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](#_Toc95039277)

[Background 3](#_Toc95039278)

[2.X.X Operating Systems 3](#_Toc95039279)

[2.X.X Popcorn 3](#_Toc95039280)

[2.X.X Summary of MInf 1 4](#_Toc95039281)

[2.X.X Distributed Operating Systems 4](#_Toc95039282)

[2.X.X Capabilities 5](#_Toc95039283)

[Related Work 6](#_Toc95039284)

[2.X.X Amoeba 6](#_Toc95039285)

[2.X.X CAP 7](#_Toc95039286)

[2.X.X Plan 9 8](#_Toc95039287)

[2.X.X Kerrighed 8](#_Toc95039288)

[2.X.X Consistancy Algorithms 10](#_Toc95039289)

[2.X.X SCOPE 10](#_Toc95039290)

[Simulation Implementation 11](#_Toc95039291)

[Simulation Results 12](#_Toc95039292)

[Popcorn Implementation 13](#_Toc95039293)

[Evaluation 14](#_Toc95039294)

[Bibliography 15](#_Toc95039295)

Chapter 1

# Introduction

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

Aim of this year’s project:

* Add encryption and authentication to the joining protocol (security between links is needed for capabilities to remain viable)
  + Evaluate security
    - Show that it is encrypted to today’s standard
    - Attempt to break in
    - Run tests to show how much it slows down the system
* Use capabilities to secure
  + Create internal resource authentication system
  + Process server, page server, etc. passes a capability with desired action to this
  + Evaluates whether action can be done, system must be incredibly fast
  + Should not be able to be circumvented – maybe put between interfaces to block things not allowed?
  + Evaluate
    - Security of the system, can it be circumvented?
    - How it ensures security
    - Slowdown of using system
      * How slowdown is reduced
    - Comparison between other systems?

Not very original so would require a more detailed evaluation or more original design

Any thoughts on adding something more original or any aspects that can be more deeply evaluated?

Implementing resource control with access control lists and a central server and compare according to scale, this could get interesting because as the latency on the network increases capabilities would become more favourable meaning they would be better for larger networks

Chapter 2

# Background

### 2.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.

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 Popcorn with a single cache coherent memory linking them together [4]. Popcorn allows for heterogeneity between nodes [4].

### 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, and 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) 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. They are unloaded when the nodes using it is removed so no additional resources are used.

Before last year’s project, Popcorn would only allow a list of nodes that should be connected to during the loading of the module. This was changed to load with no other nodes attached and established a joining protocol to allow other nodes to join. 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.

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 goals of this project highlighted in last year’s project were: “I intend to finish providing security and authentication between nodes, add flow control to the Popcorn command messages, add functionality to remove nodes, create a command utility that allows for connection of nodes in a more user-friendly way and perform a more in-depth evaluation of Popcorn including the use of real hardware. ”

### 2.X.X Distributed Operating Systems

“The users of a true distributed system should not know (or care) on which machine (or machines) their programs are running, where their files are stored, and so on. ” [6]

There are three broad categories of distributed operating systems referred to as the “minicomputer”, “workstation”, and “processor pool” models [6]. The minicomputer model is where each device on the network has multiple users a user logs into a machine. The logged in user uses that machine but also has remote access to the others on the network [6]. The workstation model is where the user has a powerful machine that does the bulk of the work but with some distributed services like a shared file system. Finally, the processor pool model is where a group of processors are available to each user, where users can use a flexible number of processors (between them and the other users on the system [6].

A problem with distributed systems is maintaining naming consistency across different machines. E.g., two user may have the same user ID (UID) on different devices, they should not however be considered the same user, or have the same privileges when accessing resources on the other machine. As a result distributed systems either login through a centralised system or use the name of the computer that they are logged into as part of their UID (it can be thought of as user@device\_name). Another solutions are to create a user of minimum privileges (e.g., demo user), however this means users not logged on to that machine have fewer privileges than ideal. It is considered that a true distributed system should have a unique UID for each user that is used across the system [6]. But this requires a shared structure that must be kept consistent across the network.

The OSI has significant overheads so message passing is generally favoured for distributed operating systems [6]. Using message passing it must be decided whether to use blocking vs non-blocking, and reliable vs unreliable messages. This refers to code that is required to wait until a message is received vs being allowed to other things in the meantime and confirming if the message was delivered vs not confirming, respectively [6].

### 2.X.X Capabilities

A capability is a descriptor that contains a unique ID that represents a server process. This ID is cryptographically linked to the process it describes in such a way that only the server can understand which process it is for [7]. This provides protection to the process as only a user or process with the correct capability can access the protected resource (another process, memory, etc.). The rights that are granted can be specific privileges to that resource (e.g. write, read-only, execute) [8].

For a system using capabilities to remain secure it must be ensured that capabilities cannot be modified, only authorised interfaces can create them, and they are only given to processes that are authorised [9].

Capabilities can provide a secure method of granting privileges. however, they are difficult to revoke after creation. For this reason, many capability-based systems either use an access control or an intermediate system (referred to as a reference monitor in EROS) to stop revoked capabilities being used [9].

### X.X Consensus Algorithms

* Consensus:
  + Majority rules
  + Crash and Byzantine failures
  + Consider that in the popcorn setup the higher up the tree the more likely you are to be right – other algorithms don’t take this into account

Consensus algorithms must ensure that the value chosen must be that only a single value that had been proposed is selected. This selection must be atomic [10].

A consensus algorithm can use a single designated node to decide the value. However, this provides a single point of failure. Instead you can have multiple nodes that decide that decide on the value, with a majority deciding the final 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.

### 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

Performed in two stages:

Promise: where a request is sent out from one node to all the others to state that it will be in charge, they all reply with a promise that this will happen.

Commit: then the node asks to commit a value to all of the nodes, they all agree, this provides a lock 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].

### 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.

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].

Chapter 3

# Related Work

### 2.X.X Amoeba

The motivation behind Amoeba was to build a system where all resources are automatically managed by a distributed operating system [10]. As a consequence, 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 are able to 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 has had several iterations of file systems however, the simpler “bullet service” that stores immutable files as contiguous bytes [10].

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].

Amoeba’s design allows for great scalability, fault tolerance, and for processes to temporarily acquire large processing power [10].

To increase reliability new objects in a directory are replicated a set number of times and distributed across different nodes [10]. Capabilities are then stored in the directory along with the files [10].

To maintain security capabilities are used. By using a sufficiently large address space and having capabilities cryptographically linked to the resource that they are protecting, this protects the capability and therefore the resource it is protecting [10]. The use of cryptography allows capabilities to be safely used within user-space, thus following the principle of minimality and simplifying the kernel [10].

A hash of the port number that the message was sent on along with a shared secret means that an adversary cannot gain access to this [10]. This can be implemented in software or hardware but there should be no way of bypassing this function [10].

Capabilities are distributed meaning that transfers cannot be detected, this means that to allow mandatory access control a system within the kernel needs to be implemented [9].

Capabilities are not used for individual pages or memory mapped structures but instead larger structures [9].

For directories, encryption key and a random value are XORed together. The result is stored in the directory itself and the capability given to the user that owns the file. When the file is requested by the user they provide their capability along with the random value. This means only the owner is able to decrypt the directory [10].

New capabilities that have a subset of rights of the other can be created by the owner [11].

Amoeba relies on the security of the capabilities and there any messages carrying them need to be encrypted to maintain the security of the system [11]. Two different systems can provide security to the network: one is a Kerberos like authentication server, and the other uses hashes. The authentication server results in a slower system however, the second system assumes a secure network and kernel [11].

### 2.X.X CAP

The virtual address space is mapped to a physical address space used across all devices [8].

CAP can have a hierarchical structure to represent processes. The position in the hierarchy is used to regulate the resources that the process has access to [12].

Capabilities can be used to restrict memory for example, each capability lists a maximum and minimum contiguous memory range that the permissions are for. A child capability can be generated from this that gives a maximum and minimum memory value relative to its parent. This way only memory in range of the parent can be used by the child. Any number of levels parent-child capabilities can be made [12].

CAP restricts users from procedures using a capability. to restrict who can run it, this is opposed to the conventional way operating systems determine each of the things done are allowed (and throws error when this happens) [12].

### 2.X.X Plan 9

Plan 9 allows for heterogeneity; different processor architectures are able to 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].

Authentication between nodes is conducted by solving encrypting/decrypting challenge messages with some key [13]. Following both of the nodes success on the challenge messages one of the nodes sends a message to the authentication server. The authentication server then sends a message to both nodes containing a conversation key. The message that the authentication server is encrypted such that only that node can get the conversation key. The conversation key is then used to encrypt the conversation from then on [13].

This method does not rely on a synchronized clock, unlike Kerberos [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].

### 2.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].

More OSes

Implementation:

“Applications can use an eventfd file descriptor instead of a pipe

(see pipe(2)) in all cases where a pipe is used simply to signal

events. The kernel overhead of an eventfd file descriptor is

much lower than that of a pipe, and only one file descriptor is

required (versus the two required for a pipe).”

Stuff on consistency algorithms

### 2.X.X Consistency Algorithms

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].

### 2.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 about encryption

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 it’s 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:

1. If the conflict regards one of the nodes involved, then that node’s node list persists
2. A connection to the node is attempted if it cannot be connected to then it is removed from each of the node lists
3. 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

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.

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.X Message Size

It is important to consider the size of the messages being sent. Large and frequent messages will cause large overheads to the network. 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. Even for a large node list e.g., 1000 nodes it would this be large?

The acknowledgement algorithm has a tiny message since by comparison, needing only the node’s address, node id, and an integer to represent the command (add or remove).

When using check neighbours, it would be possible to reduce the message size by storing what has changed since the last time the node checked. This would drastically reduce the message size.

Use check neighbours: the message size is large but can be significantly reduced as the neighbours are fixed, it does not require significantly more rounds than check random, it does not require central coordination. With reduced message size the impact on the network is going to be lower than with random check-up even with more rounds of conflict resolution.

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???

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: 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

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

# 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]. |