SharedState 2.0

M. Vonk

TU Delft

chess

SharedState 2.0

by

M. Vonk

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on 30 January 2020

Student number:	4178165	
Project duration:	December 3, 2018 – January 30, 2020	
Project supervision:	Prof.dr. K.G. Langendoen,	TU Delft, supervisor
	Drs. D. Faber,	Chess Wise B.V.
	Ing. S. de Vries,	Chess Wise B.V.
Thesis committee:	Prof.dr. K.G. Langendoen,	TU Delft, supervisor
	Dr.ir. J.A. Pouwelse,	TU Delft
	Ing. S. de Vries,	Chess Wise B.V.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Abstract

More and more devices are gaining some form of smartness either by sensing their environment on their own or by communication. An example of this is Smart Lighting, a field in which *Chess Wise* is active for over 6 years. By adding sensors and communication lighting can be controlled more efficiently. For example motion sensors to turn off lighting where no movement is detected, or a light level sensor to achieve the same light level throughout the day. This saves energy when natural light is available. Extending the wireless network with a gateway to the internet opens even more possibilities to control and monitor the wireless network.

The technology used by *Chess Wise*, called *MyMesh*, is a proprietary mesh network. One node is a small embedded device with limited resources. The *MyMesh* network is self-organizing. All nodes have the same function in the network. All nodes cooperate to get messages from node A to node B. Being self-organizing allows nodes to be added or removed at any time. By leveraging the communication between nodes their limited resources can be combined into a big shared resource.

The *MyMesh* network currently uses an protocol, *SharedState*, to combine the limited memory capacity (cache) of the nodes. During communication nodes exchange memory items, called Tokens. By moving these tokens around each node has access to the combined memory space of the whole network.

In this thesis a renewed algorithm is proposed that improves the overall storage capacity of the network. This renewed algorithm also scales better with the number of nodes compared to the current algorithm. *SharedState* used a probabilistic approach to transmit tokens and replace tokens in the cache. With the renewed algorithm per cache entry two additional fields (*age*, *rx*) are added. These fields are stored to calulate the informational value by Shannon[19].

The *age* field indicates how long ago the token was placed in the cache. The *rx* indicates how many times the token was received during that time. Using the *age*, *rx* and the average number of tokens received per round the informational value can be calculated. This informational value is higher for tokens that are received less often. Each round the cache is reordered saving the tokens with the highest informational value at the top. Tokens that do not fit in the local cache are discarded. These discarded tokens carry a low informational value.

To transmit tokens the renewed algorithm uses the cache position as the probability of ending up in the transmit message. The higher the position in the cache, and the higher the informational value, the more likely a token is to end up in the transmit message.

Using this new proposed algorithm the storage capacity now grows with the number of nodes. And more tokens are retained in smaller networks.

M. Vonk
Delft, Jan 2020

Acknowledgments

I've started my school career at HBO (similar to Bachelor). After graduating from HBO I was left with a feeling that I've not yet reached my learning potential. At that time my impression of University was that it was too theoretical. I thrived in the more practical side of things being a code-monkey. During my HBO I've already become acquainted with *Chess Wise* during my internship. During this internship, supervised by Frits van der Water, I've learned plenty and was given freedom to help out outside of my main internship goal. During that time I've met with Master and PhD students. And even got to help or assist. During this time I learned that University could still be practical, one just learns ways to motivate their decisions. *Chess Wise* motivated me to attend University for which I am very grateful.

Executing this thesis was no easy task for me. I've gotten ahead of myself a lot of times. Once a set of experiments was running in my head a new set of experiments or representations was already in place. That was not really handy since doing the experiments was just a small part of the work needed. The TU did teach me how to motivate my actions, applying them on my own is a different story.

It resulted in a nice end-result, but it has taken a lot of effort and time. I would like to thank *Chess Wise* for their time and accommodation of my thesis. During my thesis both Siebren and Diederik spent a lot of time in meetings with me to discuss results and keep me on the right track. I would like to thank them both for the meetings and valuable feedback they have provided.

I would like to thank the TU Delft, and especially, Koen Langendoen for their time and feedback. Even after the usual thesis duration Koen still found time to provide feedback or meet up either in person or via Skype.

And last but not least I would like to thank my parents for encouraging and supporting me to throughout my studies.

Contents

1 Introduction

This thesis is conducted at *Chess Wise*. *Chess Wise* has developed a wireless sensor network called *MyMesh* that is scalable, secure and energy-efficient. *Chess Wise* currently utilizes *MyMesh* to control lighting. Having all light sources, sensors and switches interconnected brings advantages. Being able to set the correct light level by sensing presence for each fixture at any time brings comfort and reduces energy consumption. This thesis focuses on a data dissemination algorithm that is part of *MyMesh*. The wireless nature brings in a lot of challenges while still having to solve problems that can also be found in wired distributed systems. A brief introduction to these subjects will be given in this chapter.

1.1. Distributed Systems

All around us devices become interconnected. This process started in the 1960s when a large expensive machine needed to be shared between users. For this to work multiple processes in the same system needed interconnection. Later devices became more affordable and network connections between devices were made. The network can be either local (LAN) or wide spread (WAN) and range from a hand-full to millions of devices in a large network like the internet. A single element of a distributed system is often referred to as a *node*. The nodes of a distributed system communicate with each other. By exchanging messages actions can be coordinated. This makes the whole distributed system act as one coherent system. One of the characteristics that are present in both small and large distributed systems is that these are dynamic: devices can enter and leave. Anyone can turn their system on or off but the internet is still there.

Having a similar notion of time in a system that consists of multiple autonomous systems is not easy any more. All devices have their own clock, without synchronization these clocks will drift apart. For example to monitor the traffic in a neighborhood two sensors are placed. The order in which the sensors are activated indicates which direction the object is traveling. Counting will be off when the order of events is reversed. Even more if the sensors are used for speed measurement the time must be equal at both devices to perform precise calculations.

1.2. Wireless Sensor Networks

Wireless Sensor Networks (WSN) are a type of distributed system in which multiple small, autonomous and energy-efficient devices communicate wirelessly. These devices usually monitor and control their environment; environmental measurements can for example be light, sound or humidity-level. These measurements are either aggregated in the network itself or transferred to another network through a node called the gateway. One form of aggregation could be to check if more than one node in an area notices a sudden high temperature increase, which could be an indication of fire. Because multiple nodes have the same observation the chance of a malfunction in a single device is eliminated. Another example could be to measure soil humidity. A farmer could create a large network of small battery or solar powered

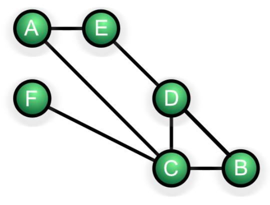


Figure 1.1: Mesh topology

nodes in his fields to report the soil humidity. This data collected through the gateway gives the farmer an indication of which parts of a field need more water.

The communication that exists between nodes and the gateway can be direct or relayed by other nodes creating a multiple hop path. By using multiple hops communication can be extended beyond the radio range of one node. If more than one hop is used for communication the nodes between the sender and receiver are used as relays. Between the sender and receiver multiple paths can exist. Usually some routing algorithm is used to determine the path with the least cost to relay communication between two nodes. If one of the relaying nodes stops working a new path must be established to restore communication. A small example network is shown in fig. 1.1. Each of the green circles corresponds to a node and a black line for the communication paths. If for instance node A and B want to communicate there are multiple paths; one consisting of two hops, through C, and one with three hops, trough D and E. So either C or D can fail whilst communication between A and B is not impeded. The shown network is a partially connected mesh; when all nodes have a direct connection to every other node it is called a fully connected mesh.

1.2.1. MyMesh

Chess Wise uses a proprietary WSN called *MyMesh*. Multiple types of actors and sensors are available for different use cases, most are developed for smart lighting. Some products are battery operated like sensors, whilst the light controllers are operated on mains.

As previously mentioned a distributed system such as a wireless sensor network has its challenges. The essential core of the *MyMesh* network, called MyriaCore, takes care of these challenges. For one the system maintains a global synchronized clock on which the network runs. By knowing when communication takes place the radio can be set to standby when other nodes are not transmitting to reduce power usage. Using periodic communication we can speak of rounds. In each round a node may transmit one message whilst receiving multiple messages from its neighbors. After a communication round ends the received messages are processed and a new message is prepared to transmit in the next round. The remainder of the time for the next round is spent sleeping, conserving energy.

The complete distributed system appears to the user as a single system. To support this system as a whole a distributed virtual machine is placed on top of the networking stack. This machine, called the Token Machine, uses Tokens as instructions. A Token has an identifier and one or multiple bytes of data. A message can consist of one or multiple tokens. This Token Machine makes it possible to run multiple applications on the MyMesh platform, which share the underlying radio link. A small figure representing the token machine can be seen in fig. 1.2. An application can show interest in a specific token identifier by registering a Token Handler. By distributing a single token the behaviour of the whole system can be changed. For example flicking a switch to put the whole building into night mode in which all interior lighting turns off. For this to happen each node in the network must receive this token. To spread tokens the token handlers can determine their own dissemination strategy.

When looking at the application on Node level it will only receive tokens from its neighbors within radio range. One of the strategies that an application can take is to forward all received tokens. This strategy is called Flooding. Flooding is a dissemination protocol in which a huge number of nodes can be reached within a short amount of time without knowing the underlying network structure. One of the challenges in flooding is to know when to stop forwarding a token. Otherwise each token will keep recirculating indefinitely. One way to limit flooding is to add a time-to-live counter to the token. Each time the token is forwarded the counter is decreased until it is zero, which indicates to stop forwarding the token. For this strategy the diameter of the network must be known. If for example the whole network from end to end has 20 intermediate hops the counter must start at least at 20 to transfer this token to all nodes. Once the counter has reached zero and flooding has stopped, nodes that join the network will not be able to receive the token. These

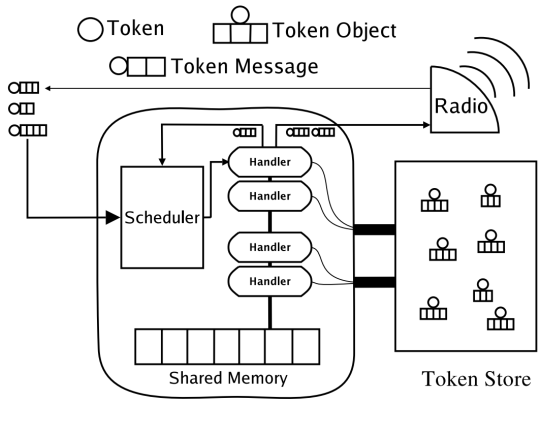


Figure 1.2: Token Machine[15]

nodes will thus not be informed.

The spreading of data between two nodes through flooding is costly. If we take fig. 1.1 as an example network, if we want to transmit a token from node A to C both E and D will repeat this token which is a waste of resources. In some cases data has to travel from or to a node. Node C can for instance be a node with more storage or a gateway to another network. To transfer data to a known point in the network it is more efficient to know the underlying network structure and route the data from A to C. In a dynamic system such as *MyMesh* this underlying structure is not something that is programmed in. The network must be able to determine this at run-time. This can be done using a routing strategy. And once the structure is known it must be updated periodically to allow nodes to enter and leave the network. The addition or removal of a node can add and remove paths between nodes. Once C leaves the network routing a token between A and B should go through E and D.

Flooding or routing a token from A to B are just two example strategies that an application can implement. The token machine allows to use different priorities and strategies to share the communication medium between applications. In this way an application moving large amounts of data, like a software update, can run in the background whilst other applications can still operate in a timely fashion. When the software update is distributed a single token with a different priority and strategy can switch the whole network to the new version at once. For light control *MyMesh* uses a proprietary dissemination strategy called *SharedState*.

1.2.2. SharedState

The *SharedState* algorithm designed by D. Gavidia [7] presents itself in *MyMesh* as a dissemination protocol. A node can distribute a token that travels throughout the whole network. By constantly moving the tokens around new nodes will also receive this token whilst they were not present in the network during the originating event. So besides being a dissemination strategy *SharedState* is also a storage system for the whole network. This gives way for a message passing pattern named Publish-Subscribe. In this pattern the Publisher does not keep track of all interested parties, called Subscribers. Instead usually a broker keeps track of the subscribers. A publisher instead publishes a message to a topic after which a broker distributes the message to the subscribed parties. With *SharedState* no broker is needed. Since the tokens travel around, each token will pass by each node in time. *SharedState* can be seen as a *Distributed State-full Publish Subscribe System* for the *MyMesh* network.

The *SharedState* algorithm is an adaptation of a *shuffle* operation. This can best be explained by example: shuffling a deck of playing-cards by a group. Each member starts out with a subset of the cards. Then all members search another member to exchange cards with, they give a subset of cards away and in return receive the same number of cards from the other member. Applied to the MyMesh network, cards are represented by tokens and communication is different. We do not exchange cards but let our neighbours know which cards we want to exchange, communication becomes one-to-many. So when a token is sent multiple receivers receive the same token, essentially creating a number of copies of the token. Copying tokens leads to the number of tokens growing each round. An advantage of creating copies is that new members without tokens will also receive tokens and join in on the shuffling. Since memory is finite at some point a node has to discard tokens when this capacity is reached. A cache-replacement strategy must be in place to determine which tokens to replace. All nodes now have to check if a token that they receive is a duplicate of what is already stored at the node. When a node receives a copy of a token that is already in its storage this is an indication that other parties also have this token. So when a node has reached its storage capacity these tokens are less favorable to store than other tokens.

Copying brings in another advantage: redundancy. Nodes can now leave the network without the network losing the tokens that are stored at the leaving node.

1.3. Gossiping

The concept of gossiping has been around for a while. In 1987 Demers et al. published a paper on ”Epidemic algorithms for replicated database maintainance”[6]. An epidemic algorithm, synonym for a gossip protocol [13], uses gossip to exchange information.

The terminology used in epidemic algorithms is inspired by spreading a virus. Each member can be in one of three states:

- *Susceptible* (S): the node is unaffected

- *Infected* (I): the node is affected and participates in spreading
- *Removed* (R): the node is affected but stopped spreading.

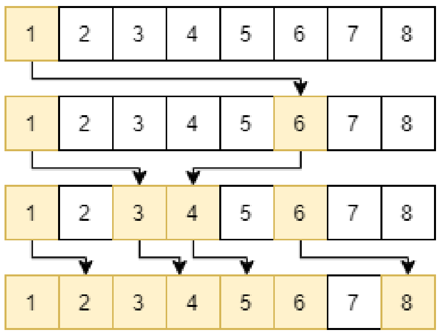


Figure 1.3: Gossip Example (White = Susceptible, Yellow = Infected)

The goal is to reach a pandemic. The susceptible nodes are uninfected. The infected nodes try to infect as much susceptible nodes as possible. Once a susceptible node becomes infected it helps spreading the virus. A removed node is not contagious any more.

In gossiping communication exists in rounds. Each round a node pairs up with another node to exchange information, depicted in rows in fig. 1.3. If a node receives gossip it deems worthy of sharing it will convey the gossip the next round when paired up with a different node. The first round only one node knows the gossip (Node 1), the second round two and the third round four. If each round all infected nodes select an uninfected node the number of infected nodes doubles. In fig. 1.3 the last row show that node 4 receives gossip from node 3 whilst already being infected. Therefore node 7 remains uninfected after 3 rounds. One example of such an algorithm previously mentioned in Section 1.2.1 is Flooding.

Demers et al.[6] split the algorithms into two types.

Simple epidemics Simple epidemics, also called *anti-entropy*, are aimed at the reduction of total disorder in a network. Any node can either be *Susceptible* or *Infected*. Each of the nodes have a single value and timestamp. The goal is to have the latest version at all nodes. Three types of algorithms are proposed, *push*, *pull* and *push-pull*. In the push variant each node pushes their content to the selected peer. If the information in the message is more recent the peer updates its value. In the pull variant each node sends a message with their current timestamp. If the peer has a more recent version it will send a reply containing its newer version and timestamp. In the push-pull these two are combined. If the pushed version is older than the copy at the peer the peer will send back its more recent version.

Complex epidemics In complex epidemics, also called *rumor mongering*, a new state is introduced: *Removed*. In the removed state the node will no longer actively publish their value. This is done to reduce traffic. To become removed two types of algorithms are proposed, *blind-coin* and *counter-feedback*. In the coin variant an infected node becomes removed with a probability of $\frac{1}{k}$. k is a value chosen between 1 and infinity. In the feedback variant the node will become removed once n nodes replied, acknowledging the transmission.

1.4. Emergent Behavior

In complex systems such as self-organizing distributed systems, the overall behavior does not rely solely on its individual components of the system but also on the interaction between them. By adding them together properties arise/emerge that were not a part of the single component.

Many examples can be found in nature, like the behavior of a swarm of birds. Researchers created simple mathematical models based on the behavior of birds, which contain just three rules: Move in the same heading as your neighbors, keep close to your neighbors and do not crash into your neighbors. These rules do not seem that powerful on their own, but by combining many birds with this set of rules into a flying swarm does result in complex patterns.

The disadvantage of these emerging properties is their unpredictability. A complex system will usually converge into a specific steady state based on a set of input parameters. It is impossible to determine which of the start conditions must be changed to land on a steady state that is more favorable. One of the strategies in designing algorithms for complex systems, called a ”genetic” approach, is to start with a simple version and improve this by small tweaks.

1.5. Problem Statement

MyMesh currently uses a probabilistic approach, *SharedState*[7], to create a distributed token-store. This token-store allows the network to store more elements than a single node can store, whilst nodes receive all tokens stored in the network over time. These two elements of behavior must be retained.

To make the number of tokens stored measurable we take n for the number of nodes, C_s for the token storage capacity of a node and a replication factor r. We say that $t = \frac{C_s n}{r}$ denotes the storage capacity of unique tokens in the network. Adding nodes(n) or storage (C_s) should lead to a higher t for scalability to work. To allow node failure the replication factor(r) should be at least 2 to allow for a single node to fail whilst retaining all elements in the network.

Various tests at *Chess Wise* have shown that the *SharedState* algorithm does not scale with the network size. Experiments with 100 nodes that retain 40 tokens each could lead to a theoretical maximum of $t = \frac{100*40}{2} = 2000$ unique tokens. In reality only around 100 to 150 tokens can be stored in the network. This is an order of magnitude lower. Adding more nodes does not increase the total storage capacity.

Chess Wise currently uses *MyMesh* to control lighting in public and commercial applications. A building consists of multiple rooms which all have multiple light-fixtures. The light-fixtures in a room can be grouped since they have the same desired light level. If for all rooms the desired light level would be stored in Shared State there could only be around 150 rooms. Controlling lighting in a building such as EWI building at TU-Delft with 23 floors could be problematic using *SharedState*.

The network wide storage capacity of the current SharedState implementation does not scale with the number of nodes in the network. What causes this unexpected upper bound and can we improve or change the SharedState algorithm to maximize the storage capacity closer to $\frac{C_s n}{2}$*?*

1.6. Structure

Being able to test and verify the differences to the *SharedState* implementation a repeatable set of experiments must be conducted. This setup and reference measurements are explained in chapter 2. After getting acquainted with the *MyMesh* network in chapter 3 related sources will be discussed in connection to *MyMesh* and *SharedState*. In chapter 4 multiple algorithms are tested and data filtering is added. In chapter 5 a conclusion is given and future work is suggested.

2 Analysis of SharedState

Distributed wireless systems such as a *WSN* are not easily measured. Due to the distributed nature one cannot determine exactly what is sent and received by each individual node by observing the radio communication alone. Messages can collide in the air. For each experiment all incoming and outgoing messages must be stored locally and gathered afterwards to be analyzed. To compare *SharedState* with other algorithms a set of specific experiments is created to improve repeatability and comparability. This chapter motivates which and why this set of experiments is chosen as a test bench.

2.1. Experiments

The goal of these experiments is to measure the number of tokens that can be stored in the network. Using four parameters which will be described in this section multiple experiments will be conducted. Each of the parameters has to be tested individually. We can only change one parameter at a time to check the influence this parameter has on the total storage capacity.

In the previously mentioned storage capacity formula $t = \frac{C_s n}{r}$ there are already two factors (n, C_s) that could impact the total network storage capacity. This will be our first two parameters, *Number of nodes* n and *Cache Size* C_s.

In the current configuration *Chess Wise* uses a *Cache Size* of 40 tokens. Adding more storage per node should increase the distributed storage capacity. Otherwise the added resource usage is not justified. Perhaps this is already in effect so testing a cache with fewer items is also advisable.

The second parameter *Number of nodes* can be described as just adding or removing nodes. However once nodes are added or removed a couple of effects take place. If extra nodes are placed between existing nodes the neighborhood of these nodes grows. During experimentation this will add an external influence. A communication round in *MyMesh* has a defined set of slots to communicate. If there are less receive slots than nodes in a given radio range the network starts scheduling. With this scheduling communication changes (this will further be explained in Section 3.1.1). This scheduling effect should be considered while testing. Therefore a third parameter *Density* is added which indicates the network density.

When scaling the $2xN$ topology using the *Density* parameter the network becomes more dense or spread out, see fig. 2.1. In fig. 2.1 a cross-section of the $2xN$ topology is shown. The top-center node 0 can reach all nodes that have a darker border (-7 to 7 for Compact, -4 to 4 for Normal and -2 to 2 for Wide). Note that using normal scaling node 0 can reach -4 and +4 of the same row but not of the row below. So node 0 has a neighbour count of: 29, 15 and 9 for respectively Compact, Normal and Wide.

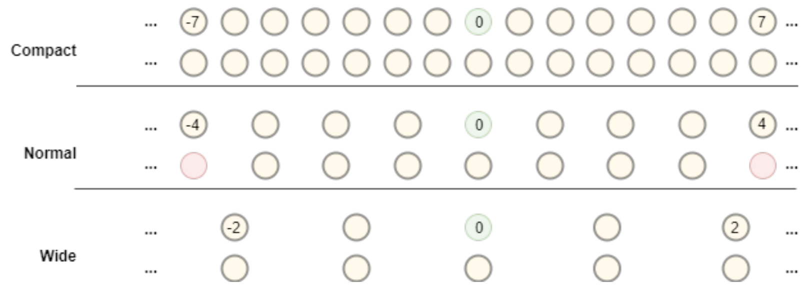


Figure 2.1: Testing Topology Neighbour graph

To minimize the effect of a changing topology when adding nodes a $2xN$ topology is chosen. In this topology there are two rows of nodes. By adding nodes a network is created with more hops but most nodes are unaffected by this change (only far left or far right gain extra neighbours). This topology bears similarity to a corridor in a building or roadside lighting. For these experiments a network size of 24, 50 and 100 nodes are used. If instead a NxN network is used with these three sets with of nodes (24, 50 and 100) a network of $5x5$ to $10x10$ can be created. Using $2xN$ we get $2x12$ to $2x50$ which allows more hops in the network.

When creating an empty test application there are no additional token handlers registered in the Token-Machine (for more info see Section 1.2.1). When no limit is added the test application could insert up to 20 tokens per round into the outgoing message. In a real-life application this is not possible, other token-handlers will also insert tokens in the outgoing message. Using the fourth parameter *Througput* the number of tokens that are sent out each round can be altered. A constant low (5) and high number (20) of tokens per message shows the difference between worst and best case.

Number of Nodes	24, 50, 100
Storage Size	20, 40, 60, 80
Density	Compact, Normal, Wide
Throughput	5, 10, 15, 20

Table 2.1: Test parameters

To check the influence of each parameter only one can be changed per experiment. With just these 4 parameters with 3 or 4 values (see table 2.1) a total set of 144 experiments is needed for each single parameter to be tested individually.

To do these 144 experiments in a consistent method one node will be selected as the seeder and deliver an impulse to the network. The steady-state response of the system will be taken as the storage capacity of the network. The conditions for the seeder node cannot change between experiments. Therefore the seeder node is in the bottom left of the $2xN$ topology such that its neighbours will be the same between experiments of the same density. This seeder node will wait till the autonomous network is created and then delivers an impulse to the network by sending out 500 unique tokens. 500 is chosen as a nice goal for the current setup and can be altered if needed later on in the experiments. Since only a certain amount of tokens can be transmitted each round the spreader starts out with its cache full of unique tokens and adds a new token each 5 rounds. After the distribution is finished a specified interval is waited before the experiment is stopped. The number of tokens left in the network after the experiment is finished will be used as the number of tokens that can be stored in the network.

2.2. Experimentation

For the analysis of *SharedState* multiple experiments are conducted. Each of these experiments must be repeatable such that future analysis and comparison on different algorithms can be made. At *Chess Wise* multiple options are available, both physical and simulated networks. A number of networks is dedicated for experiments that scale from a small network on one's desk to a board with 100 nodes to a +/-300 node network that spans the whole office floor. These networks can be monitored by so called "Sniffers" that monitor the messages sent through the air.

To simulate *MyMesh* networks a simulator called MyriaSim has been developed at *Chess Wise* to simulate the behavior of the nodes. The software developed for the nodes has a Hardware Abstraction

Layer (HAL) that is used to port the *MyMesh* to different target hardware. MyriaSim utilizes this HAL to execute the code on a PC and emulate the hardware events.

The main simulation program is a Discrete-event Simulator. The simulator uses a queue of events that need to occur at specific timestamp. When an event is processed the ”clock” jumps to the next timestamp to process the following event. In this way no processing power is lost by simulating time in which nothing occurs. The simulator has a built-in radio model that emulates delivery and radio range. When two nodes transmit at the same time the radio-model detects this as a collision and drops the messages.

2.2.1. Physical vs. Simulation

Using real-life tests to check the performance of *SharedState* is cumbersome. For each experiment the binary and parameters must be programmed to all nodes in the network. The experiment must be conducted and the data gathered for all nodes. Whilst wireless software distribution is implemented in the *MyMesh* network it would be a tedious task to gather all the data. There is also no control of the environment that could affect the experiments. The simulator has a clear advantage here, the experiments can be executed in a controlled environment and the data is stored on the computer that runs the simulator. A second advantage is the separate communication links. In the air messages physically collide and as such are lost. In the simulator these collisions are simulated. The other nodes will not receive the collided messages, but they can be logged to determine what each node sent out.

Therefore most of the testing will be done in *MyriaSim*. Information can be logged from each individual node to determine its cache content and which tokens it received and sent each round. Upon this information extended graphs and analysis can be run to optimize the algorithms. Simplified graphs are useful later on to verify the algorithm in a real-life network. These graphs can only be based upon ”Sniffing” data.

2.3. SharedState

In the dissertation of D. Gavidia the pseudo code for *SharedState* is listed, see appendix B.1. From this pseudo-code an implementation has been made for the *MyMesh* network. Over time some adjustments were made. One adjustment is the addition of ”resident” tokens. These tokens cannot be discarded from the cache. For these experiments and thesis no resident tokens are defined. A resident token will always be stored on at least one node and clouds the total storage capacity of the protocol.

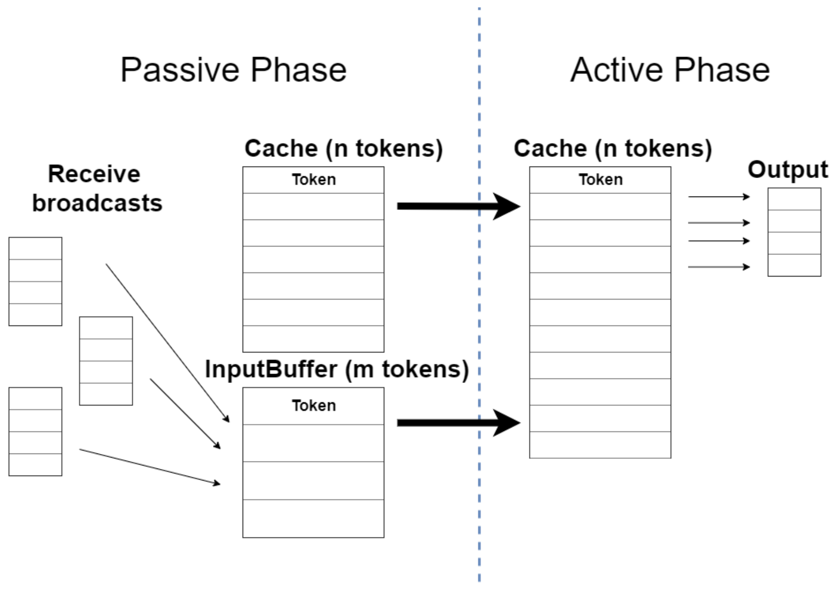


Figure 2.2: SharedState Buffers

The *SharedState* algorithm knows two phases, Passive and Active, and multiple buffers (see fig. 2.2). The InputBuffer is used to collect all incoming tokens in a round, the cache is the local storage and the output buffer is the collection of tokens that are sent out next round.

In the Passive phase received tokens are stored in an input-buffer and de-duplicated. The Active phase moves the received tokens over to the cache and tokens are put into the output-buffer.

One difference between the pseudo code and the current version is the addition of a Mark-Free flag to each of the cache items. Before the tokens in the input-buffer are copied to the cache a cleanup action is done. For each token of the input-buffer that is already in the cache the Mark-Free flag is set at the corresponding cache entry and the token is removed from the input buffer. When in the active phase the tokens from the input are moved to the cache there is potentially more free space to store incoming tokens. Tokens stored in the cache that have the Mark-Free flag can be overwritten without moving them to the output-buffer.

This feature can be reverted by setting a flag in the code. To get a feeling for comparing algorithms this part shows both the ”Frits” and ”Daniela” variant, indicating the version of *Chess Wise* and the dissertation version respectively.

2.4. Results

Running the set of 144 experiments leads to a large amount of data points which result in a cluttered unreadable graph. Therefore in this section the steady-state response of the experiments are represented as a bar graph. These numbers are comparable with real-life experiments done at *Chess Wise*. For the rest of this thesis it is more interesting to see the development of tokens in the network over time. For this two topologies are chosen with single and multi hop communication.

2.4.1. Full data-set

In the first comparison the current implementation is tested with and without the Mark-Free flag in the simulator. These results can be found in table A.1 and table A.2 in the appendix. Overall these two tables show roughly the same amount of tokens stored in the network. The Normal density fares better without the Mark-free flag whilst Compact and Wide are in favor of the addition of the flag by Frits. The overall difference is -3.2% to 1.2%. Since *Chess Wise* currently uses the extension added by Frits this data-set is visually represented in fig. 2.3.

In this figure each bar represents the number of tokens left in the system at the end of the experiment. From left to right the three network sizes are displayed with the individual cache sizes in different groups. Each of these groups is separated by an empty bar. So ”2x12: 20” shows the 2x12 topology with 24 nodes and a cache size of 20. In this group there are 12 bars that represent the topology density and throughput. Each bar being a combination of throughput and network density, ”C5” represents the transmission of 5 tokens each round in a compact network.

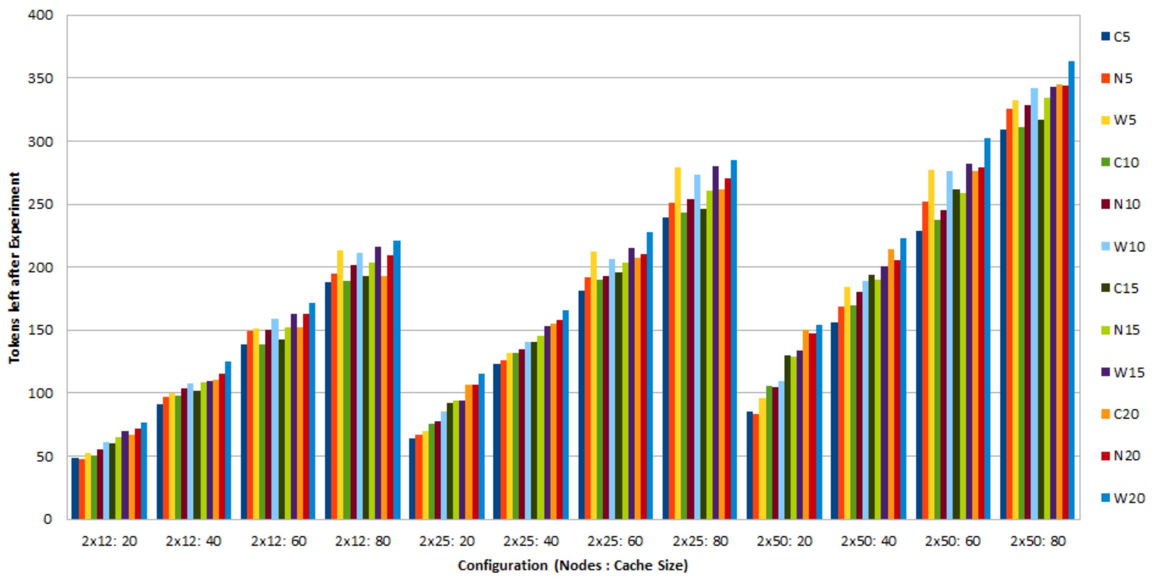
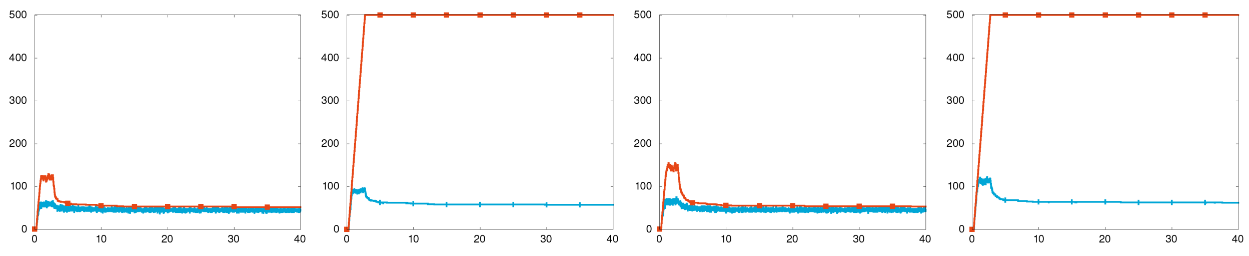


Figure 2.3: Full test measurement

These are arranged in such a way that the number of nodes or cache size increases from left to right. Adding nodes or increasing the cache size leads to a larger combined token-space ($C_s * N$ is the total token space). In fig. 2.3 adding either nodes or increasing the cache size does lead to more storage. Within the three groups an ascending line can be seen. It can however be seen that the replication does not stay the same. Looking at the left and right-most groups of bars in fig. 2.3 the node count goes from 24 to 100 and the cache size from 20 to 80. Combined a factor of nearly 16 ($\frac{100}{24} * \frac{80}{20}$). The total storage capacity of the network goes from +/- 50 to +/- 300. Only a factor of 6. In appendix A.1 the numbers used for fig. 2.3 can be found. In these tests *SharedState* has a replication factor between 6 and 22.

2.4.2. Future comparison



(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 2.4: SharedState results (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

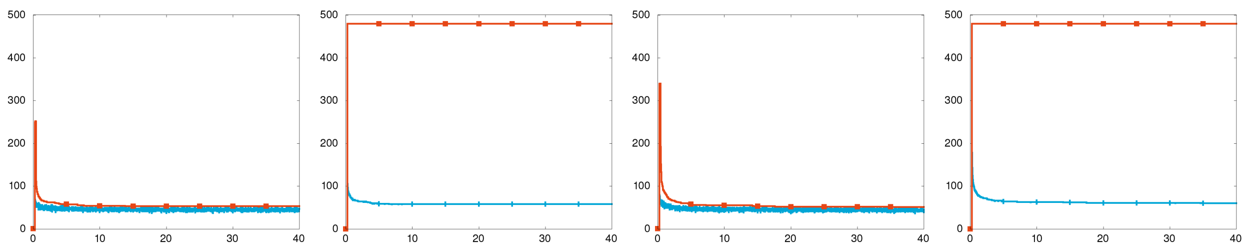
Figure 2.3 shows just the steady-state results. The seeder inserted 500 tokens into the network during an experiment, at what time these tokens were lost cannot be seen. To explain why these tokens got lost and to make the graphs comparable between algorithms a set of images over time is made. These experiments were made using two smaller networks composed of twelve nodes. In one network all nodes can reach any other node (single hop network, full-mesh). The second network is more spread out to create a multi hop network. The two remaining test parameters, cache size and throughput, are set to 40 and 5 respectively. These settings are comparable to a real-life network. The twelve nodes in the network cannot store the whole impulse injected by the seeder. With a replication factor of 2 and the used settings a storage capacity of $\frac{40*12}{2} = 240$ could be reached.

The results of these two experiments are shown in fig. 2.4. These figures show the number of tokens on the vertical axis and the time in rounds on the horizontal axis. These four figures have the same axis throughout this document such that a comparison can be made with future algorithms. The y-axis is set to 500 such that the complete impulse can be seen.

For each experiment there are two images; tokens in the air and stored. The tokens in air shows the number of unique tokens which go through the air and can be picked up by the sniffer. The tokens stored shows the number of unique tokens that are stored in the combined cache of the nodes. This is done to show how many tokens are stored in the system and how many are exchanged between nodes. If for example the storage graph indicates that there are 300 tokens stored and there are just 10 tokens in the air we do have storage, but the exchange of information is low.

The in the air graph is made using the message output of the simulator. Using this a comparison could be made to a real-life network with sniffers. The orange line (with square points) in the air plots shows the number of unique tokens over the last 10,000 sent tokens. If the network kept all the tokens that are fed in by the spreader node the orange line should reach the 500. The blue line indicates the number of unique tokens over the last 100 received tokens. This is a measure of the distribution of the tokens. If only one token was sent 100 times the blue line would be at 1, if 100 tokens were sent 1 time the line would be at 100.

In fig. 2.4 it is shown that just shy of 100 tokens are stored while the seeder is adding tokens (indicated in the storage graph with the orange line with squares). The network where the nodes are more spread out performs better than the single hop.



(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 2.5: SharedState results (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

To check if the total storage capacity is limited by the seeder a second experiment is conducted. At time 250 each node fills its own cache with unique tokens. This should result into 480 unique tokens throughout the network. In the graph fig. 2.5{b,d} we can see that indeed that at time 250 the 480 tokens are injected in the network (the orange squared graph goes from 0 to 480 at time 250). The blue line also indicates 480 at time 250. After a couple of rounds this drops back to around 80, the same when the spreader is used.

2.4.3. Discussion

The default setting in which *Chess Wise* conducted experiments has a cache size of 40 and the message is shared with other Applications that are running on the nodes. The *SharedState* dissemination algorithm can then only use around 5 to 10 tokens per outgoing message. During experiments conducted previously by *Chess Wise* a storage capacity of about 150 tokens was reached. Looking in either Figure 2.3 or the tables in appendix A.1 it is shown that the numbers of tokens stored in the network range from 91 to 189 tokens.

This shows that the real life testing is not that different from experimentation in the simulator.

From the second set of graphs (fig. 2.4) can be seen that the algorithm cannot process the 500 unique tokens. In the air plots (fig. 2.4{a,c}) the orange line (with square points) should reach the injected 500 tokens, instead it maxes out at 120 and declines to the number of tokens that are kept in the network (which can be seen in the stored plots (fig. 2.4{b,d}) indicated by the blue line). Good news is that the orange line is close to the number of tokens that are stored in the network, the orange line of the air plots lays close to the blue line of the corresponding stored graph. This shows that the tokens that are saved are also exchanged.

So it is not all bad, the network accepts a subset of tokens and as long as new tokens are fed in the capacity is somewhat higher than in the steady state. The effect between one- and multi hop is also visible, the multi-hop network outperforms the single hop network. Due to the random cache replacement there is no preference for any token, incoming tokens will replace information that could potentially be only stored at this node in the network. By having all nodes in a single hop they act upon the same information and their caches will converge to the same tokens. Due to the probabilistic nature this will always eventually happen. By adding nodes the probability will decrease.

3 Related Work

This chapter describes work related to this thesis. It contains two parts, in the first part the background of *MyMesh* and in the second part algorithms for the networking nature of MyMesh will be discussed.

3.1. MyMesh

The MyMesh network is somewhat different from other wireless sensor networks. It relies on broadcast-only traffic, a node sends out a message and each node in its radio range receives this message. This without acknowledging the reception of the message; a node does not know if anyone received this message, but more than one could have heard the message. This is somewhat different than common mesh networks like Z-Wave, which use point-to-point traffic. To allow point-to-point communication an overlay network is created to determine routes between nodes. To support an overlay network a routing protocol is used to create this overlay and it is to be updated when nodes enter or leave.

MyMesh does not create an overlay network or support routing. Each node has the same role and software to determine whether a message must be re-broadcasted; One of the easier ways has already been shown by using a TTL for flooding (Section 1.2.1). Applications on the *MyMesh* network can however implement their own routing algorithm.

3.1.1. gMAC

To support the broadcast-only nature of *MyMesh* a medium access control layer (MAC) called gMAC is used. The ”g” in gMAC stands for Gossip, the full name is ”Gossip based Medium Access Control”. To share the medium (radio communication) between nodes the gMAC provides synchronization as well as medium access.

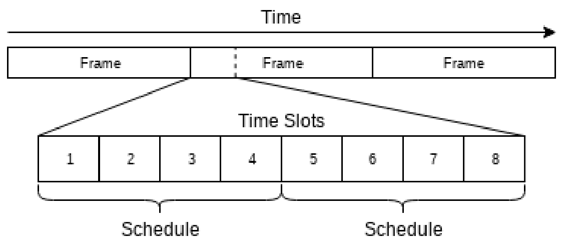


Figure 3.1: gMac Scheduling

The gMAC works in frames that have a specified duration. On a network level a frame is called a round. Using synchronization each node begins each frame at the same time. The start of a frame is used for communication after which the nodes have time to process the received messages and prepare a transmit message for the next round (active period). When a node is not communicating or processing the node is sleeping to conserve energy (idle period).

gMAC utilizes Time Division Multiple Access (TDMA) to allow more than one user to use the same medium. TDMA divides time into slots. So each frame is split up into slots, which are grouped into a schedule (see fig. 3.1). During a frame each node can use at most 1 slot to transmit its message, and listen to one schedule.

By default one slot schedule is used, as long as there are fewer nodes than slots this is ok. Each node can transmit in one slot and listen to the other slots in the schedule. When there are more nodes than slots within radio range the gMAC starts scheduling. Scheduling is done by transmission and receive scheduling. In transmission scheduling a node only transmits once in x frames. So a node starts skipping transmission of its own message to allow other nodes to transmit. A second scheduling strategy is to add an additional slot schedule to increase capacity. If a slot schedule is added a node will still listen to just one of the schedules (receive scheduling) and transmit in one of the available slots. The next frame the node will listen during the next schedule. Both strategies are used together.

gMAC can autonomously create a network. If a node is powered on it will initialize its peripherals and the gMAC starts listening to determine if there is already a network present. If there is a network present it will synchronize itself to the network and join it. When no messages are received this node starts its own network.

If multiple nodes that are not in the same radio range are started at the same time they cannot hear each other and will create two independent networks. When these nodes are brought closer to each other the networks should be merged into one network to allow communication between nodes. These networks A and B can operate in each others idle period, A can communicate while B is not listening. Since the goal is to create one network these networks must be joined. To aid this process gMAC sends out a Join message in the idle period. This join message is sent out at a random time in the idle period. By doing this at random this join message of network A will eventually land into an active slot of network B or vice versa. Based upon the network age of A it will either join the other network or wait till B joins. An application on *MyMesh* can insert data into the join message, which can be received by external sniffers, for development or debugging purposes.

3.2. Gossiping Algorithms

The focus of this thesis is data-dissemination. Gossiping can however be used for different applications as well. Three commonly used fields are: Membership management, Data aggregation and Overlay networking. Systems in which gossip algorithms are used are commonly large-scale and node failure is unpredictable. Therefore the usage of gossip protocols is rather limited and commercial implementations are hard to find.

Amazon S3 built on Dynamo is one known implementation in literature [5]. This object storage service ticks the box of being large-scale, in 2013 it stored one-trillion objects[3]. Having a gossiping algorithm also comes with some caveats. In 2008 a data-corruption lead to an outage of the S3 platform [1]. The resolution was to shutdown all servers and boot them up again. This event gives an indication of how successful but also how aggressive a gossip algorithm can be. Only a full reboot restored communication.

Examples of gossiping algorithms in different fields are:

- **T-Man[8]** Gossip-based Overlay Topology Management
- **Cyclon[22]** inexpensive membership management for unstructured p2p overlays
- **AVCOL[14]** Availability-aware information aggregation in large distributed systems under uncollaborative behavior.

VICINITY[21] by Spyros Voulgaris and Maarten van Steen is a paper that shows the influence of randomness in an algorithm. The VICINITY algorithms tries to create an overlay network using gossiping. This algorithm uses a view of neighbours. This view can be filled at random or deterministic. They experiment with multiple variants: full random, full deterministic or a mix of both. The results show that having either full random or full deterministic does not give the best result. Adding either a ”pinch” of randomness or determinism helps the performance of the algorithm.

3.3. Shared State in Components

The Shared State algorithm is more than a data dissemination protocol. It aims on an even distribution of tokens throughout the network and makes all items available for all nodes through a push-style algorithm. The *SharedState* algorithm can be seen as two parts: Distribution and Maintainance. During distribution a token should travel as fast as possible throughout the network (new elements are important). In the Maintainance part the token is still distributed such that nodes that do not have this token in their cache still have access to the token. The transition of a token from distribution to maintenance should be automatic. In reference to Demers et al. [6] this algorithm falls in the category of Push-based simple epidemics (see Section 1.3).

3.3.1. Different Approaches

During research the following algorithms were found that have some resemblance to the target application of *SharedState*. Each algorithm will be summarized after which the applicability to the two parts (Distribution and Maintainance) of the *SharedState* algorithm will be discussed.

Flooding Optimalization

Flooding is a more common algorithm in broadcast networking. Many studies have tried to optimize the number of messages sent by flooding. A couple of examples are:

Probabilistic Flooding [18] In probabilistic flooding a node uses a factor p to determine if it rebroadcasts (and thus forwards) the message. This corresponds with the complex epidemics with blind-coin described in Section 1.3. A number k can be used to allow one in k messages to be forwarded: $p_{forward} = \frac{1}{k}$. If $k = 1, p_{forward} = 1$ it is basic flooding and all messages are rebroadcasted by each node. If $p_{forward}$ goes to zero total coverage can no longer be guaranteed. The flooding can starve before reaching all nodes. In a dense network $p_{forward}$ can be set to 0.65 whilst still reaching 90% of the nodes[18]. In a paper by Crisostomo et al.[4] a mathematical approach for random networks is used to determine a $p_{forward}$ based upon the number of nodes n and the probability of connection between two nodes p_e. This research can be used as a starting point when implementing a probablisitic flooding algorithm.

Counter-based Flooding,[12] A different approach is to use feedback to stop flooding, a number k is used to determine when to stop rebroadcasting. This corresponds with the complex epidemics with counter-feedback using push pull. A node will start broadcasting once a gossip is received. Once the gossip is heard more than k times the node stops the rebroadcasting. An implementation based upon this idea is Trickle[9]. When a node receives a message it is temporarily placed on hold. The Hold time is chosen at random by each node. If within the hold time the node hears the same message two times it will discard this message without rebroadcasting. Otherwise the message will be rebroadcasted upon expiration of the hold time.

Location-based Flooding,[17] If for each node a location is known a node can determine if a message is to be rebroadcasted. When in a rectangular network the left-most node broadcasts a message this message can only travel to the right. For any node in between it makes no sense to rebroadcast a message when this is received from the node on its right side.

Of these three types of optimizations Location-based is the most efficient[16]. If no location information is available counter-based is preferred over probabilistic flooding. In probabilistic flooding the $p_{forward}$ is based upon the number of nodes and how well the network is connected. If $p_{forward}$ is too small the gossip may not reach all nodes.

```
% Blind/coin variant
on update($v$)
    $state \leftarrow$ INFECTED
    $value \leftarrow v$
    set timeout $\Delta$

on timeout
    if $state$ = INFECTED then
        $q \leftarrow$ random($P$)
        send ⟨PUSH, $value$⟩ to $q$
        if tossCoin($1/k$) then
            $state \leftarrow$ REMOVED
        set timeout $\Delta$

on receive ⟨PUSH, $v$⟩
    if $state$ = SUSCEPTIBLE then
        $value \leftarrow v$
        $state$ = INFECTED
```

```
% Feedback/counter variant
on update($v$)
    $value \leftarrow v$
    $state \leftarrow$ INFECTED
    $counter \leftarrow k$
    set timeout $\Delta$

on timeout
    if $state$ = INFECTED then
        $q \leftarrow$ random($P$)
        send ⟨PUSH, $p$, $value$⟩ to $q$
        set timeout $\Delta$

on receive ⟨PUSH, $q$, $v$⟩
    send ⟨REPLY, $state$⟩ to $q$
    if $state$ = SUSCEPTIBLE then
        $value \leftarrow v$
        $state$ = INFECTED
        $counter \leftarrow k$

on receive ⟨REPLY, $s$⟩
    if $s \neq$ SUSCEPTIBLE then
        $counter \leftarrow counter - 1$
        if $counter = 0$ then
            $state \leftarrow$ REMOVED
```

Figure 3.2: Complex epidemic, copied from [13]

Both Counter and Probabilistic based flooding are shown in fig. 3.2, where blind/coin uses probabilistic flooding and feedback/counter counter-based flooding. In these example implementations complex epidemics are used to get the latest copy of $value$ to each node in P. Each round a node selects a gossiping partner q from all nodes P, and pushes its $value$ to node q.

Broadcast Disks: Data Management for Asymmetric Communications Environments
In BroadcastDisks[2] a server transmits multiple elements over a channel. For the following examples the items are known as the letters of the alphabet (A,B,C,D,...). The basic solution for broadcasting would be to transmit the elements in a round robin fashion through the channel (A,B,...,Y,Z,A,...). This is known as a single disc. If for example A would be a more important element this could be interleaved such that the transmission becomes (A,B,C,A,D,E..). In this way A travels through the channel at a higher frequency than other elements, this could be described as two discs. Disc 1 containing element A and Disc 2 containing the rest of the alphabet. Where Disc1 has a higher priority than Disc2.

Based upon this communication channel the paper describes a way to manage a local cache. Items that are transmitted more frequently do not have to be cached, the access latency is lower when it is transmitted more often.

This works for purely asymmetric communications in which one transmitter has access to all data and regularly transmits this data to multiple clients which only listen.

pCache - Global Data Management
The pCache algorithm, which is described in Global Data Management[11], is aimed at Mobile Ad-Hoc Networks(MANETs). This type of networks is comparible to WSN's since they are formed autonomously. MANETS generally have a higher throughput and are more mobile than WSN's. Nodes are constantly on the move.

PCache is an algorithm that, just as *SharedState*, is based upon epidemic gossiping. Each node has a local buffer in which a limited set of data elements can be kept. Each of these data elements has a key, value, version and expiration. Within *MyMesh* a token has just a key and value. The application itself can however implement a version number. The PCache algorithm tries to store as much data within one hop for rapid access. It varies its buffer contents to achieve a better distribution of data elements in the network. The node that started the distribution of the data element is the owner. It is assumed that at all times the owner keeps its own data elements such that there is always at least one copy. PCache tries to store any element at most 1 hop away. If a node has interest in a specific data element it sends a request. Since pCache tries to store data at 1 hop away the direct neigbours can directly send back the data element if it is in their local cache. If no reply is received the neighbours don't seem to have it and a request that spans more than 1 hop must be done and routed back to this specific node. In this case an overlay network is needed to relay data back.

HOOD
Hood is a neighbourhood abstraction [23]. In a sensor network each node will usually save some information of their direct neighbours, like when it was last seen. In Hood a neighbourhood is defined by a set of criteria and can share a set of attributes, a node can be part of multiple neighbourhoods. For example defining a one hop neighbourhood in which the light state is shared and a two hop neighbourhood in which temperature is shared. The abstractions in Hood allow the programmer to read the shared values of each neighbour. Hood is hiding complexity by managing discovery and data sharing.

Hood uses broadcasting and filtering, when a node transmits its attributes it is only cached at the receivers if they consider it valuable. The transmitting node does not know if and who cached their attributes. To indicate this asymmetry the terms neighbour and co-neighbour are used. A neighbour is a node of which you got attributes cached, a co-neighbour has your attributes cached. Upon updating an attribute it is rebroadcasted such that the co-neighbours are updated.

Unfortunately the criteria for a neighborhood are unknown for the application of *SharedState*. The publishers and subscribers cannot be seen directly as neighbors or co-neighbors. Also the total storage capacity is bound by the number of neighborhoods that are used.

3.3.2. Distribution
Shared state uses a probabilistic approach to Distribute elements. The output buffer is filled with random entries from the cache (either when being replaced or at random to fill the message). If all caches are unique each of the tokens in the network should have an equal probability of being broadcasted.

Some applications add the network time to the token. While a token is "fresh" it will be flooded throughout the network using the time as a time-to-live indicator.

PCache distribution is based upon the Trickle [10] algorithm in which each data element which is received is put on hold. The hold time is chosen proportional to the receive strength. A signal which is weaker is assumed to be further away and yields a shorter hold time than a strong reception. During the hold time the node listens for this same data element. Upon receiving the same data element another two times it will not be rebroadcasted. If a node received less than two copies during the hold time the node rebroadcasts the data element. Because each node generally has a different reception strength the hold time will be different for each node for the same broadcast. In this way a message is flooded throughout the network whilst using less messages. The weaker message are considered to be further away. Using a shorter hold time this should result in less hops used throughout the network.

In Broadcast Disks a single server has the whole list of items and knows their priority. Being just a single system and knowing all information the scheduling of the items from the various disks can be calculated up front. This is much harder in a distributed system. Each node has their own radio range in which the tokens are distributed. This cannot be seen as a single communication channel in which elements can pass through. Grouping tokens into multiple groups, with or without priority, could be considered in this research.

3.3.3. Maintenance

In the maintenance stage of shared state a token is not fresh anymore. Each of the elements of the cache now has the same probability of being broadcasted again. When *SharedState* stores more elements than a single cache can store a cache-replacement policy comes in play. If a token is received whilst it is already in the cache it will be removed from the cache and inputbuffer, making space for other tokens. When more tokens need to be stored than free places in the cache random elements are moved to the output buffer.

The PCache algorithm adds a counter to outgoing messages which indicates when it was stored, increasing with each hop since storage. This counter is called Time From Storage(TFS). During the hold time of a data element each node saves the lowest value of TFS it received ($mintfs$). If at the end of the hold time a data element is received more than two times and $mintfs < 2$ the data element will be discarded. A low $mintfs$ value indicates that one of the neighbours has already saved this data element. By receiving a data element more than two times during its hold time we can assume that the neighbours already propagated this data element. If this was not the case the node saves the data element and re-transmits it with a probability of $e^{mintfs-2}$. If a data element is saved more than 2 hops away this is probability is 1. To remove full deterministic behaviour a node has a probability to store this data element whilst a copy might be only 1 hop away. Note that so far a data element is stored and the TFS is reset to zero upon rebroadcasting. Neighbours where the hold time is not yet reached will have their $mintfs$ reset to 0 and thus have a lower probability of storing and rebroadcasting this data element. When the hold time expires and the data element was not received more than two times and it was not saved by the probability of $e^{mintfs-2}$ the node will rebroadcast the data element with a $TFS + 1$.

3.3.4. Interesting parts

The communication used in the *MyMesh* network is different from most applications. By using broadcast only communication many algorithms found are not applicable. The basics by Demers et al.[6] give us a good basic understanding and terminology for future algorithms. Though still being random one-to-one gossiping, where *MyMesh* is many-to-many.

The PCache (and therein Trickle) has some interesting features like the hold time to limit number of messages used for flooding a message throughout the network and adding metadata to the cache. Using this metadata a more deterministic approach on cache replacement can be made. The downside being a MANET algorithm is ability to transfer more data, having an extra TFS field per data element is more costly on the *MyMesh* network.

Hood has a huge replication factor and needs a lot of memory to operate, for each of the co-neighbours a set of attributes is stored and pre-allocated.

4
Algorithms

The *MyMesh* network is a complex system. Designing an algorithm by means of exploration seems like a good approach to find a improvement that scales better with the number of nodes and lowers the replication factor. As explained previously in Section 1.4 complex systems can land into a steady state based upon a set of variables. By keeping these variables static and changing the algorithm a different steady state can be found which could be more or less desirable. Inspired by VICINITY [21] both fully random and fully deterministic approaches will be tested in the first section of this chapter. Based upon a mention of a fellow student (Kees Kroep) the second part of this chapter focuses on filtering incoming data.

These algorithms are tested using the test-bench described in Section 2.1. The data set which results form these experiments is too large to represent in a meaningful manner in this thesis. Therefore the network for the figures in this chapter consist of 12 nodes in a single hop or multi hop network. The nodes in the network have a cache of 40 tokens and can transmit 5 tokens per round. This configuration resembles the current settings used by *Chess Wise*. This network cannot save the whole impulse applied by the seeder (500 tokens). There is only space for 440 tokens in the network (seeder is excluded from total storage capacity). Using a replication factor of 2 the total network storage should reach 220 which is in the center of the figures.

In the appendix fig. C.1 shows all algorithms placed side by side on one page. The axis scales of the graphs are kept equal between all algorithms to make an easy comparison.

4.1. Algorithmic changes

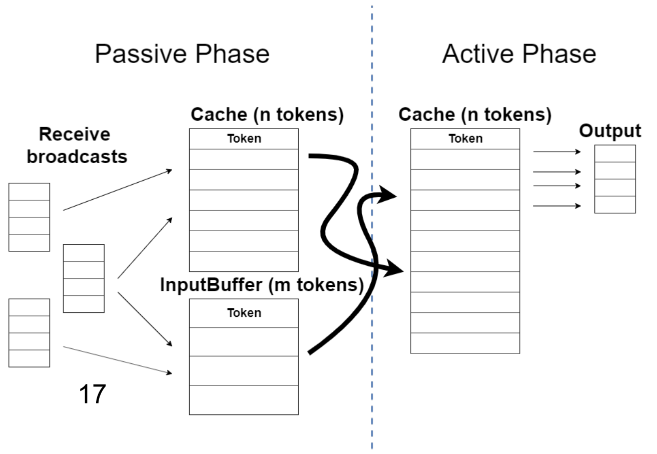
The current *SharedState* is a probabilistic storage algorithm. It is not fully probabilistic as it has some semantic purging of the cache. So in comparison to VICINITY it is random with a pinch of determinism. In this chapter both extremes (Random and Deterministic) are explored and another mix is made.

The same structure as the shared state algorithm will be used for these algorithms. There are three buffers named *input*, *cache* and *output* buffer. And there are two phases, the Passive phase places the incoming tokens into the input-buffer and the Active phase copies the tokens from the input buffer to the cache and output buffer.

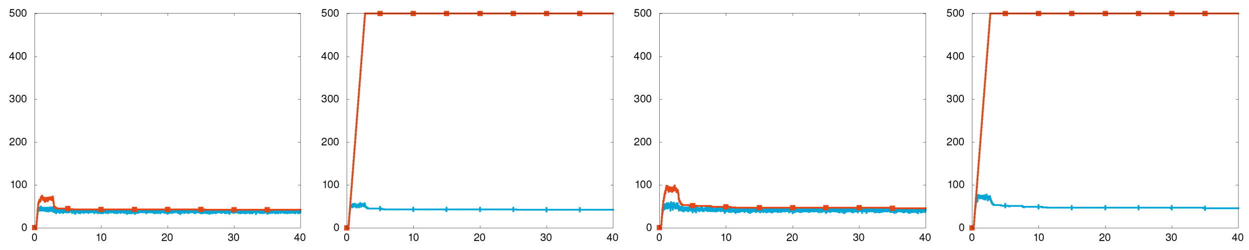
4.1.1. Random

In the random approach, shown in fig. 4.1 and pseudo-code listed in appendix B.3, a random approach is proposed. In the passive phase the received tokens are placed into the input-buffer if these are not known in the cache. In this way each token is known at most once. In the active phase the tokens in the cache and input-buffer are combined and reduced such that

Figure 4.1: Random implementation - Deduplicate, transmit and cache at random



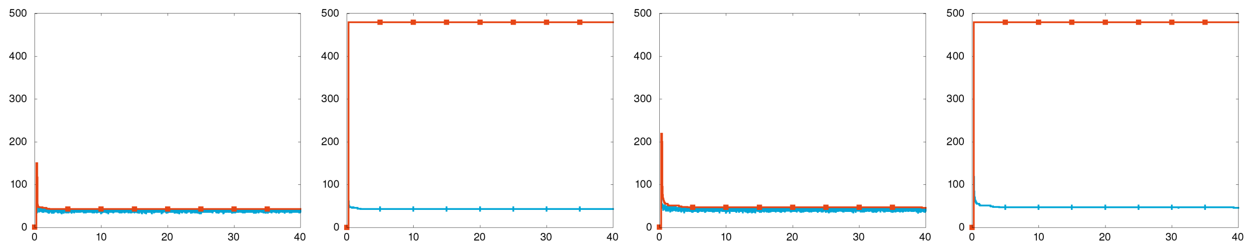
at most C_s tokens are kept. Each of the tokens carry the same weight in the selection process and are placed back at random in the cache. Since the tokens are placed back at random the top x items can be copied into the output buffer.



(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.2: Random - one seeder (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

In these experiments the cache size used is 40. Only when there are tokens injected into the network the lines in the air and storage plots reach above 40. Once the seeder stops injecting the number of tokens quickly drop to about 40 where the line stabilizes. This is way worse than Shared State (see fig. C.1 for easy comparison). In fig. 4.2 the orange line (with square points) in figures b and d shows the number of tokens injected by the seeder. This goes to 500 while the number of stored tokens just reaches above the cache size of 40. All 500 tokens are injected in the system and only about 40 tokens remain in the system. The end storage capacity is 42 for the single hop network and 46 for the multi-hop network.



(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.3: Random - all unique (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

In the second set of experiments all nodes fill their cache at time 250 with unique tokens. So the system has 480 tokens at time 250. At that time all nodes will start to exchange their tokens. The in the air figures of fig. 4.3{a,c} show that the orange line (with square points) reaches back to 40 again within a couple of rounds. Having this plot lead to the same amount of tokens as when the seeder injected tokens shows that the result is a maximum bound and not an effect of the seeder.

4.1.2. Deterministic - Entropy

In the paper by Demers et al. mentioned in Section 1.3 anti-entropy is mentioned. Here only one value is used and no caching is added. Entropy within information theory is introduced by Claude Shannon in 1948[19], and has an analogous in statistical thermodynamics.

This is useful to discuss properties of the network. For example when there is no activity in the network it could be considered in a frozen state. Being overactive as a gas state. Being in a gas state means more movement and a higher chance of losing data. Having a new token at a certain position in the network leads to an increase of entropy. That needs to flow out such that entropy is the same throughout the network.

$$S = -\sum_i P_i \log_2(P_i) \tag{4.1}$$

$$I(p) = \log_2\left(\frac{1}{p}\right) = -\log_2(p) \tag{4.2}$$

The equation by Shannon gives the total entropy eq. (4.1) or informational capacity of a communication channel. For example if a node receives 4 tokens the informational value can range from 0 to 2. This is demonstrated in eq. (4.3) in which gradually more information is added. In the first equation a single token is received 4 times, this gives an informational value of 0 since there is no information added. In the second example two different tokens are received and the entropy rises. Even more information is in the next example where one token is received twice and two tokens received once.

$$S_1 = -(\frac{1}{1}\log_2\left(\frac{1}{1}\right)) \qquad = 0 \tag{4.3}$$

$$S_2 = -(2\cdot\frac{1}{2}\log_2\left(\frac{1}{2}\right)) \qquad = 1 \tag{4.4}$$

$$S_3 = -(\frac{1}{2}\log_2\left(\frac{1}{2}\right) + 2\cdot(\frac{1}{4}\log_2\left(\frac{1}{4}\right)) \qquad = 1.5 \tag{4.5}$$

$$S_4 = -(4*(\frac{1}{4}\log_2\left(\frac{1}{4}\right))) \qquad = 2 \tag{4.6}$$

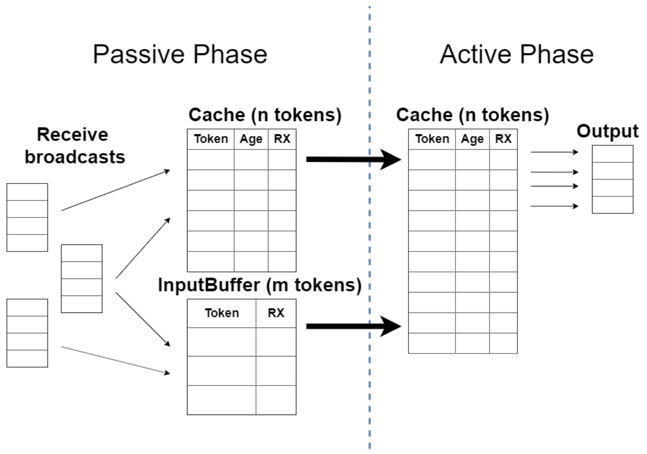


Figure 4.4: Entropy implementation - Fill output based on highest informational value

A node can only act upon data it receives. The general idea is to track the informational value of each token locally without using too much resources. A counter is added to each token in the cache and input-buffer to count the number of times the token is received. For the current round the number of tokens received is given. Using the number of tokens received N_{total} and the number of times the current token t in the input buffer is received $N_{rx}(t)$ the value p for eq. (4.2) can be calculated ($p = \frac{N_{rx}(t)}{N_{total}}$) and the informational value I can be calculated.

Once a token enters the cache the informational value should be calculated over all received tokens since placement in the cache. For this N_{total} should be saved with each token in the cache. This is rather costly and in our experiments the number of tokens in a message is static. Instead the average number of tokens per round is stored globally $N_{avg_{rx}}$ and per token in the cache t an age counter $A(t)$ is added. Using these values the informational value of the cached tokens can be calculated $p = \frac{N_{rx}(t)}{A(t)*N_{avg_{rx}}}$.

In the passive phase of the algorithm de-duplication is done in the cache and input-buffer. If a token exists in either the cache or input buffer the corresponding counter $N_{rx}(t)$ is increased. If a token is unknown it is added to the input buffer. In the Active phase the cache is filled with tokens with the highest informational value at the top and tokens with the lowest informational value at the bottom. In this way the cache and input buffer are combined and only the most unique tokens are stored. For transmission the top x items are copied into the output buffer. These items carry the highest informational value.

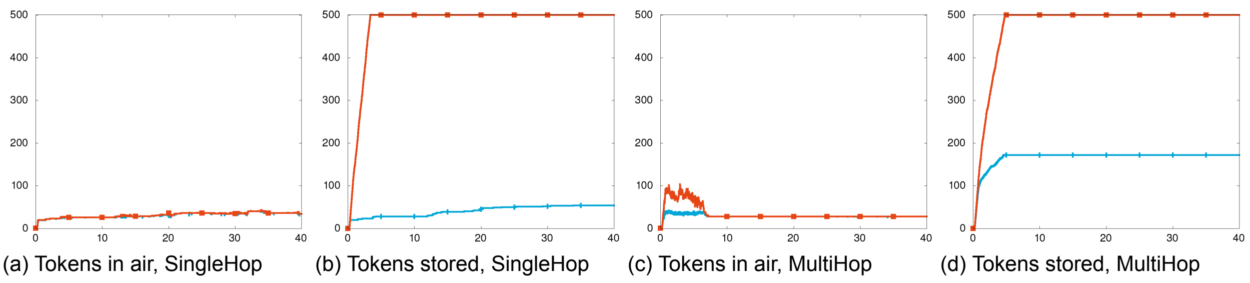

(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.5: Entropy - one seeder (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

In the figures shown in fig. 4.5 the 4 plots as shown previously for the *SharedState* (fig. 2.4) and random (fig. 4.2) algorithm are shown in the same scale. In these experiments the cache size used is 40. The in the air plots of the single and multi hop networks show that the exchange of information is low throughout most of the experiment. Less than 40 tokens are exchanged. In the multiple hop in the air figure (fig. 4.5.c) the exchange of information is higher while the seeder is adding information (depicted by the orange line, with squared points). While the seeder is injecting new tokens the multi hop does accept new information, the storage count in fig. 4.5.d rises. Once the seeder stops the exchange of information quickly freezes showing a horizontal line in the graphs.

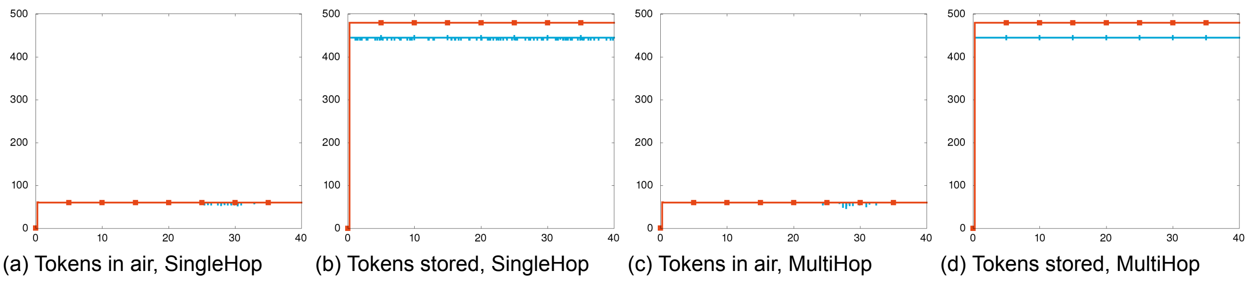

(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.6: Entropy - all unique (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

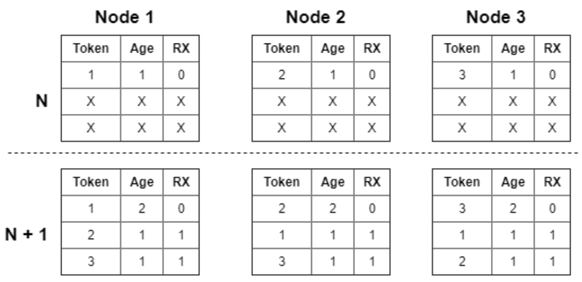
When a different seeder strategy is used where each node contains 40 unique tokens we see that nothing happens. All lines in fig. 4.6 are horizontal. For the storage figures horizontal lines can be seen as a positive outcome, no tokens are lost and a replication factor of 1 is achieved. The horizontal line in the air graphs lays at 60. This coincides with the number of tokens transmitted each round (12 nodes, 5 tokens per message). Since these lines do not exceed 60 the same information is exchanged each round. The network is in a frozen state. The orange line shows the unique tokens over the last 10000 received tokens. If each round a different token is exchanged this line would lay above 60.

4.1.3. Comparing results

Both random and entropy seem to do a worse job than *SharedState*. Only after creating the algorithms and doing the experiments the flaws of both approaches really come forward. On one hand the random algorithm does not favor any token. The latest token which is distributed by the injector is discarded with the same probability of all other tokens in the cache. So new tokens are not easily accepted. Once the seeder stops adding tokens the number of tokens quickly reduces to the cache size. All caches contain the same tokens.

The flaw in the entropy variant can be explained by an example with three fully connected nodes which transmit 1 token and can store up to three tokens. Each node has a unique token (see fig. 4.7). In round N all nodes transmit their own unique token and receive the tokens of the other two nodes. These new tokens are added to the cache and the age of all tokens is updated. Since the node never heard its own unique token its informational value will in increase and remain at the top of the cache. This token will be re-transmitted in round N+1,N+2,... etc.

Looking at the total network as one system each token has the same entropy. A token which has a high entropy level at Node 1 has a low entropy value at Node 2 and 3. The goal however was to achieve



	Node 1 Token	Node 1 Age	Node 1 RX	Node 2 Token	Node 2 Age	Node 2 RX	Node 3 Token	Node 3 Age	Node 3 RX
N	1	1	0	2	1	0	3	1	0
	X	X	X	X	X	X	X	X	X
	X	X	X	X	X	X	X	X	X
N + 1	1	2	0	2	2	0	3	2	0
	2	1	1	1	1	1	1	1	1
	3	1	1	3	1	1	2	1	1

Figure 4.7: Entropy: Why is this not working?

the same entropy value for each token at each node. In this way all tokens are evenly distributed across the network storage and the messages sent by the nodes.

There needs to be some feedback in the system. When a node transmits a token the informational value decreases at the neighbors for this token. Since the informational value decreases the neighboring nodes will stop to distribute this token and eventually discard the token.

4.1.4. Algorithm 2.0

The entropy variant creates a list of all tokens ordered by their informational value. Tokens at the top will not be discarded as long as they have a high informational value. Tokens at the bottom are less unique and have a lower informational value. These will be replaced by tokens with a higher informational value. Among these bottom tokens there are tokens that are received recently from our neighbors. By sending feedback on these tokens the entropy decreases at the sender (given that they receive the feedback). Therefore the neighbouring node will transmit this token less often. This is more the way that was envisioned by creating an even distribution of entropy across the network.

For this feedback a pinch of randomness is introduced. By not only treating the position in the cache as a measurement of informational value but also the probability of being transmitted a system is created in which the less unique tokens are still distributed to provide feedback.

Each token in the cache has a position N_c where 0 is the first entry with a high informational value and $C_s - 1$ is the last entry with a low informational value. The entry with the lowest informational value should still have a minimum probability of being transmitted, Using $P_{send}(N_c)$ the probability to be broadcasted can be calculated.

$$P_{send}(N_c) = P_{min} + P_{fact} \cdot (C_s - N_c) \tag{4.7}$$

P_{send}	= Probability of being in transmit message
P_{min}	= Minimal probability
P_{fact}	= Probability increase per cache position
N_c	= Cache position
C_s	= Cache size

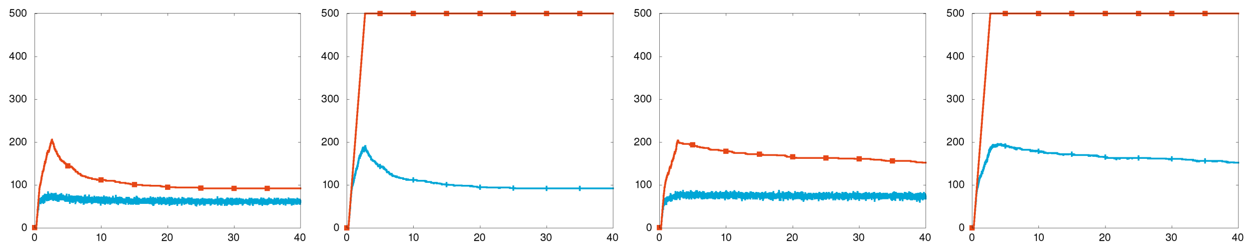
In these experiments the number of tokens which is sent out each round is static (5 tokens per message). Applying eq. (4.7) to all elements of the cache should result into the number of tokens that a node can transmit N_t.

$$N_t = \sum_{N_c=0}^{C_s-1} P_{send}(N_c) = \sum_{N_c=0}^{C_s-1} (P_{min} + P_{fact} \cdot (C_s - N_c)) \tag{4.8}$$

$$N_t = C_s \cdot P_{min} + P_{fact} \cdot \sum_{N_c=0}^{C_s-1} (C_s - N_c) = C_s \cdot P_{min} + P_{fact} \cdot \frac{C_s \cdot (C_s + 1)}{2} \tag{4.9}$$

$$P_{fact} = \frac{2 \cdot (N_t - C_s \cdot P_{min})}{C_s \cdot (C_s + 1)} \tag{4.10}$$

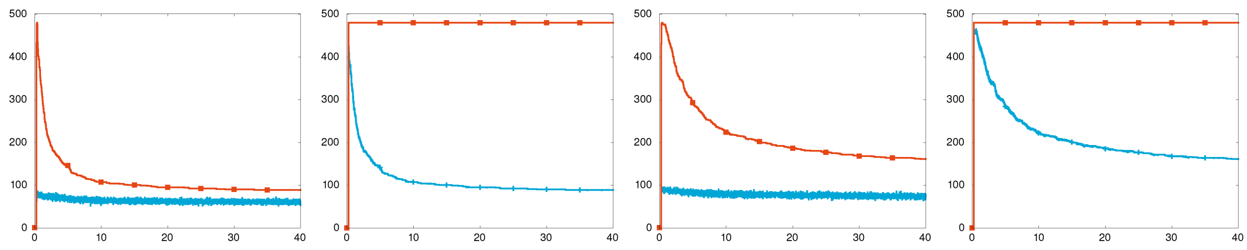
For the test setup with $C_s = 40, N_t = 5, P_{min} = 0.03$ the calculated value of $P_{fact} \approx 0.005$. For the first entry in the cache $P_{send}(0) = 0.03 + P_{fact} \cdot 40 \approx 0.22$ and the last entry equals $P_{send}(C_s - 1) \approx 0.035$



(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.8: Algorithm 2.0 - one seeder (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

Adding this feedback clearly works. From the storage logs in fig. 4.8.{b,d} it is clear the injected tokens stay in the system better and longer than any algorithm tested before. The multi hop network reaches 200 tokens in the air after which it stabilizes around 150 tokens (orange squared line in fig. 4.8.d). For a small network (12 nodes) this is a huge improvement. In multi-hop the replication factor already comes a lot closer to 2, $r = \frac{11*40}{152} \approx 2.89$. Single hop is lagging a bit behind, $r = \frac{11*40}{94} \approx 4.68$, but still an improvement over any of the other implementations.



(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.9: Algorithm 2.0 - all Unique (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

When all nodes have their own unique tokens the number of tokens that remain in the system stabilizes around 160 for multi-hop and 90 for single hop. These numbers correspond to the numbers in fig. 4.8. The seeding strategy has similar results.

4.2. Data filtering

In the previous section the single hop performance is lagging a bit behind the multi hop network. If all nodes are within one hop the data received is the same for all nodes. If all nodes behave in the same deterministic way the caches become more alike. Resulting in a reduced overall storage capacity.

A fellow student Kees Kroep suggested to add a filter on the incoming channel. If every node just processes 20% of the data the number of tokens that each node adds to its cache becomes smaller. More unique tokens saved, potentially increasing the total network storage capacity.

The proposed algorithms are all in the form of anti-entropy with push as described in Section 1.3. The paper [20] acknowledges this idea: ”The paper shows that anti-entropy protocols can process only a limited rate of updates”. In their approach they have two mechanisms. One dependent on token age, which is not applicable in the current setup (in the current tests each token only carries a unique identifier and data). The second mechanism adapts the local update rate. By adapting the local update rate its gossiping partners will receive fewer updates. If all nodes adapt this strategy all nodes will have to process fewer messages.

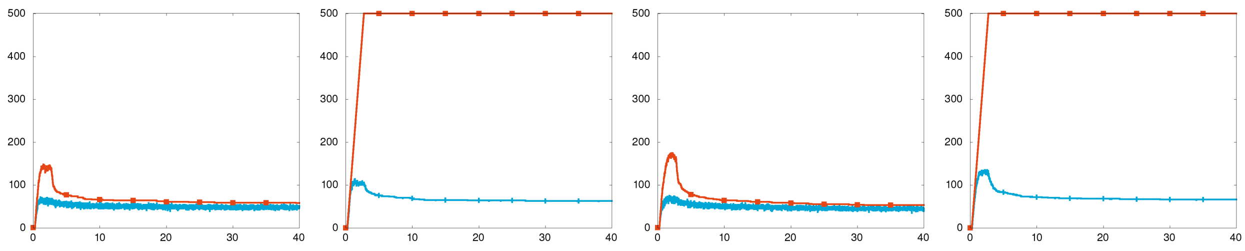
An easy way of adapting the update rate is to give each node a unique sequential ID. Based on this ID each node processes only a specific set of items in the input-buffer. In these experiments the nodes are numbered sequentially. In the $2xN$ topology the most-left two nodes have the ID 101 and

102, the next column 103 and 104. This ID increases to the right. For these experiments the filtering is based upon a fixed modulo operator. So if all nodes are labeled $0 \ldots X$ and the modulo chosen is 5 Node 0 will process tokens $0, 5, 10, \ldots$ from its input-buffer. Node 1 will process $1, 6, 11, \ldots$. When there are more than 5 nodes Node 5 will process $0, 5, 10, \ldots$ again. The modulo chosen must be smaller than the number tokens received and the number of neighbors, otherwise tokens will not be processed. In these experiments the number of nodes and the number of tokens per message is static. Therefore a static modulo operator can be used. In a future implementation this filtering should be done differently.

For each algorithm a preliminary explanation is given on how this filtering could impact the algorithm. After the graphs these explanations are tested against the results. For an easy comparison one can look at fig. C.1 and fig. C.3 in the appendix to see the differences with and without filtering.

4.2.1. SharedState

For *SharedState* an improvement is expected. By having only one fifth of the data to store in the cache each cache should become more unique.



(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.10: SharedState + Filtering - one seeder (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

The addition of this data filter to *SharedState* has almost no effect. Somehow the lines are almost the same as in Figure 2.4. It is unclear why the caches do not become more unique. Seeing that the caches are not becoming unique tokens must be lost.

There is one obvious way for *SharedState* to "lose" a token. The following example will demonstrate this. If we have a network of two nodes (1 and 2) with full caches containing all unique items except one token A. This token A is present on both nodes. Each round a node can only transmit one token. The nodes are fully connected and no messages are lost. The following exchange happens:

- Round 1: Both Node 1 and 2 select a random item (not token A) to transmit.
- Round 2:
  - Node 1 needs to make space in the cache and randomly selects token A to be removed from the cache. If removed from the cache it is placed in the output buffer. Node 1 will now transmit token A.
  - Node 2 selects a random token to replace. This token will be selected for transmission.
- Round 3:
  - Node 1 receives a random token and needs to select a item from the cache to move to the output buffer. token A is no longer present in the cache of Node 1.
  - Node 2 now receives token A, it will be marked free. Node 2 selects a random item from the cache (not token A) to transmit.
- Round 4:
  - Node 1 receives a random token and selects an item from the cache to move to the output buffer.
  - Node 2 now receives a random token, in round 3 the location of token A is marked free. Since this is seen as a free spot the received token is placed in the place of token A without transmitting token A

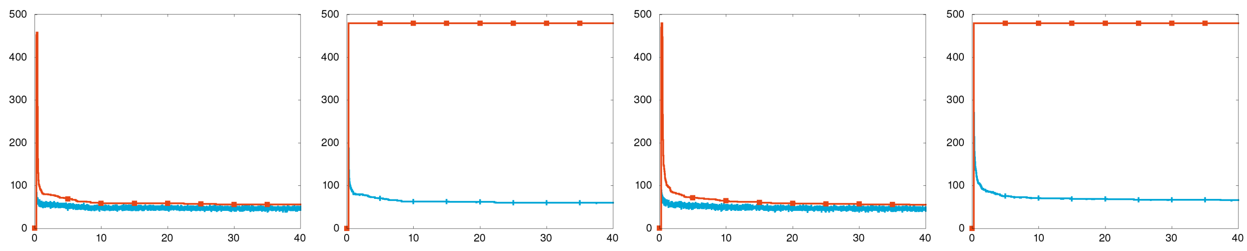
As an end result of above exchange token ”A” is lost.

In the above example there are two types of transmission: Unforced and Forced. If Forced a token will be moved from the cache and transmitted. It is no longer stored in the cache. With Unforced the token that is being transmitted is still stored in the cache. Only in Round 1 both nodes are unforced. In round 3 and 4 node 2 is unforced.

When able to transmit more tokens per round this phenomena could become even more apparent. More nodes are forced to transmit tokens. The remainder of the transmit message is filled with random unforced tokens. For example; If at round N Node 1 contains tokens A,B,C and D and Node 2 contains tokens D,E,F and G the following exchange could occur:

- Round N:
  - Node 1 is forced to transmit a random token(D) due to incoming tokens and a full cache. Node 1 selects token B as a second token to transmit.
  - Node 2 is unforced and selects two random tokens to transmit (token E and G).
- Round $N+1$:
  - Node 1 receives token E and G, it has no space and needs to forcefully transmit two random items. Token A and C.
  - Node 2 receives token D and marks its location Free. The second incoming token B is placed in the cache at the location of token D.

In this example within one round one token is eliminated. So far no messages are lost. If the message from Node 1 in round $N+1$ is lost even three tokens could be lost.



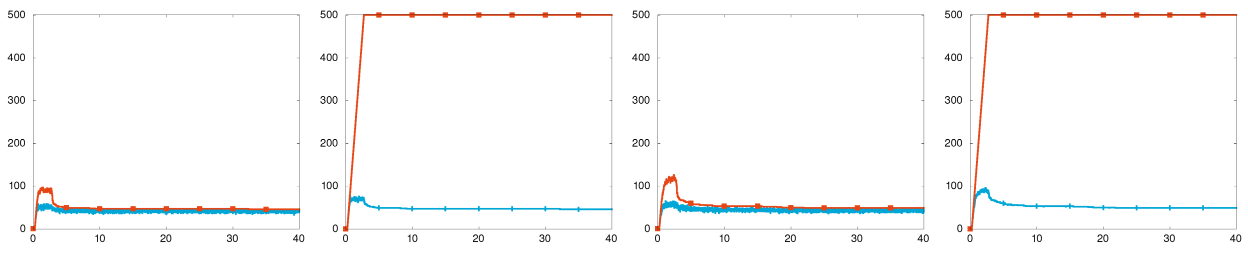
(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.11: SharedState + Filtering - all unique (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

When having all unique tokens at each node the algorithm still stabilizes at the same values as *SharedState* without filtering, the slope is a bit less steep. The seeding strategy has no effect on the total storage capacity of the network.

4.2.2. Random

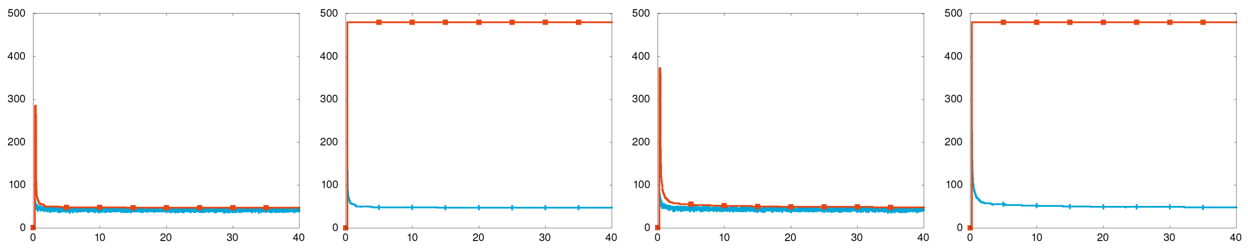
The random algorithm without filtering could only store about 60 tokens in the network whilst the seeder is injecting tokens (see fig. 4.2). Once the seeder stops the total storage capacity rapidly declines back to about 40 tokens. Data filtering should make this decline less steep since fewer tokens get overwritten. In the steady state response no real gain is expected. No token is favored above any other token and all caches become more alike.



(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.12: Random + Filtering - one seeder (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

The figures lay somewhat higher than previous experiments using the random approach. The maximum number of tokens stored is around 80, a bit of an improvement. When the seeder stops injecting tokens the number of stored tokens quickly goes back down to around 40. This was expected to be less steep. For the part where the total storage capacity is 60 and goes down to 40 this holds. For the part from 80 to 60 the line is still very steep. The more elements are stored in the network the lower the overlap between caches. Having more unique items come in will result in a steeper decline. Even with filtering.



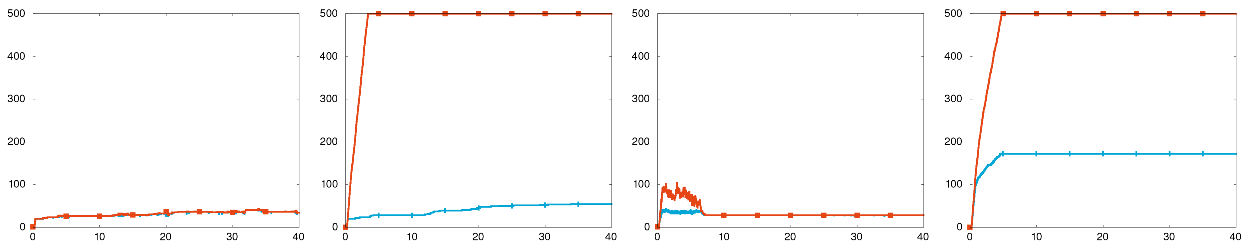
(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.13: Random + Filtering - all unique (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

When having all unique tokens at each node again the system quickly loses tokens. The slope is less steep but within 500 rounds the network is back to around 40 tokens.

4.2.3. Entropy

For entropy no real difference is expected. In fig. 4.7 it is shown that each node only transmits its own Token which it deems important to share. The other nodes will never transmit this token.



(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.14: Entropy + Filtering - one seeder (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

For entropy the added modulo filter does not really help. It even becomes worse, the number of tokens going through the air for single-hop now slowly climbs to 60. This indicates that all nodes selected their own unique tokens to transmit.

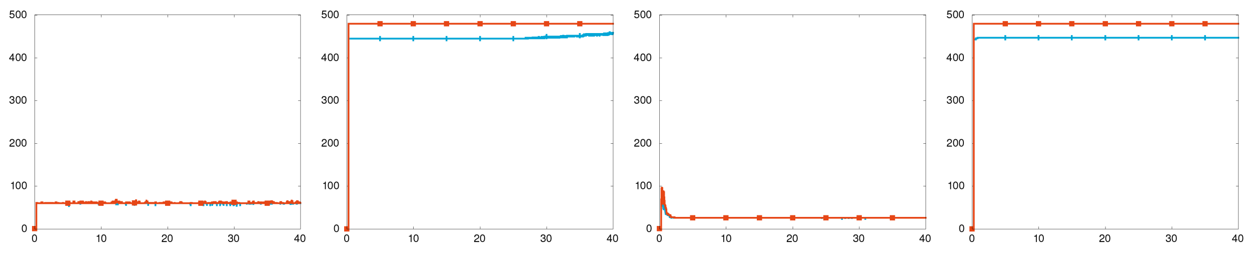

(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.15: Entropy + Filtering - all unique (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

Using all unique tokens the results stay the same as without a filter in fig. 4.6. Once all nodes created their own unique tokens at $t = 250$ each node sends out their first 5 unique tokens. This leads to 60 unique tokens through the air. The number of stored tokens stays the same. Essentially all incoming tokens are discarded. Using the entropy calculation the incoming tokens (Age 1, Receive count 1) cannot exceed the entropy calculation of tokens already in place (Age 2, Receive count 1). Filtering has no gain here.

4.2.4. Algorithm 2.0

For algorithm 2.0 a gain is expected. Each of the nodes store the tokens they deem important. The importance of the tokens is measured by the number of times the token is received whilst also being stored in the cache. This feedback mechanism will be somewhat impaired, since it only has information of one fifth of the received tokens.

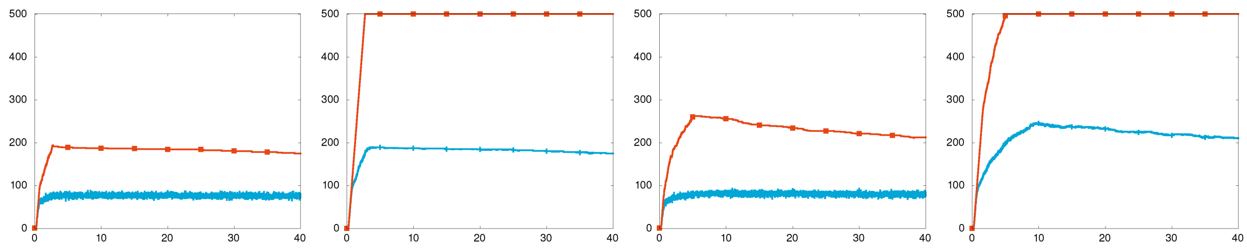

(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.16: Algorithm 2.0 + Filtering - one seeder (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

Adding a simple modulo filtering has a huge impact on algorithm 2.0. The network absorbs the data at a faster rate. And once the seeder stops injecting tokens the number of tokens that are stored almost stays the same. In the previous test without filtering (fig. 4.8) the number of tokens declined immediately once the injector stopped. Now the number of tokens stored in the network lays around 190 and there is almost no decline in the number of tokens stored (indicated by the blue line in fig. 4.16.b).

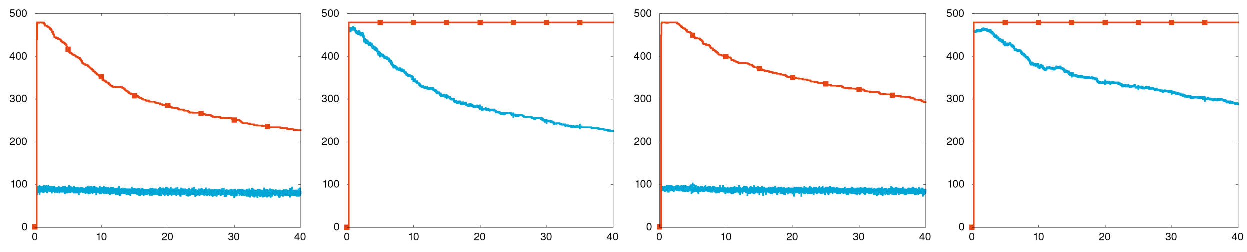
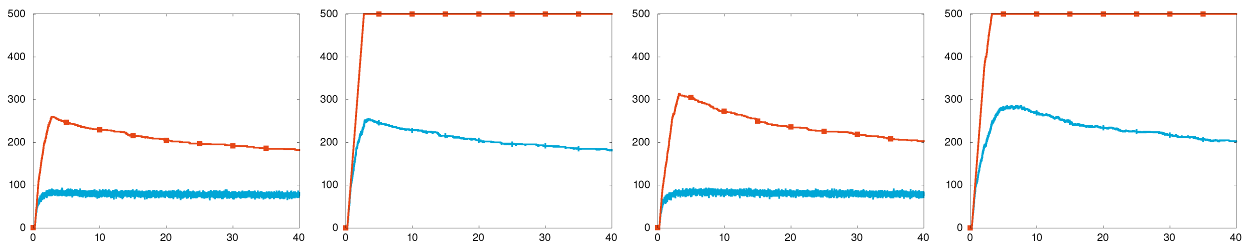

(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.17: Algorithm 2.0 + Filtering - all unique (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

When adding unique tokens to all nodes and adding filtering the number of tokens that are stored at the end of the experiment is increased. For single hop +/- 210 tokens are stored and for multiple hops almost 300. It seems however that the steady state, which was more apparent in fig. 4.8, is not yet reached.

4.2.5. Algorithm 2.0 - unfair filtering

In the previous filtering all algorithms were presented with the same elements, one in five tokens were passed through to the algorithm, and the rest was dropped. Algorithm 2.0 uses the incoming data to determine the informational value for the stored tokens. In this experiment this is changed by notifying the algorithm of the dropped tokens. These tokens are not stored but their informational value is updated if stored in the cache. This should have an even higher impact than just filtering. The algorithm has all the information and is just presented with a couple of updates to store.



(a) Tokens in air, SingleHop (b) Tokens stored, SingleHop (c) Tokens in air, MultiHop (d) Tokens stored, MultiHop

Figure 4.18: Algorithm 2.0 + Unfair filtering - one seeder (X=Time in rounds / 1000, Y=Number of unique tokens (max=500))

In the multi-hop for the first time the 300 tokens in the air is reached, and the storage stabilizes around 210 (indicated by the blue line in fig. 4.18.d). For single-hop 280 tokens in the air is reached and storage stabilizes around 190 (indicated by the blue line in fig. 4.18.b). In the first half of the graphs the tokens that are injected by the seeder are retained better. In the second half the number of tokens that is left in the system stabilizes around the same number of tokens as with the ”fair” filtering.

4.2.6. Results

All experiments were conducted in a setup with 12 nodes, in which one is the seeder. One node can store up to 40 tokens in their cache. The total storage volume is therefore $11 * 40 = 440$, as the seeder is excluded from these calculations. In the following table the end result of the experiments is shown together with the replication factor between braces.

	Single hop no Filter	**Single hop with filter**	**Multiple hops no filter**	**Multiple hops with filter**
SharedState	57 (7.72)	63 (6.98)	62 (7.10)	66 (6.67)
Random	42 (10.48)	46 (9.57)	46 (9.57)	49 (8.98)
Entropy	54 (8.15)	54 (8.15)	172 (2.56)	172 (2.56)
Algorithm 2.0	94 (4.68)	175 (2.51)	152 (2.89)	210 (2.10)
Algorithm 2.0 Unfair	94 (4.68)	182 (2.42)	152 (2.89)	202 (2.180)

Table 4.1: Tokens left in network after experiment, replication factor between braces

From this table we can see that the added modulo filter helps for almost all algorithms. For Entropy no gain is seen, the nodes still pick their own ”unique” tokens. The Algorithm 2.0 almost reaches a replication factor of 2 for both fair and unfair filtering. Having the unfair filtering did lead to an increase in storage capacity whilst injecting tokens. The additional token were however lost in the steady state response. While tokens are lost this is still a good sign, having over 220 tokens stored in the network results in a replication factor smaller than 2. Having a replication factor lower than 2 indicates that there are tokens that are stored at just one node. For these tokens there would be no replication within the network. Making these tokens prone to loss when a node leaves the network.

5 Conclusion and Future Work

5.1. Conclusion

The *MyMesh* network is a Wireless Sensor Network (WSN). The nodes in this WSN are equal, there is no hierarchy and no structure is defined. The *MyMesh* network communicates using tokens. A token consists of an identifier and data. These tokens can be distributed and stored using *SharedState*. *SharedState* is a *Distributed State-full Publish Subscribe System* for the *MyMesh* network.

The current implementation of *SharedState* has an unexpected upper-bound and does not scale when adding nodes.

This research started with an analysis of *SharedState*. In this preliminary research a test-bench is developed. This test-bench is used to test different parameters in a consistent manner. The parameters that are tested are: *Network Size*, *Network Density*, *Throughput* and *Storage capacity*.

The results from this research show that there is indeed an upper bound in *SharedState* (see table A.1 and fig. 2.3). The root cause of this phenomena is not clear from these experiments. None of the tested parameters could be tweaked to show a cause. Later on with Section 4.2.1 a feeling for the cause of the upper bound is stated. When the cache is full random tokens are moved to the output buffer to make space for new tokens. This can lead to tokens being erased from the network (see Section 4.2.1). No experiment was found to demonstrate this. Instead the focus was shifted to creating an alternative algorithm.

In chapter 4 different algorithms are tested. Based on the paper by S. Voulgaris and M. van Steen[21] a decision is made to test two algorithms that are on different ends of a scale from Random to Deterministic. Once both ”extremes” were tested a mix was made to see if a ”pinch of randomness” helps in this case. For the Deterministic variant an information theory[19] based algorithm is implemented, which stores and transmits tokens based on their informational value (Shannons Entropy). By storing and transmitting tokens with a high informational value an attempt is made to even out the informational value for each individual token throughout the network. The Random approach treats the pile of tokens as all having the same chance of being stored and transmitted.

Both of these implementations performed worse than the current *SharedState* implementation. After analysis of the Entropy based implementation in Section 4.1.3 a flaw in the approach was found. If zooming out on the network and treating all communication as one channel all stored tokens had the same informational value. When zooming in on a node one unique token had a high informational value and all others were lower. Due to transmission of the token with the highest informational value the informational value at the neighboring nodes decreased. Since the neighboring nodes have a low informational value for that specific token it will never be transmitted. This results in each node only transmitting the same ”unique” token.

With Section 4.1.4 the third implementation, called algorithm 2.0, is described. Instead of transmitting the tokens with a high informational value the informational value is used as a probability of being transmitted. Using this approach each node provides feedback to its neighboring nodes. With this feedback the informational value for that token will decrease at the neighboring node upon which it will transmit more diverse tokens.

A final, but important addition was coined by fellow student Kees Kroep, who suggested to add filtering to the incoming data. As that would help the individual caches of the nodes to become more unique. Leading to an increased storage capacity in the whole network. Each node has their own unique identifier (ID). Based upon this ID and a modulo 5 operator each node processes al data but only stores 20% based on their ID. The other 80% of data is dropped. This 80% is stored at nodes with a different ID. This filter was first applied to Algorithm 2.0, which had a huge impact. Due too having such a high gain in Section 4.2 all four algorithms were tested using filtering (SharedState, Random, Entropy and Algorithm 2.0). This filter was made more generic by only delivering 20% of the incoming tokens to each of the algorithms for a fair comparison. For the other three algorithms the impact of this filter was not really significant. For Algorithm 2.0 the unfair filtering (processing all tokens and only storing 20%) has a performance gain on the first part whilst tokens are injected over the fair filtering. Both filters perform similarly for net storage.

So far all results have been in a simulated environment. To conclude this research a last comparison is made to a real-life network of 12 nodes. This to circle back to chapter 2 where the simulation is shown to behave similarly to real-life testing. The 12 nodes are places on the wall and in a similar configuration in the simulator. fig. 5.1 shows near identical performance for Simulation (Left) as Real Life(Right).

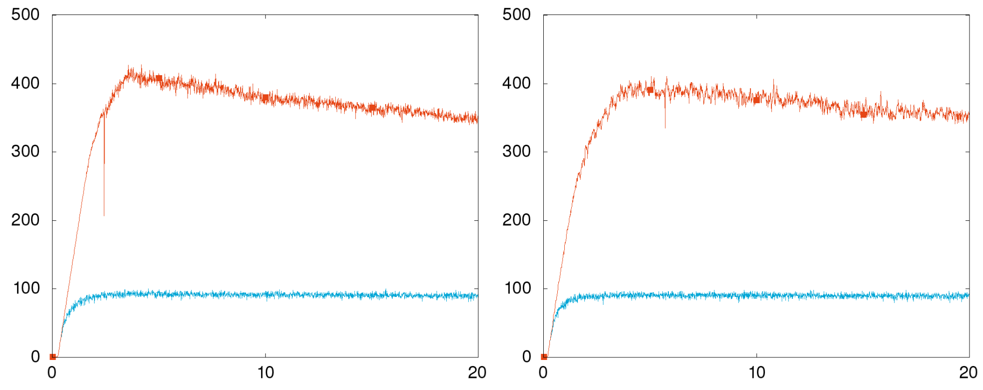


Figure 5.1: Algorithm 2.0 real life test (X=time, Y=Number of unique tokens (max=500))

With the *SharedState* implementation the replication factor goes from 6 for smaller networks (2x12) to 22 for larger networks (2x50). This can be seen in Table A.1 and Section 2.4.1. In this thesis the results of a smaller network have been shown for brevity. Note however that the rest of the experimental results are available and discussed during the thesis work at *Chess Wise*. To show that the renewed algorithm handles scaling better than *SharedState*, the results of the three node counts and densities are shown in Table 5.1. In this table the replication factor lays between 2.29 for a smaller network and 3.06 for a larger network. This is a huge improvement adding more nodes now result into an increase in the total storage capacity.

	Compact	**Normal**	**Wide**
2x12	390 (2.34)	401 (2.29)	367 (2.51)
2x25	797 (2.46)	727 (2.70)	687 (2.85)
2x50	1442 (2.74)	1348 (2.94)	1294 (3.06)

Table 5.1: Algorithm 2.0 with filtering simulation results for larger networks

5.2. Future Work

Besides total storage capacity the new algorithm should still move the data throughout the network. In the multi-hop storage graph of Figure 4.5 there was a storage capacity with a replication factor of two but no movement. With the new algorithm 2.0 all token that are stored are also transmitted (see Figure 4.8). From the used graphs in this theses we cannot see if the data that is transmitted by the nodes is also accepted at other nodes and thus travels around.

In further research this could be measured by using the inter-arrival time at each node for each token. This could be done on the data set that is generated during this thesis.

5.2.1. Improving input filtering

The *MyMesh* network is a self-organizing network. Having a nice distribution of Node IDs like in the experiment is not a given. If input filtering using a simple modulo would be implemented it is advisable to have a second algorithm distribute the numbers 0-4 in an even way across the network. Among neighboring nodes the numbers should be evenly distributed. In the current use case of *MyMesh* (light control) the nodes are at fixed positions and could be given a fixed number when deployed.

Filtering using a fixed modulo does have some caveats. Situations where networks are not fully connected and consist of multiple smaller networks that are loosely coupled can be impacted. The nodes that couple these networks only process 20% of the data received per round. The rest of the 80% cannot be passed through to the other network. Luckily the nodes transmit in a random order and the tokens they select are selected at random. So the chance that a node processes the tokens from the neighbor it ignored this round in another round is fairly high. Additional research should be conducted to see if a different filtering algorithm is better suited.

A
Tables

A.1. Test-bed Measurements

These tables show the measurements done on the test-bed. In the first column the node layout and storage capacity C_s is shown. The layout *2x12* means 2 rows of 12 nodes, totaling $N = 24$ nodes. The number between brackets is the total storage capacity $N \times C_s$, by dividing the total storage capacity by one of the numbers in the other columns the replication factor r can be calculated. The column headers show the number of tokens per message and the proximity of the nodes to each other, C is Close, N is Normal and W is Wide.

		5			10			15			20	
	C	**N**	**W**	**C**	**N**	**W**	**C**	**N**	**W**	**C**	**N**	**W**
2x12: 20 (480)	49	48	52	51	55	61	60	65	70	67	72	77
2x12: 40 (960)	91	97	100	98	104	108	102	109	110	111	115	125
2x12: 60 (1440)	139	149	151	139	150	159	143	152	163	152	163	172
2x12: 80 (1920)	188	195	213	189	202	211	193	204	216	193	209	221
2x25: 20 (1000)	64	67	70	76	78	85	92	94	94	107	107	115
2x25: 40 (2000)	123	126	132	132	135	141	141	145	153	155	158	166
2x25: 60 (3000)	181	192	212	190	193	206	196	204	215	207	210	228
2x25: 80 (4000)	239	251	279	243	254	273	246	261	280	262	270	285
2x50: 20 (2000)	85	83	96	106	105	110	130	129	134	150	147	154
2x50: 40 (4000)	156	169	184	170	180	189	194	190	201	214	205	223
2x50: 60 (6000)	229	252	277	237	245	276	262	259	282	276	279	302
2x50: 80 (8000)	309	326	332	311	328	342	317	334	343	345	344	363
Total	1853	1955	2098	1942	2029	2161	2076	2146	2261	2239	2279	2431

Table A.1: Frits

	5			10			15			20		
	C	**N**	**W**	**C**	**N**	**W**	**C**	**N**	**W**	**C**	**N**	**W**
2x12: 20 (480)	44	50	50	51	58	62	58	66	71	67	75	78
2x12: 40 (960)	90	95	105	99	105	106	104	107	115	107	120	122
2x12: 60 (1140)	137	147	154	143	149	158	147	154	162	153	164	168
2x12: 80 (1920)	182	198	211	188	201	208	195	204	211	195	211	218
2x25: 20 (1000)	65	66	70	76	80	83	93	94	95	103	109	113
2x25: 40 (2000)	120	127	135	128	133	140	140	143	151	156	158	166
2x25: 60 (3000)	149	195	205	183	194	205	195	201	213	206	212	224
2x25: 80 (4000)	236	261	272	241	258	273	251	262	280	258	266	283
2x50: 20 (2000)	83	90	93	108	107	111	130	136	132	148	150	158
2x50: 40 (4000)	155	163	187	168	184	194	196	197	203	212	210	229
2x50: 60 (6000)	231	244	256	233	257	271	257	265	275	274	284	295
2x50: 80 (8000)	301	331	340	307	327	344	323	331	346	333	346	361
Total	1793	1967	2078	1925	2053	2155	2089	2160	2254	2212	2305	2415

Table A.2: Daniella

B
Code Listings

B.1. SharedState Algorithm [7]

```
**var** $localEntry$                                          ▷ Locally generated event/entry
*Phase 1 – Update Cache*
**for all** $entry$ **in** inputBuffer **do**
    **if** cache.contains($entry$) **then**
        cache.delete($entry$)
        inputBuffer.delete($entry$)
**while** cache.slotsAvailable() < inputBuffer.size() **do**
    **var** $randomEntry$ = cache.selectRandomEntry()
    **if** outputBuffer.slotsAvailable() **then**
        outputBuffer.add($randomEntry$)
    cache.delete($randomEntry$)
cache.addAll(inputBuffer)
inputBuffer.clear()
*Phase 2 – Select Entries to Broadcast*
**if** outputBuffer.slotsAvailable() **then**
    **if** !outputBuffer.contains($localEntry$) **then**
        outputBuffer.add($localEntry$)
**while** outputBuffer.slotsAvailable() **and** !(cache - outputBuffer).isEmpty() **do**
    **var** $randomEntry$ = cache.selectRandomEntry()
    **if** !outputBuffer.contains($randomEntry$) **then**
        outputBuffer.add($randomEntry$)
```

B.2. Current Implementation

Phase 1 – Message Reception

```
for all $entry$ in message do
    if residentialEntries.contains($entry$) then
        var $residentialEntry$ = residentialEntries.search($entry$)
        if $residentialEntry$.isNewer($entry$) then
            $residentialEntry$.update($entry$)
    else
        if inputBuffer.contains($entry$) then
            var $inputEntry$ = inputBuffer.search($entry$)
            if $inputEntry$.isNewer($entry$) then
                $inputEntry$.update($entry$)
        else
            if cache.contains($entry$) then
                var $cacheEntry$ = cache.search($entry$)
                if $cacheEntry$.isNewer($entry$) then
                    $cacheEntry$.update($entry$)
            else
                if inputBuffer.slotsAvailable() then
                    inputBuffer.add($entry$)
```

Phase 2 – Active Process

```
for all $entry$ in inputBuffer do
    if cache.contains($entry$) then
        cache.markFree($entry$)
        inputBuffer.remove($entry$)
for all $entry$ in inputBuffer do
    if !cache.slotsAvailable() then
        var $randomEntry$ = cache.selectRandomEntry()
        if !outputBuffer.contains($randomEnty$) and outputBuffer.slotsAvailable() then
            outputBuffer.add($randomEntry$)
        cache.remove($randomEntry$)
    cache.add($entry$)
while outputBuffer.slotsAvailable() and !(cache - outputBuffer).isEmpty() do
    $randomEntry$ = cache.selectRandomEntry()
    if !outputBuffer.contains($randomEnty$) then
        outputBuffer.add($randomEntry$)
```

B.3. Random Implementation

Phase 1 – Message Reception

```
for all $entry$ in message do
    if inputBuffer.contains($entry$) then
        $inputEntry$ = inputBuffer.search($entry$)
        if $inputEntry$.isNewer($entry$) then
            $inputEntry$.update($entry$)
    else
        if cache.contains($entry$) then
            $cacheEntry$ = cache.search($entry$)
            if $cacheEntry$.isNewer($entry$) then
                $cacheEntry$.update($entry$)
        else
            if inputBuffer.slotsAvailable() then
                inputBuffer.add($entry$)
```

Phase 2 – Active Process

```
var $allItems$ = [cache, inputBuffer]
inputBuffer.clear()
cache.clear()
while cache.slotsAvailable() and !$allItems$.isEmpty() do
    $randomEntry$ = $allItems$.selectRandomEntry()
    cache.add($randomEntry$)
    $allItems$.delete($randomEntry$)
while outputBuffer.slotsAvailable() and !(cache - outputBuffer).isEmpty() do
    $randomEntry$ = cache.selectRandomEntry()
    if !outputBuffer.contains($randomEnty$) then
        outputBuffer.add($randomEntry$)
```

B.4. Entropy Implementation

```
struct CacheEntry = {entry, count[numCounter], totalCount}    ▷ numCounter = number of rounds
of history to store
var $currentRound$ = getRound() % numCounter
var $nextRound$ = ($currentRound$ + 1) % numCounter
Phase 1 – Message Reception
for all $entry$ in message do
    if cache.contains($entry$) then
        var $cacheEntry$ = cache.search($entry$)
        if $cacheEntry$.isNewer($entry$) then
            $cacheEntry$.update($entry$)
            $cacheEntry$.totalCount = 1
            $cacheEntry$.count = [0] * numCounter
            $cacheEntry$.count[$currentRound$] = 1
        else
            $cacheEntry$.totalCount += 1
            $cacheEntry$.count[$currentRound$] += 1
    else
        var $cacheEntry$ = {$entry$, [0] * numCounter, 0}    ▷ Create cacheEntry with zero counters
        $cacheEntry$.totalCount += 1
        $cacheEntry$.count[$currentRound$] += 1
        if !cache.slotsAvailable() then
            $mostReceivedEntry$ = cache.getByHighestCount()
            cache.delete($mostReceivedEntry$)
        cache.add($cacheEntry$)
Phase 2 – Active Process
var $numTx$ = min(outputBuffer.slotsAvailable(), cache.size())    ▷ Get number of items to transmit
outputBuffer.add(cache.getMultipleByLowestCount($numTx$)
for $entry$ in $cache$ do    ▷ Clear next Counter, update Total
    $entry$.totalCounter -= $entry$.counter[$nextRound$]
    $entry$.counter[$nextRound$] = 0
```

		$currentRound$	$nextRound$		$totalCount$
$cacheSize$	Token X	1	2	3	6
	Token Y	1	0	2	3
	⋮	⋮	⋮	⋮	⋮
	Token Z	1	0	0	1

Table B.1: Entropy/Receive count Structure

C Images

C.1. Algorithm Overview - Seeder

These figures show a network of 12 nodes in which one seeder distributes 500 tokens.

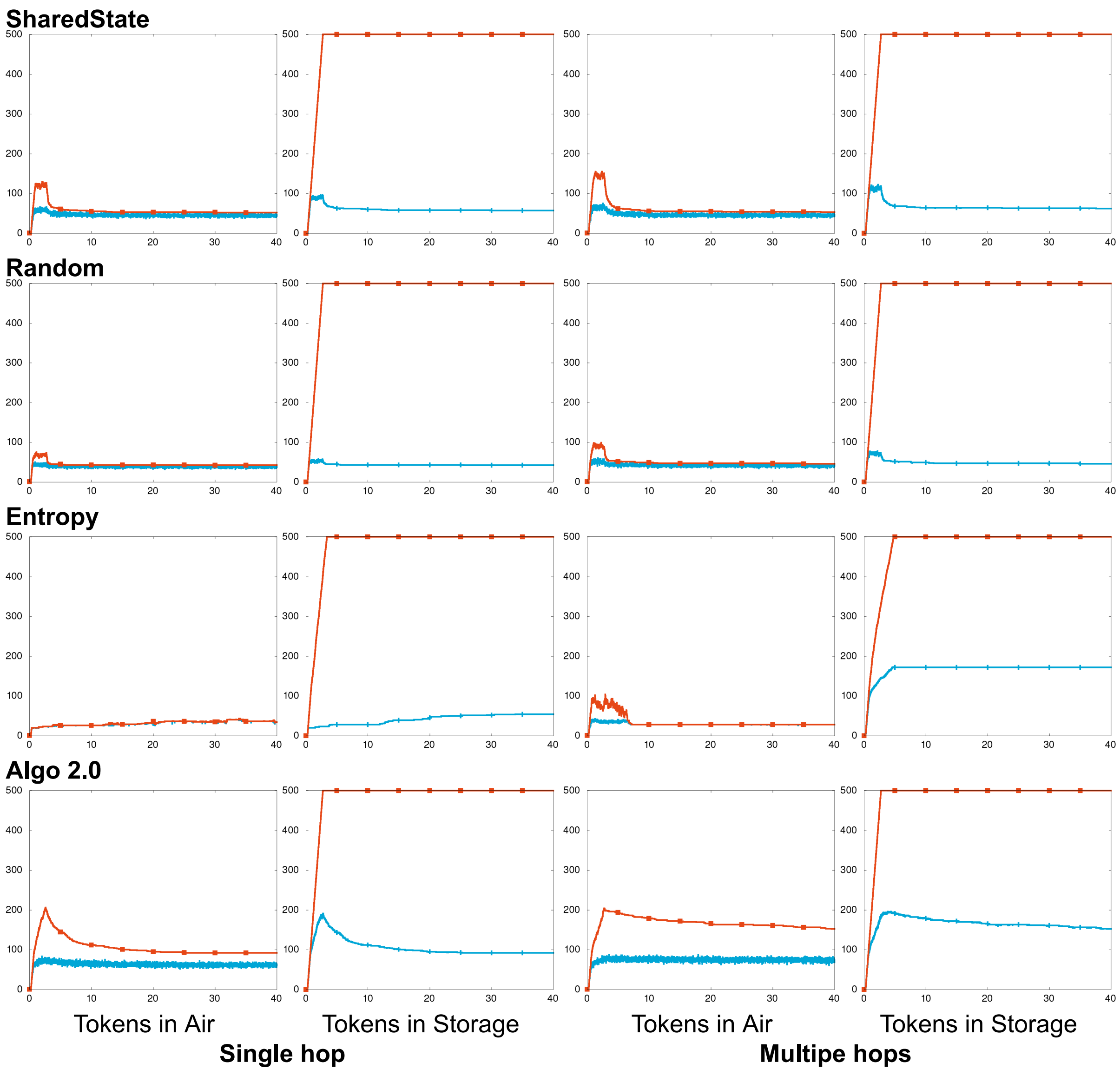


Figure C.1: Seeder without Filter

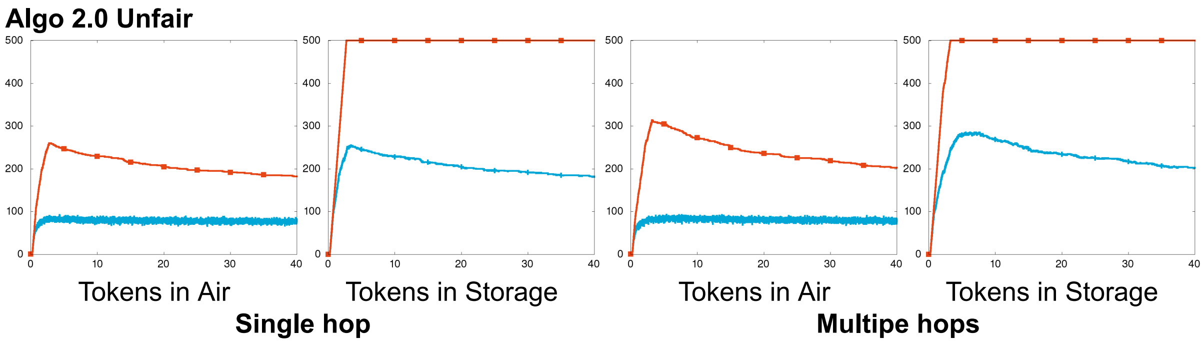


Figure C.2: Seeder with Unfair filter

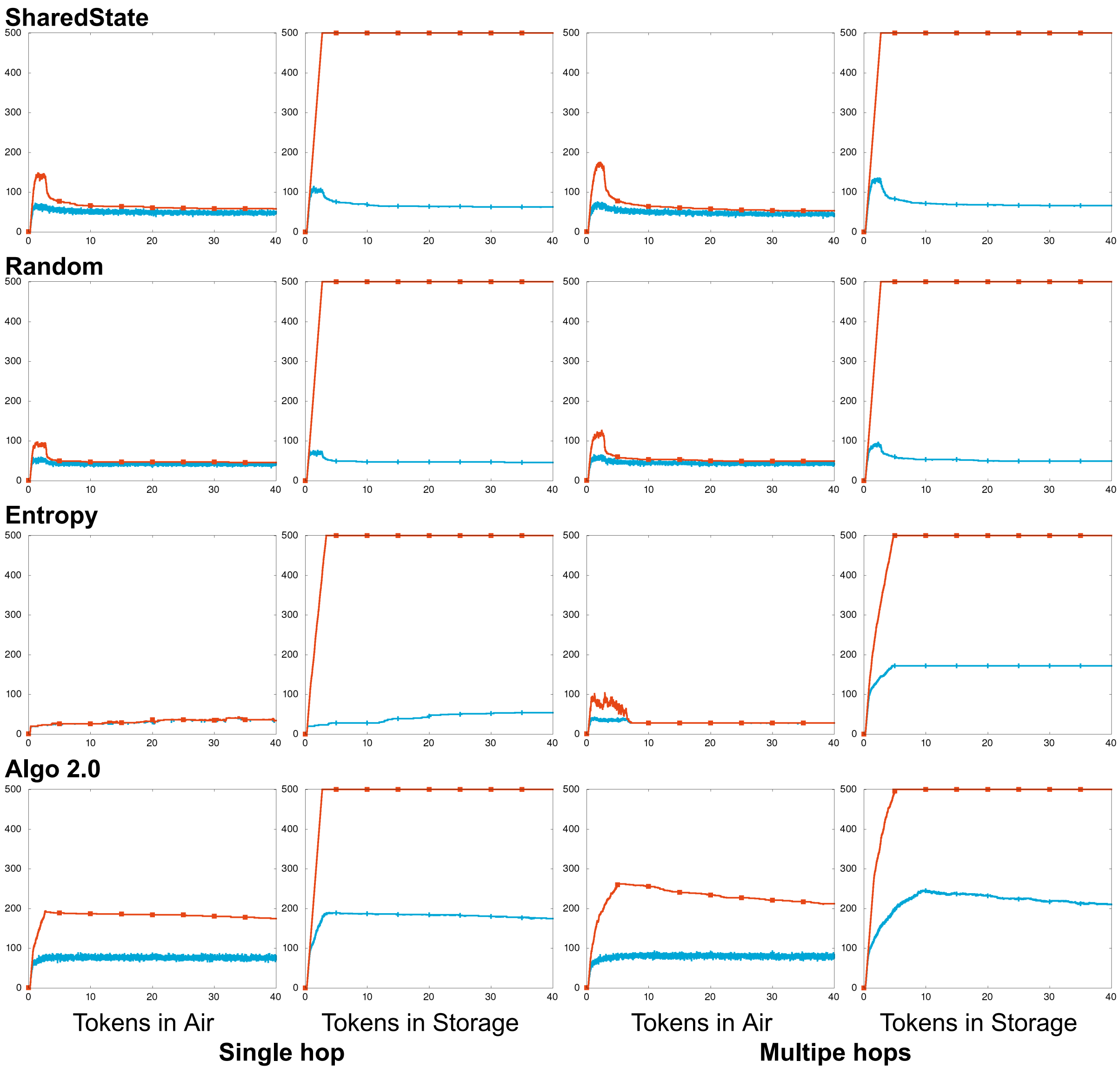


Figure C.3: Seeder with Filter

C.2. Algorithm Overview - All Unique

These figures show a network of 12 nodes, at Time=250 all nodes start distributing their own fully unique set of tokens.

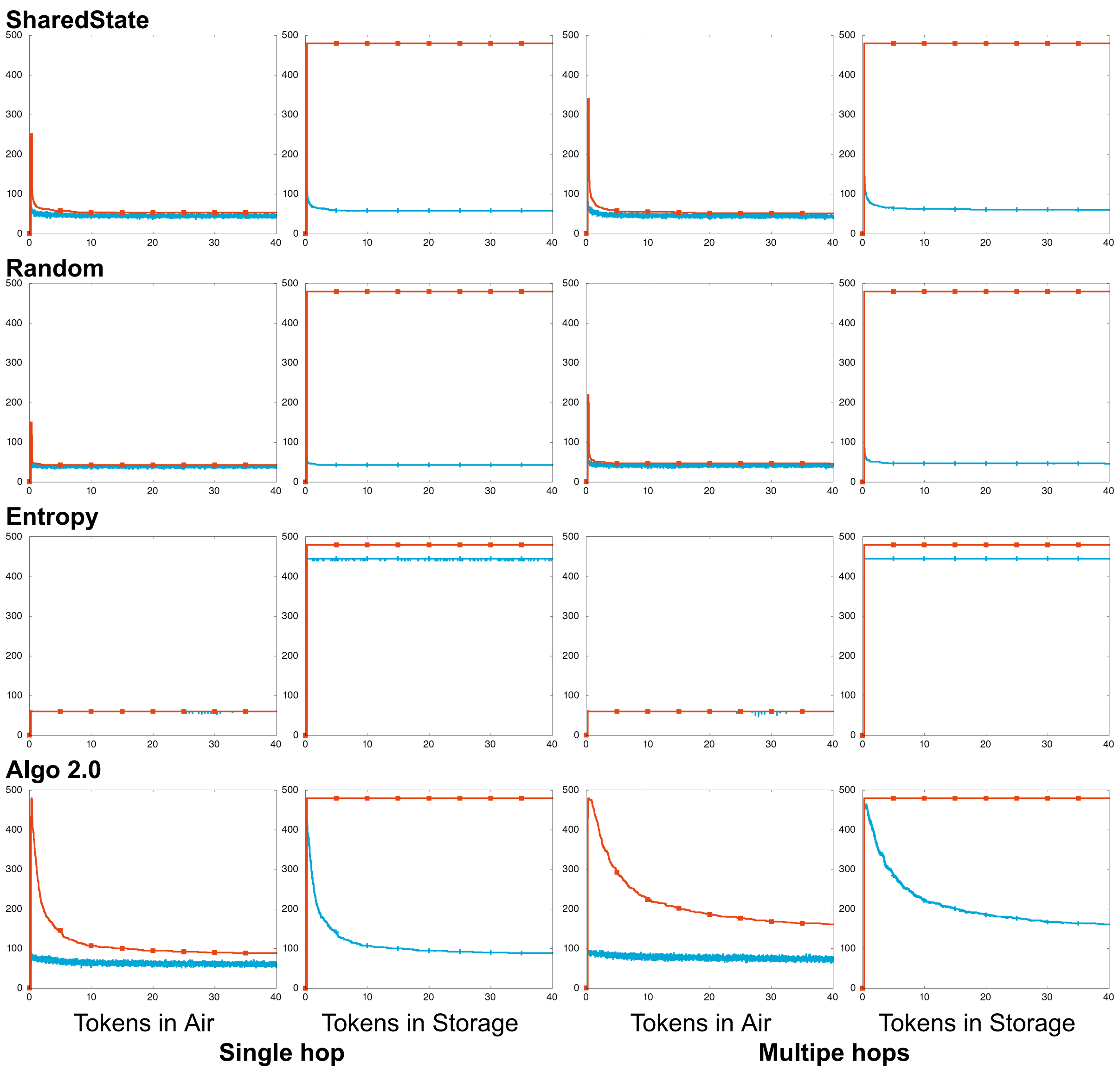


Figure C.4: All nodes unique tokens without filter

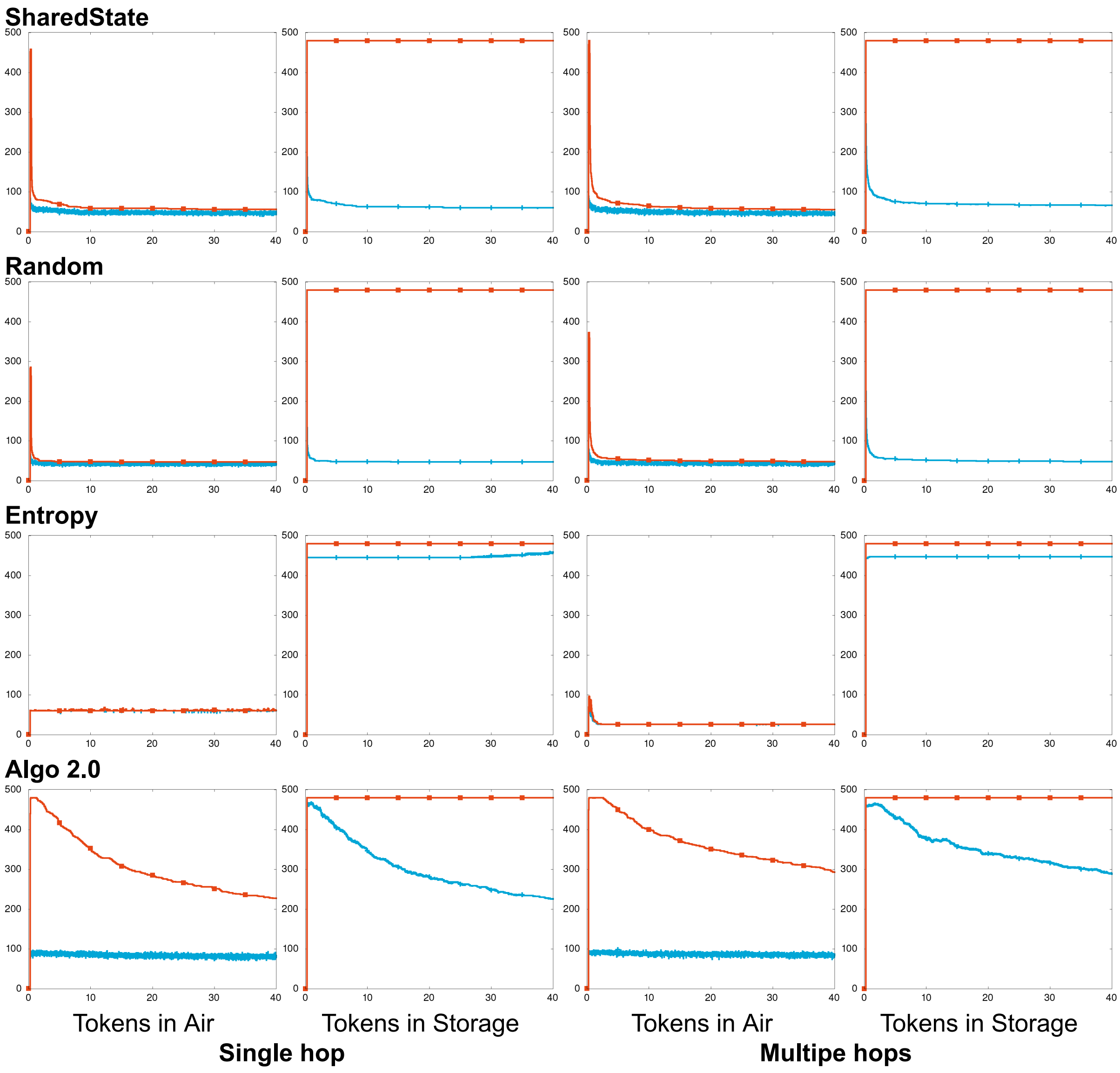


Figure C.5: All nodes unique tokens with filter

Bibliography

[1] Jul 2008. URL `https://status.aws.amazon.com/s3us-20080720.html`.

[2] S Acharia, M Franklin, S Zdonik, and R Alonso. Broadcast disks: Data management for asymmetric communications environments. *ACM SIGMOD-1995*, pages 199–210, 1995.

[3] Jeff Barr. Amazon s3 – the first trillion objects, 2008. URL `https://aws.amazon.com/blogs/aws/amazon-s3-the-first-trillion-objects/`.

[4] S. Crisostomo, U. Schilcher, C. Bettstetter, and J. Barros. Analysis of probabilistic flooding: How do we choose the right coin? In *2009 IEEE International Conference on Communications*, pages 1–6, June 2009. doi: 10.1109/ICC.2009.5198745.

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294281. URL `http://doi.acm.org/10.1145/1323293.1294281`.

[6] Alan Demers, Dan Greene, Carl Houser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Systems Review*, 22(1):8–32, 1988.

[7] D. Gavidia and M. van Steen. A probabilistic replication and storage scheme for large wireless networks of small devices. In *2008 5th IEEE International Conference on Mobile Ad Hoc and Sensor Systems*, pages 469–476, Sept 2008. doi: 10.1109/MAHSS.2008.4660060.

[8] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer networks*, 53(13):2321–2339, 2009.

[9] P Levis, Thomas H. Clausen, Omprakash Gnawali, Jonathan Hui, and JeongGil Ko. The Trickle Algorithm. RFC 6206, March 2011. URL `https://rfc-editor.org/rfc/rfc6206.txt`.

[10] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks.

[11] H Miranda, S Leggio, L Rodrigues, and K Raatikainen. *Epidemic Dissemination for Probabilistic Data Storage*, volume 8, pages 124–144. IOS Press.

[12] Aminu Mohammed, Mohamed Ould-Khaoua, and Lewis Mackenzie. An efficient counter-based broadcast scheme for mobile ad hoc networks. In Katinka Wolter, editor, *Formal Methods and Stochastic Models for Performance Evaluation*, pages 275–283, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-75211-0.

[13] Alberto Montresor. Gossip and epidemic protocols. 2017.

[14] Ramsés Morales and Indranil Gupta. Avcol: Availability-aware information aggregation in large distributed systems under uncollaborative behavior. *Computer Networks*, 53(13):2360–2372, 2009.

[15] Ryan Newton, Arvind Arvind, and Matt Welsh. Building up to macroprogramming: An intermediate language for sensor networks. volume 2005, pages 37– 44, 05 2005. ISBN 0-7803-9201-9. doi: 10.1109/IPSN.2005.1440891.

[16] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 151–162. ACM, 1999.

[17] Sangho Oh, Jaewon Kang, and Marco Gruteser. Location-based flooding techniques for vehicular emergency messaging. In *2006 Third Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services*, pages 1–9. IEEE, 2006.

[18] Y. Sasson, D. Cavin, and A. Schiper. Probabilistic broadcast for flooding in wireless mobile ad hoc networks. In *2003 IEEE Wireless Communications and Networking, 2003. WCNC 2003.*, volume 2, pages 1124–1130 vol.2, March 2003. doi: 10.1109/WCNC.2003.1200529.

[19] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27 (3):379–423, July 1948. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1948.tb01338.x.

[20] Robbert Van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 6. Citeseer, 2008.

[21] Spyros Voulgaris and Maarten van Steen. VICINITY: A Pinch of Randomness Brings out the Structure. In David Hutchison, Takeo Kanade, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, David Eyers, Karsten Schwan, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, and Bernhard Steffen, editors, *14th International Middleware Conference (Middleware)*, volume LNCS-8275 of *Middleware 2013*, pages 21–40, Beijing, China, December 2013. Springer. doi: 10.1007/978-3-642-45065-5_2. URL `https://hal.inria.fr/hal-01480790`. Part 1: Distributed Protocols.

[22] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and systems Management*, 13 (2):197–217, 2005.

[23] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A neighborhood abstraction for sensor networks. In *Proceedings of the 2Nd International Conference on Mobile Systems, Applications, and Services*, MobiSys '04, pages 99–110, New York, NY, USA, 2004. ACM. ISBN 1-58113-793-1. doi: 10.1145/990064.990079. URL `http://doi.acm.org/10.1145/990064.990079`.