A Node Consistency Algorithm for a Multiple-Kernel Operating System

Andreas Ramsøy

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2022

Abstract

This project builds on the work of last year to modify the kernel of the Popcorn multiple-kernel operating system. This year, a consistency algorithm was developed and implemented in the kernel. This algorithm was used to ensure that every node in the network has the same understanding of which nodes should be connected to the network. Several potential algorithms were analysed using a Python simulation of a Popcorn network. The results of the simulation were used to inform the choice of algorithm. This algorithm was then implemented within the Popcorn Linux kernel with some optimisations to improve performance. The system was tested and was successfully able to correct mistakes in the node list. This algorithm is run periodically by the system such that it quickly detects and repairs errors but does not introduce large overheads to the system. Work was also done towards the goal of encryption. This project is opensource and is available online for the wider Popcorn community.

Acknowledgements

I would like to thank my supervisor, Antonio Barbalace, and Karim Manaouil for all your help over the past two years. Your knowledge and support have been fantastic.

I would also like to thank all my friends and family for their support. Particularly to Marius Ramsøy and Demetris Christodoulou, your insights and support have been invaluable.

Table of Contents

Chapter 1

Introduction

The aim of this project is to develop a joining protocol for the Popcorn operating system. Popcorn is a multiple-kernel operating system based on Linux that allows a network of nodes to be connected. Last year, the project modified the kernel to allow for multiple transport protocols to be used and allowed for the adding of nodes from user-space without requiring a reboot. An algorithm was developed to forward messages through the network to each node. The algorithm was designed to be scalable and not to overburden any nodes. The algorithm was shown to be scalable and could effectively allow for nodes to be added without needing to manually create a connection on every device on the network. This is important, as needing to create a connection manually between each device when there are n devices on a network results in n^2 connections, which quickly becomes cumbersome to the user.

It was highlighted in last year's work that the algorithm did not provide guarantees of consistency. This is in terms of ensuring that every node has the same view of which nodes are connected to the network. Should an error occur that violates consistency, the system did not have a mechanism to correct it. This year, a mechanism for correcting errors was developed. Existing work was researched, and a set of algorithms was created and analysed in a Python simulation. The best algorithm was selected and implemented within the Linux kernel. The system was designed so that it could automatically detect differences in the nodes' view of the network and resolve them. Progress was also made towards the goal of encrypting messages. This project is opensource and is available on GitHub for the wider Popcorn community.

The next chapter explores background material for this project including the Popcorn operating system, last year's work and the joining protocol. In chapter 3, we discuss related work including how Popcorn fits into the field of multiple-kernel operating systems, and consensus algorithms. Chapter 4 discusses the consistency algorithms chosen. In chapter 5, we discuss the simulation that was implemented to test these algorithms. In chapter 6, we look at the results from the simulation and decide which algorithm should be implemented. In chapter 7, we discuss how the system was

implemented into the Popcorn operating system. Chapter 8 discusses the progress made towards encryption. Finally, in chapter 9, we conclude the project and summarise the progress and contributions of the project.

Chapter 2

Background

2.1 Operating Systems

All the fundamental services of the computer's software are provided by the operating system kernel (e.g., scheduling, memory management, inter-process communication) [1]. Operating systems must be exceptionally fast as to not introduce overheads to the programs that users run. They are difficult to debug and so should ideally be free from errors [1].

A so-called micro-kernel is an operating system designed to be as small as possible. As many services as possible are implemented as applications as opposed to within the kernel. This is called the principle of minimality and is used to make the development of micro-kernels easier to manage [2]. This is in contrast to larger operating system kernels, known as monolithic, where their large size and interdependencies mean an error in one system can cause errors in others [2]. Developing these systems in user-space means that the operating system is better able to detect errors and better able to recover from them.

Micro-kernels include a messaging layer for inter-process communication. Signals can be used for inter-process communication [2]. This is when a numerical value conveys a command. Signals are very fast but are more difficult to maintain as all signal handlers must be updated if the value changes [2]. Micro-kernels most often use message passing, allowing for arbitrary messages to be sent. This makes the kernel more maintainable and easier to change in future [2].

Multi-kernel operating systems are a network of independent cores that do not share resources at the lowest level [3]. They are better suited for heterogeneity of hardware, since the use of message passing allows them to not be restricted by the differences in the hardware design of different processors [3].

2.2 Popcorn

Popcorn is a multiple-kernel operating system based on Linux. It provides a single system image to the user. This is when a single view of the system is presented to the user, despite being split across multiple processors or groups of processors [4] [5]. Each node, that is a processor or group of processors, runs the Popcorn operating system with a single cache coherent memory linking them together [4].

Popcorn allows for heterogeneity between nodes, meaning that different nodes can use different instruction set architectures (e.g., a node with an ARM processor can share data with a node with an Intel processor) [4]. Heterogeneity allows for different processes to be migrated to processors that are better tailored to particular tasks [4]. The Popcorn OS is specific to the processor architecture that it is deployed on in order to support the differences in memory architectures [4]. Popcorn allows for the sharing of pages of memory using a cache coherency protocol. This protocol assigns ownership of memory to a particular node, known as the origin node, but facilitates the transfer of memory to another node, known as the remote node.

The Popcorn messaging layer only allows for single node-to-node communication but not broadcasting to multiple nodes [4]. For this reason, every node has a single connection with all other nodes in the network.

2.3 Summary of MIP 1

The first part of this project (MIP 1) worked to modify the existing Popcorn operating system to allow for multiple communication protocols to be used at the same time by nodes. Some additional data structures were added to support this. It was also modified to allow nodes to be dynamically added to the system without requiring the reloading of the kernel module.

Previously, Popcorn required all connected nodes to use the same communication protocol (e.g., TCP, RDMA, etc.) between all nodes. The modifications allowed for independent protocols to be used for different nodes. This was done in such a way that only the protocols being used are loaded. Once the network no longer has a node using that protocol then the protocol is unloaded, meaning that no additional resources are used.

A data structure was used to contain all the information regarding a node, such as the node index, address, transport protocol, and send/receive handlers. This was known as a `message_node`. Each node was stored on what is referred to as the node list. This is a dynamic structure that consists of an array of pointers to message nodes, allowing for extremely quick access. Along with the array there was also a pointer to another

node list. This allowed for an arbitrary number of nodes to be connected within Popcorn.

Before last year's project, Popcorn would only allow for a list of nodes, set when loading the message layer module, to be connected to. The module would need to be unloaded to allow for any changes to this list. This was changed to load with no other nodes attached and established a joining protocol to allow other nodes to link to the network. A proc file was used to send commands to the kernel module. Several joining protocols were considered but the final solution was chosen due to its scalability. It achieved this by each node forwarding messages to just two other nodes in the network. Each node then forwards this message to another pair of nodes until all nodes have established a connection. Each node already on the Popcorn network would attempt a connection while the node outside the network would listen for a connection. This mimicked the existing Popcorn implementation where a node with a lower node ID would attempt a connection, and the other was listening for a connection.

Evaluation of the implementation showed that the new features, which require some extra checks to take place, caused minimal slowdown to the system when compared to the previous version.

The first half of this project did not provide any method of correcting errors that may occur when adding or removing nodes within the joining protocol. It is the aim of this year's project to provide this error correction. The work builds on the progress made last year and so must continue to allow for an arbitrary number of nodes and for multiple protocols to be used.

2.4 Joining Protocol

Last year a protocol was developed to allow nodes to be dynamically added to the network from user-space. The protocol created a binary tree structure in which messages were forwarded in a distributed manner meaning no single node has a significantly higher burden of processing.

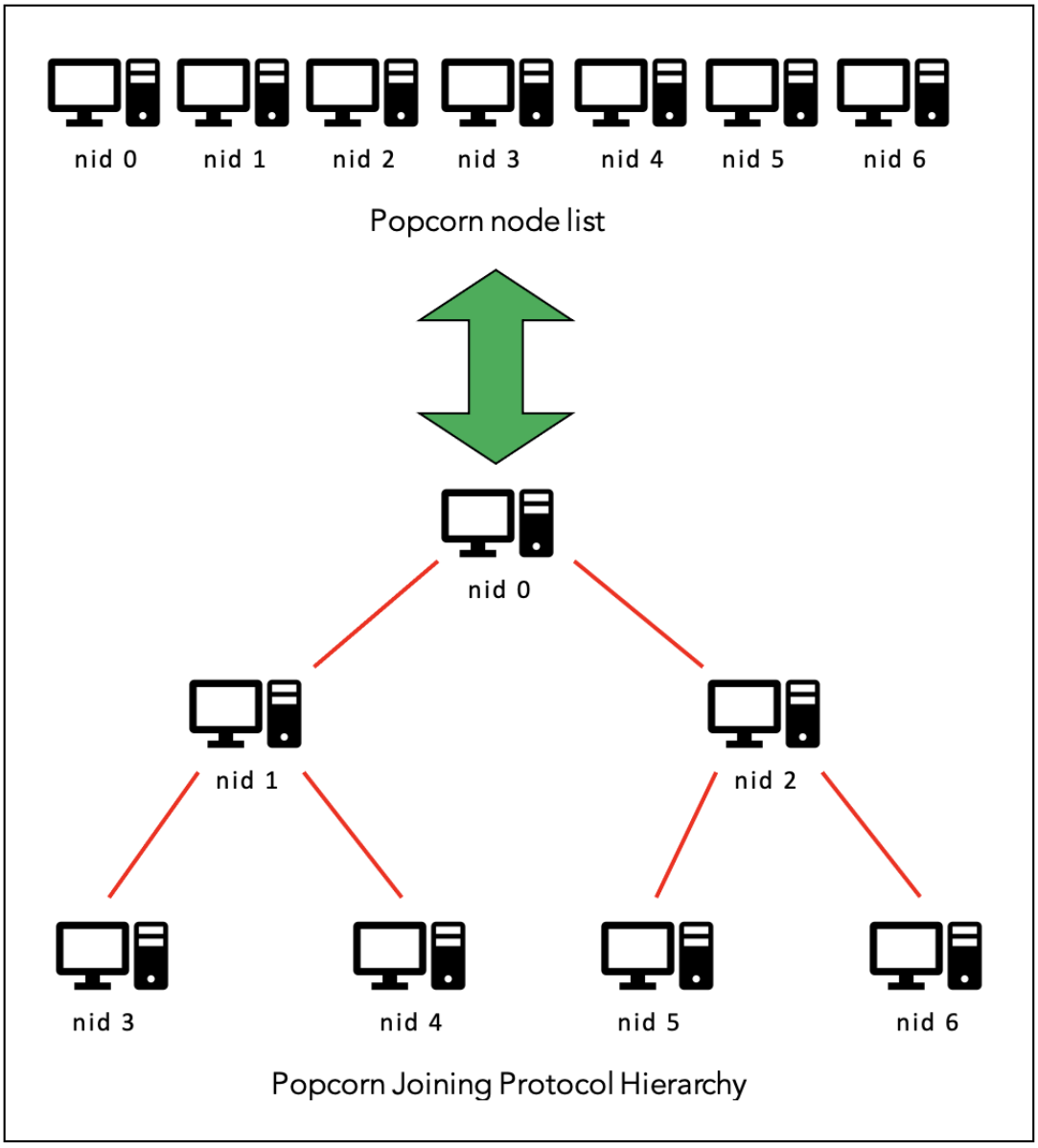

Figure 1: The upper structure shows the node list where each node is ordered in a single list by their node ID (nid). The lower structure shows how the node list is translated into a hierarchy, or binary tree, for the joining protocol. The red lines in the lower diagram show the path that messages take.

Figure 1 shows how the structure that holds the information about the nodes, known as the node list, is translated into a binary tree structure where each node forwards to only two nodes. The node list may contain gaps where previous nodes have left. When there is a gap the parent node in the tree takes responsibility for forwarding the commands for the missing node. For example, in Figure 1, if node 2 (`nid 2`) was missing then node 0 would forward messages to node 1 (as before), node 5, and node 6. The first node in the list is referred to as the instigator node. The instigator node is the first node to establish connections and begins the process of propagating the

command through the network. Should the instigator leave the network then the first node in the node list (the lowest `nid`) becomes the new instigator node.

Assuming all nodes have the exact same node list then they will forward to the correct nodes and know which node the instigator is. The existing algorithm provides no method to ensure that every node on the network has the same node list. This is referred to in this paper as maintaining a consistent node list and is the aim of this project.

The joining protocol was controlled by a proc file. A proc file allows a user to send data to a kernel module. It is an interface that appears to the user as a file. This allowed the user to write commands such as: `echo "add 192.168.1.24 socket 100" > /proc/popcorn_nodes`. This would trigger the adding of the node with that address.

The key differences with the Popcorn joining protocol are that the hierarchy of command propagation means that nodes earlier in the node list are more likely to be correct than those further on. Another difference is that multiple communication protocols can be used. For example, the transport protocol linking node 0 and node 1 in Figure 1 can fail but the node is still active and reachable by other nodes. In this scenario the propagation method of the joining protocol would not work to find errors. A solution could be to remove any such nodes, however this would result in quickly removing a large number of nodes. The fact that the protocol does not change any node IDs when a node leaves means that a mechanism to provide consistency only needs to be eventually consistent. This means that we can afford to value the performance of the system over the strict guarantee of consistency.

Chapter 3

Related Work

3.1 Related Operating Systems

There are many operating systems related to Popcorn; it is important to understand the Popcorn operating system within the context of existing work for this project.

3.1.1 Roscoe

Roscoe was a distributed operating system designed to share computing resources in a non-hierarchical manner [6]. Roscoe worked by sending messages between nodes on the network. All processors were required to be the same (however peripherals could differ) – this is referred to as a non-heterogeneous setup. Memory was not shared between nodes although a paper written in 1978 states that this decision was mostly due to hardware constraints of the time [6]. The user was not informed of where a process was running and so were presented with what appeared to be a single, powerful, computer. This is known as providing a single-system image [6].

This was similar to Popcorn in that it allows sharing of computing resources and single-system image. It differed in that it does not allow for a heterogeneous setup nor was able to share memory between nodes.

3.1.2 Barrelfish

Each core runs an instance of Barrelfish and uses message passing to maintain consistency between instances [7]. Barrelfish is a multi-kernel operating system. Unlike Popcorn, Barrelfish is not heterogeneous. However, heterogeneous versions have been proposed [8]. Messages within Barrelfish are highly optimised to the hardware architecture in order to increase speed [7].

The primary differences between Barrelfish and Popcorn are that Popcorn is based on Linux whereas Barrelfish was built from the ground-up, and Popcorn allows for heterogeneous setups.

3.1.3 Mach

Mach is a microkernel that uses message passing for inter-process communication [9] [10] [11]. It does this by using finite length queues of messages known as ports [12] [13]. Only the processes that require the port are allowed to access them. Mach has the advantage of being able to extend the message passing transparently across a network [10].

The differences between Mach and Popcorn are that Popcorn is based on Linux and is a multiple-kernel operating system instead of built from the ground up and being a microkernel.

3.1.4 Neutrino QNX

Quick-UNIX, or QNX, is another microkernel which uses message passing for inter-process communication [14]. The messages are transported using a messaging bus. Each service is modular and communicates through messages [14].

QNX being a microkernel is the key difference with Popcorn, which is a multiple-kernel operating system.

3.1.5 Amoeba

Amoeba is a system where all resources are automatically managed by a distributed operating system [15]. Consequently, users do not know which processor their programs run on, or how and where their files are stored in the system [15]. Amoeba provides a combination of the processor pool and workstation model where users can login to a particular machine but also run large jobs on a pool of processors [16] [17]. Amoeba makes use of heterogeneity by using different machines for specialised purposes e.g., devices with large storage disks are used for file storage [15].

Amoeba distributes the processing using a processor pool; this differs from Popcorn which allows for processes to be migrated between nodes. Amoeba is also a distributed operating system rather than following the multiple-kernel architecture.

3.1.6 Plan 9

Plan 9 allows for heterogeneity; different processor architectures can join the network running Plan 9 [18]. Messages are transferred between nodes in a high-level way, e.g. text, when possible, as this simplifies the kernel when dealing with different processor architectures [18]. Plan 9 interacts with services as if they are files and uses file

operations as such. This means one simple, well understood, secured protocol can be used to access almost all services [18].

Again, Plan 9 is a distributed operating system whereas Popcorn is a multiple-kernel operating system. It does however allow for heterogeneity.

3.1.7 Kerrighed

Kerrighed is an operating system for clusters [19]. It provides a single system image to the end user [19]. Kerrighed is built from Linux with some kernel modules added [19]. This has the advantage of existing programs being able to be recompiled to work on a cluster and do not require any further modification [19]. Kerrighed allows for memory sharing and message passing between nodes on a cluster [19].

Popcorn and Kerrighed are similar in terms of providing a single-system image on top of Linux; however, their methods of maintaining consistency differ greatly [5].

3.2 Consensus Algorithms

Within a distributed network of computers, it is essential to maintain consistency between nodes. Faults can occur in a system – such as the loss of a packet on a network, a software, or a hardware failure which can result in some shared data structures becoming inconsistent between nodes on the network of computers.

Eventual consistency is when data may be old, and incorrect, but eventually converges to the correct value. Strong consistency is when data is always up to date; however, this is usually at the cost of latency.

Consensus algorithms must ensure that a single value is chosen. This value must be one that has already been proposed and the selection must be atomic [20]. Atomicity refers to an action being isolated from other actions. In this instance, it refers to a value either being committed or not.

A consensus algorithm can use a single designated node to resolve differences. However, this provides a single point of failure. Another method is to have multiple nodes choose the value, with a majority deciding the outcome. The problem with this is deciding which nodes are needed to make the decision and ensuring scalability [20]. Not all messages may appear at the same time, and this may result in no single value receiving a majority.

In large scale networks, device or component failures are to be expected [21]. Designing a system to withstand failures is crucial for any scalable system [21].

3.2.1 Byzantine and Crash Failures

The Byzantine Generals' Problem refers to a hypothetical case where several generals need to coordinate an attack. Their attack will only be successful if they all attack at the same time [22]. For the case of only two generals this is commonly referred to as the Two Generals Problem. The difficulty is that the generals can only communicate using an unreliable messaging system. This is relevant to consensus as all nodes in the network are communicating via packets that may be lost. Errors can also occur in the hardware or software of the device causing inconsistencies. A crash failure is when a process within a system stops unexpectedly and does not restart [23].

An effective consensus protocol must be able to cope with Byzantine and crash failures.

3.2.2 Paxos Algorithm

Paxos is a consensus algorithm for fault tolerance in a distributed system [20]. A leader is chosen and used to determine the correct state when there is a conflict between nodes. The algorithm is performed in two stages – a promise and a commit. The promise is where a request is sent from one node to all the others to state that it will be the leader. Each of the other nodes reply with an acknowledgement. In the commit stage, the node that is the leader asks to commit a value to all of the nodes. If all nodes agree, a lock is given to the node. Paxos is widely used including within Google's Chubby protocol [24].

Paxos effectively provides a mechanism to maintain consistency. However, within the Popcorn joining protocol there is a hierarchy of nodes where the lower the node ID the more likely the node is to be correct, which Paxos does not consider when resolving conflicts. Paxos requires for a subset of nodes to be involved in resolving any conflicting values. All these nodes must communicate with each other meaning that in large systems a smaller number of nodes is chosen. However, you must then maintain consistency between these select nodes and the rest of the network. Paxos also cannot detect failures after joining has completed, including when a particular transport protocol fails. This is because it only prevents errors occurring rather than being able to fix them retrospectively.

3.2.3 Phase King Algorithm

This algorithm operates in a series of phases where each phase has two rounds. In each phase, one of the nodes is designated to be a "king". In the first round of each phase the nodes broadcast their values to all other nodes. In the second round, after having received these values, each node counts the occurrences of values to see if one gives a majority. The king of the phase broadcasts its value to act as a tiebreaker for the nodes where the number of occurrences for all the values is less than $n/2 + f$, where n is the number of nodes and f is the number of failures allowed. Each of the nodes uses

the value previously selected (or the king's value if none reached the threshold) as their new value for the next phase. Since the king rotates for each phase and there are $f + 1$ phases then you can allow for a given number of failures with at least one honest node processing it. After the final phase all honest nodes should have the same value [25].

This algorithm is useful where some nodes are liable to fail or are untrustworthy. This is because it provides guarantees on the number of nodes allowed to fail but still provide the same final value, maintaining consistency. This processing would need to be done before any value can be committed, meaning that it prevents errors rather than corrects them. Like Paxos, it gives consensus between a subset of nodes where all nodes in this subset communicate with each other. This prevents the wrong value being committed, whereas we wish to maintain consistency between all nodes. Again, like Paxos, it would not be able to determine whether an individual node or transport protocol has failed. It does not consider the structure of Popcorn where nodes closer to the instigator are more likely to be correct.

3.2.4 Lockstep Protocol

The Lockstep protocol is where each node records its actions within a given time period, known as a "bucket". They then generate a hash of the actions for that bucket and broadcast this to all other nodes. After they have received hashes from the other nodes, they reveal the plaintext actions within the bucket. If any of the hashes do not match, then the majority determines the correct game state. The Lockstep protocol is often used within real-time, peer-to-peer games to prevent cheating [26].

The bucket size can be adjusted to reduce the messages sent. This algorithm can detect and resolve errors retrospectively rather than preventing errors like in Paxos. All nodes must send messages to all other nodes for each bucket, meaning there is a large overhead. As a result, when implemented in games each player will only participate in the protocol for other players that are nearby. This is known as the "zone of control" [26]. The protocol is designed for untrustworthy nodes but can used in the same way to detect errors and repair inconsistencies. Since Popcorn is between trusted nodes, this algorithm could be performed without the use of cryptographic hashes.

3.2.5 Proof of Work

Bitcoin and similar blockchain protocols use proof of work to maintain consistency over the network. Blockchains are distributed networks where nodes do not trust one another. They maintain consistency by making the connected nodes perform a difficult task, usually, to determine an input that when joined with the value to be stored, creates a hash that begins with a specific number of zeros. The network is protected by the fact that a bad actor would need to have more than 50% of the processing power of the network in order to be able to write incorrect data [27].

However, the energy consumption of this methodology is significant. It has been found that it is not possible to reduce the difficulty of the proof of work problem without degrading security [27].

3.2.6 SCOPE

Structured Consistency Maintenance in Structured Peer-to-peer systems, or the SCOPE protocol is already deployed in several different Peer-to-Peer systems (P2P) [28]. SCOPE was designed to maintain the consistency of a mutable data structure across a P2P network [28]. SCOPE has three operations to maintain consistency on data structures: subscribe, unsubscribe, and update. Nodes use these operations to register an interest in a particular data object (meaning that they will be notified of any changes to them), remove that registration of interest, and notify the network of a change to a data object, respectively [28]. Each node within the network sends messages according to a particular route through the network organised as a tree. When a node is detected to have failed, the network assigns a new node to the position of the lost node. This node then queries the child nodes to detect and repair any inconsistencies by comparing their shared data structures.

In SCOPE, the network is designed as a series of trees where each contains the nodes that store the replica of a particular data object. When an update to the data structure occurs, the message is propagated through the tree structure, where each node updates, forwards the message, or if it is a leaf node, stops forwarding [28]. This is similar to the method employed by Popcorn from the previous year's work. SCOPE uses multiple replicas which are not needed within Popcorn. This is because the node list structure that we need to maintain consistency of is replicated across all nodes. This algorithm is well suited to the existing design of Popcorn; however, it contains some aspects designed for P2P that are redundant for Popcorn, such as multiple tree structures and replicas. SCOPE only repairs mistakes in the network when it detects a node is lost. This does not consider when particular transport protocols fail and therefore some mistakes may go undetected.

3.2.7 Summary

A variety of existing algorithms have been developed and are already in use within the field of consensus. However, all but SCOPE do not consider the hierarchical structure of the Popcorn joining protocol. Paxos and Phase King only prevent errors occurring and are not able to resolve errors when they occur later. Another case would be that a particular transport protocol fails, meaning none of the algorithms that prevent errors would be able to resolve this. Inspiration for an algorithm can be taken from SCOPE to use the passing of messages down the hierarchical structure without the unneeded aspects designed for P2P networks, such as additional replicas. The Lockstep protocol provides an interesting approach whereby you can verify if nodes have the same value by comparing between nodes. This could be modified such that each node on the

network compares their node lists. This will eventually converge and does make use of the hierarchy of nodes within the network.

There are several viable solutions for Popcorn. We will first define and justify which algorithms are most likely to be effective and then run experiments to determine which would be most appropriate for Popcorn.

Chapter 4

Consistency Algorithms

The algorithms chosen were based on those researched but modified to be better suited to the joining protocol and the Popcorn operating system. There are several key considerations in Popcorn specific to the consistency algorithms. The first of which is that gaps can occur in the node list. Any algorithm must be able to cope with missing nodes within the list. The second is that multiple protocols can be used. Finally, the algorithm must be robust enough to cope with protocols that fail. Should an error occur resulting in a link between nodes failing then it must not remove the node when it may still be accessible by others.

4.1 Acknowledgement Algorithm

This algorithm was inspired in part from the hierarchical messaging of the SCOPE protocol. It was designed to be easily integrated into Popcorn. It consists of each node sending an acknowledgement of the message once it, and all its children, have performed the action. This follows the existing binary tree structure within the Popcorn joining protocol. This means that each command to add or remove a node is propagated through the network. Each node performs the action as it is received. Once the final nodes in the binary tree structure, the leaf nodes, have performed the command they send an acknowledgement to their parent. This propagates backwards through the network such that when a parent has an acknowledgement from both its children it then sends its own acknowledgement. The command has been successful if the instigator node receives acknowledgements from all its children.

If no acknowledgement is received after a timeout period, then that node will retransmit the message. The timeout period is calculated based on the number of nodes that the message must be forwarded to, that is the number of levels within the tree structure of the network. This repeats until either an acknowledgement is received, or a maximum number of attempts is received. If the maximum number of attempts is reached, then the node that has just been added is removed from the network. The

connection to the node that did not send the acknowledgement is checked and is also removed if it is not responding. This ensures the consistency throughout the entire network after the acknowledgement of the commands are received. This algorithm would require O(log n) time to complete as each message is forwarded to two other nodes in the network. This algorithm provides strong guarantees on the consistency of the network; however, it cannot detect mistakes after they have been made. When errors occur, it is easy to locate exactly where they occurred, as that will be the node waiting for an acknowledgement. When the final acknowledgement is received by the instigator node it is guaranteed that the operation has completed. This is useful feedback for a user of the system.

Since each addition needs to be entirely completed before the next node can be added this will mean long wait times, particularly during the initialisation of the Popcorn network.

4.2 Check Random Algorithm

The check random algorithm works by the instigator generating a random offset value and forwarding this in a message to all other nodes. This message is forwarded in the same manner as a message to add or remove a node from the network. Each node, once it receives this message, calculates the node it should check, and it forwards its own node list to the node for comparison. If there is a gap in the node list such that the node ID that it was requested to check is not present, then it finds the next node that is present. This loops back round to zero if it goes over the length of the node list. The algorithm took inspiration from the method of SCOPE probing nodes once a node had left the P2P network. However, it was considered that introducing randomness may be able to detect errors faster.

When a node receives another's node list, it checks for inconsistencies with its own. When there are differences, they are resolved by first checking if the node is still active and then choosing the node list with the lowest node ID. This is because the lower the node ID, the closer it is to the instigator node. Using the joining protocol where messages are forwarded from the instigator means nodes closer to the instigator will have the most up-to-date node list.

By using an offset from a node ID, it means that nodes will be checked reasonably evenly and avoids many nodes being left unchecked while others are being checked multiple times.

Since the offset value is random, nodes will typically check different nodes with each pass. As they are corrected, errors will generally reduce with each pass. This can be proven, since messages are passed through a tree structure with the instigator at the root. Each message can fail to be passed on each edge. This means the nodes closer to

the root are more likely to be correct. By randomly checking and deciding that the lower node ID wins, then node lists closer to the root will replace the value of the higher node ID node. As a result, the message will gradually pass through the network until all nodes are consistent.

This algorithm requires a central coordinator, the instigator node, to generate a random offset. It then requires the entire node list of each node to be passed with each check. This node list may be large in size. The random offset must change with each run of error correction.

An issue with this algorithm is that previous runs of error correction may be undone if another node, with the incorrect value, checks the newly corrected node. E.g., if node 0 corrects node 4 (which has a mistake), the following round of error correction node 1 (which has a mistake) puts the error back on node 4. A simulation will be needed to determine is this is a significant problem.

4.3 Check Neighbours Algorithm

Similarly, to the check random algorithm, check neighbours operates by each node sending its node list to its neighbours. E.g., for node 4 its neighbours would be node 3 and node 5 (if they are present on the node list). If there is a gap in the node list, then the next available node is the neighbour. The node list loops back on itself so the first and last nodes are neighbours. This algorithm resolves conflicts, once found, in the same way as check random where the lowest node ID takes precedence.

This algorithm ensures that every node in the list is checked twice by different nodes. The fact that each node checks its neighbours when the node list is structured as a binary tree means that every node will always be checking a sibling/child node pair, or a sibling/parent pair. This means that you always check a node in a different branch and a different level of the tree structure, where differences are more likely to occur.

As with the previous algorithm, errors may be pushed through the network; however, with sufficient rounds of error correction it will converge to the correct values.

It differs from the previous algorithm by not needing a single node (generally the instigator node) to initialise a check. It does not require an offset value to coordinate as all nodes know exactly which nodes to check. This means that each node would be able to decide how often to run error correction independently of the others and does not require central coordination. It also requires less waiting than the check random; this is because nodes which are neighbours are close within the tree structure (the same level ±1), and so should receive messages at approximately the same time. Whereas for check random it needs to wait until all nodes have finished as they can be checked in any order.

4.4 Summary

Three algorithms were developed from the research done. The first, the acknowledgement algorithm, like Paxos or Phase King, can prevent errors occurring. It can provide feedback to the user when it has completed the command in the joining protocol. Check random and check neighbours are designed to compare different nodes' node lists in order to detect errors after they occurred. Check random requires a central coordinator.

Chapter 5

Simulation Implementation

5.1 Implementation

To determine which protocol would be most appropriate to implement, a simulation was created. The simulation was written in Python. It consisted of a data structure containing the Popcorn network. This network contained a list of nodes, known as the master list. This is the actual state of the network. Each node was represented by a `PopcornNode` object. This object contains a node list containing that node's view of the network. When created, each node object is assigned its own unique identifier; this was the time it was created. The unique identifier is used to distinguish two nodes that during the lifetime of the network had the same node ID.

The `PopcornNetwork` class had methods to check the number of conflicts in the network. It did this by moving through each node and checking its node list against the network's master node list. Where there were any differences, this was recorded as an inconsistency in the node list.

5.1.1 Drop Rate

The simulation was able to set a drop rate for the network. This is the proportion of messages dropped by the network. Within the simulation this is designed to represent the messages dropped, corrupted, or hardware or software failures which lead to messages not being processed. This also results in some links between nodes being broken, which represents a particular protocol of a node failing while still being able to communicate with other nodes. One condition is that when adding a node, the first message, to the instigator node, is never lost. This is a fair assumption, as if a node was not able to make a connection with the first node, then it has not managed to successfully connect.

5.1.2 Trials

The program randomly chose to add or remove a node with equal probability, apart from when there is only one node left in which case adding is guaranteed. The node would be added to the first gap in the network as per the protocol developed last year. If a node is to be removed, then one is randomly selected from ones connected to the network (i.e., on the master node list). The name of the checking algorithm, and the trial number is used to generate the random seed; this ensures easy replication of the data. Each trial ended once a node list length of 100 nodes had been reached.

Although Popcorn has asynchronous events, the simulation was designed so that everything occurs in a fixed order. This drastically reduced the complexity of the program. Instead, the simulation processed the results in a depth-first manner recording an artificial timestamp of each event. Each node only changes their own node list for one node ID per add or remove command. Following the joining protocol, if a node is not present in the node list then a node will forward to its children until the end of the list is reached.

5.1.3 Measurements

The number of nodes that were inconsistent was recorded along with the number of attempts or rounds of a particular algorithm that was needed for the operation to complete, and the number of flooded nodes – that is when many messages reach a node at the same time which may mean it becomes overwhelmed. The length of the node list was also recorded. This data is outputted to a CSV file which was then processed further. Several trials were used for each algorithm and drop rate; this was to ensure that the random structure of the network did not bias the results. All the results of the trials are combined and averaged.

The time that messages are sent is calculated based on when the message started and how many nodes it must have travelled through. This is trivial to do when the network is a tree structure. Using the timing results, we can detect if a particular node receives a large number of messages in a short period of time.

A single datatype was used for the algorithms where subclasses implemented the functions for the error detection and correction. A common data structure was created to represent a consistency checking algorithm. This allowed a common interface between the different algorithms. The methods of note are `check_up` and `error` and these detect and fix errors respectively.

Three algorithms were implemented for this simulation, and they are compared in the following chapter.

Chapter 6

Algorithm Comparison

The simulation was run on each of the algorithms with five trials for each algorithm and each drop rate. The drop rate refers to the rate at which messages are lost within the simulation. This represents the failures of software and hardware, including when a node's connection to a set of nodes fails, due to a transport protocol failing, and so can only communicate with some nodes on the node list. The drop rates were set to values of 0.4, 0.2, 0.1, 0.05, and 0.0. This is the probability of an error occurring. These values were chosen to show how the algorithm degrades as the number of failures increases. Each trial ends when 100 nodes are reached. The use of multiple trials ensures that the results are statistically significant. A different seed is set for each trial. Otherwise, the structure of the node list would not be different for each trial, meaning that large gaps in the network in a trial could skew results.

The algorithms are compared based on the message size, message frequency, attempts or the number of rounds that the algorithm required, flooding of nodes, authentication, and time taken. They are compared both using the structure of the algorithm and the results from the simulation.

6.1 Message Size and Frequency

It is important to consider the size of the messages being sent. Large and frequent messages will cause large overheads to the network, which degrade the performance of Popcorn.

6.1.1 Acknowledgement Algorithm

The acknowledgement algorithm has a comparatively small message size, needing only the node's address, node ID, and an integer to represent the command (add or remove). The messages are transmitted down the hierarchy of nodes in the same manner as the joining protocol. If a message is lost, it will cause nodes waiting for an

acknowledgement to timeout and all retransmit. When this occurs in leaf nodes or those close to the leaves then the number of messages retransmitted will be large. Since half of all nodes in a binary tree are leaf nodes then likeliness of this is high. This can be mitigated by the algorithm sending an acknowledgement of the message and then another acknowledgement once the child nodes have replied. When no messages are lost every acknowledgment is only sent twice per node – one to acknowledge the message and another when the child nodes have completed the action.

In summary, with mitigations, this algorithm can ensure consistency of the node list with few messages, and each of these messages is small in size.

6.1.2 Check Random

Check random contains `n` unique identifiers for a node list of `n` nodes, along with an integer offset value. The offset value is the randomly generated value that each node adds to its own node ID to determine which node to check. The message size scales linearly; a subset of the node list could be compared to reduce the message size. However, this would be at the cost of reducing the probability of the error being detected. Instead, a checksum could be used to compare a single value before triggering a full check. This algorithm would first require the offset value to be transmitted through the network in the same manner as the joining protocol. Then every node must check every other node. This results in two messages per node. If multiple rounds of this algorithm are needed, then the frequency must be multiplied by this value.

6.1.3 Check Neighbours

This algorithm performs in the same way as the check random algorithm except it does not require the central coordination for the random offset value. This means that the message size scales according to the number of nodes in the node list (with the same possibility of the checksum optimisation as with check random). The frequency of messages is also one per node per round, where multiple rounds may be needed to converge to the correct value.

6.1.4 Summary

The acknowledgement algorithm has a smaller, and constant, message size with respect to the number of nodes in the node list. It also has a lower frequency of messages sent $O(n)$ where n is the number of nodes. With some minor optimisations the check random and check neighbour algorithms can achieve a $O(1)$ message size when there are no errors. When there are errors the message size is large at $O(n)$, the frequency is also $O(n)$ per round of conflict resolution.

6.2 Attempts or Rounds Taken

The number of attempts or rounds taken to perform any algorithm must be measured as it has an impact on the frequency of messages and the overall viability of the algorithm. Here we make a distinction between an attempt and a round. An attempt refers to any command that terminated in a message that was dropped within the acknowledgement algorithm. The experiment was initially run to find the number of attempts in the check neighbours and check random algorithms. However, since these two algorithms are not guaranteed to detect all errors, any previous errors were carried onto the following attempt for the simulation. For this reason, all errors were corrected before moving onto the next command. The number of rounds refers to the number of times that the algorithm needed to be run before all errors were resolved.

6.2.1 Acknowledgement Algorithm

For the acknowledgement algorithm the number of attempts is tracked. Figure 2 and Figure 3 show the average and maximum number of attempts in the acknowledgement algorithm respectively. These graphs show how the algorithm degrades as the node list increases in length and the number of failures (drop rate) increases. If the algorithm was not able to cope with the number of errors, we would expect the number of attempts to increase at least linearly. Even for high drop rates both the average and maximum number of attempts levels off. This indicates that the algorithm scales well even under significant failures and a large node list. The maximum number of attempts was higher than expected; however, this would not be to the level that the algorithm would not be feasible.

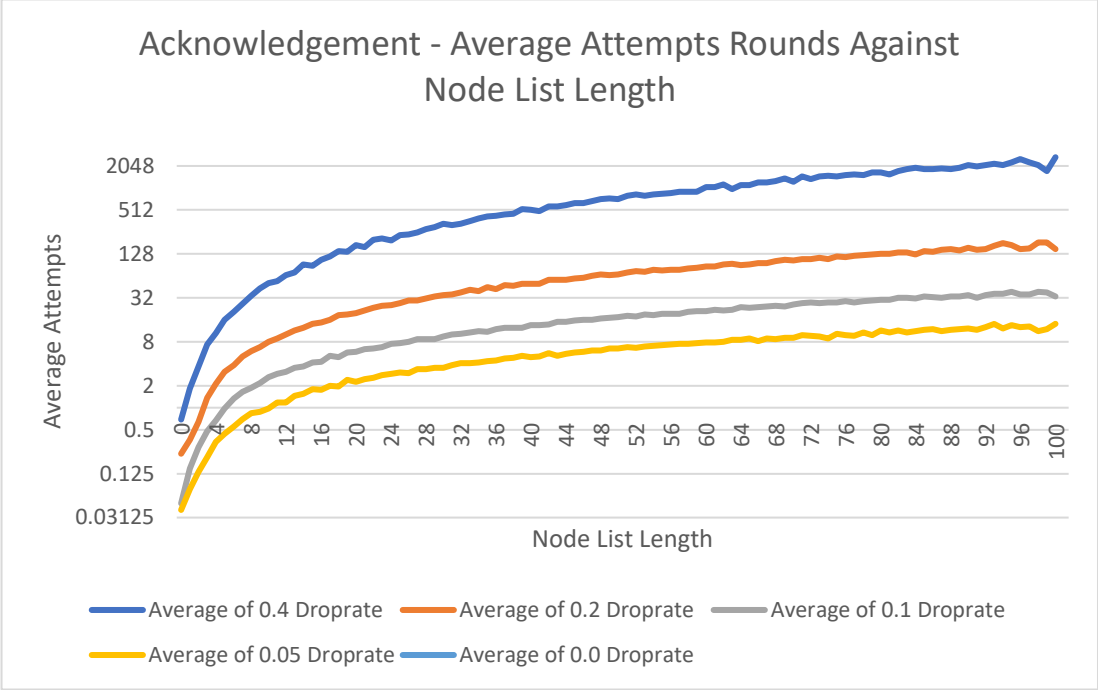

Figure 2: Average number of attempts against node list length for the acknowledgement algorithm. Plotted on a logarithmic scale as the drop rate of 0.4 is significantly higher than the other values.

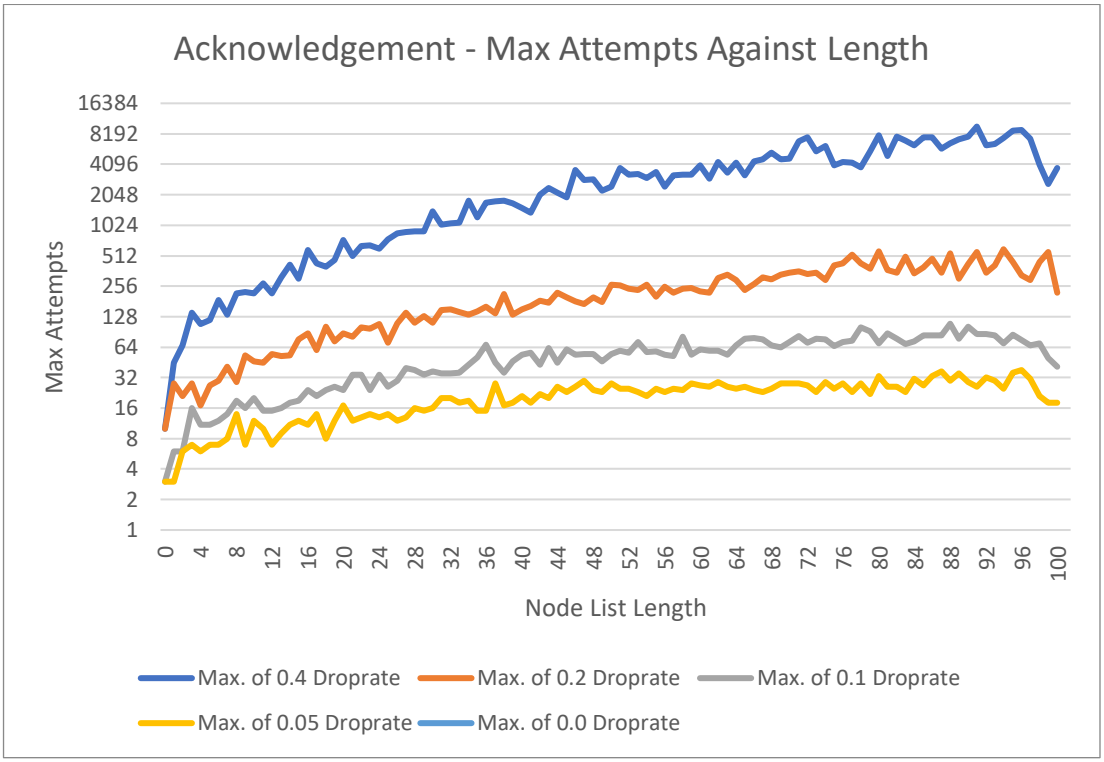

Figure 3: Maximum attempts against node list length for acknowledgement algorithm. Plotted on a logarithmic scale.

6.2.2 Check Random and Check Neighbours

These two algorithms are evaluated together in this section as they detect and correct errors in a similar way. Their graphs are shown together with the same axis boundaries for ease of comparison.

Figure 4 and Figure 5 show the average number of rounds for the check random and check neighbours algorithms respectively. The average shows that check random marginally outperforms check neighbours with fewer rounds required. The number of rounds also increases marginally faster for check neighbours, indicating that it does not scale as well. The average values appear to show a linear trend where the number of rounds required increases with the length of the node list. This was expected, as when errors occur closer to the instigator node (and therefore most nodes have not received the command) it can take multiple rounds of the algorithm for the correct value to move through the network (e.g., if 3 adjacent nodes have a mistake then the node in middle will not have received the correct value after the first round).

The simulation terminated the trial once a length of 100 nodes was reached. For this reason, the node lists of length close to 100 are less reliable as they have fewer datapoints. This also explains the dip in the number of rounds towards the end of all the graphs. To gain a better understanding of the impact that this would have on a real implementation we compare these algorithms for the average and maximum values for a node list of length 85 nodes with a 0.05 probability drop rate. The results are shown in Table 1. This is only to give an indication of how many rounds may be required. Although the table shows better performance, it is clear from the graphs on the following pages that check random generally outperforms check neighbours. It is also worth noting that for a 0.05 drop rate check random never exceeds 5 rounds for check random or 6 rounds for check neighbours at any point during the simulation. This shows that in a low failure network few rounds are ever required to fix the network.

ALGORITHM	AVERAGE ROUNDS	MAXIMUM ROUNDS
Check Random	1.11	3
Check Neighbours	1.08	2

Table 1: Average and maximum number of rounds required to repair a network of 85 nodes with a drop rate of 5%.

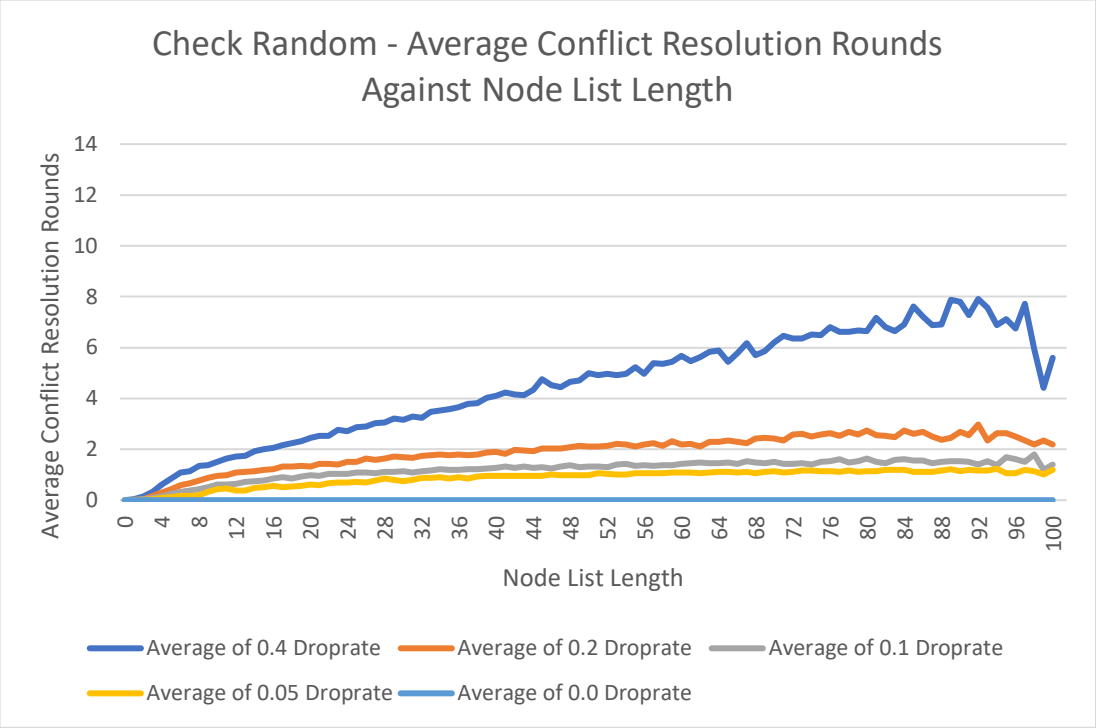

Figure 4: Average number of rounds required to resolve all conflicts using the check random algorithm.

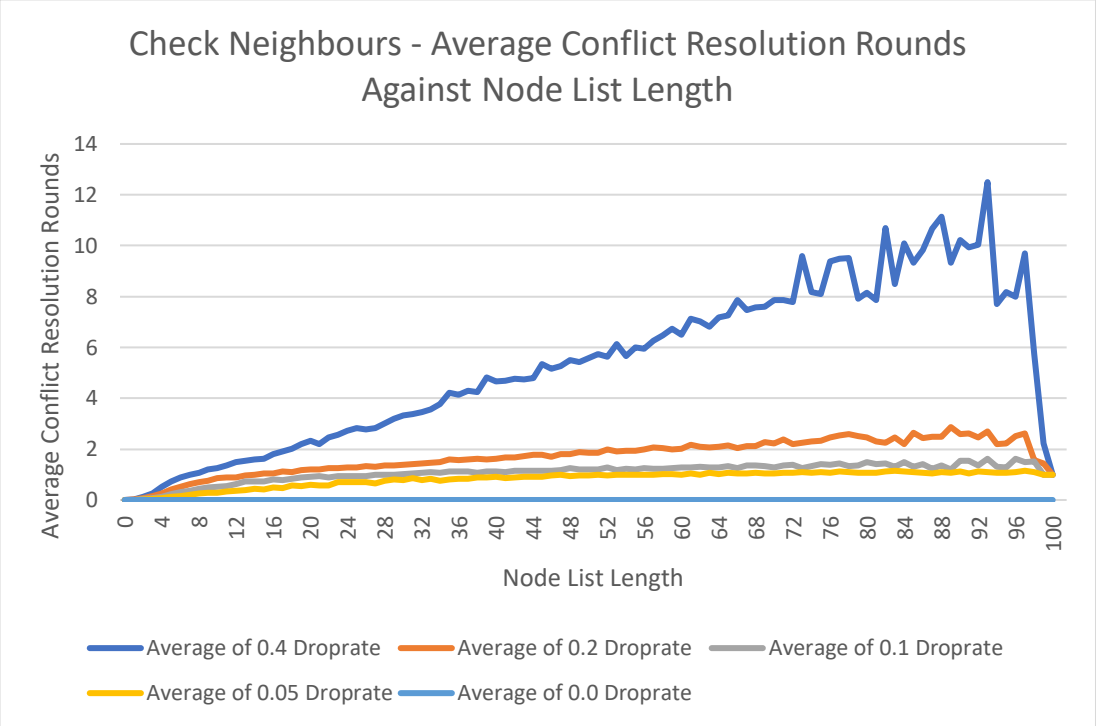

Figure 5: Average number of rounds required to resolve all conflicts using the check neighbours algorithm.

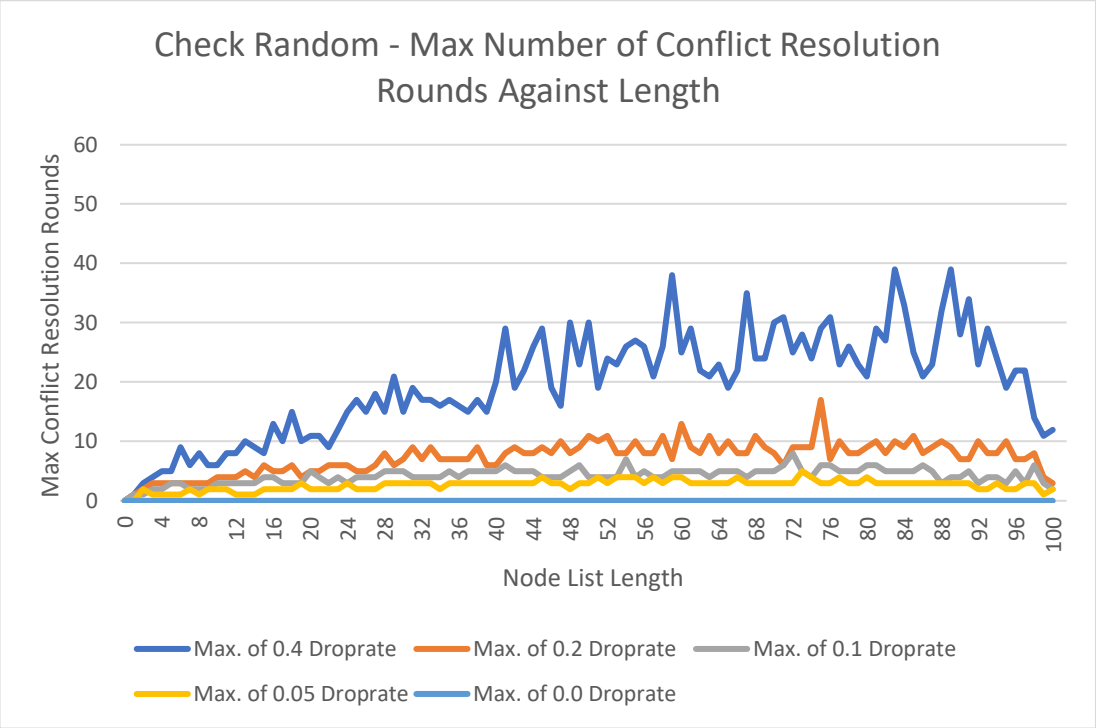

Figure 6: Maximum number of rounds required to resolve all conflicts using the check random algorithm.

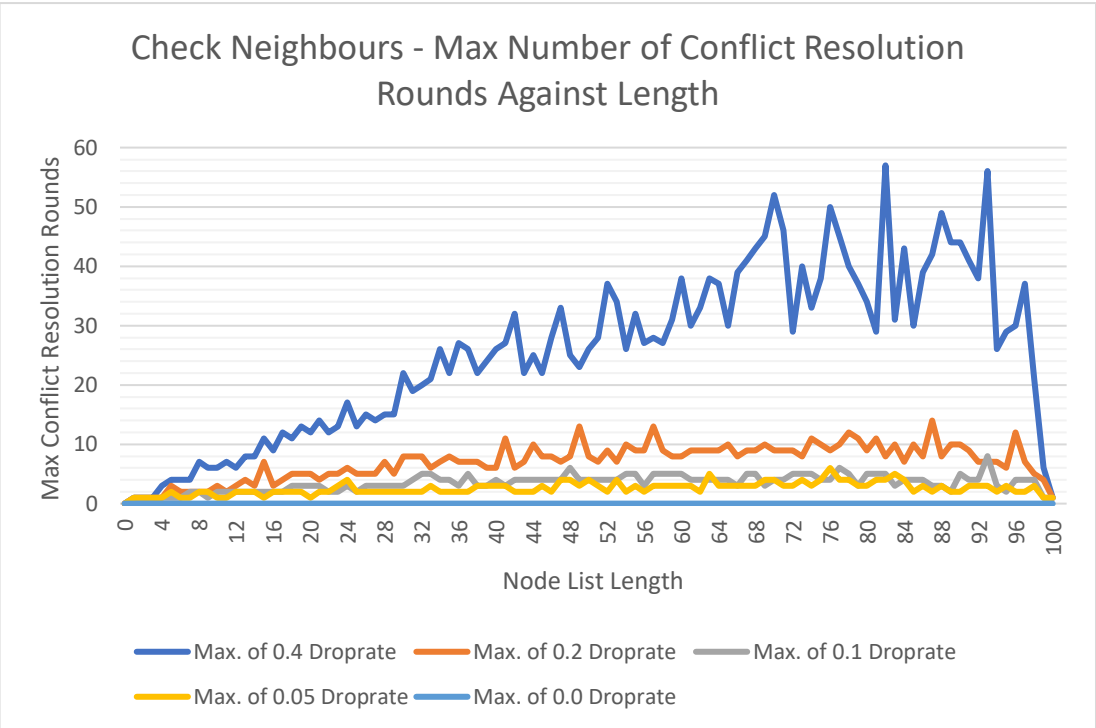

Figure 7: Number of rounds required to resolve all conflicts using the check neighbours algorithm.

This shows that in an implementation of these algorithms, a small number of corrections would be needed even in a large node list. The maximum number of rounds taken could be used in the implementation to maximise the probability of resolving all errors.

The maximums, shown in Figure 6 and Figure 7, show that the maximum number of rounds needed generally stays below 10 for all but 0.4 drop rate. This is also confirmed by the values in Table 1. The graphs do show that extremely high drop rates (which can indicate how well the algorithm should perform on very large networks) can result in a high number of rounds. However, the check random algorithm will easily be able to scale the number of rounds as each node coordinates its own checks.

6.2.3 Summary of Attempts and Rounds

In summary, the number of attempts for the acknowledgement algorithm grows quickly for large node lists. Check random marginally outperforms check neighbours in terms of the number of rounds needed for a consistent node list. The maximum number of rounds in each of these algorithms does not significantly increase as the node list length increases. Check random may perform poorly with high drop rate or very large networks, due to requiring coordination by a single node. Check neighbours however, does not have this problem.

6.3 Flooding Nodes

In this paper flooding of nodes refers to when a node receives a sufficiently large number of messages that it becomes overwhelmed. This is important to determine in order to assess the scalability of different algorithms.

6.3.1 Acknowledgement Algorithm

This algorithm did not cause flooding of nodes with the exception of some isolated cases. It is likely that these are isolated cases where the node had to retransmit the acknowledgement multiple times. Flooding within the simulation is registered when the number of messages within a time period is greater than some arbitrary value. It only provides an indication, but these results show that it is unlikely that this algorithm would result in nodes within the network becoming overwhelmed.

6.3.2 Check Random

With the exception of when the number of rounds of conflict resolution was very high check random rarely caused any nodes to become overwhelmed. As the algorithm utilises the random offset value every node is unlikely to check a node which has already been checked. This lowers the chance of any single node being flooded with

messages. However, when the number of rounds is very high, nodes find themselves being repeatedly checked. This explains the results in the simulation but is unlikely to be problematic if implemented. This is because few rounds are ever required, and the number of messages did not reach a level that a node would not be able to cope with.

6.3.3 Check Neighbours

Again, check neighbours did not cause significant flooding in the simulation. This is because each node only checks two other nodes. Check neighbours produced similar results to check random. For this reason, the algorithm also shows that it would be able to scale without overburdening individual nodes.

6.3.4 Summary of Flooding Nodes

All three of these algorithms performed well in this area and due to their designs are unlikely to cause individual nodes becoming overwhelmed.

6.4 Time Taken

It is important to consider the time taken either to add a node to the network or detect and resolve errors in the node list, as this will have an impact on the usability of the system.

6.4.1 Acknowledgement Algorithm

The acknowledgement algorithm works by preventing errors occurring while adding a node. For this reason, only one node can be added at a time; any other node being added must wait until the action is completed. Each message takes $O(\log n)$ time to send a message. This is because both the action and the acknowledgement travels along a binary tree structure. A benefit in the design of this algorithm is that when the acknowledgement reaches the instigator node it is guaranteed that all nodes in the network have performed the command. This is very useful to the user as they can get confirmation of when the command has been successful.

6.4.2 Check Random

Check random must first propagate a message with the offset value before running the check. Check random has a central coordinator, the instigator node; this means that the time taken for a single check to be performed is the time taken to propagate a message across the binary tree structure which is $O(\log n)$. Each check is then done independently by each node in constant time. As seen when comparing the number of rounds, this algorithm does not require many rounds to complete. However, the checks can only begin after the random offset value has been propagated through the network, resulting in a longer delay.

6.4.3 Check Neighbours

Check neighbours does not require any central coordinator and each node can decide independently when it wishes to run a check. This means that the time taken is simply the number of rounds needed, multiplied by the time taken to perform one check. We found in the previous section that the number of rounds is low, even for large node lists. The time taken to perform one check is small compared to the other algorithms as it does not require traversing the binary tree structure of the node list. This means that the check neighbours algorithm is considerably faster than the other algorithms.

6.4.4 Summary of Time Taken

Check neighbours clearly is the fastest at resolving inconsistencies within the node list. However, the acknowledgement algorithm has the benefit of being able to inform the user when the command has been completed – and therefore no errors in the node list.

6.5 Authentication

Although the authentication of the messages has been left to future work it is important to consider how these algorithms can be adapted in future to facilitate authenticated commands. The acknowledgement algorithm could allow for messages to be signed. This allows each node to easily verify the legitimacy of the command it receives. It also means that every node can verify that every child node has performed each command.

For the check random and check neighbours algorithm, this is more difficult. When adding or removing a node a command can be signed. For the check random and check neighbour algorithm they can store the signature given in the command and relay this when checking other nodes. Only valid signatures would be considered and therefore it is possible to cryptographically verify all commands with these algorithms. The downfall with this is that a signature would need to be kept for the lifetime of a node. This should not require a significant amount of memory and would therefore be an acceptable solution.

6.6 Summary

In terms of message size and frequency the acknowledgment algorithm clearly outperforms the other two algorithms. However, the check neighbours and check random algorithms allow for significant optimisations. The number of attempts for the acknowledgement algorithm was large but not to the point that this would make the

algorithm infeasible. The check random and check neighbours algorithms performed well, where the number of rounds required to perform the algorithm does not become large even for large node lists. For time taken the check neighbours outperforms the others. This is because each node can independently perform the action without the need for any central coordinator, such as in the check random algorithm.

Unlike the other two algorithms, the acknowledgement algorithm aims to prevent errors occurring rather than retrospectively correcting errors. It was noted during the implementation of the acknowledgement algorithm that if an error did occur the algorithm had no way of correcting it and the error grew exponentially with each subsequent change to the node list. An operating system should be robust and able to handle any error. A hardware or software failure or a bug in a related system could cause an error in the node list. A system using the check random, or check neighbours would be more robust to this kind of error.

None of the algorithms had significant problems with particular nodes within the network being strained.

For the time taken the check neighbours outperforms the other algorithms. The acknowledgement algorithm however, is the only algorithm that has the benefit of the user being able to have confirmation of when the command is completed.

Check neighbours was chosen as the best algorithm, due to its ability to be able to perform checks where each node can act independently. This results in far faster checking and correction times. Although on average it takes marginally more rounds to perform a correction, since a central coordinator is not needed the check is able to be performed faster. The algorithm allows for significant optimisation in terms of message size and each node can individually decide when to run a check.

Chapter 7

Popcorn Implementation

7.1 Implementation

The check neighbours algorithm was implemented into the Popcorn Linux kernel. It was implemented by creating a function that added the index, address, transport protocol, and a random token to a list. The `check_and_repair_popcorn` function then took the linked list of nodes previously generated, packaged them into a Popcorn message and sent them to the neighbouring nodes. When a node receives a message with the details of other nodes it checks each one to see if there are any inconsistencies between its own node list and its neighbours. If its node ID is lower than its neighbour, it runs the `check_and_repair_popcorn` function to send its node list to its neighbour so that it corrects its node list. If the node list is higher, then it corrects its node list. Each message must have a fixed size, for this reason each message can carry 10 nodes (this value can easily be changed). If there are more than 10 nodes, then another message is sent until all are sent. A list was used as it was considered that sending the most recent changes first will detect errors faster. When there are not enough nodes to fill a message then it is padded with dummy values (set to -1). The transport name is sent with the check so that if a connection is made then the node knows which transport protocol to use. This preserves the work done last year allowing for different transport protocols to be used by different nodes.

7.1.1 Token

In the previous year's project, a randomly generated token was used to ensure that a node outside of the network waiting for Popcorn nodes to connect them only connect to nodes within the network. This was done by the instigator randomly generating a token string which it sent to the new node and passed along with the command to all the other nodes in the network. That way the node being added knew only to add the nodes with the correct token and aborted if there were too many failed attempts. This was extended by storing the token within the node list so that it could be recalled with every check of the node list. When a node is establishing a connection due to a check

in the node list it only does so if the token is correct, providing security to the node list. These messages are currently unencrypted so do not provide security in isolation, but encryption has been left as future work. A future addition could be to sign the tokens so that every node can cryptographically verify if a node should be added.

7.1.2 Preliminary Check

In order to reduce message size a checksum of the message was produced. This is known as a preliminary check. Comparing a single value significantly reduced the message size, as each node does not need to send the entire node list each time it performed a check. A cryptographic hash was considered, however they are computationally expensive, and the nodes are trusted so do not need the guarantee of security. So instead, the randomly generated tokens were used. Each token is 16 bytes long and randomly generated meaning that each bit has a 50% chance of being a one. If each of these tokens are XORed together then this will give a 16-byte representation of the entire node list. Since the result will be entirely random then the chance of a collision is 1 in 2^{48} (or 1 in 281 trillion). This means that any differences are almost guaranteed to be detected. The number of bytes used can be reduced in future to improve performance. This method would not be able to detect if the order is incorrect; however, this is unlikely to ever happen as each node is sent its index value from the instigator. This is inexpensive to compute and provides a unique value that can be used to check the node list is correct without needing to transmit the entire node list.

7.1.3 Repeated Checks

In order to periodically check the network, the function to run a preliminary check needed to be repeatedly called. At first a kernel timer was used. Within the Linux kernel a timer can be used to call on a function after a specified amount of time. The timer can continuously rerun itself in order to run a check every set time period. This was implemented, however, timers run in what it known as an atomic context. This means that the kernel cannot sleep or wait. Sending a message through Popcorn involves the use of semaphores where you are required to wait until the semaphore is released. The timer caused the system to crash when trying to send a message as it was trying to wait within an atomic context. This was replaced with a kernel thread which looped infinitely sleeping for a set period of time between each preliminary check. If the prelim check fails, then it runs a full check of the node list. This was successful in being able to periodically run checks.

The thread was placed in the messaging layer along with a lock that can prevent changes to the node list. The kernel thread will only run checks when this lock is not engaged. This means if the user knows changes will not occur in the node list then they can send the command `lock` to the `/proc/popcorn_nodes` file and periodic checks of the node list will stop. This allows the system to conserve resources when

the user knows changes will not occur. The node list can be unlocked for changes by sending `unlock` to the proc file.

Running checks too frequently would result in reduced performance in Popcorn applications whereas running checks too infrequently will risk the node list becoming inconsistent. The system was designed so that with each change to the node list, or a failed preliminary check the time was recorded in a value called `time_of_last_change`. The number of seconds until the next check is calculated as $2^{current\_time - time\_of\_last\_change}$, with a minimum value of 2 seconds and a maximum of 5 minutes. The use of an exponential means that checks will occur frequently when close to when a change last occurred. Thus, increasing the chances of detecting a mistake but increasing the time lowers the overhead caused by the system the longer the time since the change was made. Should a node list receive a check and find that it must update its node list that will trigger it to check its neighbours. As a result, the system is able to quickly detect and resolve differences in node lists with minimal overheads. This work minimises the impact that a check has on the system; however, it could be future work to be able to adjust the frequency of checks based on the activity of a node so that nodes that are idle are able to take the strain of maintaining the node list's consistency.

7.2 Evaluation

As the analysis of the algorithm was done when choosing the algorithm, this section will only discuss the evaluation of the final implementation. In order to check that the system worked we needed to create a system to deliberately create errors within the node list. This was done by creating a command called `add_no_prop` that when sent along with an integer value to the input proc file for the messaging layer triggers the adding of a node without running the joining protocol. This means that the nodes are connected but do forward the details of the new node to any other nodes. Another command called `check_prelim` was created for debugging that would trigger the correction algorithm. However, the system also runs preliminary checks automatically.

Testing was done by connecting a series of virtual machines. The algorithm was able to detect and initiate a connection between the nodes and therefore repair the node list when an error occurred. This shows that the algorithm worked correctly. The system was able to detect the inconsistency in the node list within two seconds and immediately corrected the error. The time taken to send a full check – that is the comparison of the full node list and not just the checksum – was on average 0.11 seconds.

It is critical that this new system does not introduce significant overheads to the Popcorn system. In order to test this, we ran an experiment where we repeatedly migrate a process between two nodes. In order to measure the time taken we did this

several thousand times per trial. The results showed that the new implementation resulted in only a 0.32% increase in time taken to run the experiment. This shows that there was not a significant decrease in performance introduced by this new system.

Chapter 8

Encryption

An initial aim of this project was to add encryption and authentication to the protocol. The aim of this was, in combination with the consistency algorithm and joining protocol, that Popcorn would be safe to use on any network. Currently without encryption, details of any process running on Popcorn would be exposed and at risk of being modified in a network. The encryption was also needed to implement a form of capabilities for Popcorn, which was the original aim of this project. This chapter has been added to describe the work done towards the goal of encryption as it took a significant amount of time from the project.

Research was done on methods to encrypt data within the Linux kernel, and capabilities in various other operating systems. Encryption was then implemented within the Popcorn kernel. The payload of messages sent in Popcorn were encrypted with the initialisation vector (IV) also sent in the message. The payload was encrypted using a symmetric key. The AES algorithm was chosen to encrypt the data as it is currently the standard algorithm within the field of symmetric cryptography. Any message could then be received, decrypted, and then processed as normal by the existing functions. The symmetric keys are stored within the `message_node` structure in the node list. Each node-to-node link in the network has a unique key. This means that there would be `n-1` keys, where `n` is the number of nodes in the network, stored on each node. This is needed as should one node leave the network then it should not be able to view or access any of the existing connections. Requiring the entire network to update a single key would require a great deal of coordination between nodes. This system would cause minimum slow down to the system and meant that most functions would not need to change to facilitate the encrypted messages. The fact that messages were encrypted and decrypted immediately before being sent to the transport protocol being used meant that the system was modular and improved maintainability. This system is also invariant of the transport protocol that was used. This means that it only relies on one robust implementation rather than being implemented multiple times in TCP, RDMA, etc.

This system could not be tested due to this encryption API being for a later kernel version. It was attempted to use Transport Layer Security (TLS) within the TCP specific protocol; however, this again required a later kernel version. It would also mean that other transport protocols such as RDMA would not be encrypted. A final attempt was made to introduce encryption by using an encryption library within the current kernel version. This was designed to allow user-space programs to perform encryption. As a result, the memory allocator for the encryption needed to be called from user-space. This system was partly implemented. However, using a call to user-space to allocate memory for encryption is likely to introduce side-channels and is considered bad practice in the security community. Upgrading the kernel version of Popcorn would have taken considerable work beyond the range of this project. The testing of the encryption system has been left for future work. The implementation for encrypting and decrypting messages has been left in the kernel code but disabled by a macro.

An important contribution of this chapter is that encryption should not be introduced into Popcorn within this kernel version. An attempt to do so is likely to introduce side-channel vulnerabilities; instead, it is recommended that the kernel is first updated to a version that allows for encryption within the kernel.

Chapter 9

Conclusion

This year, the aim of the project was to provide a consistency algorithm to resolve errors on the node list. Every node on the network must have the same view of which nodes are connected to the network. An algorithm was selected due to its scalability and low overhead ability to resolve conflicts between a node's view of the network as it occurred on the node list. The algorithm was implemented in the Popcorn Linux kernel. Optimisations were made in order to reduce the size of messages sent by the algorithm and the system was set to automatically check itself for errors. It did this in a way that found errors quickly but adjusted the time between checks in order to lower overheads to the system. The system was tested by connecting a single node to the network and running a check. This check then detected the differences in the node lists and the nodes with incorrect node lists connected to nodes that they had not connected to.

The main contributions of this project were the analysis of the algorithm, the implementation of the algorithm, and the research into encryption. This paper laid out the reasoning behind the choice of algorithm, how it best fits into the existing Popcorn system and how it relates to the work carried out last year. The research into encryption is also an important contribution as any attempt to introduce encryption should only be done after the kernel is upgraded to a newer version. The research found that attempting to use user-space encryption functions within the kernel is more likely to introduce side-channel vulnerabilities. Encryption was implemented into the kernel and so it is available for future work.

I had no prior experience of kernel programming or operating systems before the start of this two-year project. Implementing the algorithm in Python before implementing within the kernel made for considerably easier development this year compared to last year. Going forward, when implementing kernel or any complex project I think it is beneficial to implement in a higher-level language where debugging is easier. This greatly simplified the process when it came to kernel programming in this year of the project. In terms of improving the work of this project I think it would have been

beneficial to plan out the system at the beginning of the two-year project. Often time was lost due to assumptions made in the previous year so planning long term would have improved this. Many problems during kernel development were due to lack of experience with kernel programming and operating systems. Although this made the research more interesting, it often led to false leads (such as with the encryption implementation) and therefore lost time.

This project is opensourced, available on GitHub and will be shared with the wider Popcorn community. Future work of the project is to allow for other commands to be added to the joining protocol e.g., to be able to seamlessly switch transport protocols without disconnecting. Future work would also include more testing of the system with real hardware.

Bibliography

[1] E. Novikov and I. Zakharov, “Verification of Operating System Monolithic Kernels Without Extensions,” 30 October 2018. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-03427-6_19. [Accessed 6 November 2020].

[2] B. Roch, “Monolithic kernel vs. Microkernel,” 2004. [Online]. Available: http://web.cs.wpi.edu/~cs3013/c12/Papers/Roch_Microkernels.pdf. [Accessed 2 November 2020].

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach and A. Singhania, “The Multikernel: A New OS Architecture for Scalable Multicore Systems,” October 2009. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1629575.1629579?casa_token=I7_hNx4wHdsAAAAA:0SVwWy0PBIxp-ZjoK3g9NLYR0uT1tJUHc29C2HBgPjo_VysRDtqGmfp1-3Swdqh6lng4qYOkTf3vKg. [Accessed 8 November 2020].

[4] M. Sadini, A. Barbalace, B. Ravindran and F. Quaglia, “A Page Coherency Protocol for Popcorn Replicated-kernel Operating System,” 2013. [Online]. Available: http://www.popcornlinux.org/images/publications/marc2013_camera_ready_fixed.pdf. [Accessed 21 October 2020].

[5] A. Barbalace, B. Ravindran and D. Katz, “Popcorn: a replicated-kernel OS based on Linux,” 2014. [Online]. Available: https://www.linuxsecrets.com/kdocs/ols/2014/ols2014-barbalace.pdf. [Accessed 19 April 2021].

[6] M. H. Solomon and R. A. Finkel, “The Roscoe distributed operating system,” December 1979. [Online]. Available: https://dl.acm.org/doi/10.1145/800215.806577. [Accessed 29 March 2022].

[7] S. Peter, A. Schüpbach, D. Menzi and T. Roscoe, “Early experience with the Barrelfish OS and the Single-Chip Cloud Computer,” 2011. [Online]. Available: https://people.inf.ethz.ch/troscoe/pubs/marc11-barrelfish.pdf. [Accessed 8 November 2020].

[8] A. Barbalace, A. Murray, R. Lyerly and B. Ravindran, “Towards Operating System Support for Heterogeneous-ISA Platforms,” April 2014. [Online]. Available: http://www.popcornlinux.org/images/publications/sfma14.pdf. [Accessed 21 October 2020].

[9] R.F.Rashid and H.Tokuda, “Mach: A system software kernel,” 15 June 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0956052190900045. [Accessed 20 October 2020].

[10] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan and D. Bohman, “Microkernel Operating System Architecture and Mach,” 30 April 1992. [Online]. Available: https://courses.cs.washington.edu/courses/cse451/15wi/lectures/extra/Black92.pdf. [Accessed 8 November 2020].

[11] R. Rashid and H. Tokuda, “Mach: A system software kernel,” 15 June 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0956052190900045. [Accessed 20 October 2020].

[12] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, “Mach: A New Kernel Foundation For UNIX Development,” 1986. [Online]. Available: http://cseweb.ucsd.edu/classes/wi11/cse221/papers/accetta86.pdf. [Accessed 8 November 2020].

[13] “Openmach Git Repository,” [Online]. Available: https://github.com/openmach/openmach/blob/master/include/mach/message.h. [Accessed 8 November 2020].

[14] R. Krten, “Getting Started with QNX Neutrino: A Guide for Realtime Programmers,” 2008. [Online]. Available: http://jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl/Neutrino/getting_started.pdf. [Accessed 8 November 2020].

[15] S. Mullender, G. v. Rossum, A. Tananbaum, R. v. Renesse and H. v. Staveren, “Amoeba: a distributed operating system for the 1990s,” May 1990. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/53354. [Accessed 28 May 2021].

[16] A. S. Tanenbaum and R. V. Renesse, “Distributed Operating Systems,” December 1985. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/6041.6074. [Accessed 17 June 2021].

[17] A. S. Tanenbaum, R. v. Renesse, H. v. Staveren, G. J. Sharp and S. J. Mullender, “Experiences with the Amoeba distributed operating system,” 1 December 1990. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/96267.96281. [Accessed 10 June 2021].

[18] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey and P. Winterbottom, “Plan 9 from Bell Labs,” no date. [Online]. Available: http://9p.io/sys/doc/9.html. [Accessed 2 September 2021].

[19] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, G. Utard, R. Badrinath and L. Rilling, “Kerrighed: A Single System Image Cluster Operating System for High Performance Computing,” 1 June 2004. [Online]. Available:

https://link.springer.com/chapter/10.1007/978-3-540-45209-6_175. [Accessed 13 October 2021].

[20] L. Lamport, “Paxos Made Simple,” 1 November 2001. [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/paxos-simple-Copy.pdf. [Accessed 19 March 2022].

[21] S. Fu, “Failure-aware resource management for high-availability computing clusters with distributed virtual machines,” 10 January 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731510000031?casa_token=4okvzjR2ZFoAAAAA:ICTd-3vz0WJf2GFnv42wBADO0UNfslDkbkNxMg0Gs6NTqSlllJKKjl_g_gQO5I-yMRI80ew. [Accessed 7 February 2022].

[22] L. Lamport, R. Shostak and M. Pease, “The Byzantine Generals Problem,” 1982. [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/The-Byzantine-Generals-Problem.pdf. [Accessed 29 March 2022].

[23] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” 1 March 1996. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/226643.226647. [Accessed 29 March 2022].

[24] M. Burrows, “The Chubby lock service for loosely-coupled distributed systems,” 6 November 2006. [Online]. Available: https://static.googleusercontent.com/media/research.google.com/en//archive/chubby-osdi06.pdf. [Accessed 20 March 2022].

[25] P. Berman, J. A. Garay and K. J. Perry, “Towards Optimal Distrubted Consensus,” 1989. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.463.9356&rep=rep1&type=pdf. [Accessed 20 March 2022].

[26] H. Lee, E. Kozlowski, S. Lenker and S. Jamin, “Multiplayer Game Cheating Prevention with Pipelined Lockstep Protocol,” 2003. [Online]. Available: https://link.springer.com/chapter/10.1007/978-0-387-35660-0_4. [Accessed 20 March 2022].

[27] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf and S. Čapkun, “On the Security and Performance of Proof of Work Blockchains,” October 2016. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2976749.2978341. [Accessed 20 March 2022].

[28] X. Chen, S. Ren, H. Wang and X. zhang, “SCOPE: scalable consistency maintenance in structured P2P systems,” 13 March 2005. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1498434. [Accessed 4 February 2022].