This is the Project Title

Andreas Ramsøy

**MInf Project (Part 2) Report**

Master of Informatics

School of Informatics

University of Edinburgh

2022

Abstract

This skeleton demonstrates how to use the infthesis style for undergraduate dissertations in the School of Informatics. It also emphasises the page limit, and that you must not deviate from the required style. The file skeleton.tex generates this document and can be used as a starting point for your thesis. The abstract should summarise your report and fit in the space on the first page

Acknowledgements

Acknowledgements go here.

Table of Contents

[Introduction 1](#_Toc99460801)

[Background 2](#_Toc99460802)

[X.X Operating Systems 2](#_Toc99460803)

[2.X.X Popcorn 3](#_Toc99460804)

[2.X.X Summary of MInf 1 3](#_Toc99460805)

[2.X.X Distributed Operating Systems 4](#_Toc99460806)

[X.X Consensus Algorithms 5](#_Toc99460807)

[X.X.X Byzantine and Crash Failures 5](#_Toc99460808)

[X.X.X Paxos Algorithm 5](#_Toc99460809)

[X.X.X Phase King Algorithm 6](#_Toc99460810)

[X.X.X Lockstep Protocol 6](#_Toc99460811)

[X.X.X Proof of Work 7](#_Toc99460812)

[X.X.X SCOPE 7](#_Toc99460813)

[2.X.X Consistency Algorithms 8](#_Toc99460814)

[X.X Popcorn Joining Protocol 9](#_Toc99460815)

[Related Work 10](#_Toc99460816)

[X.X.X Roscoe 10](#_Toc99460817)

[2.X.X Amoeba 10](#_Toc99460818)

[2.X.X CAP 12](#_Toc99460819)

[2.X.X Plan 9 12](#_Toc99460820)

[2.X.X Kerrighed 13](#_Toc99460821)

[Simulation Implementation 15](#_Toc99460822)

[X.X.X Algorithms 16](#_Toc99460823)

[X.X.X Acknowledgement Algorithm 16](#_Toc99460824)

[X.X.X Random Check-up Algorithm 17](#_Toc99460825)

[X.X.X Check Neighbours Algorithm 18](#_Toc99460826)

[X.X.X Contrast of Algorithms 19](#_Toc99460827)

[X.X Evaluation of Algorithms 19](#_Toc99460828)

[Simulation Results 22](#_Toc99460829)

[X.X Message Size and Frequency 22](#_Toc99460830)

[X.X Flooding Nodes 23](#_Toc99460831)

[X.X Attempts Taken 23](#_Toc99460832)

[X.X.X Acknowledgement Algorithm 24](#_Toc99460833)

[X.X.X Check Random and Check Neighbours 25](#_Toc99460834)

[X.X.X Summary of Attempts 29](#_Toc99460835)

[X.X Time Taken 29](#_Toc99460836)

[X.X.X Acknowledgement Algorithm 29](#_Toc99460837)

[X.X.X Check Random 29](#_Toc99460838)

[X.X.X Check Neighbours 29](#_Toc99460839)

[X.X Scalability 29](#_Toc99460840)

[X.X.X Acknowledgement Algorithm 29](#_Toc99460841)

[X.X.X Check Random 30](#_Toc99460842)

[X.X.X Check Neighbours 30](#_Toc99460843)

[X.X Summary 30](#_Toc99460844)

[Popcorn Implementation 32](#_Toc99460845)

[Evaluation 33](#_Toc99460846)

[Conclusion 34](#_Toc99460847)

[Bibliography 35](#_Toc99460848)

Chapter 1

# Introduction

The aim of this project was to develop a joining a protocol for the Popcorn multiple-kernel operating system. 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.

It was highlighted in last year’s work that the algorithm did not provide guarantees of consistency. Should an error occur the system did not have a mechanism to correct it. This year a mechanism for correcting errors was developed. Existing work was researched, a set of algorithms were selected and analysed in a simulation. The best algorithm was selected and implemented within the Linux kernel.

This is the introduction

The aims of the project. Subsection of previous project summary, better in background so can explain concepts? Short summary of last year’s work

The motivation of the research/why it’s important. Outline of how the paper is structured

Chapter 2

# Background

## X.X 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 of the services 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 compared to larger operating system kernels that their large size and interdependency means 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. This allows 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]. Multi-kernel operating systems 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.X.X Popcorn

Popcorn is a multi-kernel operating system based on Linux. It provides a single system image to the user despite being split across multiple processors or groups of processors [4]. Each node, that is a processor or group of processors, run 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 and not broadcasting to multiple nodes [4]. For this reason, every node has a single connection with every with all other nodes in the network.

### 2.X.X Summary of MInf 1

The first part of this project (MInf 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 reloading 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 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 forwarding messages to just two nodes each in the network, each node it passes it to 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 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 was the aim of the project this year to provide this error correction.

### X.X.X 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 order to propagate messages quickly and distributing the messages so that no single nodes have a significantly higher burden of processing.

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 node in the node list becomes the new instigator node.

Assuming all nodes have a consistant node list then they will forward to the correct nodes and know which node the instigator is.

The key difference with the Popcorn joining protocol is that the hierarchy of command propagation means that nodes earlier in the node list are more likely to be correct than those further on. 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 is only a new process that wishes to migrate to a node that has just been added needs to be connected to this node. This means that for the consistency mechanism we can afford to favour speed of adding a node over the strict guarantee of strict consistency.

Diagram

Description automatically generated

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 for the joining protocol. The red lines in the lower diagram shows the paths that commands will follow through the network.

Chapter X

# Related Work

## X.X Related Operating Systems

### X.X.X Roscoe

Roscoe was a distributed operating system designed to share computing resources in a non-hierarchical manner [3]. 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 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 [3]. The user was not informed of where a process was running and so were presented with want appeared to be a single, powerful, computer. This is known as providing a single-system image [3].

### X.X.X Barrelfish

Each core runs an instance of Barrelfish and uses message passing to maintain consistency between instances [15]. Barrelfish is a multi-kernel operating system. Unlike Popcorn, Barrelfish is not heterogeneous. However, heterogeneous versions have been proposed [16]. Messages within Barrelfish are highly optimised to the hardware architecture in order to increase speed [15].

### X.X.X Mach

Mach is a microkernel that uses messaging passing for inter-process communication [8] [18]. It does this by using finite length queues of messages known as ports [19]. 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 [18].

### X.X.X Neutrino QNX

Quick-UNIX, or QNX, is another microkernel which uses message passing for inter-process communication [20]. The messages are transported using a messaging bus. Each service is modular and communicate through messages [20].

### X.X.X Amoeba

The motivation behind Amoeba was to build a system where all resources are automatically managed by a distributed operating system [10]. Consequently, users do not know which processor their programs run on, or how and where their files are stored in the system [10]. 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 [6].

In Amoeba users log into a terminal computer. This device does all the low latency computation whereas the group of computers known as the “processor pool” does all the larger computations [11]. Amoeba makes use of heterogeneity by using different machines for specialised purposes e.g., devices with large storage disks are used for file storage [10].

### X.X.X Plan 9

Plan 9 allows for heterogeneity; different processor architectures can join the network running Plan 9 [13]. 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 (however, binary can still be used for large transfers of data) [13]. Plan 9 interacts with services as if they are files and uses file operations as such. This means one simple, well understood protocol can be used to access almost all services [13]. In Plan 9 the use of the same secured protocol to represent services rather than the use of firewalls means security is implemented in any Plan 9 service from the start [13]. There is no superuser, each individual server must ensure security (physical access to the server does give special permissions) [13].

### X.X.X Kerrighed

Kerrighed is an operating system for clusters [14]. It provides a single system image to the end user [14]. Kerrighed is built from Linux with some kernel modules added [14]. This has the advantage of existing programs being able to be recompiled to work on a cluster and do not require any further modification [14]. Kerrighed allows for memory sharing and message passing between nodes on a cluster [14].

## X.X Consensus Algorithms

Within a distributed network of computers, it is essential to maintain consistency between nodes. This is needed when there exists some aspect of the system which may result in a fault – such as the loss of a packet on a network.

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 [10].

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 chose the value, with a majority deciding the outcome. This presents the problem of deciding how many of such nodes are needed [10]. 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 and not exceptional [15]. Designing a system to with stand failures is crucial for any scalable system [15].

Within large computing clusters checkpointing can be used to allow processes to restart with minimal impact or loss. However frequent checkpointing leads to large overheads which may reduce performance of the system as a whole. As a result there has been a movement towards adapting the system configuration according to node availability and failures [15].

A hierarchical structure is used to coordinate the nodes [15].

When considering which nodes to group together to perform takes: randomly allocating them to groups can be done however this does not use any information on the reliability and so is not an effective way of doing this [15].

Another method is to allocate based solely on reliability, this must be tuned [15].

Protocols using “rumour spreading” where the updates are only transmitted to a subset of known nodes have been used. This provides partial consistency [15].

Periodic heartbeat messages are used to check the network, this creates a large amount of messages sent across the network [15].

### X.X.X 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 [10]. For two generals this is also known 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 [10].

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

### X.X.X Paxos Algorithm

Paxos is a consensus algorithm for fault tolerance in a distributed system [10]. Paxos requires a leader to ensure liveness. 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.

This way if some nodes are not present on the network if another node tries to acquire the same lock, then at least some of the nodes will not be able to fufil the promise and so will not be able to aquire the lock.

Paxos is widely used including within Google’s Chubby protocol [11].

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.

### X.X.X Phase King Algorithm

This algorithm operates in a series of phases where each phase has two rounds. In each phase one of the nodes are 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. If the number of occurrences of the value is less than , where is the number of nodes and is the number of failures allowed. Each of the nodes use 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 phases then you can allow for a given number of failures with at least one honest node processing it. After the final all honest node should have the same value [11].

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.

### X.X.X 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. [13].

The bucket size can be adjusted to reduce the messages sent. This algorithm allows for error correction rather than error prevention. 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” [13]. The protocol is designed for untrustworthy nodes but can used in the same way to detect errors and use a majority is able repair inconsistencies.

### X.X.X 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 particular 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 [14].

The energy consumption is significant. It has been found that it is not possible to reduce the difficulty of the proof of work problem without degrading security [14]. For this reason the Proof of Work algorithm was not considered further for Popcorn.

### X.X.X SCOPE

Structured Consistancy Maintainance in Structured Peer-to-peer systems, or the SCOPE protocol is already deployed in several different Peer-to-Peer systems (P2P) [15]. SCOPE was designed to maintain the consistency of a mutable data structure across a Peer-to-Peer network [15]. P2P systems must replicate data across several nodes as any node is liable to leave the network. Within P2P networks this can cause several problems: the hotspot, the node-failure, and the privacy problem. The hotspot problem is due to different data objects having different popularities which causes some nodes to become overloaded while others are under utilised. Since the data structure we are trying to replicate across Popcorn is the same for all nodes and present on all nodes hotspots from accessing the data will not be an issue. The node-failure problem concerns itself with recovering from a node dropping out of the network. This is relevant to Popcorn as any node has a chance of failure. For Popcorn it was engineered last year to index each of the nodes such that the index does not change while the node is connected to the network (it would need to leave and re-join the network to be assigned a new ID). The node list data structure is replicated on every node, this means that should a node drop out then only the Popcorn processes running on that node will be affected (resolving this will likely be future work but is not the subject of this project). The rest of the Popcorn network will be unaffected by the lost node. Finally, the privacy problem in P2P networks concerns itself with obscuring the location or identity of the other nodes. This is not directly applicable to Popcorn as each node knows the address of all the others [15].

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), removes that registration of interest, and notify the network of a change to a data object, respectively [15].

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 then it stops forwarding. This is very similar to the method employed by Popcorn from the previous year’s work. This results in O(log2n) messages being sent within each tree [15].

SCOPE – standard in P2P, also discuss the BNoC or whatever it is called that builds on it

Some random checking algorithm (probabilistic approach)

Then add some more that I have come up with

Chapter X

# 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, would mean 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. Encryption is required for the safe deployment of Popcorn.

Research was done on methods to encrypt data within the Linux kernel

Chapter 4

# Simulation 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 of containing the Popcorn network. This network contained a list of nodes, known as a 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. If there was a node present that was not on the master node list then it would be counted as an “excess node”. Nodes that were missing on a node list but present on the master node list were counted as “missing nodes”. Finally, nodes that had the incorrect unique identifier were known as “incorrect nodes”.

The simulation was able to set a drop rate for the network. This is 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. One condition is that 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.

The program randomly chose to add or remove a node with equal probability, except 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 checking algorithm used and the trial number is used for the random seed, this ensures easy replication of the data.

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. Each node only changes their own node list for one node id per add or remove command. As a result it was only necessary to ensure order between the add or remove commands. The Popcorn joining protocol divides the network into a hierarchical tree structure, the simulation navigated this in a depth-first manner.

Following the joining protocol is 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.

The number of nodes that were recorded as excess, missing, or incorrect are recorded along with the time taken for the add or remove operation to complete, the number of flooded nodes – that is when a large number of messages reach a node at the same time which may mean it becomes overwhelmed. Also, the length of the node list is 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 particular 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 are caculated based off of 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. This is able to detect flooding of nodes within the network, this is when a node may become overwhelmed if too many messages are sent at once.

A single datatype was used for the algorithms where subclasses implemented the functions for the error detection and correction.

## X.X.X Algorithms

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 these detect and fix errors respectively.

Three algorithms were implemented and one control (where no error correction is applied.

### X.X.X Acknowledgement Algorithm

This algorithm consists of each node sending an acknowledgement of the message once it has performed the action and all its children have also sent an acknowledgement. This means that each command to add or remove a node is propagated through the network (each node performing the action as it is received), once a leaf node is reached an acknowledgement is sent to its parent. This propagates backwards through the network such that when a parent has an acknowledgement from both its children it then sent it’s acknowledgement. The command has been successful if the instigator node receives acknowledgements from all of its children.

If no acknowledgement be 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 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 is not able to detect inconstancies in the network after they occur. When errors occur it is easy to locate exactly where they occurred as that will be the node waiting for an acknowledgement.

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.

### X.X.X Random Check-up Algorithm

The random check-up 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, 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 (loops back round to zero if it goes over the length of the node list).

When a node receives another’s node list it checks for inconsistencies with it’s 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. The exception is if the difference is regarding one of the nodes, in this case it always wins. This is because the lower the node id the closer to the instigator node it is and so is more likely to be correct.

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 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 that lower down. As a result the message will gradually pass through the network until all nodes are consistent.

It was considered to make each node independently choose another node to check. However, the node list has a binary tree structure. This means that as messages are passed down the tree the chances of an error occurring increases. Since the size doubles with every level of a binary tree this means that each node at any given time has a 50% chance of encountering a #TODO: \* it was considered that randomly checking any node (as opposed to using an offset) would favour leaf nodes (or those close to leaves) as each level has twice the nodes of the others. We could weight these so that nodes closer to the root are favoured but this is likely to causd flooding, this is why the offset was selecteds

This algorithm requires a central coordinator, the instigator node, to generate a random offset and it requires the entire node list of each node to be passed with each check. The random offset must change with each run of error correction. The size of the node list will not be significant, but it represents a considerably larger message size than the acknowledgement algorithm. Another issue with this algorithm is that previous runs of error correction may be undone: say 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. This is somewhat mitigated by each of the nodes checking if the conflicting node is active on the network first but it may mean that the network is slower to converge to the correct solution.

### X.X.X Check Neighbours Algorithm

Similar to the of the random check-up 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 then the next available node is the neighbour. The node list loops back on itself so the first and last nodes are neighbours.

As with random check-up, this algorithm when it resolves conflicts as follows:

If the conflict regards one of the nodes involved, then that node’s node list persists

1. A connection to the node is attempted if it cannot be connected to then it is removed from each of the node lists
2. Otherwise, the node lists are updated with that of the lowest node id

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.

As with the previous algorithm, errors may be propagated 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 random check-up, 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 random check-up it needs to wait until all nodes have finished as they can be checked in any order.

### X.X.X Contrast of Algorithms

The acknowledgement algorithm allows for messages to be signed allowing each node to easily to verify the legitimacy of the command it receives along with being able to verify that every child node has also performed the action. For the check random and check neighbour’s algorithm this is more difficult. The authentication is left to future work however this is an important consideration. 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.

## X.X Evaluation of Algorithms

The acknowledgement algorithm was measured on the number of attempts were taken to add or remove a node. The other two algorithms were measured on the number of rounds of the algorithm needed before the node list was consistent. The algorithms are measured differently because the acknowledgement algorithm ensures consistency whereas the other algorithms detect and repair mistakes to the network.

#TODO: consider the use of saving the most recent instruction for that node (e.g. add/remove, doesn’t scale well if you want to add more commands and just because a command is the most recent doesn’t mean it is the most likely to be correct, higher in the node list always does – need to justify this better

#TODO: \* considered checking from leaf nodes upwards to the instigator - instigator would have to message half of all nodes which produces undue strain on network - so back propagation algo is better

#TODO: • compare with no check and repair as a baseline

#TODO: • messages could be dropped on way to instigator, the simulation does not cover this as what the instigator has is deemed as the correct list

#TODO: • equal chance of node joining or leaving the network (except cannot remove the last node)

#TODO: • a logging system for bugs but also for neatly writing output

#TODO: • while implementing the back prop algorithm it was noted that any mistake was amplified and not detected, any incorrect detection or simple hardware failure would mean a severe failure of the system. This means a combiniation would be better

#TODO: • found bug that exists in last years implementation where if the instigator node is removed then the not all branches are updated (as the other branch of the zeroth node is not followed)

#TODO: • note that for the encryption, it is better not to implement an allocator as this could act as a side channel, wait for future version

#TODO: • time taken is calculated rather than the time of the simulation so then it is invariant of the speed of the computer

#TODO: • measuring the number of rounds of resolution give an idea of how frequently it is needed

#TODO: • the number of attempts is measured, in the final development this would result in removing a node but we want to see how this is done

#TODO: • we must divide the number of messages and floods by the number of times it took to resolve conflicts, should we?

#TODO: • we assume that the first message is never lost (otherwise it would not have connected)

Chapter X

# Simulation Results

The simulation was run on each of the algorithms with drop rates of 0%, 5%, 10%, 20% and 40% with 5 trials for each algorithm and drop rate. These values were chosen to show how the algorithm degrades as the quality of the network degrades. They are chosen to be extreme values for this reason. Each trial ends when 100 nodes is reached. The use of multiple trials ensures that the results are not due to randomness, having a different seed set for each trial. It also means that the structure of the node list is different for each trial as large gaps in the network in a trial could skew results – this is why multiple trials are used.

The algorithms are compared based on the message size, number of messages, flooding of nodes, and attempts/rounds taken.

Since neighbours frequently send messages to each other, an optimisation of the system would be place low latency devices near each other (i.e., devices that are physically closer). This means optimisation between the node list would also make the check neighbours algorithm more efficient.

SINCE ACKNOWLEGEMENT IS WAITING FOR THINGS TO COMPLETE, WHEN IT DROPS A MESSAGE LOWER IN THE NETWORK THEN ALL THE PARENTS WILL ALSO TIMEOUT!

## X.X 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 degrades the performance of Popcorn.

The check neighbour and the random check-up algorithm have similar message sizes: n unique identifiers for a node list of n nodes. Random check-up will also contain an integer offset value. This 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. This could be optimised to reduce the message size based. The message frequency of these two algorithms is the same with n messages being sent per round of conflict resolution.

It is possible to optimise the previous two algorithms by hashing the node list and passing this to the node that it is checking. If these values do not match, then it triggers a full check. This drastically reduces the messages size, making it constant with respect to the node list length.

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). Since it passes its messages according to a binary tree, the number of messages scales logarithmically. However, if a message is lost, this will cause nodes to timeout and retransmit previous messages. When this occurs in leaf nodes or those close to the leaves then the number of messages retransmitted will be large.

In summary, the acknowledgement algorithm has a smaller message size (O(1)) and lower frequency of messages sent (O(log n)). With some minor optimisations the check random and check neighbour algorithms are able to achieve a O(1) message size and a messages sent across the network of O(n) per round of conflict resolution. Further optimisations could be used to reduce the number of messages sent across the network. This could be done by reducing the frequency of checks based on the previous commands received and previous mistakes found.

## X.X 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. Within the simulation a node is classed as flooded if it receives a large number of messages within a given time period. Time is simulated within the program so number of flooded nodes is arbitrary, it only provides a comparison between different nodes.

PROVIDE COMPARISON BETWEEN NODES

## X.X Attempts Taken

### X.X.X Acknowledgement Algorithm

For the acknowledgement algorithm the number of attempts is tracked. An attempt was defined as any command that terminated in a message that was dropped. This means that as the node list grows the number of attempts will grow exponentially. As with each level of the binary tree structure there are twice as many messages sent.

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

Figure 1 and Figure 2 show the average and maximum number of attempts in the acknowledgement algorithm respectively. These graphs show how the algorithm degrade as the node list increases in length and the number of dropped messages increases.

### X.X.X Check Random and Check Neighbours

The experiment was initially run to find the attempts as with the acknowledgement algorithm. However since check random and check neighbours are not guaranteed to detect all errors any previous errors were carried onto the following attempt. For this reason, all errors were corrected before moving onto the next command. Instead for these algorithms the number of rounds of the algorithm that were needed were recorded.

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 3 and Figure 4 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 it can take multiple rounds of the algorithm for the correct value 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 final values are less reliable. 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 80 on with a 5% drop rate. This is because 80 is a large number of nodes but there are fewer data points closer to 100 and the 5% drop rate is the closest to a real world network. The results are shown in Table 1.

|  |  |  |
| --- | --- | --- |
| ALGORITHM | AVERAGE ROUNDS | MAXIMUM ROUNDS |
| Check Random | 1.15 | 4 |
| Check Neighbours | 1.08 | 3 |
|  |  |  |

Table : Average and maximum number of rounds required to repair a network of 80 nodes with a drop rate of 5%.

This shows that in a realistic implementation of the network there would be a small number of rounds needed even for large node lists. The maximum number can be used in the implementation to maximise the probability of resolving all errors.

The maximums, shown in Figure 5 and Figure 6, show that the maximum number of rounds needed does not scale exponentially and for low drop rates does not reach excessive values. This is also demonstrated by the values in Table 1.

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

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

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

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

### X.X.X Summary of Attempts

## X.X 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.

### X.X.X 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 node takes ///////////////////

### X.X.X Check Random

Check random does not need to wait for a reply from another node which significantly reduces the time taken. 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). As seen in the previous section the number of rounds needed is low and so has a O(log n) time.

### X.X.X 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 extremely small compared to the other algorithms.

## X.X Scalability

### X.X.X Acknowledgement Algorithm

Of the three algorithms the acknowledgement algorithm scales the worst. If an error occurs then node that passed the message will timeout. However, the added time will cause all previous nodes to also timeout and retransmit the message. Since an error is more likely to occur the larger the network then it will result in more timeouts.

### X.X.X Check Random

Due to a central coordinator being needed to decide on the random offset value the check random algorithm must propagate messages through the Popcorn network. This takes O(log n) steps to send through the network. This provides good scalability for this algorithm.

### X.X.X Check Neighbours

The check neighbours algorithm is the most scalable as each node can independently decide when to run a check. This makes for O(1) operations. The number and time taken is invariant to the number of nodes in the list. Although the number of rounds does increase as the node list increases it remains low even for a large number of nodes.

## X.X Summary

Summarise each section

Check neighbours was chosen on balance, because of…… despite …… because WHY IT DOES NOT MATTER

It was noted during the implementation and testing of the acknowledgement algorithm that when errors occurred they grew extremely quickly, this is because the algorithm has no method of detecting and fixing errors after the fact. The algorithm, when correctly implemented, should guarantee the consistency of the node list. However, using an algorithm so dependent no error occurring for something as critical as the operating system???

Found during developing the acknowledgement algorithm that when it goes wrong then errors are amplified as it further rounds of error prevention cannot recover from earlier errors. It seems sensible to assume that errors may occur somewhere in the kernel at some point, also Popcorn only requires eventual consistency. When this is a kernel process and so needs to be extremely robust it is better to opt for a more robust system of error correction rather than just error prevention. The experiments show that the number of rounds of conflict resolution are fairly minimal even for high loss network.

Chapter X

# Popcorn Implementation

Use a hash(ish) of the data rather than passing it all of it to reduce message size, store changes since last check

#TODO: storing the token in the node list, this way a new node can verify that the command was correct. This needs to be taken for the node that it is sending to rather than the node it is checking – add in future signed tokens that can be revoked

#TODO: could not set a timeout on the socket because the options is only available on the next version of the kernel – need to stress that updating would be an entire project in itself

#TODO: using a timeout in it’s current form would mean that timeouts could occur while a message is being processed, the next kernel version provides the ability to safely set a timeout for sockets

#TODO: initially only stored recent changes to the node list, then implemented the hash of nodes

They reply if there is a mistake, need to prove that this will terminate – lowest id always wins so this is true.

Chapter 4

# Evaluation

Discuss testing of the final implementation

Should I stress experience with kernel programming? Did do it last year but over two years, six-month academic years, 1/3 of time on project, equates to only 4 months full time work. So still quite new to kernels

Considerations: kept coming across problems caused by assumptions made by the Popcorn before last year’s project. Looking back I could produce something a lot better.

Put in presentation: learnt about symbols, general debugging, etc.

Chapter X

# Conclusion

The code will be opensourced and shared with the wider Popcorn project.

Add what I have learnt and the things I would do differently over this project and the whole project.

# 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. Ramsøy, “A Scalable Node Management Structure and Joining Protocol,” 25 April 2020. [Online]. Available: https://drive.google.com/file/d/1g8kiPErOtCLB8Fy-v3gVgQo5qDO3VJSe/view?usp=sharing. [Accessed 22 October 2021]. |
| [6] | 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]. |
| [7] | J. S. Shapiro, “The EROS System Structure,” 11 January 2007. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.5173&rep=rep1&type=pdf. [Accessed 15 October 2021]. |
| [8] | R. Achermann, D. Cock, R. Haecki, N. Hossle, L. Humbel, T. Roscoe and D. Schwyn, “mmapx: Uniform memory protection in a heterogeneous world,” 1 June 2021. [Online]. Available: https://sigops.org/s/conferences/hotos/2021/papers/hotos21-s08-achermann.pdf. [Accessed 23 September 2021]. |
| [9] | J. S. Shapiro, J. M. Smith and D. J. Farber, “EROS: a fast capability system,” 12 December 1999. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/319151.319163. [Accessed 14 October 2021]. |
| [10] | 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]. |
| [11] | 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]. |
| [12] | R. Needham and R. Walker, “The Cambridge CAP Computer and its protection system,” November 1997. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1067625.806541. [Accessed 27 September 2021]. |
| [13] | 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]. |
| [14] | 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]. |
| [15] | 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]. |
| [16] | 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]. |
| [17] | 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]. |
| [18] | 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]. |
| [19] | J. Corbet, “Popcorn Linux pops up on linux-kernel,” 5 May 2020. [Online]. Available: https://lwn.net/Articles/819237/. [Accessed 25 October 2020]. |
| [20] | 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]. |
| [21] | 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]. |
| [22] | 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]. |
| [23] | 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 10 April 2021]. |
| [24] | “Openmach Git Repository,” [Online]. Available: https://github.com/openmach/openmach/blob/master/include/mach/message.h. [Accessed 8 November 2020]. |
| [25] | “Structure of monolithic kernel, microkernel and hybrid kernel-based operating systems,” 17 July 2008. [Online]. Available: https://en.wikipedia.org/wiki/Monolithic\_kernel#/media/File:OS-structure2.svg. [Accessed 8 November 2020]. |
| [26] | Shubham, “What is kernel - monolithic and microkernel,” 29 January 2018. [Online]. Available: https://medium.com/@shrimantshubham/what-is-kernel-microkernel-and-monolithic-kernel-66c6de358b43. [Accessed 18 April 2021]. |
| [27] | 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]. |
| [28] | B. Ravindran, “Replicated-kernel Linux,” [Online]. Available: http://popcornlinux.org/index.php/replicated-kernel-linux. [Accessed 19 April 2021]. |