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. wants research to be done into distributed algorithms to limit the problems stated above for their centralised systems. However, there are some technical constraints placed on this implementation, 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 $134\% \pm 3\%$ of the throughput, with only $54\% \pm 3\%$ of the average memory for a five-node system when normalised by CPU usage, compared to a model of the current system. This indicates that this implementation is a workable algorithm for WitFoo's situation, context, and constraints.

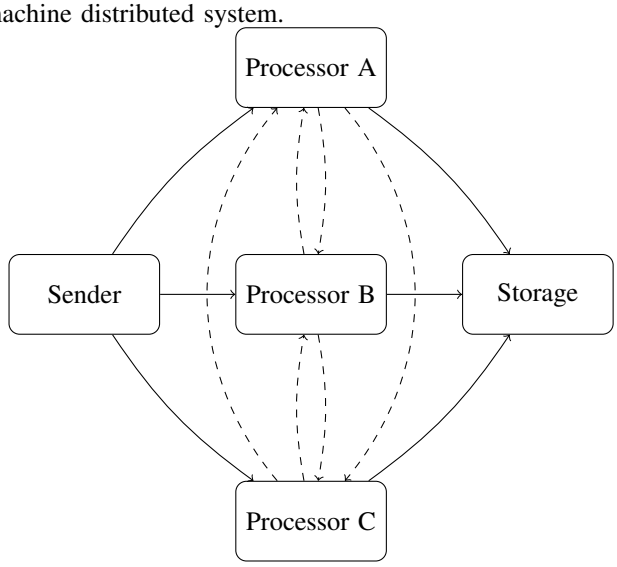
I. Introduction

WitFoo Inc. is a cyber-security analytics company that processes information from third-party systems to identify attacks on these organisations. WitFoo has proposed this research project to find an optimal distributed algorithm to replace their centralised system, because simply adding machines in parallel limits what each processor knows, and can lead to attacks being missed. To further explain why this is the case, a high-level understanding of the system is needed. The current system converts third-party *logs* into *artefacts*, and then converts these into *nodes* and *edges*. The following is a basic example of the flow from a *log*, to an *artefact*, to *nodes* and an *edge*.

Log: "Jane Doe, logged into server-1"
↓
Artefact: {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 in persistent storage for further processes to find attacks. WitFoo wants to limit the number of insertions into this storage, as thousands of these logs are processed every second, which would lead to a high workload for these further processes. This means all nodes and edges are stored in memory on the processor, then flushed together in one combined insertion to the persistent storage. Because of this constraint, the issues with a distributed algorithm start to present themselves.

If two machines flush the same information to the storage without knowledge of each other, the data would be duplicated, causing issues for those later processes. Therefore, this research project looks into a possible solution[1] that does not allow duplication of data. This means that there must be some way for each machine in the network to determine who should 'own' a piece of information, so that all information about it can be sent to that owner. This requires connections 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,

[1]Because 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 current system. This means that the implemented system should process the same number of artefacts per second or more, while using the same or less average memory in total. Secondly, memory use needs to be balanced between all machines in the system, ensuring that machines in the network 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. These are the objectives that this project aims to achieve, and they have shaped the research questions in Section II.

To keep this project well contained, a number of common issues with distributed systems are ignored or have assumptions made about them. Firstly, the possibility that machines can maliciously send information to other machines is ignored. This is because the system will run within WitFoo's network, so only WitFoo's machines will be in that network. This means that these machines can have random faults, but cannot send malicious messages. Next is the requirement for every node to be told global information, or for setup procedures to learn who is in the network. Again, this is because all machines are WitFoo's and can be preloaded with the global information. This also means that the number of machines in the network must be static. Lastly, artefact information in this implementation needs to be saved on disk, or stored in a way that faults do not cause this information to be lost. This is because it would be impossible for an implementation to be equivalent to the centralised system if information is lost. These assumptions have been proposed or accepted by WitFoo, therefore they do not invalidate this implementation.

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 (***RG***), questions (***RQ***), and a research metric (***RM***) for each question. These parts help ensure that the objectives set by WitFoo are met to the level they want. These parts are as follows:

RG: *To develop an equivalent distributed consistency algorithm that performs comparably to the current system, for WitFoo's analytics processing system.*

RQ1: *How would a distributed implementation ensure equivalence to WitFoo Inc.'s current analytics system?*
RM1: *A comparison of the final state of the nodes and edges between the implemented distributed system and a non-distributed system.*

RQ2: *How does this algorithm minimise the memory needed for processing, while balancing this memory between all the nodes in the network?*
RM2: *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.*

RQ3: *What implementation allows for comparable throughput of artefacts, while maintaining the fault tolerance and scalability of the network?*
RM3: *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]. Anti-entropy 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 one of the first consistent hashing algorithm, and it 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 all three of these 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]. Both Ring and Rendezvous Hash can work well in this context, such that Rendezvous was used in early simulations due to its simplicity. 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 do not allow for a good backup information design, which would be needed for any fault-tolerant system. 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.

M. Coluzzi et al. [17], [18] have named Jump Hash, AnchorHash, and DxHash as the state-of-the-art consistent hashing algorithms. This project chose DxHash for its implementation. Therefore, a more in-depth explanation of how the algorithm works is given in Section IV-A.

B. Research Process

A good design can only be achieved if a good research process is conducted. 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. These papers were normally foundational to the research topic, and were read to understand where the topic began. 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 process of looking at a paper's citations to find newer paper was continued untill the state-of-the-art was found. This helped ensure that better papers were not missed simply because they had received less attention. This gathered a wide range of papers, more than are shown above, which gives this paper grounds for its claims.

IV. Proposed Solution

A. Design and Implementation

DxHash by C. Dong et al. [21] 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. To understand the design, some surrounding formalisation is needed. 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

$$m \in N : m = \langle m.running, m.location \rangle \tag{1}$$

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:

$$u = 2^{\lceil \log_2 n+1 \rceil} \tag{2}$$

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

$$t \in NSArray : t = \langle i, m \rangle : i \in \mathbb{N}, m \in N, i \leq |N| \tag{3}$$

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 and will return the empty set. Continuing on, if DxHash finds that the machine at index i is not in N_W, it will generate a new index and repeat the process. If DxHash finds that the machine at i is in N_W, it will return $m = \{m \in N_W : NSArray[i] = m\}$. To meet fault tolerance, backup locations are needed from DxHash, which are not defined by C. Dong et al. [21]. Therefore, a new function is defined to get the top b unique backup locations, where $b \leq n$. Pseudo-code for DxHash setup and lookup is found in Algorithm 1.

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 $h \in H$, or nothing if it has no entry. The *directory-register()* function, given a location $l \in N$, a hash $h \in H$, and an owner $o \in N$, adds or replaces a mapping from h to o in D_l. Further clarification can be found in the pseudo-code in Algorithm 2. Note that these directories are not saved on disk, only in memory and can be lost due to crashes. This keeps the directory functions able to handle requests quickly.

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:

$$a_n \in a_N : a_n = \langle a_n.keys, a_n.info \rangle \tag{4}$$

Algorithm 1 Functions for DxHash

$NSArray \leftarrow \emptyset$
$u \leftarrow 0$
b
procedure DXHASHSETUP(N)
    $u \leftarrow 2^{\lceil \log_2 |N|+1 \rceil}$
    $i \leftarrow 0$
    for $m \in N$ **do**
        $NSArray[i] \leftarrow m$
        $i \leftarrow i + 1$
    end for
end procedure

procedure DXHASHLOOKUP($hash$)
    $prng \leftarrow newPRNG(hash)$
    $i \leftarrow prng.next() \mod u$
    while $NSArray[i] = \emptyset \vee NSArray[i].running = \mathbb{F}$ **do**
        $i \leftarrow prng.next() \mod u$
    end while
    return $NSArray[i]$
end procedure

procedure DXHASHLOOKUPTOP($hash, b$)
    $prng \leftarrow newPRNG(hash)$
    $i \leftarrow prng.next() \mod u$
    $result \leftarrow \emptyset$
    while $NSArray[i] = \emptyset \vee NSArray[i].running = \mathbb{F} \vee |result| \leq b$ **do**
        $result \leftarrow result \cup NSArray[i]$
        $i \leftarrow prng.next() \mod u$
    end while
    return $result$
end procedure

Algorithm 2 Directory Functions

$m \in N$
$D_m \leftarrow \emptyset$
procedure DIRLOOKUP($hash, loc$)
    if $loc \neq m$ **then**
        return
    end if
    return $D_m[hash]$
end procedure

procedure DIRREGISTRY($hash, owner, loc$)
    if $loc \neq m$ **then**
        return
    end if
    $D_m \leftarrow D_m \cup hash \mapsto owner$
end procedure

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, which will not be formalised, as it is outside the scope of this project.

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 entities can be added to this set. This data also gets flushed to the database, which is again out of scope here, but flushing does reset E_m afterwards. This is defined with pseudo-code in Algorithm 3.

Algorithm 3 Entity Functions

$m \in N$
$E_m \leftarrow \emptyset$
procedure SENDNODE(a_n, loc)
    if $loc \neq m$ **then**
        return
    end if
    $E_m \leftarrow E_m \cup a_n$
end procedure

procedure SENDEDGE(a_e, loc)
    if $loc \neq m$ **then**
        return
    end if
    $E_m \leftarrow E_m \cup a_e$
end procedure

procedure FLUSH(loc)
    if $loc \neq m$ **then**
        return
    end if
    $sendToStorage(E_m)$
    $E_m \leftarrow \emptyset$
end procedure

All of this allows us to define the pseudo-code for the core implementation of handling artefacts in this distributed system, as shown in Algorithm 4. Note that the number of backups b for each directory is needed for fault tolerance, but is an open implementation design choice, so it is left as b.

Algorithm 4 first finds all the locations, including backups, where another machine would have registered the owner if this artefact node already had one. The system then checks all of those locations to see if an owner is known. If so, the node is sent to that owner, and if not, the current machine becomes the owner. Finally, every location for each key, including the backups, is updated with the owner. An important note is that only one owner should ever be returned across all the keys and backups. if not, one is chosen and all others are replaced with that owner. Edges are much simpler, as there is only one possible key for each edge, so the owner can simply be the location given by DxHash.

There are also some functions that are not formally defined, such as the $hash()$ 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 $sendToStorage()$, which performs the bulk insert of all the nodes and edges into the storage.

Algorithm 4 Function for Handling the Artefacts

```
1:  m ∈ N
2:  b
3:  procedure HANDLEARTEFACT(artefact)
4:      a_N ← artefact.extractNodes()
5:      for a_n ∈ a_N do
6:          owner ← ∅
7:          for k ∈ a_n.keys do
8:              loc ← dxHashLookupTop(hash(k), b)
9:              for l ∈ loc do
10:                 owner ← dirLookup(hash(k), l)
11:             end for
12:         end for
13:         if |owner| = 0 then
14:             owner ← owner + m.location
15:         end if
16:         for k ∈ a_n.keys do
17:             loc ← dxHashLookupTop(hash(k), b)
18:             for l ∈ loc do
19:                 dirRegistry(hash(k), owner, l)
20:             end for
21:         end for
22:         sendNode(a_n, owner)
23:     end for
24:
25:     a_E ← artefact.extractEdges()
26:     for a_e ∈ a_E do
27:         loc ← dxHashLookupTop(hash(a_e.key), b)
28:         sendEdge(a_e, loc)
29:     end for
30: end procedure
```

B. Meeting Objectives

The question now remains whether this implementation achieves the goal stated above. ***RQ1*** 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.

RQ2 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. The directories do add some extra memory, but each entry is only stored on b machines. This leads to less memory being used per machine, as the memory is spread over a number of machines. 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, ***RQ3*** asks whether the system will still be comparable if the number of machines increases or machines fail. The number of machines in the system is generalised to N; therefore, the design does not change if n increases or decreases. This should also not affect the throughput of the system, given a realistic value of b, such as 3. It also handles fault tolerance, as the backups give multiple locations for all directory information, meaning directory information is not lost or missed because a machine was down.

C. Method

Because this project is based on WitFoo's context, there are few comparable research projects. Therefore, the most effective way to test the proposed design works is through a model implementation. WitFoo has provided a model network within their system for this project.

There are five processing machines in total in the model network. There is also a persistent storage machine and a conductor machine that passes the artefacts to the processors. 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 modelling time of 2 minutes, and a 60-second wait until all the processors have fully flushed their information. The model was repeated 40 times for each configuration of 2 to 5 processing nodes. There was also a single-machine implementation to compare against, which uses the same code, but places the artefact nodes and edges directly into the entity buffer to be flushed.

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 model distributed and model 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, the variability in the results is large. Therefore, this project chose to limit the CPU utilisation available to each node to 200%, and to take the ratio of the metrics above against the average CPU usage. This helps make the two different model systems more comparable, even if they are not running at maximum resources. For clarity, the final value for the comparison between the single-node system and the distributed system with s machines is calculated as follows:

$$finalValue = \frac{metric_s}{metric_1} \cdot \frac{avgCPU_1}{avgCPU_s} \tag{5}$$

Where $metric_s$ is the average given metric from the model with s machines, and $avgCPU_s$ is the average CPU between all the node in that model. With $metric_1$ and $avgCPU_1$ being the same value but in the centralised model.

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. These cannot be presented here, as they fall under the NDA.

V. Evaluation and Discussion

A. Evaluation

The results from the final simulation are shown in Fig. 1, and in Table VIII in Appendix A. These simulations show that the throughput of each node becomes a fraction of the single-node system's throughput. The total throughput drops initially with two machines, but picks up afterwards, before beginning to flatten out.

This result for throughput per node 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 information being stored locally is $1/n$, leading to the value decreasing asymptotically.

Comparing the results to the research metrics, for ***RM1***, all tests returned 100% equivalence between the distributed and single-node systems for an ingestion of 10,000 artefacts. This means that ***RQ1***, equivalence between the two systems, is confirmed to be met by this implementation.

For ***RM2***, the memory per node follows a similar but lower path to the throughput, decreasing towards a limiting value. This decrease is large enough that even the total memory use of a five-machine system is only $54\% \pm 3\%$ of the single-node system. The results also had a standard deviation of 6.18 MB, compared to the single node's 109.57 MB. 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 ***RQ2***, minimal memory and load balance, has been met.

Lastly, ***RM3*** 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. Fig. 1 suggests that this throughput will continue to increase with more nodes, even if more slowly over time. This shows that ***RQ3*** has been partially met, supported by the theoretical fault tolerance of the design.

B. Discussion

These simulations do have some strengths and weaknesses, and this section discusses them to show whether these results are more or less valid. Firstly, the result for ***RQ3*** is limited by the small size of the network. A five-node system is very small, even in WitFoo's context. Therefore, it is difficult to make

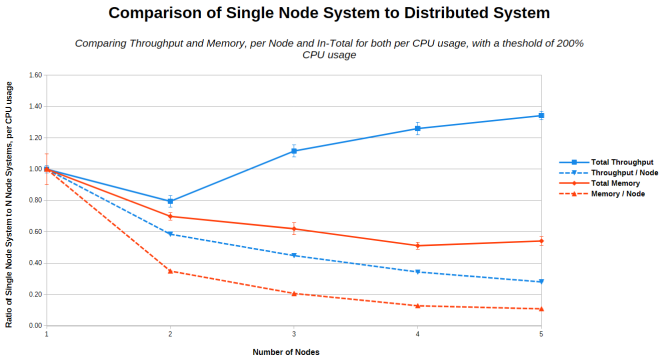


Fig. 1. Comparison of the distributed system as a ratio to the single-node system, from two to five nodes ($\alpha = 95\%$)

strong claims about scalability at this size. Having simulated data for this implementation in a large network match these value, or just having a larger network of 10, 20, or 50 nodes would allow for a stronger claim.

However, the design proposed in Section IV-A and the actual implemented code are not optimal. There are many places for small improvements with little to no trade-off, such as batching requests, short-cutting local lookups, and not using the directory system for artefact node types with only one key. Because these were not included in the implementation for this result, it shows that even a weaker version of this design can still meet the requirements well. This provides a claim for ***RQ3*** that this result is a somewhat lower-limit for the throughput.

A weakness for ***RQ1*** is that the ingested data was the same 10,000 artefacts repeated over and over. This was to keep the data the same between simulations, but it does mean that only a combined total of about 2,000 nodes and edges needed to be created. This is a very small amount compared to the size of the data WitFoo deals with, and could have missed issues in the equivalence between the two systems. Therefore, a much larger data size, or dynamically changing data, would better show the equivalence between the model systems.

The model also had a flushing period of 20 seconds and a run time of only 2 minutes. This limits the justification for ***RQ2***, as longer-running simulations could show stale data accumulating and increasing the average memory use. Therefore, running the simulations for longer would help show that this implementation works over a longer period.

VI. Conclusion

This paper proposes a DxHash-based DHT implementation for distributed ownership in analytic processing. The model results show 100% equivalence between the model centralised system and the distributed one, with only $54\% \pm 3\%$ of the memory use for a five-machine system. This distributed implementation also had more balanced memory use, with a standard deviation of 6.18 MB compared to 109.57 MB.

Total throughput for this five-machine system was $134\% \pm 3\%$ of the model centralised system. While this increase does slow down per node, it shows this system can reach

comparable rates. 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 modelled. 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 interesting direction for future work is Weighted DxHash from [21]. This is where each node has a weight from 0 to 1, which is the probability that a lookup landing on that node will return it (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 machines of differing power. Another interesting idea is relaxing the constraint that the system is a static network, allowing machines to join and leave the network freely. This could help WitFoo's system scale more dynamically as incoming traffic changes over time. This future work would help generalise the use of consistent hashing algorithms for distributed processing, and allow 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. Ravishankar, "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.

[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. Cilingiroglu, 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.

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

- 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, as the project progressed, its scope was refined in discussion with WitFoo, and the original tasks and timelines no longer matched the work that was needed. Therefore, a revised timeline and task list is presented, prepared partly 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. The original tasks and due dates are retained above for comparison.

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 time spent before the project scope was refined, this milestone was mostly completed successfully.

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

Week	Hours	Description (Task)
Feb 23rd - Mar 1st	7	Gossip (1)
Mar 30th - Apr 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

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 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 3, 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.

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

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 22nd - 28th	8	Large batch test (9)
Sep 29th - Oct 4th	N/A (10)	Repeat tests (9)
Sep 29th - Oct 4th	N/A (12)	Writing (10)
Total	25.25	.

TABLE VII
'VALIDATION AND REPORTING OF IMPLEMENTATION' MILESTONE, HOURS WORKED

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' milestone 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 8 should not be counted, leaving 50 hours in total. Specifically, task 7 was 2.75 hours under, task 9 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.

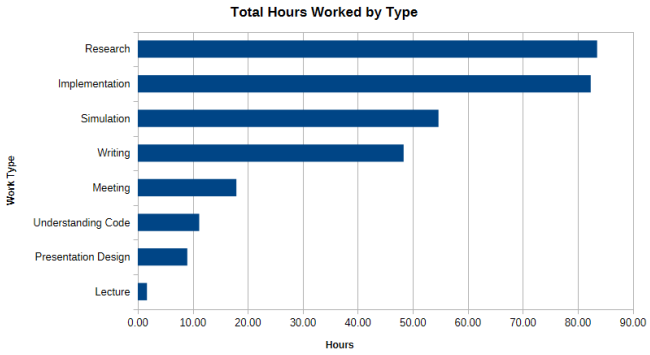


Fig. 2. Bar graph of the total number of hours worked by type

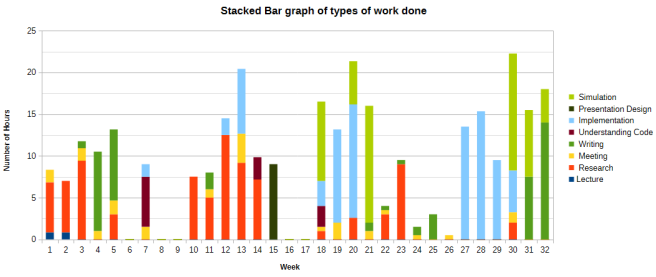


Fig. 3. Stacked bar graph of the number of hours by type per week

Nodes	Throughput (art/s)	CI 95%	Memory (MB)	CI 95%	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
PER-NODE SIMULATION RESULTS FOR ONE TO FIVE NODES