Department of Computer Science


Submitted in part fulfilment for the degree of MEng.

# Swarm Memory

Harry Burge

Version 1.0, 2020-November

Supervisor: Simon O'Keefe

# Acknowledgements

# Contents

# List of Figures

# List of Tables

# Executive Summary

# 1 Introductury Material

## 1.1 Introduction

Swarms are an increasingly important area of research for society, as the world moves towards a distributed technology future. The research of swarms within a technology setting can be broadly divided into two partitions, these are intelligence and mechanics.

Swarm intelligence can be viewed as the research into highly distributed problem solving[2, 5]. This is ever more becoming relevant as computer systems start to level out in sequential performance [7] and parallelism is embraced, satisfying the demand of the age of big data [9].

Swarm mechanics heads more down the robotics side and can be seen as the study of practical implementation of swarms, whether that be movement or communication within the swarm. This is on the rise in the industry, as society's pace increases and manual labor is automated out, whether its drone delivery to inpatient customers or mapping areas in dangerous environments [1].

These two areas often are highly integrated rather than being disjoint from each other, and are rarely seen in their pure form. An example of a pure form of swarm intelligence can be seen in [5] with network routing protocols. This project will be focusing mainly within said grey area, but predominantly in intelligence dealing with a practical problem.

Focusing down on distributed and local memory of swarm-like agents, that will be covered by this project. Most research has gone down the route of optimization on distributed problem-solving algorithms, as compared to practical applications of storage of abstract ideas as a collective. As one of the key reasons for using a swarm is redundancy, which is often assumed and boasted about, rather than proven specifically within the memory domain.

This relative lack of research into collective memory, seems to this author to be a glaring hole in the foundations of a complex and interchangeable subject. The need for more research into collective memory can be seen

in how invaluable it would be to applications like mapping of a dangerous area [2]. By being able to handle the loss of agents and the collection of data on agents with limited memory, we could ensure efficient solutions that have high redundancy compared to other implementations.

An explanation for swarm based memory management solutions to be a less developed area of study is the existence of subjects like cloud-based and raid based storage systems. The argument for having these two subjects partially-separated is due to the nature of a swarm's locality and ever-changing network style, compared to a server farm. Storage of data on an ever-changing network of devices is a hard task to complete, handling loss of connection between servers, reliability to access of the data and loss of services, whether it be non-correlated or correlated failures [9].

Most elements of cloud storage policies at a high level of abstraction could work effectively within a swarm based environment. However as explained above, key adoptions would need to be changed to create said effective policy. A prime example of the type of algorithm is "SKUTE" from [10]. "SKUTE" will be the main inspiration for this project's solution, too storage on a highly dynamic network.

The objective of this dissertation is to merge three areas of study into an effective/suitable storage policy for swarm-like agents in a setting with high locality and dynamic connection behaviours. Then to perform multiple analyses on the created policy using a variety of simulations to gauge its capability compared to the desired abstract behaviour.

To complete the objective of this report we will programmatically break down the problem described in Section 1.2 into solvable tasks. Therefore the report will be structured as follows, firstly we'll define a clear and concise problem, of which incompaces all disgruntlements described above. Secondly, bring the reader up to date with relevant literature and explain key concepts that will be required for a complete understanding of where the proposed solution has been derived and why said solution might be a relevant stepping stone for future research. This will be covered in the two sections, Section 2.1 of which goes into detail about current cloud based storage technologies and Section 2.2 of which will delve into more of a background behind swarms, specifically relating to the problem and possible solutions. We'll then go through the methodology of the proposed solution's design. How it is supposed to act and react in different scenarios, and the reasons for why. This leads to analysing how effective the proposed policy has kept to standards derived in the methodology section, and whether it solves the problem defined in Section 1.2.

## 1.2 The Problem

The problem this report will cover is defined as follows. We need to create a storage policy for agents of a swarm, to be able store directional abstract data as a collective, without complete duplication.

This problem can be split into two separate sub-problems. One is the handling of data duplication throughout the swarm as to be able to control for failures of agents, and provide a mechanic for recovery from said failures.

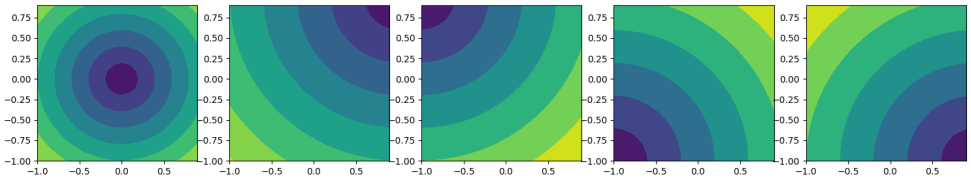

Figure 1.1: Data duplication density based off distance to datas desired location

The second is controlling where the duplicated data is within a world compared to where the data is needed in that world. A visualization of this can be seen in Figure 1.1, this shows the density of duplications within an area based on distance from the desired position of that data. As the reader can see the further we get from the desired point we should have less duplicates of data compared to agents without duplications of that data.

There are lots of examples where a problem like this could arise, a good example of where this could be applied is operating in a mine filed [2]. Near the start of the agents operation they will need to map out where potential mines might be and record their location. Obviously once an agent finds a mine we would want it to spread the word of the location of said mine. However due to memory restrictions the agents can only know about a minimal amount of mines, therefore agents closer to the mine should prioritize knowing its whereabouts compared to mines it may never come across. Now bringing this into a more of a practical solution if agents fail individually for example running out of battery, or as a group from a mine going off. We still need the swarm to not lose data, in the case of our example it would still be relevant to know where a possible crater could be.

With this increase in complexity, the storage policy needs to be able to withstand the fluctuations and locality of the swarm. With reliable redundancy to handle correlated (Mine detonation) and non-correlated (Power loss) failures.

# 2 Litreture Review

## 2.1 Cloud/Backup storage policys/schemes

Like most things in computer science, cloud storage started off relatively simple (In nowadays terms). As the years have progressed so has the demand and use of the cloud, varying from everyday people storing files to large businesses storing harvested data. This led to the need for much sophisticated storage solutions, which can handle the increasing file size and frequency of use. A component that increased this complexity was the Legal Services Act. 2007 [11]. This enforces that cloud storage suppliers must provide reliable and fast data collection for users. Not only that but to also provide near guaranteed longevity of stored data. In this section we will focus on cloud-based storage policies and some background terminology needed for this report.

Firstly the reader will need to understand the difference between correlated and non-correlated failures. A non-correlated failure is when a device fails independently and with no relation to other failures with the system as a whole. An example of this within cloud based computing, a server node can shut down due to a software failure, therefore we lose connection and access to the node's data, typically this is completely independent of other servers in the rack. A correlated failure as the name suggests, is when a device fails with other devices with relation linking why they have failed. A typical example of this in cloud computing would be mass restarts because of a power surge from a storm. The power surge event is what links the failures together, therefore making it correlated. To be correlated doesn't mean they need to be geographically close together, however within our swarm solution will mean geographically close failures.

To control for these two types of failures there are many different solutions providing different features. Locally what is typically used is a RAID system for local failures within a node, and then a replication policy [9] for internode duplication. These in tandem provide stable node storage and redundancy for when a possible node failure acquires.

The most common forms of replication policies will take a piece of data, decide whether the data needs to be duplicated and if so will completely

copy the data onto another storage device. This provides a backup in case of failure on either one of those devices. A simplistic approach would be a random replication policy, where data is randomly chosen to be duplicated, typically duplicated within the same datacenter, so on a neighboring rack. This is an efficient design policy for handling non-correlated errors, however, lacks the robustness against correlated errors, and without a tracking of global duplications can lead to over used storage. We can mitigate for correlated errors by allowing for duplications to happen over data centers, however, this leads to downsides which will be explained below. An algorithm like random replication, is substantial for long-term storage where popularity of data and distance to users are averagely the same for all data items.

Two key concepts of availability and popularity are not taken into account when using a random replication policy. When cloud based storage transitioned from a semi-local backup system to a worldwide daily driver. Random replication can't withstand the variability of how users interact with files nowadays. An example of why these concepts are needed in today's age, are videos. If a video is hosted in one country and replicated within said country then international viewers might have delays to their streaming. If we then tried to compensate for that by hosting it in multiple countries, those other countries might not view the video as much. This therefore means we are wasting storage space of which we could use for other more popular videos. This therefore leaves us trying to maximise for both situations, and a new replication policy is required.

We will be looking into two different algorithm concepts, of which try to handle the maximisation problem. Both withstand non-correlated failures well because of the nature of replication, so we will only be looking into the other effects. The algorithms have been abstracted from papers on handling "Distributed key-value store" [14], where you have key-value pairs on multiple devices on a network where duplication only leads to more fault tolerance of the data stored.

The first approach uses a privileged level of control where it uses its global knowledge to make decisions about whether and how to duplicate items [9, 12]. This doesn't have to just be data replication, but the same principles can be seen within schema changes [13]. Due to the nature of having a privileged user, the control of the policy is a lot easier to make specific behaviours be exhibited and to be understood. This means we have higher guarantees for correlated failures unless on the master node, however this can be handled dynamically by assigning another master, touched upon in Section 2.2. Availability and popularity are handled by the policy coded onto the master node.

Having an approach that uses a master, doesn't work effectively for a

swarm. This is because the change from a server network to a swarm is quite a drastic change. Servers running over network generally have complete connectivity e.g. Global scope. Also servers have a constant power supply compared to the average swarm agent. When restricting the masters scope we have to rely on messages over other agents which will first of all reduce processing capability and power loss will be more significant to agents closer to the master, possible leading to a cut off from command [1]. It also doesn't fit into the ideology of a swarm, this will be talked about in Section 2.2.

The second approach doesn't rely on a privileged member and can be adapted for locality. We follow a distributed control approach where each node (In the case described below its each key-value pair) has its own controller. Following the approach used with "SKUTE" as proposed in [10], it can make four decisions per data item. These decisions are; Migration, Suicide, Replication, and Nothing.

Migration is the move to a lower cost or more redundant servers. Suicide is the removal of itself, this is usally be because of to many duplicates. Replication is when the data decides that it needs to be duplicated and sent to another node and Nothing is as the name indicates.

Because of the highly distributed nature of said approach, when coming to suicide we need to handle the edge case where the only two nodes with duplicates make the decision to suicide at the same time, therefore leaving us with no replications. This is where a consensus algorithm comes into play. Paxos [15] is an example of how both nodes couldn't suicide at the same time, therefore leading to the ideal case in this example of only one duplicate.

Within the domain of server storage networks an approach like "SKUTE" is less commonly used because it adds complexity which is not needed within a global and consistently powered network. Most data warehouses would prefer to have one server running as a controller and other servers running at full capacity compared to all servers at a slightly lower capacity due to extra self computation. However this approach allows for greater redundancy because of having no single point of failure.

An approach like this is highly adaptable towards a homogeneous swarm. However with heterogeneous swarms the previous algorithm may work a lot better. Differences between the swarm types will be explained in Section 2.2.

Moving towards local redundancy and optimisation is the stagnated study of RAID. This is where we change orderings of multiple storage discs to gain redundancy and/or performance increases. This grouping of disks is

called a RAID array and can be structured in a multitude of ways. Common structures are labelled as RAID levels and give different attributes based on what functionality you are pursuing [**?**].

One of the key components of multiple RAID levels is the use of parities [8]. This is where a function (typically an XOR) is done on two or more sets of data to create one or more parities. The function has a property thus that if a tolerant amount of disks are lost the data that was lost can be reconstructed. In the case of data A, data B and A XOR B, if data B is lost then can be reconstructed using data A and A XOR B. With different levels and functions more than one disk can fail and still retain data however if failures go over that then all data is lost. RAID is predominantly used internally within a storage node to provide redundancy against disk failures and increase speed of writes that are typically on hard disks, because of cost and possible reads after failure of heads.

The methodology is that as long as the nodes individually are redundant enough and then there is control on a higher level with our replication schemes, that is a sufficient redundancy. RAID could be adapted to run over multiple different storage nodes, however the complexity compared to performance is heavily in the favour for the above methodology for storage networks. We will come back to the possible uses of RAID internally and externally in Section Improvements.

// Finish rewriting // improvements refrence

## 2.2 Swarm robotics

As explained in the introduction, the study of swarms are split into two subsections, mechanics and intelligence. Both explained broadly in the Section 1.1.

Continuing on the discussion about swarm intelligence. Typically swarm intelligence focuses on solving abstract problems, like the traveling salesman problem, in a local distributed manner compared to a global manner. From the papers read by the author, it seems that predominantly the topics undertaken are testing distributed solvers compared to already researched solutions to see how they measure up. An example of this within the TSP domain is a genetic algorithm versus AS-TSP [5]. These algorithms provide benefits and drawbacks compared to their counterparts.

The concept of swarm intelligence is creating a solver to a problem using a distributed algorithm that can rely on natural parallelism, doesn't rely on global knowledge and is adaptable on the fly, compared to their counter-

parts. A good example of where these algorithms excel is the networking domain, because of the high parallelism and need for adaptability [5].

The other subsection can be broadly known as swarm mechanics. This encompasses all of which swarm intelligence doesn't cover. Swarm mechanics focuses on problems that are less abstract and are usually in the domain of physical implementation [2, 4]. Swarm robotics can be seen as the same as swarm mechanics, however, it doesn't have a broad enough scope/name to fit everything that can be researched in this author's opinion.

Two arguments to justify the naming of swarm mechanics are as follows; within this domain we focus on the emergent behaviour of a swarm compared to the solution it might give. The second is that swarm robotics doesn't encompass the study of swarm behaviour in nature [5, 22].

Delving deeper into swarm robotics we have three different methodologies, heterogeneous, homogeneous and hybrid [1]. These can be adopted in multiple ways, however, we will focus upon the adoption of these methodologies on decision making and physical attributes of agents. Firstly we will talk about the physical adoptions.

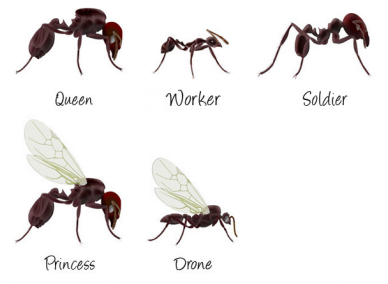


Figure 2.1: Example of a hetrogenus ant colony. https://www.pinterest.co.uk/pin/777363585651532845/

A heterogeneous swarm is defined by having differences between agents of the same swarm, as in Figure 2.1, whether physical or mental [1, 5]. These occur commonly in nature and are less studied [5], because differences in agents are a rarely needed property in research-based problems. For real-world solutions, heterogeneous swarms can be of great use, allowing other agents to pick up the slack of the swarm, or complete tasks that other swarm members cannot complete. A good example as described in [1] with a mother ship being a navy boat and a swarm of quadcopters. The boat picks up for the slack of the swarm by being able to transport them longer distances than the swarm could normally cope with.

The argument against heterogeneous swarms is that there is a tendency to over rely on the differences of agents. Running with the example from

[1], the agents rely on the naval boat to be able travel longer distances and have to have a recharge point. Without the boat the swarm fails after some time. The decision to have a heterogeneous swarm in this case is valid because if the naval ship is lost, then something has already gone horrendously wrong.

To have physical differences in agents, is to breed efficiency. However this can only hold true if the vulnerability of losing too many agents of the same type, that's work is key to the survival of the swarm, is mitigated. This vulnerability can be controlled for in multiple ways. Common rules are found in nature's swarms and can be extracted from them. These are; jobs need to be either interchangeable between all types of agents however some agents are more efficient at that job [22] or the jobs that are specific need to be non-essential for the colony's survival.

Ants typically fit into this category where ants have different types, as shown in Figure 2.1. Some ant species, like Leaf-Cutter Ants, even have subcategories within a category of type. Leaf-Cutters have workers that will specialise in certain tasks like fungus farming. The best example of showing the interchangeability of these roles is when major ants do worker jobs when there is a significant loss of workers [5]. This leads us more towards a hybrid approach which is explained later on in this section.

Because practical implementations of what humans can achieve currently in their robots, they don't have the adaptability that biology can provide, without making agents too complex.

Homogeneous swarms are defined by each agent being the same. This is found less often in nature, except for at the microbial level, and is commonly found in man-made agents. Because of biology's natural adaptability compared to current standards of robotics, semi-heterogeneous swarms are exploited better [1, 5]. Therefore we maximise for the floors of our current technology, however with a sufficiently complex agent homogeneous swarms are the most optimal, but that is getting into speculative futuristic technologies of self replication, advanced intelligence and nano size.

Homogeneous swarms benefit from maximum redundancy, this is because if any agent goes down there are still an entire swarm's worth of agents to take its place. With this benefit of redundancy we acquire some possible losses in efficiency which could have been exploited with agents with specific hardware. The design for homogeneous agents is a complex one, either the agent is too simplistic therefore loses efficiency in their tasks. Or they are too over engineered to the point that all agents have the ability for every specialism and may never need said hardware. There is a thin line between the two, where either we lose practical power of the swarm or we have to invest more into the swarm than it actually needs. An example to

show this dynamic is if we have agents that need to mine and farm, they all have hands so can do the task at a suboptimal speed. We then as an improvement give them picks and hoes instead of arms. This gives us an increase in mining and planting speed, but why would the miners need a hoe.

Within the practical implementation of swarms, everything gets a bit messy. Usually there is not a clear cut framework or design that a swarm is designed to be like. They are designed to be as efficient as we can make them to be in any type of problem faced, this is where research and engineering have a bit of a disconnect. This is where hybrid approaches come into their own.

A hybrid approach tries to exploit the benefits of both heterogeneous and homogeneous designs without the downsides. Carrying on from our example of farmer and miner swarms, a hybrid approach to the physicality of an agent would go something like as follows. Each agent would have exactly the same body and could wield either a pickaxe or a hoe, but the key point is that a miner could become a farmer if it was required. The reader might wonder why physical hybrid approaches aren't regarded as the best of all approaches. This is because when bringing a theoretical solution into the real-world we gain massive complexities. How can we guarantee job type distribution? What about the complexity of changing between job type hardware? These are all questions that are needed to be explored by the person creating the hybrid swarm. Usually it is easier to go for the simpler solution and deal with the possible loss of either efficiency or adaptability.

Moving on to the control/decision making of a swarm following these approaches. Homogeneous control follows the purest form of swarm robotics, where each agent has control of its own decision making based on what other agents in its locality are doing, sometimes labbelled distributed intelligence. This therefore creates an emergent/structured behaviour of the swarm even though each agent is acting of its own fruition and can usually only see locally. A simple example of this is [23].

Heterogeneous control is also very simple in nature, where certain agents control other agents' decision making. This can be handled in multiple ways, two promoniante ways are hivemind control [18] and royalty/hierarchical control. Hivemind is where one agent controls the entire swarms decision making e.g. The hivemind controller will say the swarm needs to move to the left and then the agents handle that the way it decides. In a hierarchical approach it follows the same style as hive mind however deals with scalability better. Where we somehow have a power structure of certain agents being sub leaders of leaders. This control structure leaves itself vulnerable in the same way as its physical implementation, however, also

has the fear of bad actors and power loss distributions over the swarm. These still can affect homogeneous swarms however is less of a threat than on a heterogeneous controlled swarm.

Hybrid approaches to control of a swarm are just heterogeneous control policies that can adapt to changes of leaders and are usually designed for homogenous/hybrid swarms. The choice of leader(s) is usually done through a consensus algorithm [15] rather than based on any form of physical or mental difference. This allows for homogenous style bots to act in heterogeneous fashion. This can create more deterministic behaviours compared to emergent behaviors of homogeneous control, and can also help with the power loss distribution problem by re-electing leaders in different locations to help distribute where messages are relayed.

Humans themselves are a great example of a fully hybrid based swarm both in design and communication. Though humans have variations in characteristics they can be seen as pretty homogeneous in terms of the tasks that they can perform, obviously removing edge case actions like child birth. Tools and knowledge can be spread between humans to make the swarm more efficient and an agent can specialize in a certain area. However, if some agents are lost other agents can replace them by using the same tools and knowledge from the remaining agents of that specialism. Also, the natural power-based structure of humans fits a hybrid model in terms of electorship of some kind, and not of genetics (Except with royalty, however, this is more of a label rather than a genetic difference). The leaders aren't needed for every single action so fit into a usually hierarchical power structure, compared to something of a hivemind model.

# 3 Approach and Motivation

## 3.1 Inital Soloution Ideas

Initially before reading much litreature the idea the authour had was to create an adaptable version of RAID, where drives could come in and out of scope and still have guarantes on redundancy. The design would be as follows, for each agent that has space in memory and two agents in its vicinity, we would create a parity of two data points in the other agents that were not duplicates of either eachother or data already in the primary agent. Then each agent would check what agents are around it and check for if any of the paritys are missing an agent that builds its parity. If so then recreate the lost data, else if both agents are lost then delete the parity, otherwise do nothing.

After reading into cloud based computing and hypothsising about the result of said algorithm it is clear that this algorithm wouldn't be good enough of a soloution and would be difficult to deal with the directionality of the data. This is because said algorithm doesn't take into account coordinates of the agents, and the spreading of the data relys heavily on connection losses, so within a semi-stable swarm this would not acquire enough.

This therefore leads us into replication policys, which can provide us directionality and duplication even on a stable network. Because of the nature of a swarm though usally it is semi stable if tlaking about iteration to iteration time scales, we can therefore look at the problem as a stable network problem but have the ability to control for failures. There were two algorithm types that seemed to be the best for a swarm, however one relied on hetrogenus/hybrid control with a controller telling other agents where to move/duplicate items. This would work effectively on a hetrogenusus swarm, but on a homogenusus swarm we would have to rely on masters only controlling there locality or masters be able to talk across the global network hoping packets from agent to agent.

This is the reseaon for picking a ditriubuted control approach where each agent makes its own decisions based off its locality. The benefits of this type of design are that we would have an average power loss over the swarm, because there is no hoping packets to singular points, focusses on higher

parraelism. A downside to using this algorithm is that we can't gaurantee no data loss because in certain conditions an agent might not send off its data due to its locallity. Even with the significance of the drawback this was the algorithm picked due to the all round ability for the algorithm to be adaptable and the benefits of what it does do compared to other algorithms un-ability.

We will model our design of "SKUTE" however we will have to change a few bits. We will take out migration because of added complexity, however this would be good to add in the future for handiling of swarm induced failures. Swarm induced failures would be filures that we couldn't account for with simple designs like an agent being in contact with no other agents therefore it can't pass on its data and leads to a vunrability. Going on from this problem we might get a case where agents in the locality of an agents has full memory this would lead to the same sort of disconnect as having no agents around. This could be mitigated for by either using a priority based system based of factors around the agent, using migration of data onto agents that have more free space on the public memory or a hybrid of the both where is an agent is maxed out and a high proirty store is coming in migrate a lower priority data out a take in the high prority data.

// Possibly change to account for the migration addidtion

## 3.2 Analysis of soloutions

We will analys soloutions propsed using varying scientific methods. We will however not be doing p-value tests due to not having other algorithms to be able to compare it too. We could do tests between each algorithm propsed however the outputs are too subjective to be analysed with those types of results meaningfully.

The analysis will be done by changing diffrent factors or diffrent algorithms, doing ten runs with ten thousand iterations. Within the ten runs we will average these results to be able to give us infomation like per data duplications globally, duplication distance to data points, number of agents and density of duplications about data point. This will overall give us a good anaylsis on how these algorithms perform, when changing factors like non-correlated failure rate, movement styles and correlated failures. It will also show us infomation on how changing certain paramters affects the bahavouir of the algorithm like total duplication.

Even though this pretty much highlights all behavouirs of the algorithm we will also need to create a analysis for seeing instability with in the runs. This

is because the data collected is at the end of the iteration of a agent and doesn't take into account which agents had what data, it only mesaures that there was for example five duplications. Therefore we need to see how many times that data might change agents to account for the stability of the "network". The stabilty is needed because excessive amounts of passing data is likely to cause errors of transmission and will increase power draw over the network.

We will then review this data to see how well we think it performed to the expectations we have set out in Section 1.2 and show it can be adjusted to users needs.

// Talk about specific things of analysis

## 3.3 Motivation

In the previous sections we have described a quite clear picture as to why the study of swarms is becoming more of an integral part for leading us to the future. With sequential computation reaching its limits [7], and the world becoming increasingly data rich, the need for distributed problem solving is heavily required. As technology gets more efficient, the ability to make swarms become exponentially more viable, especially with recent advances in nano technology/biology.

Swarm robotics is limited by the technology of its time. The use of research is limited and far between in our day and age. We research this for the possible future where technology can support and utilize the fullest potential that swarms can bring to the table.

The main uses nowadays are within surveillance [21, 18], delivery or military [1]. However, to see the full scope of what man-made swarms could do we look to the future. Whether it be for space exploration [20], nano-robots in medicine or in the parallel/distributed software domain [19].

It is for these reasons that I have decided to contribute my part to this extensive and breakthrough field, and will hopefully see it grow to its fullest potential.

# 4 Design

## 4.1 Simulation Infomation

This section will describe the environment of which the agents will be simulated in. The simulation is a 2D representation of a flat surface where agents can move freely around. Agents are randomly located on the surface within the center 75% area. Points are then selected on the surface, and at the start of simulation the closest agent to that point learns the data, that data can never be learnt again other than from passing information from agent to agent. The data is directional so when stored in agents memory the coordinates of where it originated is saved as well. The directionality is needed later for policy calculations.

Agents are homogenous, with a small connection radius around them. The world is width/height of 2, agents connection radius is 0.25. We are assuming that connections between agents are perfect, and that each agent can simultaneously respond to incoming packets and send outbound packets.

Agents have two partitions of memory, public and private. This is mainly a symbolic partition rather than an actual necessity. Private memory is data that has been learnt directly by said agent, and public memory is information that the agent has learnt from another agent. Both public and private memory cannot have duplicate data in themselves or across partitions. This partition is symbolic because it is not required, however useful to have. It allows us to not require memory management of which having the memory as one partition might require. An example of this in a one partition model, if an agent's memory is full with lots of public information and tries to learn a private data point, we would have to construct a way of dealing with these collisions. This is because of the priority difference between holding duplicate data compared to the gathered data, this is because the gathered data can only be learned once. With this partition we don't have to deal with this problem, however we do lose efficiency in how much information an agent can store. For the purpose of our testing and implementation, this efficiency is not a priority and is overlooked.

The control structure of the simulation runs 10,000 iterations. Each iteration

all agents are put into a parallel for loop to do its iteration, they are selected randomly in order to achieve the most real life-like simulation of individual agents that is possible for the author to use. The reason for not having the agents run in all different threads is because the scheduling policy has a tendency to prioritize a few agents over all other agents therefore over a certain time period one agent might have completed 500 steps and another just 1, this is unlikely in real life implementations. As a future improvement it would be good to have hardware that can support the number of agents all at the same time, compared to having to restrict the frequency because of the scheduling algorithm.

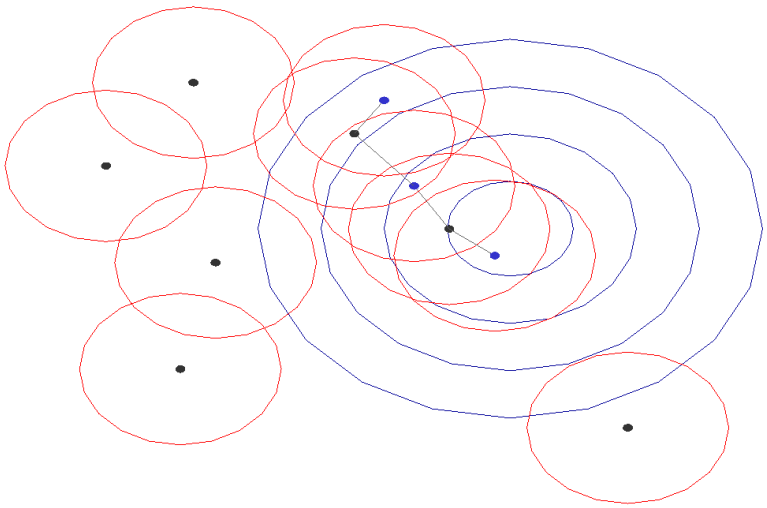

Figure 4.1: Example of simulation looks when running

In Figure 4.1 we can see an example of what the 2D simulation looks like. Dots are agents, and the red circle around them is their connection radius of which they can talk to other agents, these will be omitted from other images of the simulation to keep it cleaner and more understandable. The blue concentric circles are an example of the datas positional attribute, these will be omitted for a red circle in other images for the same reason as beforehand. The connection lines between the agents are just showing that data can pass between these two agents. These lines change colour based on what packets are being passed across them, this is for debug purposes and most likely won't be seen in any images. As a simple visualisation of how data is being passed between the agents, in this example the agent is blue signifying that it holds the duplicate of blue data, e.g. the data from the concentric circle data point.

## 4.2 Static Heursitic

This solution takes heavy inspiration from [10], we take the methodology of each agent controlling its own data and how it wants to distribute it. We do this using the actions of Replication, Suicide and Nothing. Migration is taken out due to the natural movements of the swarm, we don't want all data to be funneled back to the point, this is to deal with correlated errors.

We then use a heuristic to decide whether to do either three of these actions. In this case we have two heuristics based off these factors:

- Amount of agents in connection range that have a duplicated data compared to not having duplicated data
- Distance to datas target area
- Agents around average available public memory

We then used a weighted sum, to calculate the heuristic value. If the replication heuristic is above the replication threshold then we replicate and same for suicide, else do nothing. The weights of either action are set such that they sum to one, so they can be seen as what percentage of the output should this value account for. The reason for this algorithm being called static is that, mappings of where duplicated should be is deterministic and doesn't rely on any previous knowledge.

A pseudo code version of this can be seen in Algorithm 1. In this we can see how the agents work on an individual basis. Every step we check whether we have learnt any new private data, if so then duplicate that data onto all agents around. If this doesn't happen because of external factors like no other agents around, it will class that data has just been learnt in the next iteration. This is done to gain as much redundancy as quickly as possible, if a correlated failure happened early and we didn't do this, there is a high probability that all that data could be lost, so we transfer it to all agents and take the possible transfer energy cost losses.

We then move onto the factors mentioned in the bullet points above, in lines 8-10. For this iteration we focus on one data point in public memory, and in line 18 we switch to a new item in public memory. This means we are sequentially going through the public data and checking if we should do an action to it, over iterations equal to size of public memory stored. This is not good for scaling however for the size of data that we are using for testing it is addicaute, this will be talked about more in Section **??**. To gain the information needed for these three factors the agent broadcasts a packet to other agents in its vicinity. They will then respond by sharing

whether they have a duplicated version of the data in their public memory, and how full their memory is. To check whether they have duplicated data is a simple look if they have a data item with the same id in private or public memory. Once collated at the originating agent we can work out the specified values as shown in the code. There are improvements that can be made to the handling of broadcasting and replying to mean we aren't using as much power draw, however we will not go into this in the project and will be explained a bit more in the Section further improvements.

After getting the three factors we make sure they are on a scale from 0 to 1, and rearrange them to grow higher when we need to do a certain action. For example duplications go up when there is a higher density of duplications therefore on the replication heuristic we want it to be the other way around hence (1-dupes).

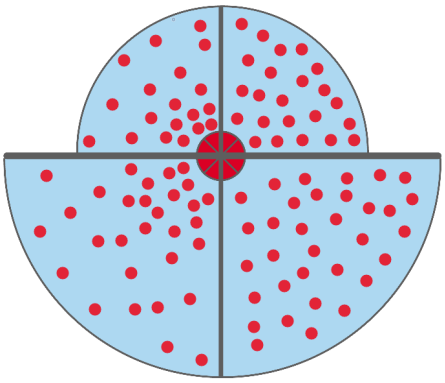

Figure 4.2: Example of changing of replication and suicide threshold on a uniform agent density

With changes to different weightings and thresholds, we get significantly different behaviours. An example of this is the increase of the replication threshold means that duplications will spread out less, avoid higher duplication density areas and not spread when agents are saturated in other data. The higher the suicide factor means there is less of a duplication density diffrence across the network.

For an easier understanding of the heuristics behaviour when changing the threshold values we can look at Figure 4.2. This is not a full representation of how the heuristic will work, this diagram shows what the duplication density would be like over a uniformly spaced swarm with exactly the same number of data in public memory. The smaller red dots are data duplicates originating from the datas area which is signified by the large red circle. When the replication threshold is increased in size, this can be seen as changing the range that agents can have duplicated data, as visualized with the blue circle getting smaller. Suicide threshold increasing, increases

the diffrence in density across distances from the data area, as visuallised from left to right being lower to higher.

Weightings can also be changed to fit the behaviour style of the heuristic, for example if you wanted to favour just distance information you could change values accordingly. Having this weighted sum and threshold lends the algorithm to being optimised with a genetic algorithm, with a fitness function dependent on the results that a user might want. This would have to be done for all different applications types because the difference in weights changes the behaviour. For our runs we use, b = 0.45, 0.45, 0.1 and p = 0.3, 0.7, b was picked arbitrary based off preliminary guesses of what would be best, and p was picked to make the average values duplications ratio and distance to data be roughly equally weighted.

Originally “suicide data” in Line 16 of Algorithm 1, was done using Paxos [15]. This was used to ensure that in a scenario where all of one data duplications are in a locality and could all be deleted at once, therefore having no duplications left of that data, wouldn’t occur. However from preliminary testing this didn’t make much of a difference in our scenario, but should be added into any practical applications of this algorithm for more guaranteed redundancy.

// Further improvements refrence

## 4.3 Dynamic Heuristic

## 4.4 Dynamic Heursitic with Migration

# 5 Analysis

## 5.1 Static Heursitic

## 5.2 Dynamic Heuristic

## 5.3 Dynamic Heursitic with Migration

# 6 Conclusion

# 7 Main

## 7.1 Simple Scattered Memory Policy

Taking on from the design of algorithm presented in [10], I applied a simple distributed storage policy onto a swarm that takes into account positional data. This policy has two actions that it can perform, suicide and replication, suicide is when a piece of data decides using a heuristic that it is not worth being stored in an agents memory so deletes itself, replication is where based on a heuristic the data believes it is worth spreading the infomation so will replicate to one member of the swarm.

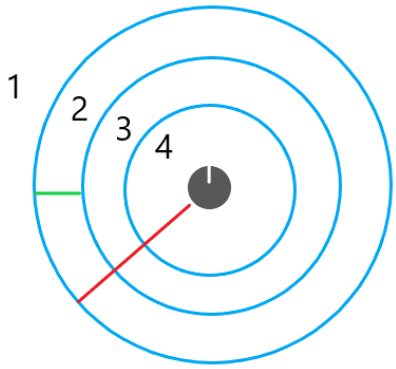


Figure 7.1: Poistional data heuristic for agents

The heuristic mentioned above follows these rules, and Figure 7.1 helps visuallise the conecept. At the grey agent in the middle is an example of when the data is learned, when learned this data will have a poistion that it was learned a radius at which you want the allowed duplications (including own duplicate) of that data to be one. The red line signifys that distance and the value in this example four is the maximum duplications of that data. The green line is just an easy way to compute which band an agent might be in.

As an agent moves closer through the gradient of allowed duplicates it will want to have more duplicates in its area, therefore allowing for a higher concentration of said duplicated data in that area. We can then use this

simple heuristic per item of data to be able to get the distribution that we want, one implementation of this is described below in Algorithm **??**. The reason for having a random replication when the allowed duplications equal the actual duplications is to try and breakthrough the boudarys at the edge where allowed dupes equals 1.

Without this replication we would have data sit inside the data area to much, this can be seen on some prilimany test results Figure **??** and Figure **??**. Blue line is amount of agents with that duplicated data. Purple line is the mean distance of agents with duplicated data from the data point, Green area is the standard deviation and yellow/orange is the data range. A guassian filter has been passed over the results due to the mild fluctions every iteration creating a harder to see results.

From the results shown in Figure **??** we can see that the distance from the area point likes to stay within side the data_radius bounds, this was tested on diffrent sizes of data radius and max duplications allowed and still found to hold true. We then tested the algorithm which has a random replication value to be able to try push through the borders of the data radius. We ran these on two diffrent values of replication value for five times on each, Figure **??** is with a random replication value of 10% when allowed duplicates = actual duplicates. Figure **??** is the same values as before however with a replication value of 50% to see how roughly this will effect the results.

We can see that having the replication value higher keeps data at a more consistent distance away from the data point rather than dealing with the fluctuations of the circluar movement of the agents. From personal observations of the algorithm running it was observed that the data transfers became more volatile with the higher replication number. Volatility meaning that data was replicated therefore making actual duplications to high then suiciding to get back down to the correct value, this would repeat often. In a real world soloution this would not be ideal due to data transfers costing energy and possible data loss, this means that we need to design an algorthim that can take this into account more.

When comapring Figure **??**, Figure **??**, Figure **??** and Figure **??** we can impliy that the algorithm change doesn't affect the max distances like our desired outcomes want. With this algorithm we do get the desired affect when dealing with non-correlated errors as shown in **??**. In this graph we can see as the number of agents is decreasing the number of replications trys to fight that change, however some improvements might want to be made when a more thourgh investigation is done. Correlated errors were not tested currently due to the failures in distance from the data point, we want a larger min max range of values before this is tested.

## 7.2 Improved Heuristic

To improve from the last design was to create a new heuristic that can take more into account and have scalable values, that can be modifyied to get diffrent behavoiur from the swarm network. This was done by identifying diffrent factors that we want to play a role in the choose to replicate and the chose to commit suicide. These were:

- Amount of agents in connection range that have a duplicated data compared to not having duplicated data
- Distance to datas target area
- Agents around average available public memory

The updated code can be seen in Algorithm 1, with this we can see that there are diffrent heurisitics for both suicide and replication. With the above factors we made sure that they were bounded [0, 1] by doing ratios of the max value. Then we had parameter values that sum to one as to show the percentage of what the parameter effects the heursitic, this can be seen in lines 12 and 15. After these have been summed together the value is bounded between [0, 1] which we then set a threshold value for at what point we want the code benith to activate, e.g. Replication or Suicide. Changing of these parameters creates diffrent charatersistics as an example can be shown with just replication and no suicide, Figure **??** and Figure **??**.

With this we can see that making the replication threshold higher decreses the likelhood of replication and gives a semi bound in a uniform density swarm and within non-uniform density swarm. In figures shown you will see there is usally a black agent close to the red cricle (Data target area), this is because the agent that learned the data has it stored in private memory so isn't counted as changing colour to show it is a duplicated piece of data in public memory. With both Figures we can see that suicide is definetly needed due to movement of agents as in Figure **??** and within Figure **??** we can see its needed due to all agents in an area around the point have the data. This goes against what this storage policy is meant to provide, we need a way to vary duplicate density within that semi-bound around the data target area.

To this we add in the suicide option which only takes into account the first to features meantion in the bullet points above. This will change the actions of the swarm to create a better policy as described in Section 1.2.

As we can see in Figure **??** adding in the suicide ability and changing the

threshold value creates diffrent duplication densitys within the replication bounds. In simple terms it changes the desnity of duplications closer to the data target area if the density of agents is uniform in desnity. When talking about non-uniform densitys it wil also take into account the desnity of agent duplications, however basically applies the same affect of having less duplicates where that is wanted, an example of this can be seen in Figure **??**.

With this current design it leads itself to diffrent observations based of parameter values. The right values need to be picked for the algorithm to work effectivaly, for example sometimes if suicide threshold goes to low it leads to massive instability with repeating duplicates and suicides which is wasted bandwidth and energy. This problem could be taken into account by also taking into account when the data was last replicated and allowing a sort of grace period within the heuristic to be able to allow natural movements to take those unstable agents apart. The heurisitc and threshold values lends itself towards a learning algorithm for example a gentic algorithm to pick the best value for the certain task or wanted behavoiur.

As a prelimanry test run on an average of five runs, Figure **??**, we can see that we have managed to create an algorithm that is more stable to the periods of the swarms movement, in our case circles. This was designed into the simulation to be able to test an algortihms abilitys to be able to handle fluctions in a repeating manner that can be visible. We can also see that the range from maximum and minimum distances has now been increased which was something we wanted in for improvement from Algortim **??**. However we can see that the mean value is roughly within the middle of the maximum and minimum value which leads me to believe that the algorithm isn't performing as expected. We prefurably want the mean to tend towards being closer to the minimum rather than in the middle, this is to show the desnsity diffrence of duplicated data. This will have to be investigated further as to whether it is the nature of the heuristic or whether it was the values of the parameters which were picked manually.

There is also an example of this algorithm running on a non-correlated errors in Figure **??**. With this we can see roughly that the duplicates stay at about 40% of the total number of agents even with non-correlated errors happening at a very extreme rate. An observation within the simulations is some failures of agents can affect the total value suggnificantly until it gets into a stable state. Due to the nature of the heuristic it is deterministic and likes to hang in an optimum as much as possible, therefore some loss of agents doesn't change much. However sometimes an important agent is removed the duplicate layout changes quite quickly changing the layout across the swarm in a ripple like effect, until it gets back to an optimium (Stable state).

# A Code apendix

**Algorithm 1** Agent's control loop

**procedure** STEP
    move()

    **if** Learned new private memory data **then**
        Replicate item to all agents in connection radius
        **return** True

    $dist \leftarrow$ euclidean distance to data
    $dupes \leftarrow$ (local) duplicates on agents / number of agents
    $pubspace \leftarrow$ (local) average space available / max public memory

    **if** $(1 - dupes) * b1 + ((\sqrt{8} - dist)/\sqrt{8}) * b2 + pubspace * b3 > repthreshold$ **then**
        Replicate item to all agents in connection radius

    **if** $dupes * p1 + (dist/\sqrt{8}) * p2 > suithreshold$ **then**
        Suicide data

    Iterate to next public memory data

    **return** True

# B Results apendix

# Bibliography


[1] J. C. Barca and Y. A. Sekercioglu, “Swarm robotics reviewed,” Robotica, vol. 31, no. 3, pp. 345–359, 2013.

[2] V. Kumar and F. Sahin, "Cognitive maps in swarm robots for the mine detection application," SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme - System Security and Assurance (Cat. No.03CH37483), Washington, DC, 2003, pp. 3364-3369 vol.4, doi: 10.1109/ICSMC.2003.1244409.

[3] H. Wang, D. Wang and S. Yang, “Triggered Memory-Based Swarm Optimization in Dynamic Environments,” in Applications of Evolutionary Computing, M. Giacobini, Ed. Berlin, Germany: Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 2007, pp. 637–646.

[4] D. A. Lima and G. M. B. Oliveira, "A probabilistic cellular automata ant memory model for a swarm of foraging robots," 2016 14th International Conference on Control, Automation, Robotics and Vision (ICARCV), Phuket, 2016, pp. 1-6, doi: 10.1109/ICARCV.2016.7838615.

[5] E. Bonabeau, M. Dorigo, and G. Theraulaz, Swarm Intelligence: From Natural to Artificial Systems. Cary, NC, USA: Oxford University Press, 1999.

[6] L. Xiang, Y. Xu, J. Lui, Q. Chang, Y. Pan, and R. Li, ‘A Hybrid Approach to Failed Disk Recovery Using RAID-6 Codes: Algorithms and Performance Evaluation’, Association for Computing Machinery, vol. 7, p. 11, 2011

[7] C. Mims, ‘Why CPUs Aren’t Getting Any Faster’, MIT Technology Review, 2010. [Online]. Available: https://www.technologyreview.com/2010/10/12/199966/why-cpus-arent-getting-any-faster/. [Accessed: 01-Dec-2020].

[8] U. Troppens, W. Müller-Friedt, R. Wolafka, R. Erkens, and N. Haustein, ‘Appendix A: Proof of Calculation of the Parity Block of RAID 4 and 5’, in Storage Networks Explained: Basics and Applic-

ation of Fibre Channel SAN, NAS, ISCSI, InfiniBand and FCoE, U. Troppens, Ed. Chichester: Wiley United Kingdom, 2009, pp. 535–536.

[9] J. Liu and H. Shen, "A Low-Cost Multi-failure Resilient Replication Scheme for High Data Availability in Cloud Storage," 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), Hyderabad, 2016, pp. 242-251, doi: 10.1109/HiPC.2016.036.

[10] N. Bonvin, T. G. Papaioannou, and K. Aberer, A Self-Organized, Fault-Tolerant and Scalable Replication Scheme for Cloud Storage. New York, NY, USA: Association of Computing Machinery, 2010.

[11] Legal Services Act. 2007.

[12] A. Prahlad, M. S. Muller, R. Kottomtharayil, S. Kavuri, P. Gokhale, and M. Vijayan, 'Cloud gateway system for managing data storage to cloud storage sites', 20100333116A1, 2010.

[13] B. Czejdo, K. Messa, T. Morzy, M. Morzy, and J. Czejdo, 'Data Warehouses with Dynamically Changing Schemas and Data Sources', in Proceedings of the 3rd International Economic Congress, Opportunieties of Change, Sopot, Poland, 2003, p. 10.

[14] 'Key-Value Scores Explained', HazelCast. [Online]. Available: https://hazelcast.com/glossary/key-value-store/. [Accessed: 02-Dec-2020].

[15] L. Lamport, 'The Part-Time Parliament', in Concurrency: The Works of Leslie Lamport, New York, NY, USA: Association of Computing Machinery, 2019, pp. 277–317.

[16] D. Agrawal and A. E. Abbadi. The tree quorum protocol: An efficient approach for managing replicated data. In VLDB'90: Proc. of the 16th International Conference on Very Large Data Bases, pages 243–254, Brisbane, Queensland,Australia, 1990.

[17] S. Lynn, 'RAID Levels Explained', PC Mag, 2014. [Online]. Available: https://uk.pcmag.com/storage/7917/raid-levels-explained. [Accessed: 06-Dec-2020].

[18] J. Hu et al., Eds., HiveMind: A Scalable and Serverless Coordination Control Platform for UAV Swarms. ArXiv, 2020.

[19] D. Calvaresi, A. Dubovitskaya, J. P. Calbimonte, K. Taveter, and M. Schumacher, Multi-Agent Systems and Blockchain: Results from a Systematic Literature Review. Cham, Switzerland: Springer Interna-

tional Publishing, 2018.

[20] L. A. Nguyen, T. L. Harman and C. Fairchild, "Swarmathon: A Swarm Robotics Experiment For Future Space Exploration," 2019 IEEE International Symposium on Measurement and Control in Robotics (ISMCR), Houston, TX, USA, 2019, pp. B1-3-1-B1-3-4, doi: 10.1109/ISMCR47492.2019.8955661.

[21] M. Y. Arafat and S. Moh, "Localization and Clustering Based on Swarm Intelligence in UAV Networks for Emergency Communications," in IEEE Internet of Things Journal, vol. 6, no. 5, pp. 8958-8976, Oct. 2019, doi: 10.1109/JIOT.2019.2925567.

[22] D. Jackson and F. Ratnieks, 'Communication in ants,'Current Biology,vol. 16, pp. 570–574, 2006.

[23] C. W. Reynolds, Flocks, Herds, and Schools: A Distributed Behavioral Model. ACM, 1987.