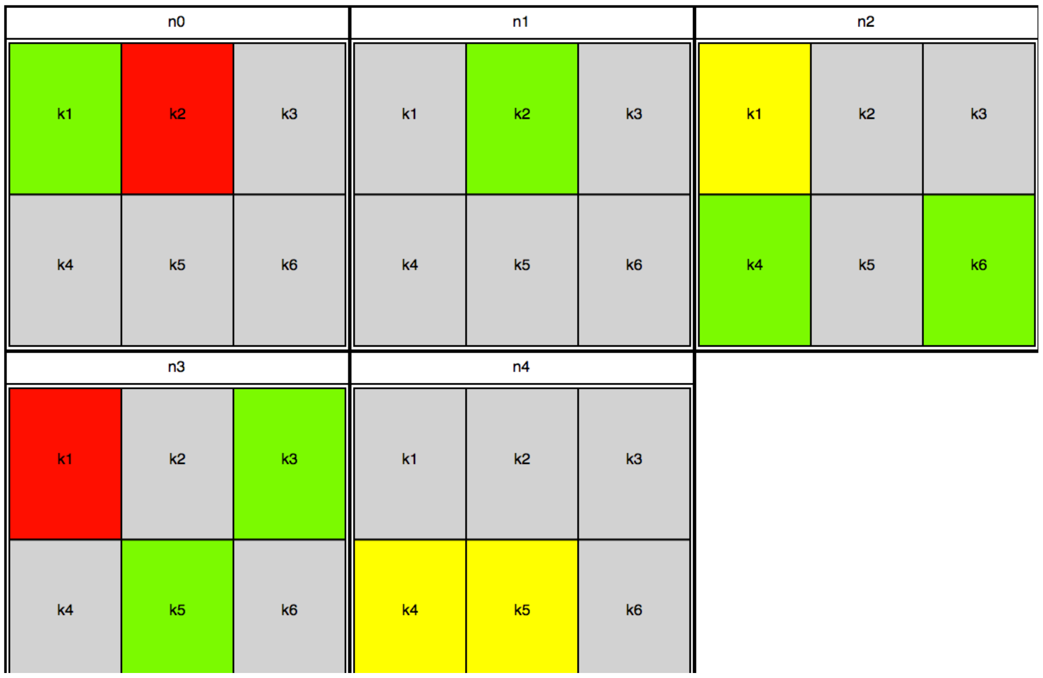

Figure 16 Third stage of an example bad log

Figure 16 depicts the third stage of the bad log example, in which n0 and n3 have still yet to be repaired. Both n2 and n4 both have stale values.

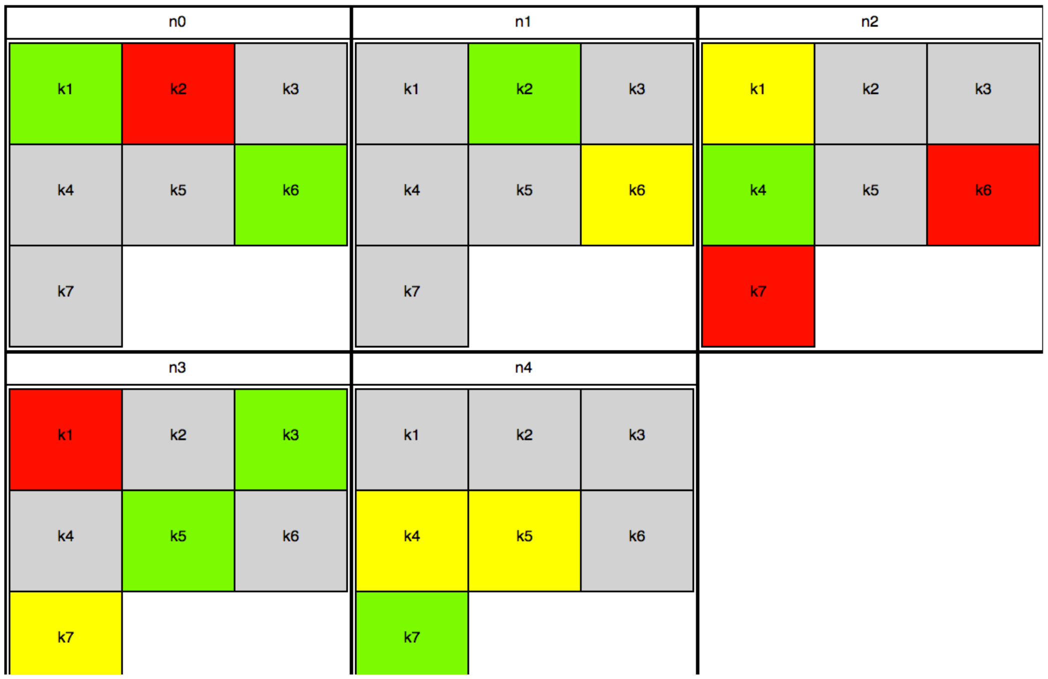

Figure 17 Final stage of an example bad log

Figure 17 shows the steady state of the bad example. It is clear that n0 and n3 were never repaired, n2 has multiple old values, and scattered nodes have stale values. Samples of both line series and node layout logs can be found in Appendix A.

4. Conclusion

Throughout this work, we have evaluated existing problems in distributed systems, focusing at last on the visualization of data consistency in a specific subset of those systems. We have also explored modern data visualization tools and implemented them in practice to produce a modular prototype to be expanded upon by future researchers.

4.1 Future Work

As outlined before, one major strength of our chosen visualization library is data processing. Currently, the visualization authority is not leveraging that functionality, but work should be done to accommodate this in the future.

The next step is to construct a communication handler to proceed the log stage, one which can actually visualize TCP/UDP communications in real time in our distributed system. The XMLHttpRequest tools seem to be the most promising path towards this goal. The handler should be set up to respond to these HTTP requests with appropriate header and the next piece of data. The difficulty in this task lies in ordering the data in the correct manner, such that the next piece of data returned is the most recent communication following that just processed.

One further project would be to assimilate the alternative weakened-consistency measures outlined in the research on current consistency evaluation work. It may be more helpful to provide Δ or Γ metrics in order to provide a full evaluation of each different consistency measure when appropriate.

The last large project identified thus far specifically concerns the line series visualization. Ideally, the VA should continue to receive data without running into memory or space constraints, so utilizing the class identifier method, the VA should implement a garbage-removal-like-scheme in which data too old to be pertinent is removed and other data is displayed instead.

Past these projects, minor graphics improvements such as transitions between communications and dynamic text sizing could be beneficial to the end user of the visualization tool.

References

[1] D. Terry, "Replicated data consistency explained through baseball," *Commun. ACM,* vol. 56, no. 12, pp. 82-89, 2013.

[2] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 6, 2007.

[3] I. Stoica *et al.*, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, pp. 17-32, Feb, 2013.

[4] L. Gladenning *et al.*, "Scalable consistency in Scatter," *Proc. 23rd ACM Symp. on Operating Syst. Principles*, ACM, 2011.

[5] D. S. Wallach, "A survey of peer-to-peer security issues," *Software Security—Theories and Syst.*, Springer Berlin Heidelberg, pp. 42-57, 2013.

[6] E. Sit and R. Morris, "Security considerations for peer-to-peer distributed hash tables," *Peer-to-Peer Syst.*, Springer Berlin Heidelberg, pp. 261-269, 2002.

[7] M. Castro *et al.*, "Secure routing for structured peer-to-peer overlay networks," *ACM SIGOPS Operating Syst. Rev.*, vol. 36, pp. 299-314, 2007.

[8] W. Golab *et al.*, "Eventually consistent: not what you were expecting?" *Commun. ACM*, vol. 57, no. 3, pp. 38-44, 2014.

[9] W. Golab *et al.*, "Client-centric benchmarking of eventual consistency for cloud storage systems," *IEEE 34th Int. Conf. Distributed Comput. Syst.*, pp. 493-502, 2014.

[10] J. Welch, "Distributed shared memory," class notes for CSCE 668, Dept. Comput. Sci. and Eng., Texas A & M Univ., 2015.

[11] L. Francl. (2011, August). "D3 for Mere Mortals" [Online]. Available: http://www.recursion.org/d3-for-mere-mortals/

[12] S. Murray. (2012, December, 30). "D3 – Scott Murray – alignedleft" [Online]. Available: http://alignedleft.com/tutorials/d3

Appendix A: Example Logs

line_good_log.csv

```
Client,Request,Key,Timestamp,Action,Value
0,put,x,1,req,t1
3,put,x,5,req,t2
1,get,x,19,req,
0,put,x,24,resp,t1
1,get,x,39,resp,t1
2,get,x,41,req,
3,put,x,54,resp,t2
2,get,x,56,resp,t1
4,put,x,68,req,t3
2,put,y,75,req,t1
3,get,x,84,req,
4,put,x,96,resp,t3
2,put,y,109,resp,t1
3,get,x,117,resp,t3
0,get,y,132,req,
0,get,y,159,resp,t1
4,get,y,164,req,
1,get,x,171,req,
1,get,x,179,resp,t3
4,get,y,184,resp,t1
```

line_bad_log.csv:

```
Client,Request,Key,Timestamp,Action,Value
0,put,x,1,req,t1
3,put,x,15,req,t2
1,get,x,19,req,
0,put,x,24,resp,t1
3,put,x,39,resp,t2
1,get,x,41,resp,t1
4,put,x,54,req,t3
2,get,x,56,req,
2,get,x,68,resp,t1
2,put,y,75,req,t1
5,get,x,84,req,
4,put,x,96,resp,t3
2,put,y,109,resp,t1
5,get,x,117,resp,t1
0,put,y,132,req,t4
0,put,y,159,resp,t4
4,get,y,170,req,
4,get,y,176,resp,t1
1,get,y,184,req,
0,get,x,190,req,
1,get,y,197,resp,t1
0,get,x,204,resp,t1
```

node_good_log.csv:

```
Node,Key,Value
0,k1,val1
0,k2,val1
1,k2,val1
3,k1,val1
3,k3,val2
2,k1,val2
0,k1,val2
1,k2,val2
4,k4,val1
4,k5,val1
3,k5,val1
2,k4,val1
2,k6,val1
2,k7,val1
1,k6,val1
0,k6,val1
3,k7,val1
4,k7,val1
```

node_good_log.csv:

```
Node,Key,Value
0,k1,val1
0,k2,val2
1,k2,val3
3,k1,val3
3,k3,val2
2,k1,val2
0,k1,val4
1,k2,val4
4,k4,val1
4,k5,val1
3,k5,val2
2,k4,val2
2,k6,val1
2,k7,val1
1,k6,val2
0,k6,val3
3,k7,val2
4,k7,val3
```