

Horizontal Scalability using DxHash for an Analytics Processing System

Dylan Costello

Computer Science and Software Engineering

University of Canterbury

Christchurch, New Zealand

dco119@uclive.ac.nz

Abstract—The issues that arise from a centralised system are many, but a few notable ones are a single point of failure, limited scalability, and a concentration of network traffic onto one machine. WitFoo Inc. currently runs a single machine for their analytics processing system, which they want to distribute in order to limit the problems stated above. However, there are some technical constraints placed on the new system, namely low average memory, load balance between the machines, high processing throughput, and limited database interactions. These constraints help ensure that WitFoo’s processing system is horizontally scalable and can be expanded upon if needed.

These constraints focused the research on distributed hash tables and the consistent hashing algorithms used for them. While this is somewhat outside the normal use case of distributed hash tables, this project looks into how they can be extended to fit this context. The research suggested that DxHash was the best state-of-the-art consistent hashing algorithm, as it allows for predictable backup locations. In simulations on WitFoo’s internal test network, an equivalent DxHash implementation was able to reach $134\% \pm 3\%$ throughput, with only $54\% \pm 3\%$ of the average memory for a five-node system, compared to the current system. This indicates that this implementation is a workable algorithm for WitFoo’s situation, context, and constraints.

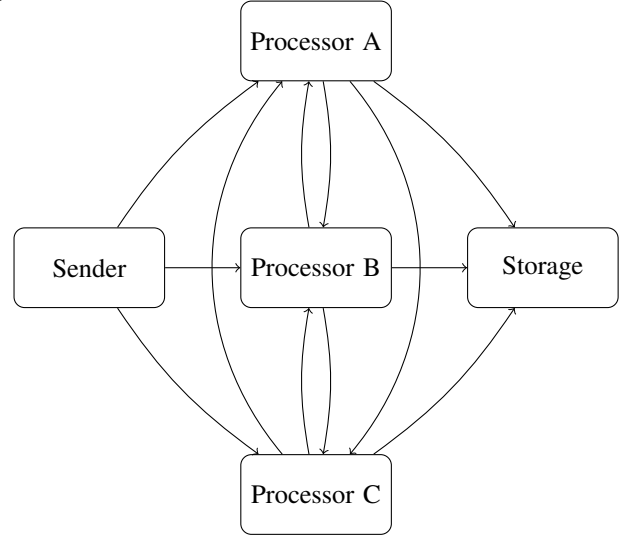
I. INTRODUCTION

WitFoo Inc. is a cyber-security analytics company that currently has a single machine running their analytics processing system. This is an issue, as if this machine has a fault, the whole pipeline stops. It would also limit how the system can scale over time, and it places a heavy load on the machine. For these reasons, WitFoo wants to distribute this analytics processing across multiple machines. However, simply adding machines in parallel causes issues with the implementation. This is because the current system converts third-party *logs* into *artefacts*, and then converts these into *nodes* and *edges*. Here is a basic example of the flow from a *log*, to an *artefact*, to *nodes* and an *edge*.

```
“Jane Doe, logged into server-1”
    ↓
{User: ‘Jane Doe’, Asset: ‘server-1’}
    ↓
Node:{Type: ‘User’, Name:‘Jane Doe’},
Node:{Type: ‘Asset’, Name: ‘server-1’},
Edge:{Source: ‘Jane Doe’, Target: ‘server-1’},
```

These nodes and edges are then saved to another machine for persistent storage. This alone does not cause issues. How-

ever, WitFoo wants to limit the number of insertions into the storage. This means all nodes and edges are stored in memory on the processor, then flushed together in one combined insertion to the persistent storage. This flushing means that if two different machines flush the same information to the storage, the data would be duplicated, leading to issues later on. Therefore, this research project looks into what a possible solution¹ could be that does not allow duplication of data. This means information must be passed between the processors, as shown in the following diagram for a three-machine distributed system.



There are also extra constraints placed on the distributed system to ensure that it can still be effective in an industry setting. Firstly, the artefact throughput and memory use of the total system must be comparable to, if not better than, the single-machine system. This is so WitFoo does not need to purchase and set up a large number of machines to replace a single one. Secondly, memory use needs to be balanced between all machines in the system. This ensures that machines in the distributed system are not left idle while others are overloaded. Lastly, the system should be fault tolerant. While this is not critical for this project, WitFoo would like the system to be able to handle random machine faults, removing

¹Because of the industry nature of this project, it is covered by an NDA; therefore, some information about the context or solution may not be included in this paper.

the single point of failure in the current system. These are the objectives that this project aims to achieve.

To keep this project well contained, there are a number of these issues that will be ignored or have assumptions made, some of which are the following. Firstly the idea that machines can maliciously send information to other machines. This can be ignored as all machines will be WitFoo's machines, which can have random faults, but cannot send malicious messages. Next, is requiring every node to be told global information. This is because again, all machines are WitFoo's, and can be preloaded with the global information, such as hashing algorithms or seeds. Lastly, distributed systems as well as threaded programs can cause a number of race conditions, if not careful. Therefore it can be assumed that all important information has mutual exclusive locks around them.

This paper has the following sections. In Section II, further background on how this project was conducted is explained, and the research questions for this project are given. In Section III, the history of distributed systems is discussed, to give context for how this project arrived at its answer. In Section IV, the final design and algorithm are explained, along with the method used to test them. In Section V, the design is evaluated and discussed to determine whether it meets the given constraints. In Section VI, the project is concluded and future work is suggested. Lastly, Appendix A covers how the project was managed over the year.

II. BACKGROUND AND OBJECTIVES

This is an industry project that has been proposed by and for WitFoo. Therefore, the context and constraints of the solution are given by WitFoo and their analytics processing environment. This project has been done by myself (Dylan Costello), with supervision from University of Canterbury staff. The project does not have existing data to work from. However, access to WitFoo's current system is available to view and improve upon. The objectives for this project have been given by WitFoo and are broken into a research goal, questions, and a metric for each question. Each of these parts helps ensure the objectives are met. These parts are as follows: The objectives for this project have been given by WitFoo and are broken into a research goal, questions, and metrics for each question. Each of these parts are to help ensure the objectives are met. These parts are the following,

Goal: *To develop an equivalent distributed consistency algorithm that works optimally for WitFoo's analytics processing system.*

Question 1: *How would a distributed implementation ensure equivalence to WitFoo Inc.'s current analytics system?*

Metric 1: *A comparison of the final state between the implemented distributed system and a non-distributed system.*

Question 2: *How does this algorithm minimise the memory needed for processing, while balancing this memory between all the nodes in the network?*

Metric 2: *Measuring the memory use over time of each node in an implementation, and calculating each node's average and the standard deviation in the memory used.*

Question 3: *What implementation allows for comparable throughput of artefacts, while maintaining the fault tolerance and scalability of the network?*

Metric 3: *Measuring the number of artefacts an implementation processes per second, compared across multiple network sizes with differing levels of faults occurring.*

III. RELATED WORK

A. History of Distributed Systems

The idea of distributed systems has been around for a long time, and many solutions have been proposed. Some early ideas came from A. Demers et al. [1], who gave the three gossip protocols of direct mail, anti-entropy, and rumour mongering. While direct mail is mostly just called broadcast now, and rumour mongering is mostly unused, anti-entropy has seen further improvements from A. Das et al. [2]. Gossip has also been implemented in Apache Cassandra, as explained by N. Padalia [3]. While these epidemic protocols could work in this context, the speed at which information changes can make these algorithms unusable.

Another early idea for distributed systems was that of a single leader with multiple replicas. This research started with the Paxos consensus algorithm by Leslie Lamport [4], followed by [5], which allows the network to decide who is to be the leader. Following this, other leader election algorithms were investigated, such as [6]–[9]. However, as WitFoo requires load balance between all the machines, and having a leader creates an imbalance in workload, this approach was not looked into further.

The final research area for distributed systems is consistent hashing, originating from D. Karger et al. [10]. This was the first consistent hashing algorithm that focused on caching information for the early internet. Following this paper came Chord by I. Stoica et al. [11] and Kademlia by P. Maymounkov et al. [12], both built on the idea of a distributed hash table. While these specific algorithms can work, they are designed for much larger systems and include multi-hop lookup, which would be overly complex for what WitFoo is looking for. Around the same time, Rendezvous/Highest-Random-Weight Hashing was developed by D.G. Thaler et al. [13], as well as Ring Hash, again by D. Karger et al. [14]. Rendezvous Hash can work well in this context and was used in early simulations to compare consistent hashing against the algorithms stated above. Improvements to consistent hashing have since been researched. However, Rendezvous Hash is still commonly used as a baseline for comparison today.

Newer consistent hashing algorithms, such as Jump Hash by J. Lamping et al. [15], use very little memory but cannot handle random faults. Some of Jump Hash's improved designs [16]–[18] do allow for random faults, but the backup are distributed in an imbalanced way. The Maglev Hash algorithm

by D.E. Eisenbud et al. [19] is perfectly load balanced, but again is designed for much larger systems. Lastly, AnchorHash by G. Mendelson et al. [20] has minimal disruption, but in this context, backup information would need to be placed on all nodes to achieve this minimal disruption, which brings us back to a broadcast algorithm.

B. Research Process

How research is done is almost as important as what research is found. Therefore, this section explains how the papers in this report were found. All topics usually started with general internet searches to gain a surface-level understanding of the topic through articles and webpages. This was done through Google searches and WitFoo's provided 'internet-research' web scraper.

After gaining some surface-level concepts, research papers were reviewed using Google Scholar searches and queries. Once a handful of relevant papers were found, the last step was to look through the papers that cite them to find improvements upon them. This helped ensure that better papers were not missed simply because they had received less attention.

IV. PROPOSED SOLUTION

A. Design and Implementation

DxHash was the chosen consistent hashing algorithm for this project. Therefore, a brief explanation of the algorithm is needed to better understand the conclusions drawn here [21]. Firstly, the set of all machines in the network is N , with the number of machines being $n = |N|$. An element in N is a tuple containing its current running state and its location, or

$$m \in N : m = \langle m.running, m.location \rangle \quad (1)$$

such that $m.running \in \{\mathbb{T}, \mathbb{F}\}$. Therefore, the current working set of machines can be defined as $N_W = \{\forall m \in N | m.running = \mathbb{T}\}$.

Moving on to DxHash, this consistent hashing algorithm uses the given hash as the seed of a pseudo-random number generator (PRNG). This PRNG then generates numbers from zero up to an upper bound called u . This upper bound is the smallest power of two that is strictly greater than the maximum number of nodes n . This can be calculated with the following:

$$u = 2^{\lceil \log_2 n + 1 \rceil} \quad (2)$$

which finds u in terms of n . The PRNG then generates a number i and checks an array called the $NSArray$ at index i . The $NSArray$ is an array of size u that maps the index i to a machine m . The $NSArray$ can also be thought of as a set of tuples, such that

$$t \in NSArray : t = \langle i, m \rangle : i \in \mathbb{N}, m \in N \vee \emptyset \quad (3)$$

Note that not all indices will map to a machine, as u will always be larger than n based on (2). Therefore, indices of n or greater will have no mapping to a machine. Continuing on, if DxHash finds that the machine at index i is not running, it will generate a new number and repeat the process of checking its state. If DxHash finds that the machine at the index is alive,

it will return the index, or a mapped value for the location of the machine. A pseudo-code algorithm for DxHash lookup is found in Alg. 1.

Algorithm 1 Function for DxHash

```

1:  $NSArray \leftarrow \emptyset$ 
2: procedure DXHASHLOOKUP( $hash$ )
3:    $prng \leftarrow newPRNG(hash)$ 
4:    $i \leftarrow prng.next()$ 
5:   while  $NSArray[i] = \emptyset \vee NSArray[i].running = \mathbb{F}$ 
     do
6:      $i \leftarrow prng.next()$ 
7:   end while
8:   return  $NSArray[i]$ 
9: end procedure

```

Next, the set of all possible hash values is H , which in this context can be thought of as the set of integers from 0 to $2^{64} - 1$. Following that, the directory formulation is denoted by the set of all directory entries over the whole network, D , with D_m being the subset of D that contains all of the directory entries on a machine $m \in N$. Each directory entry is a mapping from a hash to a machine, $D \subseteq H \mapsto N$. This allows a *directory-lookup()* function on a machine m , which returns the location of the owner given a hash h , or nothing if it has no entry. The *directory-register()* function, given a location l , a hash h , and an owner o , adds or replaces a mapping from h to o in D_l . Further clarification can be found in the pseudo-code in Alg. 2.

Algorithm 2 Directory Functions

```

1:  $m \in N$ 
2:  $D_m \leftarrow \emptyset$ 
3: procedure DIRLOOKUP( $hash, loc$ )
4:   if  $loc \neq m$  then
5:     return
6:   end if
7:   return  $D_m[hash]$ 
8: end procedure
9:
10: procedure DIRREGISTRY( $hash, owner, loc$ )
11:   if  $loc \neq m$  then
12:     return
13:   end if
14:    $D_m \leftarrow D_m \cup hash \mapsto owner$ 
15: end procedure

```

Next, an artefact a is in the set of all possible artefacts A , such that $a \in A$. The set of nodes in a is defined as a_N , such that $a_n \in a_N$ is a single node in the artefact a . Each node is a tuple of two pieces of information, the set of keys and the information in this artefact, as follows:

$$a_n \in a_N : a_n = \langle a_n.keys, a_n.info \rangle \quad (4)$$

The set of keys is a subset of all possible keys K , such that $a_n.keys \subset K$. Each key uniquely identifies an artefact

node, but different artefacts may contain different keys for the same node. The information is the novel data for the system that will be used for further processing, for example ‘20 bytes sent’.

Finally, there is a total network entity set E , which is defined as a subset of all possible nodes and edges, $E \subset A_N \cup A_E$. Again, there is one for each machine in N , such that $E_m \subset E, \forall m \in N$, and items can be added to this set. This data also gets flushed to the database, which is out of scope here, but flushing does reset E_m afterwards. This is defined with pseudo-code in Alg. 4.

All of this allows us to define the pseudo-code for the core implementation of handling artefacts in this distributed system, as shown in Alg. 3. There is a *hash()* function, which can be any standard digest algorithm. There are also *artefact.extractNodes()* and *artefact.extractEdges()*, which convert the artefacts into nodes and edges. Lastly, there is *sendToStorage(E_m)*, which performs the bulk insert of all the nodes and edges into the storage.

Algorithm 3 Function for Handling the Artefacts

```

1:  $m \in N$ 
2: procedure HANDLEARTEFACT(artefact)
3:    $a_N \leftarrow \text{artefact.extractNodes}()$ 
4:    $a_E \leftarrow \text{artefact.extractEdges}()$ 
5:    $owner \leftarrow \emptyset$ 
6:   for  $a_n \in a_N$  do
7:     for  $k \in a_n.keys$  do
8:        $loc \leftarrow dxHashLookup(hash(k))$ 
9:        $owner \leftarrow owner + dirLookup(hash(k), loc)$ 
10:    end for
11:    if  $|owner| = 0$  then
12:       $owner \leftarrow owner + m.location$ 
13:    end if
14:    for  $k \in a_n.keys$  do
15:       $loc \leftarrow dxHashLookup(hash(k))$ 
16:       $dirRegistry(hash(k), owner, loc)$ 
17:    end for
18:     $sendNode(a_n, owner)$ 
19:  end for
20:  for  $a_e \in a_E$  do
21:     $loc \leftarrow dxHashLookup(hash(a_e.key))$ 
22:     $sendEdge(a_e, loc)$ 
23:  end for
24: end procedure

```

The question now remains whether this implementation answers the research questions and goal stated above. **Question 1** asks for an implementation that is equivalent to the current system. This implementation achieves that by sending all information about a node to a single machine, so there can be no duplication or overwrites.

Question 2 asks whether the implementation can use minimal memory while being load balanced. Firstly, the implementation has low memory use, as the consistent hashing algorithm only needs the state of the network to find a location.

Algorithm 4 Entity Functions

```

1:  $m \in N$ 
2:  $E_m \leftarrow \emptyset$ 
3: procedure SENDNODE( $a_n, loc$ )
4:   if  $loc \neq m$  then
5:     return
6:   end if
7:    $E_m \leftarrow E_m \cup a_n$ 
8: end procedure
9:
10: procedure SENDEGE( $a_e, loc$ )
11:   if  $loc \neq m$  then
12:     return
13:   end if
14:    $E_m \leftarrow E_m \cup a_e$ 
15: end procedure
16:
17: procedure FLUSH( $loc$ )
18:   if  $loc \neq m$  then
19:     return
20:   end if
21:    $sendToStorage(E_m)$ 
22:    $E_m \leftarrow \emptyset$ 
23: end procedure

```

The directories do add some extra memory, but each entry is only on a single machine (without backup), leading to less memory being used overall. This implementation also balances memory by using the randomness of DxHash to distribute the directories and edges, and through how ownership of artefact nodes is assigned: if a node does not yet exist, it is saved locally. Therefore, this implementation should in theory be balanced between all the machines, given enough artefacts to process.

Lastly, **Question 3** asks whether the throughput of the system will still be comparable if the number of machines increases or machines fail. To ensure this, a small change needs to be made to Alg. 3. The change is simple, have *dxHashLookup()* return a list of machines in the order they occur in the PRNG generation sequence. Then use all of these locations for *dirLookup()*, and check whether any of them have data about the owner. Then update all of the top returned locations with the owner of the artefact node, whether the current machine became the owner or found the existing owner. This small update allows information to be placed in backup locations, so it is never lost if a single machine goes down. The extra requests would increase the time the algorithm must wait for responses. However, batching these requests (and all other requests) would reduce this delay.

B. Method

Because this project is based on WitFoo’s context, there is little comparable research. Therefore, the most effective way to test that the proposed design works is through simulations. WitFoo has provided a model network in their system for sim-

ulations in this project. The implementation on this network is written in Go and uses NATS as the message broker between nodes, as this is the standard at WitFoo.

There are five processing machines in total in the simulation network. There is also a persistent storage machine that uses Apache Cassandra to store all information, which again is standard at WitFoo. Each machine has 8 GB of memory and a 4-core ‘Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz’ CPU.

The implementation also had some parameters that were defined for this context. These include a flush interval of 20 seconds, a simulation time of 2 minutes, and a 60-second drain wait until all the processors have fully flushed their information. The simulation was repeated 40 times for each configuration of 1 to 5 processing nodes.

The metrics measured for each machine were the heap memory used, a timestamp, the first and last artefacts handled by each machine, and the total number of artefacts processed.

A comparison function was also created to compare the final state of the database between the distributed and centralised systems. This was done by comparing all artefact keys, to check for duplicates in the distributed system and whether either system had artefacts the other did not. From this, the ratio of correct to incorrect artefacts can be found, giving a measure of how equivalent the systems are.

Because of the number of custom variables that needed to be tested, such as thread count, artefact ingestion speed, drain time, and so on, variability in the answer is great. Therefore, this project chose to limit the CPU utilisation available to each node to 200 % CPU usage.

All files and scripts created for this project have been tracked in WitFoo’s internal git repository, and are therefore easily accessible and auditable for WitFoo’s tests and reviews.

V. EVALUATION AND DISCUSSION

A. Evaluation

The results from the final simulation are shown in Fig. 1, or in Appendix B on Table VIII. These simulations show that the throughput of each node becomes a fraction of the single-node system’s throughput. However, as the number of nodes increases, the rate of decrease flattens out.

This result for throughput makes sense, as any distributed system will have a much higher processing time than a centralised system. In this context, as the number of nodes increases, almost all artefacts will need to request information from other machines to complete processing. This decay of the throughput per node makes sense, as with every added node, the chance of the information being stored locally is $1/n$, leading to the value decreasing asymptotically.

Comparing the results to the research metrics, for **Metric 1**, all tests returned 100% equivalence between the distributed and single-node systems for an ingestion of 10,000 artefacts. This means that **Question 1**, of equivalence between the two systems, is confirmed to be met by this implementation.

For **Metric 2**, the memory per node follows a similar path to the throughput, decreasing towards a limiting value. This shows that memory is being distributed between all the nodes,

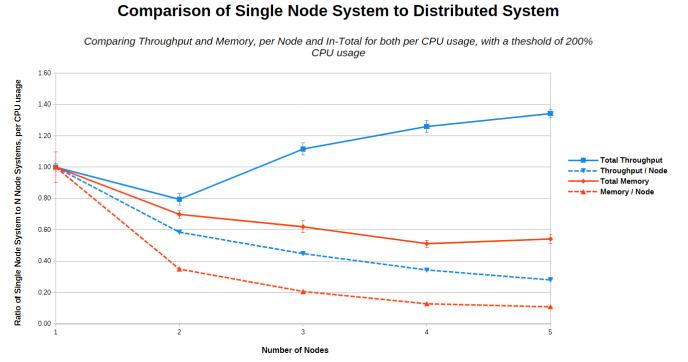


Fig. 1. Comparison of the Single Node system as a ratio to the Distributed System, from two node to five node

reducing it to $54 \pm 3\%$ of the single-node system. The results also had a standard deviation of $6.18MB$ compared to the single-node’s $109.57MB$. This shows that the load balance between the machines is very good, as each machine handles a very similar amount of work to its peers. This shows that **Question 2**, of minimal memory and load balance, has been met.

Lastly, **Metric 3** has not been fully implemented due to time constraints. However, the theory should still hold given the metrics that were measured, namely throughput and scalability, as the artefact rate of the system increases with more nodes, reaching $134 \pm 3\%$ of the original system. The graph suggests that this throughput will continue to increase with more nodes. This shows that **Question 3** has been partially met, supported by theory and some simulation data.

B. Discussion

While there is evidence that this DxHash implementation can work, some questions remain about how well it works. The equivalence between the distributed and centralised systems is limited by the small sample size of artefacts. The fact that it reaches 100% at this scale is a good sign. However, testing with a more dynamic artefact generator could give stronger results. This is because the current test sends artefacts to the processors at a flat rate, where as the real system ebbs and flows with the third-party systems. Therefore, this is a positive data point, but more varied ingestion sizes and speeds would better model a real implementation for WitFoo.

Limiting the maximum CPU was an interesting decision, as it changed the way both systems worked. Without this limit on CPU usage, the single-node system was able to reach 800%. Conversely, as the distributed system was mostly waiting for replies from other machines, increasing the number of threads greatly increased its throughput. While this generated higher numbers for both, it made it difficult to obtain consistent data. The CPU limit was therefore intended to give a fair comparison between the systems. However, it does mean both were somewhat limited in their ability. Running without a limit on CPU usage or the number of threads could show the

real performance of the two systems, which would be more important to WitFoo.

Some of the variables for these tests have limited justification, such as how long the tests were run, the drain time, and the flush period. While these were the same for every test, it is important to ask whether they were the right choices. For example, if the tests were run for longer, would the throughput slow down? Would a longer flush period remove the low-memory advantage of this implementation? These are important questions to answer for a real implementation.

The design shown above is by no means a perfect implementation, there are a number of places where a different approach would produce better results with little to no trade-off. Some examples are batching requests to other machines, looking up local data without a network request, and skipping the directory algorithm when an artefact node has only one key. Therefore, this result is more of a proof of concept, which might explain why the total throughput is lower than expected.

The positive of this result is that it clearly shows that the memory use of the system is much lower and very balanced. This was one of the critical constraints WitFoo had for this project, while throughput was secondary. Therefore, even with the weaknesses stated above, this should still be a good implementation, or at least a good starting point, for WitFoo's distributed processing system.

VI. CONCLUSION

This paper proposes a DxHash-based DHT implementation for distributed ownership in processing. The simulations show equivalence between the current system and the new one, with lower total memory use. This implementation also has similar memory use across each of the nodes in the system, allowing for balance between the machines. Total throughput increases as nodes are added, although it was lower than expected in this proof-of-concept implementation, and the improvements discussed in Section V-B are expected to narrow this gap. The system does show scalability of these metrics as the number of nodes increases. However, with only five nodes, it is difficult to say whether this holds for larger networks. Lastly, the design of DxHash allows for fault tolerance in theory. But due to time constraints, this was not tested. Therefore, this paper finds the proposed distributed processing approach to be a promising foundation for horizontally scaling WitFoo's processing system, with fault tolerance and larger-scale testing remaining as future work.

While the tests show positive results, there is always room for improvement and further study. One obvious direction is to rerun this system with more nodes, more artefacts, and faster machines to see if the results hold at larger scales. More interesting future work could look into Weighted DxHash from [21]. This is where each node has a weight from 0 to 1, and this weight is the probability that it will return that node (based on the same PRNG), otherwise it moves on to the next value. This would help distribute load more evenly depending on how powerful each machine is. This could also

lead into graph theory, to find which weights produce the optimal balance between non balanced machines. Another interesting idea is relaxing the constraint that all nodes in the system have global information, allowing machines to join and leave the network freely. This could help WitFoo's system scale more dynamically as incoming traffic changes over time. All of this future work would help generalise the use of consistent hashing algorithms for distributed processing, allowing for more horizontally scalable systems.

REFERENCES

- [1] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987, pp. 1–12.
- [2] A. Das, I. Gupta, and A. Motivala, "Swim: scalable weakly-consistent infection-style process group membership protocol," in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 303–312.
- [3] N. Padalia, *Apache Cassandra Essentials: Create your own massively scalable Cassandra database with highly responsive database queries*. Packt Publishing Ltd, 2015.
- [4] L. Lamport, *The part-time parliament*. New York, NY, USA: Association for Computing Machinery, 2019, p. 277–317. [Online]. Available: <https://doi.org/10.1145/3335772.3335939>
- [5] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.
- [6] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Trans. Computers*, vol. 31, no. 1, pp. 48–59, 1982.
- [7] S.-H. Lee and H. Choi, "The fast bully algorithm: For electing a coordinator process in distributed systems," in *International Conference on Information Networking*. Springer, 2002, pp. 609–622.
- [8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.
- [9] M. Vojdani and Y. Taj, "Dr: Divided ring leader election algorithm," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2009, pp. 90–99.
- [10] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 654–663.
- [11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 149–160. [Online]. Available: <https://doi.org/10.1145/383059.383071>
- [12] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International workshop on peer-to-peer systems*. Springer, 2002, pp. 53–65.
- [13] D. G. Thaler and C. V. Ravishanker, "Using name-based mappings to increase hit rates," *IEEE/ACM Transactions on networking*, vol. 6, no. 1, pp. 1–14, 1998.
- [14] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 654–663. [Online]. Available: <https://doi.org/10.1145/258533.258660>
- [15] J. Lamping and E. Veach, "A fast, minimal memory, consistent hash algorithm," *arXiv preprint arXiv:1406.2294*, 2014.
- [16] O. Ertl, "Jumpbackhash: Say goodbye to the modulo operation to distribute keys uniformly to buckets," *Software: Practice and Experience*, vol. 55, no. 3, pp. 559–570, 2025.

Milestone	Due Date
Initial Research	11th of May
Research and Design	13th of July
Implementation	27th of July
Create Validation System	7th of September
Validation	21st of September

TABLE I

TIMELINE OF ALL THE MILESTONES FOR THE PROJECT

Milestone	Due Date
Research into Distributed Algorithms	27th of May
Simulations & Comparison of Algorithms	13th of July
Research Select Algorithm	14th of August
Implementation into WitFoo's System	7th of September
Validation and Reporting of Implementation	14th of September

TABLE II

UPDATED TIMELINE OF ALL THE MILESTONES FOR THE PROJECT

- [17] M. Coluzzi, A. Brocco, A. Antonucci, and T. Leidi, "Mementohash: A stateful, minimal memory, best performing consistent hash algorithm," 2024. [Online]. Available: <https://arxiv.org/abs/2306.09783>
- [18] —, "Binomialhash: A constant time, minimal memory consistent hash algorithm," in *2025 IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2025, pp. 87–94.
- [19] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingeroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: a fast and reliable software network load balancer," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. USA: USENIX Association, 2016, p. 523–535.
- [20] G. Mendelson, S. Vargaftik, K. Barabash, D. H. Lorenz, I. Keslassy, and A. Orda, "Anchorhash: A scalable consistent hash," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 517–528, 2021.
- [21] C. Dong, F. Wang, and D. Feng, "Dxhash: A scalable consistent hash based on the pseudo-random sequence," 2023. [Online]. Available: <https://arxiv.org/abs/2107.07930>

APPENDIX

These are the tasks that were created at the start of the project to estimate the work needed, with Table I showing the timeline of the project.

Research and Design

- Researching distributed database systems, estimated time: 20 hours.
- Researching integrating the microservice into the system, estimated time: 20 hours.
- Researching topologies for the system, estimated time: 20 hours.
- Trialling an implementation of the design in a model service, estimated time: 30 hours.
- Understanding the codebase of WitFoo Inc.'s system, estimated time: 10 hours.

Implementation

- Implementation of the system with two nodes, estimated time: 10 hours.
- Implementation of the system with four nodes, estimated time: 15 hours. (Optional)

Validation

- Creation of the validation of *yield* and *t-visibility*, estimated time: 17 hours.
- Creation of a failure system so that random nodes crash, estimated time: 18 hours.
- Creation of the validation of latency and throughput, estimated time: 17 hours.
- Generation of WitFoo Inc.'s internal validation reports, estimated time: 5 hours.
- Generation of reports on the research question metrics, estimated time: 8 hours.

However, due to the nature of the project, changes in what was expected by WitFoo, and further misunderstandings arising from that, these tasks and timelines did not match the work that was needed. Therefore, a revised timeline and task list is presented, somewhat in hindsight. This led to a mapping of the original tasks to updated ones, a change in milestones, and a change in due dates, shown in Table II. Should you consider this unfair, please use the original tasks and due dates.

Research and Design

- (1) Researching distributed algorithms, estimated time: 20 hours.
- (2) Researching the surrounding services of WitFoo's system, estimated time: 20 hours.
- (3) Researching the selected algorithm's implementation, estimated time: 20 hours.
- (4) Simulation implementation of the design in a model service, estimated time: 30 hours.
- (5) Understanding the codebase of WitFoo's system, estimated time: 10 hours.

Implementation

- (6) Implementation of the system with one to five nodes, estimated time: 25 hours. (The two original implementation tasks were combined into one, and their hours added together.)

Validation

- (7) Validating the memory and load balance of the implementation, estimated time: 20 hours.
- (8) Validating the failure system, so that random nodes crash, estimated time: 20 hours.
- (9) Validating the throughput of the implementation, estimated time: 20 hours.
- (10) Generation of reports on the research question metrics, estimated time: 10 hours.

The 'Research into Distributed Algorithms' milestone involved looking into a range of distributed algorithms that could be used by WitFoo, as shown in Table III. This totals 48.5 hours, which covers tasks 1, 2, and 5, with a combined estimate of 50 hours. Specifically, task 1 was around 4 hours over, task 2 was about 3 hours under, and task 5 was 2.5 hours under. Therefore, apart from the misunderstanding in the specification, this milestone was mostly completed successfully.

The 'Simulations & Comparison of Algorithms' milestone involved creating a simple script to run gossip, leader election, and hashing simulations on a personal computer, and measuring the output. This took 53.6 hours, plus an extra 13 hours that were done past the milestone date. However, most of this time was spent running the simulations, which

Week	Hours	Description (Task)
Feb 23rd - Mar 1st	7	Gossip (1)
Mar 30th - Mar 1st	7.5	Understand Code (5)
Apr 20th - 27th	7.5	Hashing, Gossip (1)
Apr 27th - May 3rd	8	SWIM and phi accrual (2)
May 3rd - 10th	N/A (18)	Incorrect Research
May 10th - 17th	8.7	Meetings and leader (2)
May 17th - 24th	9.8	Hashing (1)
Total	48.5	.

TABLE III

‘RESEARCH INTO DISTRIBUTED ALGORITHMS’ MILESTONE, HOURS WORKED

Week	Hours	Description
Jun 15th - 21st	8.5	Gossip simulation implementation
Jun 22nd - 28th	8	Leader simulation implementation
Jun 29th - Jul 5th	13.1	Hash simulation implementation
Jul 5th - 13th	24	Running sims
Jul 14th - 18th	N/A (13)	Continued running sims
Total	53.6	.

TABLE IV

‘SIMULATIONS & COMPARISON OF ALGORITHMS’ MILESTONE, HOURS WORKED

some may not consider ‘work’. This milestone only has task 4 at 30 hours, which is well under the real time taken. But again, running the simulations took 24 hours, so without that the estimate was about right. Therefore, this milestone was a partial success; however, future projects should allow time for running simulations.

The ‘Research Select Algorithm’ milestone had only one task, task 4, which had an estimated time of 20 hours. As shown in Table V, 18 hours were spent on this task. This means this milestone was a success.

The ‘Implementation into WitFoo’s System’ milestone had task 6, which had an estimated time of 25 hours. As shown in Table VI, 38.3 hours were spent, with an additional 5 hours past the due date. The reason for this is that implementing a distributed system in a new environment requires a lot of setup and deployment scripts, so the first week was spent creating the framework for the implementation. Therefore, this milestone should be considered a partial failure.

The ‘Validation and Reporting of Implementation’ mile-

Week	Hours	Description
Jul 20th - 26th	4	Looked into AnchorHash
Jul 27th - Aug 2nd	9.5	Maglev, DxHash, Jump
Aug 3rd - Aug 9th	1.5	Wrote explanation of solution
Aug 10th - Aug 14th	3	Continued writing
Total	18	.

TABLE V

‘RESEARCH SELECT ALGORITHM’ MILESTONE, HOURS WORKED

Week	Hours	Description
Aug 24th - 30th	13.5	Implemented groundwork
Aug 31st - Sep 6th	15.3	Functional system
Sep 7th - 14th	9.5	Bug fixing
Sep 15th - 21st	N/A (5)	Logging & more bugs
Total	38.3	.

TABLE VI

‘IMPLEMENTATION INTO WITFOO’S SYSTEM’ MILESTONE, HOURS WORKED

Week	Hours	Description
Sep 15th - 21st	17.25	Initial tests (7)
Sep 22th - 28th	8	Large batch test (8)
Sep 29th - Oct 4th	N/A (10)	Repeat tests (8)
Sep 29th - Oct 4th	N/A (12)	Writing (10)
Total	23.25	.

TABLE VII

‘VALIDATION AND REPORTING OF IMPLEMENTATION’ MILESTONE, HOURS WORKED

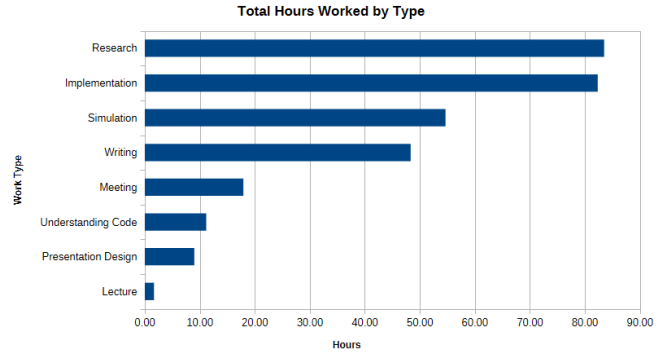


Fig. 2. Bar graph of the total number of hours worked by type

stone had tasks 7, 8, 9, and 10, which had a combined estimated time of 70 hours, with the real hours shown in Table VII. However, because the fault tolerance part was not included, task 9 should not be counted, leaving 50 hours in total. Specifically, task 7 was 3.75 hours under, task 8 was 2 hours under, and task 10 was 2 hours over. However, because half of the work was done past the due date, and one task was not completed, this milestone was greatly missed, making it a clear failure.

Overall, the management of this project has been somewhat poor, with changes to the specification, deadlines set in hindsight, and work done past due dates. However, this was mostly due to the implementation and simulation hours overrunning. Therefore, investigating simulation software, and how to properly implement and test simulations, would have been a useful part of the research. Given that the research side was handled somewhat better, the management of this project was average overall.

This project requires 300 hours, and summing the hours given above, including the time past the due dates, gives 208.7 hours. With 60 hours done before the milestones were set up, and an extra 50 or so hours for writing, presenting, and meetings, this comes to about 300 hours for this course. For a better view of how the hours were spread over the weeks, Fig. 2 and Fig. 3 show the work done by type over the weeks.

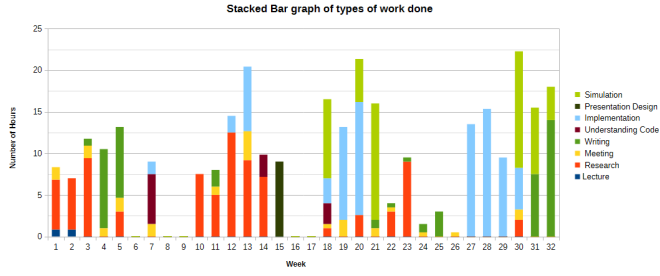


Fig. 3. Stacked bar graph of the number of hours by type per week

Nodes	Throughput (art/s)	CI	Memory (MB)	CI	Avg CPU
1	5,621.92	138.31	349.86	33.95	83.54
2	3,287.05	105.31	122.22	4.34	122.90
3	2,519.69	72.76	72.26	4.34	100.67
4	1,931.42	55.09	44.74	1.97	91.13
5	1,574.46	29.76	37.91	1.92	87.15

TABLE VIII
CAPTION