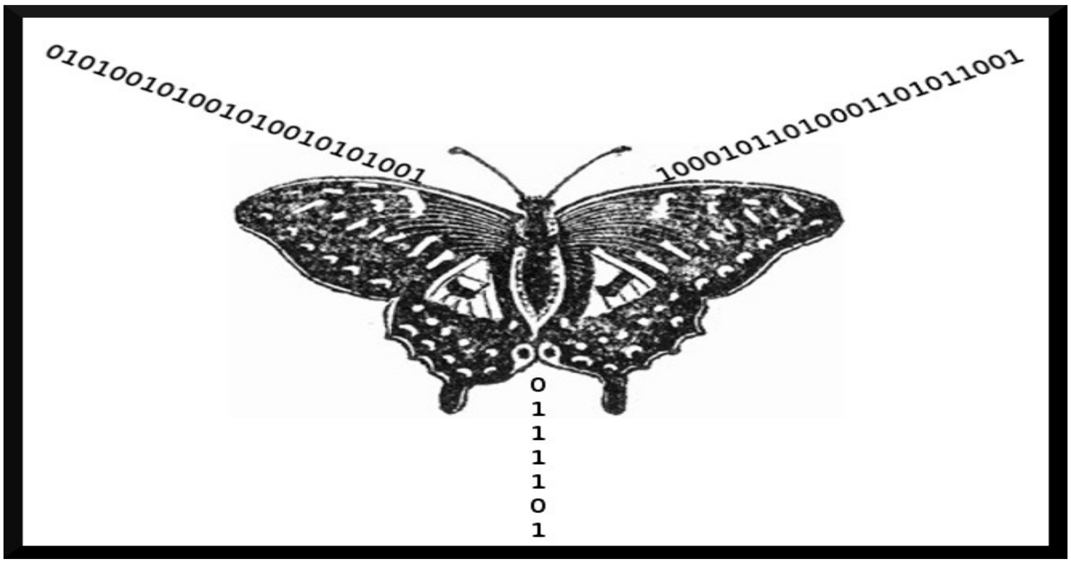
CHALMERS



# Session Management Algorithms
for Solving *K*-Token Dissemination Using Network Coding

Master of Science Thesis in the Programme Computer Systems and Networks

GUILLERMO BARREDO GARCÍA

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

Session Management Algorithms
for Solving *K*-Token Dissemination Using Network Coding

Guillermo Barredo García,

Examiner: Elad Michael Schiller

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Elad Michael Schiller, for giving me the opportunity to carry out this project. Without his encouragement, persistence, patience and guidance, this project would not have materialized.

I am also particularly grateful for the assistance provided by Iosif Salem whose constructive comments and suggestions throughout the realization of this thesis have been an enormous help to me.

Last but not least, I would like to thank my parents for supporting and encouraging me through my entire life.

Abstract

In this thesis report we consider k-token dissemination algorithms that makes use of *network coding*. The *k-token dissemination* problem consists of propagating a total number of k tokens to all nodes in the network. The tokens are distributed between one or more nodes before the algorithm is executed, and the final goal is that all nodes must eventually have the same set of k tokens.

Network coding is a recent technique that, implemented in a proper way, helps to save bandwidth and improves the speed of distributed computation. The network model consists of a network that can change completely from round to round, therefore the nodes do not know anything about their neighbours. Moreover, when a node broadcasts a message, it does not know which are the receivers, thus, it is not possible for the nodes to know which are the tokens that their neighbours need. By using *network coding*, the time needed to achieve the final goal (all nodes possess the same k tokens) is drastically reduced in comparison to a simple random forwarding algorithm, which, as its name infers, randomly broadcasts the tokens possessed by a node.

We study the *session management* problem, and present two algorithms to solve it. The problem consists of limiting the total number of sessions that concurrently coexist in the system. In the context of dissemination problems and network coding, we consider a session as an index according to which the information is codded by the session initiator. Since in dynamic networks nodes can crash and recover, we wish to allow sessions to accomplish their tasks, while limiting the amount of overall system resources in use.

We propose solutions for the session management problem, and by that facilitate the solution of the k-token dissemination problem in dynamic networks. By solving the *session management problem*, we can also solve the k-token dissemination problem in a more robust way, in the sense that the algorithms can deal with several types of failures, such as, *crashes* and *crashes-recoveries.*

Contents

1 Introduction

In existing computer networks, information is transmitted from the sender to the receiver through a set of intermediary nodes, which are responsible for forwarding the data in order to deliver it to the final destination. Normally, information received by intermediary nodes is stored and forwarded, this method is known as *store-and-forward*. In general, computer networks rely on routing schemes which allow the nodes in the network to select the right destination when a packet needs to be sent or forwarded.

A recent technique, known as *Network Coding*, breaks with the traditional paradigm of routing, in the sense that the packets are no longer needed to be treated as untouchable atomic packets since *Network Coding* permits the packets to be mixed with the aim of saving bandwidth. Therefore, when using *Network Coding*, the intermediary nodes have a more important task than merely acting as switches that receive information from an input link and then relay that information to an output link or set of output links. *Network Coding* is based on that, from the information-theoretic point of view, there is no reason to not use the intermediary nodes as encoders [3].

Network Coding has had a great impact in several areas of research such as networking, coding theory, complexity theory, cryptography, etc. due to its vast application potential [8]. It has been developed in various directions, and new and different applications continue to emerge [19] [7] [20] [18].

In this report, one of the algorithms introduced by Haeupler et al. [15] is studied. This algorithm solves the problem of k-token dissemination in dynamic networks by means of *Network Coding*. The problem is defined as follows: initially there are a total number of k tokens in the network, some/all of these tokens are held by one or more nodes. The main goal is that at the end of the execution of the algorithm all participating (correct) nodes must possess the same set of k tokens.

The network model that this algorithm is intended for, represents many modern networks where the topology of the network is changing constantly. The topology of the network may change totally from round to round, hence it offers a challenging scenario, since the nodes in the network do not know neither which are their current neighbours nor which neighbours they will have in the next round. This leads to a model where the only knowledge that nodes have is the tokens that they have received so far.

In the algorithm showed by Haeupler et al., faulty nodes are not considered. For that reason, we present a *session management problem.*

1.1 Related Work

Haeupler et al. [15] does not consider how to know that all nodes in the network possess the same knowledge or not. The authors of the studied algorithm assume that after a finite number of rounds, all nodes will gather the same information, but there is no mechanism implemented in the algorithm that can actually verify this. We provide two different mechanisms that allow the nodes to know when all nodes in the network possess the same set of coded tokens and plaint-text tokens.

Haeupler et al. [15] does not consider node failure. It is assumed that both, the nodes responsible for starting sending coded tokens and the rest of the nodes, will not crash throughout the execution of the algorithm. We present two algorithms that contain a failure detection mechanism that allows the nodes in the network to sense and deal different types of failures failures, i.e. crash-stop, crash-recovery and crash-reboot.

Conan et al. [9] introduce a distributed failure detector which periodically provides to all (correct) nodes in the network, a list of processes or nodes that are suspected to be unreachable. This way the (correct) nodes are able to know whether there has been a failure, a partition or a disconnection in the network. The failure detector used by our algorithms makes use of counters instead of sending a list of suspected unreachable processes. One of those counters is the *Incarnation number* counter, which, through the execution of the algorithm, can reach the maximum value of the data-type that defines it. This leads to the *Wrapping Around Problem* [16, 10, 4, 11], which, along with its solution, is presented in the following sections.

On the other hand, if failures are allowed in the system, another problem emerge: *work disruption*. This issue arises when a node that crashed, recovers and interrupts the work done by another node. As a result , we introduce the concept of *stable leader election*, which was first proposed by Aguilera et al. in [2].

The *Greedy-forward* algorithm [15] considers one leader or identified node at a time. This identified node is the node responsible for starting sending network coding packets. Since in our system, nodes can fail, letting a single node send coding packets at a time can lead to a noticeable inefficient performance. Consequently, one of our presented algorithm permits to have more than one identified node in the system at a time, so, in case that the identified node, which is currently carrying out the network coding phase, crashes, another identified node can take over.

1.2 Our Contribution

We provide two solutions for knowing when all nodes in the network have the same information. The first mechanism is used by the nodes to check whether they have the same set of tokens or not. On the other hand, the second mechanism helps the nodes to figure out whether they posses the same information about the coded tokens belonging to a session or not. A solution to the *wrapping around* problem, which slightly differs from other previously presented solutions, is also given.

Our main contribution is the *session management* problem and the algorithm that solves it. The problem itself consists of limiting the amount of overall system resources in use, in a dynamic network where nodes can crash and recover. Solving the *session management* problem will serve us as a channel to resolve the initial *k-token dissemination* problem. In addition, the *session management* algorithm, presented in this work, possesses extra properties compared to the studied algorithm, such as: *self-organization and-recovery* [1, 6]. These properties provide the algorithm with some extra robustness, which results to be essential in dynamic networks.

1.3 Structure of the report

The remainder of this report is organized as follows. In the second section, we show the background of this report together with our contributions. The third section contains a detailed description of the system settings is given. The next section is *Network Coding*, where a brief introduction to *network coding* and *random network coding* is presented. In addition, this third section also includes some applications that employ the *network coding* technique. In the fourth section, we define the problem that our algorithm needs to solve. Following, in the fifth section, we present basic problems that are associated to the k-token dissemination and session management problem, along with their solutions. The next two sections present, respectively, the design, explanation, lemmas, and proofs of two approaches to solve the studied problems. Finally, the last section of the report consists of the conclusions and advices for future work.

2 Background

Haeupler et al. [15] show how to make use of *network coding* in order to improve the performance of distributed computation in a dynamic network model, which is previously presented by Kuhn, Lynch and Oshman [21].

The dynamic network model has very specific and restrictive system settings which make this model representative of the highly dynamic and non-converging nature of many modern networks, e.g., Wireless Sensor Networks (WSN). These system settings consist of allowing the network to change completely in every round, but it is subject to the constrain that the network must remain always connected, i.e. the topology might change in every round but this changes can not end up with two or more sets of nodes that are not connected between them. In each (synchronize) round, each node selects a message from a pool of messages, and broadcasts it to its neighbours for that particular round. In addition, the sender of the message does not know who are going to be its recipients, and therefore, it does not know whether any of its neighbours already has the message that it is going to broadcast or not. The fact that the broadcast is "anonymous", makes this problem particularly challenging.

Along with the dynamic network model, Kuhn et al. [21] show how to solve the problem of k-token dissemination in such model. The problem consists of disseminating k tokens, in such a way that eventually all nodes in the network possess the same set of tokens. The proposed approach to solve the problem is called, *token forwarding*, and it is probably the most "natural" approach. It consists of broadcasting a token during $O(n)$ rounds, so after $O(nk)$ rounds all the tokens are disseminated. Furthermore, Kuhn et al. provide a more general lower bound $\Omega(n\ log\ k)$ that applies even if the algorithm is operated under a centralized control.

On the other hand, Haeupler et al. [15] show that this lower bound cease to hold when tokens can be broadcast together. This is done by using *network coding*, which allows to send out random linear combinations of tokens. If the size of a token is $O(log\ n)$, then in order to solve the k-token dissemination problem the algorithm will need $O(kn/log\ n)$ time to finish. This bound clearly outperforms the $O(nk)$ provided by Kuhn et al. In addition, the authors of [15] show that the greater the size of the token is, the faster can be disseminated. They state that using *network coding*, k tokens can be disseminated in $O(k(n\ log\ n)/d)$ time. Therefore, if the size of the token is equal to $O(n\ log\ n)$, the k-tokens will be disseminated in $O(k)$ time which also outperforms the lower bound $\Omega(n\ log\ k)$ associated to token-forwarding algorithms.

Both papers, [15] and [21], talk about T-stable networks, where the topology of the network does not change during a T-interval of time. If the network is stable for a certain period of time T, then it is possible to use other algorithms that improve the running time. For instance, according to [21], the complexity of the algorithms used for this kind of networks is $O(nk/T + n)$ time. In contrast, Haeupler et al. [15] show that *network coding*, instead of achieving a factor-T speed-up, can achieve a factor-T^2 speed-up. In this thesis, the described algorithm has been designed for networks that change in every round. Networks that are T-stable are outside the scope of this work. In case, the network only changes once every T rounds, then extra features should be added to the algorithm in order to make it more efficient.

2.1 K-indexing

K-indexing is one of keys to solve in an efficient way the k-token dissemination problem with *network coding.* Every coded packet has associated a coefficient vector, that eventually will be used in order to decode the k-linearly-independent coded tokens. All nodes in the network must agree on which index is associated to which token, otherwise there will be inconsistencies when doing Gaussian Elimination[1]. When talking about index, we refer to the basis vector that is associated to each token, so, for instance, the basis vector associated to the first token should be $\langle 0, ..., 0, 1\rangle$, to the second token $\langle 0, ..., 1, 0\rangle$ and so on. The size of this vector will be equal to the number of tokens that need to be coded.

Once all nodes in the network agree on what token has what index, then *network coding* can be used to disseminate such tokens. The problem is how to make that all nodes agree on the same indexes for the same nodes. Haeupler et al. show two different approaches:

1. Naive solution: All nodes give unique IDs of size $O(log\ n)$ to their tokens. Then they broadcast repeatedly the smallest $\Omega(b/log\ n)$ tokens they have heard about, where b is the size of the message. After n rounds of flooding, all nodes will have the unique IDs of the $(b/logn)$ smallest tokens, and therefore they will be able to agree on the indexes associated to these tokens. But this approach is only a $(log\ n/d)$ faster than the forwarding algorithm which does not make use of *network coding.* This is because the process of agreeing on the smallest tokens is repeated $k(log\ n/b)$ times, to this it is necessary to add the $O(n)$ time[2]

[1]A extended explanation of the reason why Gaussian Elimination is needed is given later on this report.

[2]With high probability.

that *network coding* phase takes. Consequently, the total time required to disseminate k tokens is $O(nklog\ n/b)$ or $O((log\ n/d) * (nkd/b))$.

2. Gathering tokens: A more efficient solution proposed by Haeupler et al., where instead of making all nodes to participate in the indexing problem, tokens are gathered in a single node which is the responsible for assigning the indexes to them. In order to propagate and gather the tokens, Haeupler et al. make use of a simple random forwarding algorithm, which turns out to be the same as the one used for disseminating k-tokens when *network coding* is not exploited. The authors explain in the paper, how efficient this algorithm is at the beginning, but as the time passes, nodes receive more and more packets that they have already received. The expected time to disseminate all the tokens using this technique is $O(nkd/b)$. Hence, it needs to be combined with *network coding* in order to achieve a noticeable improvement in the expected time. Haeupler et al. proposes to take advantage of the efficiency of the random forwarding algorithm in its early stage in order for a certain node in the network to gather an enough amount of tokens, and later send the gathered tokens using *network coding*.

2.2 Leader Election

When treating with dynamic networks, generally the methods and algorithms used to solve problems in conventional distributed systems are not applicable. One clear example of this is the leader election algorithm, where all correct nodes in the network must eventually agree on the same leader which is responsible of carrying out a specific task.

According to Haeupler et al.'s [15] *greedy-forward* algorithm, once a node has gathered b^2/d tokens, this node is identified as a leader, and it is the responsible for broadcasting the coded packets. The process of electing a leader will be finish within at most $O(n)$ rounds if no other node proposes itself as a leader. It can be the case, that two nodes are able to gather b^2/d tokens and thus they will have to "fight" for the leadership which will be decided with respect to, for instance, the greatest lexicographic ID. This will lead to a total of $O(2n)$ rounds (in the worst-case scenario), in order for all nodes to agree on the same leader. In fact, there is not a leader election phase as such, but once a node reaches the threshold of minimum number of tokens needed to start the *network coding* phase, it, indeed, indexes the tokens and starts sending random linear combinations of them. Therefore, as the reader may notice, one node can disrupt all the previous work done from another node with greater ID.

On the other hand, [15] does not deal with failures. What happens when the leader goes down while sending random linear combinations of tokens? How do the other nodes realize that the leader went down? What happens if an old leader after going down "wakes up" again and it turns out that there is already a node responsible for the *network coding* phase?, etc. These are open questions that will be answered throughout this report.

In addition, Haeupler et al. base the good performance of its algorithm on the synchronization feature of the network and the high probability of finishing the *network coding* phase in $O(n)$ rounds. But in a more realistic scenario, where networks are not completely synchronized and where not all nodes are able to decode all the tokens after $O(n)$ rounds, problems may arise. For example, suppose that after $O(n)$ rounds sending random linear combinations of packets, the leader stops with the *network coding* phase, but one or more nodes in the network have not received all the necessary linear independent vectors in order to decode the packets. Moreover, as stated in the paper, once the *network coding* phase has finished, all broadcast tokens are removed from consideration. Consequently, some of the k initial tokens that need to be disseminated to all tokens in the network will never be possessed by at least one node and hence, it will lead to the impossibility of solving the k-token dissemination problem.

2.2.1 Stable Leader Election

One of the first conclusions that can be derived from the text above is that once a node has already started the *network coding* phase, we do not want another node with greater ID to disrupt all the work done by such node. There are two case scenarios where this can occur. The first one is when two nodes reach the threshold of minimum number of tokens needed to start the *network coding* phase, in an interval no greater than $n - 1$, where n is the total number of nodes in the network. In addition, none of the two nodes has previously received a coded packet, since once a coded packet is received, the node will continue with the *network coding* phase started by another node. The second case scenario has to do with the disconnection and re-connection of an old leader, i.e. a node that started the *network coding* phase but it disconnected before finishing. If this "old" leader re-connects when there is a current leader, and the reconnected leader has greater ID than the current one, then it will disrupt all the work done by the newer leader.

For that reason, it is necessary to introduce the notion of *stable leader election* [2]. Aguilera et al. propose three different algorithms that are intended to provide stability to the leader. A leader election algorithm is said to be stable, if once a leader is elected, it remains being the leader until it disconnects or crashes, independently of the behaviour of other nodes. The model showed in [2] consists of a distributed system which is partially synchronous, processes have a drift-free clock and there is an upper bound B on the time within a process executes a step. Furthermore, Aguilera et al. claim that the algorithms presented in the paper are self-stabilizing, a feature that our algorithms also have. Even though, stable leader election algorithm introduced in [2] seems a decent solution for the problem mentioned above, we need another algorithm that is more suitable for solving the problem here presented. In particular, a solution that could be easier to integrate with the *network coding* and random forwarding phases of the original algorithm. The idea is to use a *piggybacking* technique that allows the algorithm to work with the *network coding*/ random forwarding and leader election in a coexisting way.

2.3 Gaussian Elimination

Since, in order to encode packets during the *network coding* phase the packets are randomly linearly combined, an algorithm for solving linear equations is needed. As mentioned earlier, the packets sent during the *network coding* phase are composed by a coefficient vector and a vector with the combination of the tokens that the coefficient vector indicates. For instance, if the coefficient vector, c, is equal to $\langle 1, 0, 1\rangle$, that means that the vector that contains the combination of tokens will have a combination of the first token and the third one. Note that if a node has only received this packet, it will not be able to decode the tokens, since it does not have enough number of *equations* to solve the system. Thus, a node will be able to retrieve useful information about one of these tokens, if more linear independent packets are received. The algorithm used to carry out this task is Gaussian Elimination algorithm.

2.4 Failure Detection

The algorithms that have been designed through the life cycle of this project make use of a failure detection technique in order to provide correct operation.

2.4.1 Failure Models

In a highly dynamic system, and in any system in general, mainly five types of failures can take place:

1. Crashes. The processor or, in this case, a node stops functioning and it never starts again.

2. Omissions. Omission errors take place when one or more actions that are carried out by a node fail.

3. Time errors. These type of failures take place when responses arrive outside the specified time interval.

4. Crashes and Recoveries. A node halts, but eventually recovers.

5. Arbitrary or Byzantine. A node may fail in an arbitrary way, including sending arbitrary data its neighbours in the network.

2.4.2 Failure Detector

Once we know which type of failures can occur in our system, it should be stated which failures our algorithm will deal with.

Looking at the condition of the nodes, the same (crash) failure is treated in two different ways:

1. Regular node. If a regular node crashes, the rest of the nodes will not notice it, since it is just a node among many, whose failure will not affect the performance of the algorithm. Note that all the tokens that the crashed node possessed before crashing and which were not possessed by any other node, will disappear forever from the system.

2. Node responsible for having started the *Network Coding Phase*. If a node that is responsible for broadcasting random linear combinations of tokens goes down before sending enough linear independent vectors in order for the rest of the nodes to be able to decode all its tokens, it must be known by the rest of the tokens in the network. Otherwise, the rest of the nodes will be waiting until they have enough packets to decode the coded tokens, but since the leader is down, these packets will never arrive and therefore, the awaiting nodes will not move forward.

Moreover, Conan et al. [9] present in their paper an architecture of local and distributed detectors for mobile networks. These detectors give the network the ability to detect failures, disconnections and partitions. Partitions are not allowed in our network model, but in case this extra feature wanted to be added, we recommend the reader to have a look at [9]. Conan et al. propose an eventually perfect unreliable partition detector that exploits information provided by a failure detector and disconnection detector in order to identify partitions. Contrary to the failure detector given on our report, where no list of processes is kept, the unreliable failure detector shown in [9], regularly provides, for each process p in the network, a list of processes suspected to be unreachable.

2.5 Wrapping Around Problem

The two different approaches presented in this report use an *incarnation number* counter in order to differentiate between correct nodes and nodes that just joined the network or have crashed and recovered. Since the size of the data type used for the counters is limited, it is necessary to find a solution for when the bound value is increased by one.

Herman et al. [16] propose two different approaches to deal with this problem. They have a bounded global clock with domain $[0, L]$. When the event where the clock rolls over from L to zero takes place, it disrupts the converge-to-max protocol presented on the paper [16]. The two techniques showed by Herman et al. are obatined from the literature of self-stabilizing phase-clocks and they consist of:

1. Redefining comparison of clock values in the clock protocol to behave modulo $L + 1$ [10].

2. Letting the event of a clock that reached L initiate a system reset, after which all clocks begin from zero [4].

In [11], the authors also deal with the *wrapping around problem* and make use of wrap around flags that indicate when a process has wrapped its counter. Dolev et al. state that even though the size of the counter is 64-bits and therefore practically infinite, this assumption is not valid within the scope of self-stabilization. The reason why it does not hold is because a single transient-fault may lead a counter to reach the above mentioned large number at once, and therefore it will disrupt the whole self-stabilizing algorithm avoiding it to reach a safe configuration.

2.6 Estimating N

In this project we assume that we have a good estimation of N, where N is an upper bound on the number of nodes in the network. Bellow we present three different approaches of how this problem can be solved.

In [15], the authors propose an algorithm for the case of n-token dissemination, where the nodes ignore the value of n. It consists of starting guessing an upper bound $n = 2$, and then the nodes count the number of node IDs using token dissemination. If there is a failure, i.e., the number of received tokens from different nodes exceeds n, the n estimation is doubled and the algorithm restarted. This process is repeated every time a failure is detected. A similar technique can be used together with the proposed algorithm in order to make it more realistic and efficient.

Dolev et al. [12] give a solution to this problem, in a self-stabilizing group communication system for ad-hoc networks, in which the algorithm needs to know N in order to reach stabilization. The authors state that having a more accurate upper bound on n will ensure that the system will react in a faster way to addition/removal changes. They also indicate that estimating n will increase the number of short messages, which will lead to a trade-off between the time needed to estimate n and a special type of messages, called scouters.

In addition, Baldoni et al. [5] consider the problem of counting the number of nodes. The authors propose a more recent technique based on "energy transfer" to calculate the size of the network on anonymous dynamic network. They use the *energy-transfer* technique to deal with the dynamic environment and the absence of unique IDs. The solution proposed in [5] is leader based and where each node owns a fixed energy charge. A node discharges itself by exchanging at most half of its charge with its neighbours. Energy is not created or destroyed, i.e. after every round, the sum of energy contained in each node is the same. Nodes discharge themselves by exchanging at most half of their charge with their neighbours. On the other hand, the leader is responsible for absorbing all the energy, and thus, it does not transfer its energy to its neighbours. In order to calculate the number of nodes in the network, the leader measures the energy received.

2.7 Our Contributions (in detail)

With respect to the related work, the following points describe our contributions:

1. **K-indexing.** In both algorithms presented on this document, the technique used to gather tokens is similar to the one used in the studied algorithm. Nodes randomly forward plain-text tokens until a *session* becomes stable. Note that there are no longer leaders but sessions and session creators. The creator of a session is the responsible for carrying out the indexing.

2. **Stable Leader Election.** The way of how a node is elected to start sending random linear combinations of tokens is completely different to the approach used in the studied algorithm. Instead of directly starting with the *network coding* phase once a node gathers enough number of tokens, a node will become eligible. And, if certain requirements are fulfilled, then it will create a session. Once all nodes in the network agree on this session, the session will become stable and the creator of the session will be allowed to start with the *network coding* phase.

3. **Piggybacking.** Piggybacking is the technique used in network transmission, and more specifically, in the network layer of the OSI model. It consists of adding the acknowledgements of previous received packets to the data frame, thus instead of sending a confirmation in an individual frame, i.e. a frame containing ACK information, the acknowledgement is appended with the payload of the next packet that the receiver will sent to the emitter. This practice saves bandwidth since it requires fewer frames [23].

   This technique allows us, for instance, to agree on the node responsible for broadcasting random linear combinations of tokens, while plain-text and coded packets are sent, without interrupting neither the network coding phase nor the random forwarding phase. The main advantage of using piggybacking is that the total number of packets used is reduced and hence, the efficiency of the algorithm is incremented.

4. **Gaussian Elimination.** Considering that the base q used to encode the packets is equal to 2, the general Gaussian Elimination algorithm can be simplified. In addition, a minor modification has been carried out in the algorithm in order for the nodes that have the same set of linear independent packets to detect that they have the same information.

   The algorithm is well known and therefore it will not be described in depth on this report. Basically, it consists of three main operations:

   I Find the lexicographically lowest row that has a pivot corresponding to the iteration of the algorithm. That means, in the first iteration of the algorithm, we look for a pivot in the first column, in the second iteration we look for a pivot in the second column and so forth. The general algorithm does not look for the lexicographically lowest row, instead, it uses the first row found that has a pivot. The reason why this lexicographical order has been implemented is for the nodes that have received the same set of coded packets to have the same matrix independently of the order in which they have received those packets. This will be explained with more detail later on the report.

   II Once the row with the pivot has been found, it must be placed in the correct row. If it turns out that the selected row is already placed in the correct row, nothing is done, otherwise, the positions of two rows are swapped.

   III XOR all the rows that have the same leading coefficient as the selected row, with the selected row itself. Hence, all rows that had the same leading coefficient but were not selected, end up with these coefficients equal to zero.

5. **Failure Detector.** The algorithms that will be shown and described later on this report, implement a detection failure technique that allows the nodes in the network to notice if the node responsible for sending coded packets has crashed or not. A proper explanation of how the failure detector is implemented will be given later on this document.

   On the other hand, nodes that crash and resume or reboot later are also tolerated. It might be the case, that a node, independently of its condition, crashes and eventually recovers being able to retrieve complete, partial or none of the tokens that has been sent while it was absent. It is assumed, that no node that has gone down will recover after a certain number of rounds, i.e. if a node crashes it has a certain time to recover, if this time is exceeded then it is assumed that this node will remain in a crashed state "forever".

   Our failure detector has two properties: Completeness and Accuracy. Completeness refers to the fact that every crashed node is eventually suspected by every correct node. And accuracy refers to the fact that no correct node is ever suspected. This will hold as long as no partition occurs in the network.

6. **Wrapping Around Problem.** Redefining comparison technique, first presented in [10] will be the one used to deal with the *wrapping around problem.* Moreover, a similar technique to the system reset introduced by Arora and Gouda in [4], can be used in the algorithm for the cases where two incarnation numbers are not comparable[3].

Finally, our main contributions are the *session management problem* together with two different approaches to solve it. The *session manager* is used to solve the k-token dissemination problem providing a robust solution that is capable of detecting certain failures. In addition, both algorithms presented on this paper are self-organizing and-recovering. A strong property that allows the system to eventually reach a legitimate state, independently of node failures.

[3]In this project, we assume that all incarnation numbers that exist in the system are comparable

3 System Settings

We consider a distributed system formed by nodes or processors: p_1,p_2,...,p_N, where N is a known upper-bound on the number of nodes. Nodes may *crash-stop*[4], *crash-reboot*[5] and *crash-resume*[6], but always respecting the upper-bound N. During the execution of the algorithm, processes that have not crashed are said to be correct. All nodes in the system have unique identifiers.

The nodes in the network may send and receive messages to and from their set of neighbours. The communication between two nodes p_i and p_j, where $i \neq j$, is modelled by a FIFO queue. Whenever a processor p_i broadcasts a message m to its neighbours, m is appended to all queues of all the neighbours of p_i. Whenever there is a pending message in the queue, this message is immediately processed by the node. For each neighbour, there exists a queue.

At every round, the network topology may change, however it is assumed that it is always connected. The network topology is defined by a connected undirected graph G. For every round r, any node in $G(r)$ can reach any other node that also belongs to $G(r)$. Partitions in the network are not allowed. It is assumed that there are no collisions in the system, hence for the sake of simplicity the nodes are connected by directed communication links. When communication takes place, nodes broadcast anonymous packets through these links to all their neighbours, where a packet sent from p_i to p_j will make use of the (directed) link l_{ij} and a packet sent form p_j to p_i will make use of the (directed) link l_{ji}. We assume that in every round all correct nodes broadcast only one packet to their neighbours, and since the network is always connected, all correct nodes expect to receive at least one packet. In addition, a packet sent by a node is received by all its correct neighbours, therefore the network is assumed to be free of packets losses. It is assumed that the local operations are insignificant and thus, they do not affect the communications in the network.

The system possesses the property of self-organization and-recovery [1, 6]. If any of the previously mentioned failures occurs in the system, the system should recover and exhibit a desired legal behaviour in a finite number of steps. The algorithm executed by the nodes in the system does not need to be initialized, since regardless the of its initial state, it eventually reaches a

[4]A node that crashes and stops is a node that after crashing will be never part of the network again.

[5]A node that crashes and reboots is a node that after crashing it may join again the network with an initial configuration state.

[6]A node that crashes and resumes is a node that after crashing it may join again the network with the same state as the state it had just before crashing.

correct state. Once the system has reached a correct state, it will remain in this correct state as long as no fault occurs.

Every node p_i executes an algorithm which is composed by a sequence of *steps*. A step may start several local computations, but ends with a single computation either *send* or *receive* of a packet.

4 Network Coding

In the following section, a more detailed description of *network coding* will be given, along with an introduction to some of its applications.

Network coding is a relatively new technique that can significantly improve the networks' throughput, efficiency and scalability. In the classical paradigm of routing, the packets are simply relayed when they are received, however, when using *network coding*, the nodes of a network gather a certain amount of packets and mix them for transmission.

In order to clearly show how *network coding* can outperform routing, a typical example, known as "the butterfly network" is described in the following paragraphs.

Suppose there is a total number of six nodes in a directed oriented graph, as shown in Figure 1. Where $S1$ and $S2$ are the sources of the network, i.e. the ones responsible for injecting information into the network, and $T1$ and $T2$ are the sinks. The capacity of each edge is one bit, thus the maximum data that one node can send to a neighbour in every time slot is only one bit. For this example, $A = 1$ and $B = 0$, are transmitted from $S1$ and $S2$ respectively.

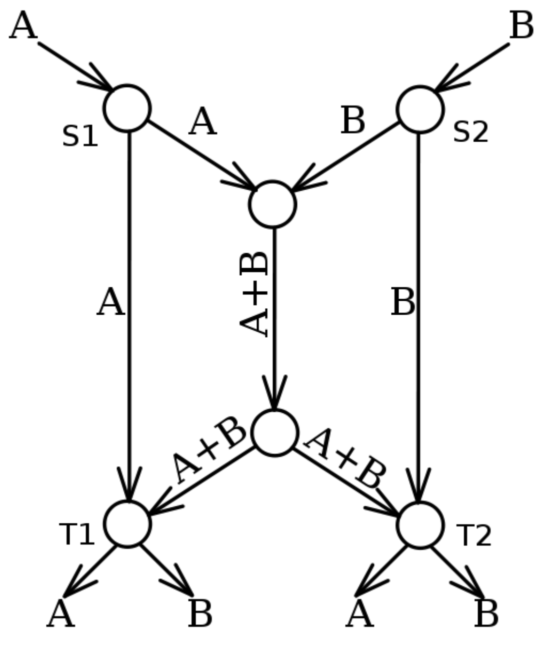


Figure 1: Butterfly network.

If classical routing is used, after the first round one of the central nodes will receive A and B, and the exterior nodes ($T1$ and $T2$) only A or B. In the next round, the central link will be able to carry only one value or bit, that will be either A or B. In case A is transmitted the left destination ($T1$) would receive A twice, and the same will happen to the right ($T2$) destination if B is sent. It is trivial to see that there is no routing scheme that can transmit both A and B simultaneously to the two final destinations.

On the other hand, if *network coding* is used, when the first intermediary node receives A and B will use the XOR operation and generate a bit that will be equal to $A \oplus B = 1$. When $T1$ receives $A\ XOR\ B$, it will simply use XOR (i.e. $A \oplus A \oplus B$) to retrieve B and $T2$ will similarly retrieve A. It is evident that when using routing, three more bits have to be sent in order for $T1$ and $T2$ to recover A and B, whereas *network coding* only needs to send 9 bits to achieve it. Therefore, 25% of bandwidth can be saved in this basic scenario by using a simple *network coding* scheme.

4.1 Random Network Coding

When the network is dynamic or the topology of the network is not known and some information need to be broadcast to all nodes in the network, *random network coding*[17] can be used to solve the problem in an efficient way. *Random network coding* is a powerful coding scheme which provides a close to optimal throughput. When *network coding* was first stated in [3], it was presented as a technique to achieve optimality. The scenarios mentioned by Ahlswede et al. describe networks containing sources and sinks. The rest of the nodes are considered as intermediate nodes, which act according to a predefined coding scheme.

On the other hand, *random network coding* is also a type of coding scheme which, in contrast to “regular” coding schemes, behaves as a decentralized algorithm. When *random network coding* is deployed in the network, all the nodes function in a same manner. Nodes transmit random linear combinations of the packets they receive. These packets are mainly divided in two parts, one is the coefficient part and the other the coded part. The coefficients are chosen from a *Galois* field. Once a node has received enough number of packets, it can decoded them and thus, retrieve the original information generated by the sources.

4.1.1 How fast does information spread?

Haeupler, introduces in [14] a simple projection analysis technique that shows how fast information spreads when *network coding* is used and therefore, how much time is needed for all the nodes in the network to be able to decode all coded tokens.

Haeupler states that the right way to look at the spreading of information is to look at the orthogonal complement of the coefficient subspaces. In [15] and [14] a definition of knowledge is given, which says that a node u knows about a coefficient vector $\overrightarrow{\mu} \in F_q$ if it has received a message with a coefficient vector $\overrightarrow{\mu}'$ that it is not orthogonal to $\overrightarrow{\mu}$. Note that the bigger the q is, the more the probability to learn something new.

Right after, in [15], the Haeupler et al. give a lemma that basically shows that any node that knows about $\overrightarrow{\mu}$ will pass that knowledge to their neighbours with probability at least $1 - 1/q$. Finally, it is proved that the *network coding* algorithm with $q \geq 2$ solves the k-indexed-broadcast problem in an always connected dynamic network with probability at least $1 - q^{-n}$ in time $O(n + k)$.

4.2 Applications

Despite *network coding* is an arguably recent technique, it has already been implemented and proved that it is useful in many different areas:

1. *Avalanche*: It is a research project carried out at Microsoft, which is claimed to provide a cost effective, scalable and very fast file distribution solution compare to existing *P2P* systems[13]. The authors of the project propose a solution for the well known problem of Peer-Assisted file delivery systems, where the last “rarest pieces” of a file are harder to obtain. This, at the end, will result in slower downloads. In order to fix the problem, *network coding* is used. Instead of simply distributing the blocks of the file, peers transmit linear combinations of the blocks they already hold together with a tag that indicates the parameters used in the combination. This approach clearly solves the problem, since a peer does need to find specific pieces; any subset of encoded pieces suffices.

2. *COPE* [19]: It is an architecture for wireless mesh networks that utilizes *network coding* to increase network throughput. The routers mix the information content in the packets before forwarding them. The example given in the paper to briefly explain how COPE works consists of three computers: Alice’s computer, Bob’s computer and a computer that acts as a relay. Alice wants to transmit a packet to Bob and Bob wants to do the same with Alice. If *network coding* is not used, the relay will receive one packet at a time(otherwise there will be a collision) and

forward it. Four transmissions will be needed in total in order for Alice and Bob to receive each others packets. However, if *network coding* is used; for instance, Alice will send first the packet and the relay will receive it, but instead of directly forward it, the relay will hold it for a certain period of time. During this period Bob will send his packet, and the central computer will only have to "XOR" the packets and forward the resulting packet. By using *network coding*, the number of transmissions is reduced from four to three. From this simple example, it can be also observed that using COPE leads to bandwidth savings.

3. *Spatial Buffer Multiplexing* [7]: It is a technique to reduce buffer utilization and delay. This approach makes use of a scheme where the intermediary nodes have no buffering capabilities for queuing transient packets (in contrast to traditional approaches). In spatial buffer multiplexing, the buffering and coding is implemented at the source and the authors of the paper claim that this will compensate for packet loss at any downstream buffer-less link.

4. *CTCP* [20]: It is a reliable transport protocol based on *network coding.* It incorporates same features of TCP such as reliability, congestion control and fairness but it additionally improves TCP's performance in lossy, interference-limited and/or dynamic networks. The authors affirm that the combination of TCP transport layer and *network coding* yields to performance gains in the presence of interference.

5. *AdapCode* [18]: It is a reliable data dissemination protocol that makes use of *network coding* in order to reduce the total amount of traffic during the process of code updates. The main idea behind *AdapCode* is that the *network coding* schemes change according to the quality of the links.

   Hou et al. state that the broadcast used when doing troubleshooting, which requires frequent upload of new code, must be fast, reliable and minimal in terms of network bandwidth consumed. *AdapCode* is intended for wireless sensor networks, it takes advantage of the fact that *network coding* reduces the total amount of traffic although increases the local computation, which is ideal for such kind of networks since the communication is slower and needs more energy compared to local computation.

   The *network coding* methodology used in [18] consists of the random combination of N coefficients and the computation of the linear combination of N packets. A node dynamically decides on N based on the number of neighbours it has. In other words, *AdapCode* adaptively decides on its coding scheme using the local knowledge of each node. In the paper, *AdapCode* is compared with *Deluge*, which is a state-of-the-art protocol used to propagate new code images, in TinyOS version 2. *Deluge* can disseminate data with 100% reliability at a speed of approximately 90 bytes per second. And according to Hou et al. the results of the comparison show that *AdapCode* uses less packets than Deluge to disseminate a image of the same size. An example of a code image of 1024 packets is given, which is sent by *AdapCode* with a reduction up to 40% of the total number of packets that are needed to disseminate the same image using *Deluge*.

In order to achieve 100% reliability, *AdapCode* makes use of Negative-ACK. Nevertheless, it is worth to mention that, in the paper, the authors do not address temporary node failures or reboots. In addition, in contrast to the network model that this report is based on, they consider that there is a single source of data, instead of having more than one source disseminating data at the same time. The reason why Hou et al. consider only one source is because normally the scenarios that are presented in wireless sensor networks, contain a single source which is responsible for broadcasting the packets.

5 Defining the problem

The following section provides a definition of the *k-token dissemination* problem, together with some of its most relevant key points. Moreover, the core problem of this thesis is presented: a *session management problem*. Solving this problem will help us to resolve the problem here studied, the k-token dissemination problem.

5.1 K-token Dissemination: Problem description

The problem that needs to be solved is based on the dynamic network model first proposed by Kuhn et al. [21] and on the *greedy-forward* algorithm given by Haeupler et al. [15]. As explained in section 2 (*Background and Our Contributions*), the problem that needs to be solved is the *k-token dissemination problem*, where there are a total number of k tokens distributed throughout the network and all nodes need to possess them eventually. *Network coding* can be used to solve the problem in a efficient way. However, it can not be directly used. First it is necessary that a node gathers enough number of tokens in order to start sending random linear combinations of tokens.

The algorithms described in this thesis are based on the algorithm provided by [15]. However, the k-token dissemination algorithm shown by [15] does not explain how to deal with some of the problems that raise in networks of such characteristics, such as: When do all nodes know that they have the same set of tokens?, When does the identified node, which is responsible for broadcasting up to b^2/d know that the rest of the nodes can decode all b^2/d tokens?, How do all nodes know whether the node responsible for broadcasting random linear combinations of tokens has crashed?, etc. Hence, we want our algorithm to be able to deal with the following problems:

- If the node responsible for starting sending random linear combinations of tokens crashes, all correct nodes should eventually notice it.
- All nodes must agree on the same elected node before this starts broadcasting random linear combinations of tokens.
- The system should be self-organizing and-recovering [1, 6].
- If a node that has crashed suddenly recovers, it shall not disrupt the self-organizing and-recovering condition. In other words, the node may make the system to enter in a non-legitimate state, but after a finite number of rounds the system will exhibit a desired legal behaviour.

5.2 Session Management Problem

In the context of the k-token dissemination problem, a session represents a node that is able to start with the network coding phase because it has reached the minimum number of tokens needed to do so. When a node reaches such threshold, we say that this node is eligible. In the moment that a session is created, every node in the network will eventually notice this event and allocate buffer space to store the information associated to the session. One of the reasons why the sessions must be handled is that the set of eligible nodes is unknown and thus, if every eligible node creates a session, all nodes need to allocate buffer space to store the information associated to each single session. For instance, in the k-token dissemination problem each node stores all the coded packets (coefficients and coded tokens) associated to a session in order to decode the tokens. Consequently, in a system where the nodes have limited resources, we do not want all (eligible) nodes in the network, which can start with the network coding phase, to immediately create a session. Therefore, it is needed to limit the total number of sessions existing in the system at the same time to a bounded number, S, of sessions. In this document, we propose two different approaches, one for when $S = 1$ and the other one for a general S. The advantages and disadvantages of using one or another approach will be shown later on this report.

We define a *session* as an expirable lease that grants a node the permission to perform a "dissemination" task. In addition, only eligible nodes can create a *session*. The session manager bounds the number, S, of sessions that may concurrently exist in the system. The exact problem definition considers three main events:

1. **Session Creation**. There are two preconditions that must be fulfilled before this event is triggered. The first precondition is that a node must be eligible. In the context of the k-token dissemination problem, in order for a node to become eligible, it has to gather the minimum number of tokens needed to start sending coded tokens. The second precondition is that the limit of current sessions in the system is not exceeded if the eligible node creates a session. If a node satisfies these two preconditions, then it is allowed to create a session. In the context of k-token dissemination problem, once a node has enough tokens, if the limit of current sessions in the system has not been reached, then the node can immediately create a session. Once a session is created, the *session creation* event is triggered. Both the creator of a session and the rest of the nodes must allocate buffer space for the information associated to the created session.

Finally, the creator of the session needs to wait for the session to become stable[7] in order to start sending random linear combinations of tokens. Meanwhile, the rest of the nodes in the network must verify that the creator of a session stays eligible and connected. Furthermore, there can only be one creator of a session performing its task at a time.

2. **Session Termination**. A session should terminate, once all correct nodes have received all the information associated to the task. In the context of k-token dissemination problem, this means that a node responsible for a session has to be sure that all nodes in the network are able to decode all tokens associated to that session before terminating it.

   In addition, a node, which is not responsible for a session, can not deliberately end such session, unless the session has expired. Once the node responsible for a created session is completely sure that all nodes in the network have received the information associated to the task, then it must end such session. As soon as a session is ended by its creator, the rest of the nodes in the network must eventually realize of this event and remove the session and its associated information from the system.

3. **Session Expiration**. The precondition for a *session expiration* event to be triggered is that a node realizes that the creator of a session, which currently exists in the system, is not connected to the network. Since the creator of the session is the only one capable of terminating the session, the action that needs to be carried out by the rest of the nodes once this event is triggered consists of removing the session from the system within a finite time. Otherwise, the session and its associated information will remain in the system "forever".

[7]A session becomes stable when all nodes agree on that session for at most $2(n-1)$ consecutive rounds in a highly dynamic network. This will be explained with more detail in Section 7.

6 Basic Techniques

In this section, we show different basic problems and their respective techniques to solve them. These problems consist of minor issues associated to the more general studied k-token dissemination and session management problems, which need to be solved in order to provide a correct functioning of the algorithms. We present a comparison problem for knowing when all nodes in the network have the same information, along with two solutions for such problem. The first described solution is used by the nodes to know when all nodes have gathered the k-tokens. The second solution provides the nodes a mechanism to know whether all nodes possess the same information associated to a particular session. Finally, we talk about the *wrapping around* problem and its solution.

6.1 Knowing that All Nodes Possess Same Information

The two approaches for solving the k-token dissemination and session management problems have in common the way they deal with the problem of knowing when all nodes have the same information.

When Haeupler et al. [15] describe their algorithm, *greedy-forward*, they assume that after $O(n)$ rounds of network coding phase, all nodes will be able to decode the tokens. But what if it takes more time for the nodes to be able to decode the tokens, or on the other hand, it takes less time. It is necessary that all nodes in the network know exactly when they have the same set of k tokens so the algorithm can finish. In addition, it is also needed that the creators of sessions know the exact moment when all nodes can decode the tokens associated to their sessions.

6.1.1 Comparing Tokens

On one hand, in order to check if all nodes have the same set of tokens, these nodes append to the packet a hash value of the ordered list of tokens that they possess together with a *Time To End* counter. Two nodes agree to have the same list of tokens if the hash value is the same. Every time there is an agreement between two nodes, the maximum of the two counters is taken and is decreased by one, otherwise is set to[8] N. In this way, when the counter reaches value 0, it means that all nodes have the same set of tokens, and therefore the algorithm must trigger an event indicating that all nodes in the network possess the same set of tokens.

[8]Where N is an upper-bound on the number of nodes in the network.

6.1.2 Comparing Coded Tokens

On the other hand, a creator of a session needs to know when the rest of the nodes have received enough number of linear independent packets so they can decode the tokens that those packets contain. Hence, it is necessary to implement a similar technique as the previous one. But this time, instead of just having plain-text tokens, there are coded tokens together with their respective coefficient vectors which need to be compared. It is not possible to lexicographically order these vectors and hash them as done in the above mentioned approach. The reason why this cannot be carried out is because it is possible that two nodes have received different packets but they actually have the same information or knowledge. For instance, suppose that a node u receives two packets[9] $\langle 1,0,0,1\rangle$ and $\langle 0,1,1,0\rangle$, and another node v receives also two packets: $\langle 1,1,1,1\rangle$ and $\langle 0,1,1,0\rangle$. It can be observed, that both nodes do not have enough packets to be able to decode at least one token, but they have the same knowledge about the tokens. In order words, in terms of knowledge, it can be stated that both nodes know the same, but the vectors that they have are not the same.

For this reason, it is necessary to devise a method that allows the nodes to know whether they have the same knowledge about the coded tokens, even though they have received different packets. The solution proposed for this problem consists of calculating and keeping the *Gaussian Elimination*[10] matrix of the packets received so far. This matrix will be hashed and sent together with the *Time To End* counter (as well as when comparing the set of tokens).

Moreover, it should be mentioned that this technique is not only useful for the creator of a session, but also for the rest of nodes. This is because, if the creator crashes, the rest of nodes eventually will notice it due to the fact that they all will have the same knowledge about the same coded tokens and if they can not decode all the tokens, they will realize that the node responsible for the network coding has crashed. Thereby, this method can be also used as a fault detector with *weak* completeness because it can be the case, that the elected node crashes, but before doing it, it has sent enough linear independent packets to the rest of the nodes in order for them to be able to decode all tokens, and consequently, the correct nodes will not detect that the elected node has crashed.

[9] We only look at the coefficient vector.

[10] As explained in the Introduction, the Gaussian Elimination algorithm employed to calculate the matrix has a small modification in comparison to the general algorithm.

6.2 Wrapping Around Problem

Both proposed algorithms make use of incarnation numbers. These incarnations numbers are counters that tell the nodes whether a node just joined the network after crashing. Since the size of the data type used for the counters is limited, it is necessary to find a solution for when the bound value is increased by one. For the sake of simplicity, suppose that the *incarnation number* is represented by 2^8 bits value and that the current value of the *incarnation number* is 255. The next time a node creates a session, it will increase by one the *incarnation number*, resulting in zero value. When comparing the new created session with *incarnation number* equal to 0 and older ones, by default, the comparison will return that the newest created session is the oldest one, which is not true. Therefore, the comparison method must be modified in order to solve this problem.

6.2.1 Comparing Incarnation Numbers

To solve the problem, we need to redefine the comparison technique when the incarnation numbers that need to be compared are close to the boundary. An incarnation number is close to the boundary if it belongs to the interval $(I-R, (I+R) \mod (I+1)]$. In order to use the redefined comparison method, at least one of the incarnation numbers has to belong to the interval $(I-R, I]$ and the other one has to belong to $(I-R, (I+R) \mod (I+1)]$. If one of the two incarnation numbers that need to be compared does not belong to the big interval, then a regular comparison method is used.

Suppose there exist two nodes p_i and p_j that are neighbours. And p_j sends a packet to p_i containing $IncarnationNumber_j$, which belongs to the interval $(I-R, I]$. When p_i compares $IncarnationNumber_j$ with its incarnation number, $IncarnationNumber_i$, which belongs to the interval $(I-R, (I+R) \mod (I+1)]$, instead of directly comparing them, it uses a redefined comparison method that behaves modulo $I+1$, where I is the maximum value the incarnation number can reach. In addition it is required to add R to the incarnation number, where R represents the maximum difference between incarnation numbers[11]. Furthermore, R should satisfy two conditions: $R \ll I$ and $R \geq S$, where S is the maximum number of sessions allowed in the system at the same time. Thereby, the resultant comparison method will look like this: $(IncarnationNumber_j + R) \mod (I+1)$ compared with $(IncarnationNumber_i + R) \mod (I+1)$.

[11]We assume that all the incarnation numbers existing in the system are comparable, i.e. $\forall\, p_i, p_j, \nexists\, IncarnationNumber_i$, $IncarnationNumber_j$, s.t. $|IncarnationNumber_i - IncarnationNumber_j| \mod (I+1) > R$.

For sake of simplicity, suppose we have the previous scenario where the maximum value of I is 255, and the incarnation number, $IncarnationNumber_i$, of a processor p_i is equal to 255. Now suppose, that the maximum allowed difference between incarnation numbers is two, $R = 2$. A node p_j creates a session and therefore, it increases the incarnation number by 1, $((255 + 2) \mod (255+1))$, $IncarnationNumber_j = 1$. When p_i receives p_j's packet and vice-versa, they will make use of the redefined comparison method: $(0 + 2) \mod (255 + 1)$ compared with $(255 + 2) \mod (255 + 1)$. The result of the comparison shows that the session created by p_i is older than the one created by p_j.

Finally, in order to give some intuition for how large R should be, suppose that $R < S$. In addition, suppose that there currently exists S sessions in the system, and all sessions have a unique incarnation number, where the first session is s_1 and its incarnation number is $IncarnationNumber_1$, and the last session is s_S and its incarnation number is $IncarnationNumber_S$. The difference between both incarnation numbers is S ($IncarnationNumber_S - IncarnationNumber_1 = S$). Since S is greater than R (and $R > 1$), it can be the case that we are comparing $IncarnationNumber_1$ and *IncarnationNumber*$_S$, where $IncarnationNumber_1$ is equal to I, and hence, it belongs to the interval $(I - R, I]$. And where $IncarnationNumber_S$ is equal to $I + S \mod (I + 1) = S - 1$, therefore, it does not belong to the interval $(I - R, (I + R) \mod (I + 1)]$. When comparing both incarnation numbers, since $IncarnationNumber_S$ does not belong to the interval, the regular comparison method is used. The comparison method will return that $IncarnationNumber_S$ is lower than $IncarnationNumber_1$, i.e. *IncarnationNumber*$_S$ is older than $IncarnationNumber_1$ which is not true. Moreover, if $R = S$, the comparison may also fail if the difference between $IncarnationNumber_S - IncarnationNumber_1$ is greater than S. Thus, it is necessary to set a large enough value to R, in order to avoid these scenarios where the comparison method misbehaves.

7 Algorithm Design: One Session at a Time

In this section, a first approach to solve the k-token dissemination and session management problems is described. The main algorithm consists of several algorithms and whose main purpose is to provide a stable session once a node has reached the threshold of minimum number of tokens needed to start the network coding phase and all nodes agree on that particular session. It is stable in the sense that no other node will start its own network coding phase even though it has also reached the threshold. While there is an agreement, the nodes send network coded packets, if not, packets containing plain-text tokens are sent. Moreover, this algorithm provides a mechanism to have a *back-up* session in case the current session terminates or expires.

7.1 Session Management: One Session at a Time

For this problem, it is allowed to have in the system only one session at a time. Thus, the nodes keep information associated to only one session. In addition, there exists the possibility for the nodes to switch to a *back-up* session, in case the node responsible for the an existing session crashes or terminates the session. However, the nodes do not allocate buffer space for the *back-up* session and its associated information until the session leaves the *back-up* condition to become principal session. The associated information of a session consists of all the coded packets (coefficients and coded tokens) received by a node, together with the generated Gaussian Elimination Matrix is needed to decode such tokens. The main goal of the algorithm is that all nodes agree on the same session for the session to become stable. And consequently, the node responsible for such session can start with the network coding phase.

7.1.1 Failure-Free Legal Execution

Here, we explain how the algorithm should behave when no failures take place in the network.

- When a node, p_i, gathers the minimum number of tokens needed to start broadcasting random linear combinations of tokens, an abstract event is triggered. This event indicates that the node is ready to start sending coded tokens, i.e. the node is eligible. Once this event is triggered, if there is no stable session existing in the system and the existent session (if any) is lexicographically lower, then the node creates a session, s_i. This session will be broadcast by its creator to all its neighbours.

- Once p_i broadcasts s_i, all its neighbours immediately receive it. The goal of broadcasting s_i is that all nodes eventually agree on s_i.
- While all nodes try to agree on one session, plain-text tokens are sent (piggybacking technique).
- Once all nodes have agreed on s_i, this session becomes stable and the node responsible for s_i, i.e. p_i, is allowed to start sending random linear combinations of tokens.
- When all nodes have enough number of linear independent tokens in order to decode all tokens belonging to a particular session, s_i, p_i will terminate s_i. This event will eventually be noticed by all nodes in the network, and thereby, the session will eventually become obsolete and removed from the system together with its associated information.
- After the termination of a session, If all nodes have agreed on a *back-up* session, such session becomes the principal session.

7.1.2 Non-Failure-Free Legal Execution

A node can be affected by three different types of failures: *crash-stop*, *crash-reboot* and *crash-resume*.

7.1.2.1 Crash-stop. If a node is affected by a *crash-stop* failure, once the node crashes, it will never be part of the network again. Depending on the node affected by this failure, the failure will lead to two different consequences.

1. If the crashed node was not responsible for any session (*principal* session or *back-up* session), the rest of the nodes will not even notice that another node has crashed.
2. If the crashed node, p_i, is the responsible for a stable *principal* session, s_i, or *back-up* session, bs_i, the rest of the nodes will eventually notice that the node responsible for s_i or bs_i is not longer in the system. Once the rest of the nodes realize that no node is maintaining session s_i or bs_i, the session expires and therefore, they will remove it from the system. In the case of s_i, its associated information (if any) will be also removed.

7.1.2.2 Crash-reboot. A node p_i may crash and reboot. The main difference between this type of failure and the previous one is that after crashing, the node may join the network again with all its variables set to their respective initial values. After the node reboots and unless the current *Incarnation number* is equal to 0, all nodes will have to agree on a new *Incarnation number*.

As in the previous failure, if p_i was responsible for a session s_i or bs_i that still exists in the system after the node crashes, this session will eventually disappear from the system.

7.1.2.3 Crash-resume. A node p_i may also crash and resume. That means that after crashing, the node may join the network with the same state as right before crashing. During the time that the node is absent, the rest of the nodes may make some progress (e.g., creating and removing *principal* and *back-up* sessions, etc.). Depending on the progress made, the crashed-resumed node may recover all, partial or none of the tokens associated to a session.

- If p_i has an older *Incarnation number*[12] than the rest of the nodes, it will lead to a disagreement in the network and the calculation of the new *Incarnation number*.
- If p_i has the same *Incarnation number* as the rest of the nodes, nothing will happen.
- If p_i created a session s_i before crashing, once the node has resumed and as long as all nodes agree on the same incarnation number, the node will keep maintaining the session and broadcasting random linear combinations of tokens until all correct nodes in the network can decode all tokens associated to that particular session.
- If p_i has a session, s_j, that the rest of the nodes still agree on, p_i will have the chance to retrieve the tokens belonging to that session. This chance will depend on two factors: *a*) the node, p_j, responsible for that session does not crash. And *b*) p_j notices on time that p_i still needs to receive more packets in order to decode the tokens associated to the session.

[12]As mentioned in sectin 5.2, an incarnation number is a counter that tells the nodes whether a node just joined the network after crashing.

7.2 White and Black nodes Algorithm

Before defining the main algorithm, it is necessary to know how information is spread in a highly dynamic network. In order to do that, the *White and Black nodes* algorithm is described in this section. For the sake of simplicity, the proofs used in the main algorithm will make use of the lemma of this basic algorithm. In addition, this algorithm is also going to be exploited by the second approach, which can be viewed in the next section.

Intuition. If a black node is neighbour of a white node, then the white node becomes black.

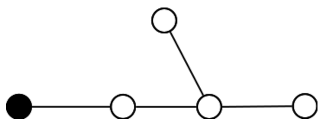
Lemma 7.1. *In an always connected dynamic graph, where the graph can change completely from round to round, suppose an initial configuration where all nodes are coloured with white color but one, which is coloured with black. We say that after at most $N-1$ rounds, where N is the number of nodes in the graph, all nodes will be black.*

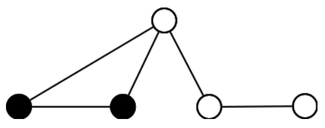
Proof. Since the graph is always connected, in every round, at least one black coloured node is the neighbour of at least one white coloured node, i.e. in every round at least one white coloured node becomes black. Thus, after at most $N-1$ rounds we can assert that all the nodes in the network are black. □

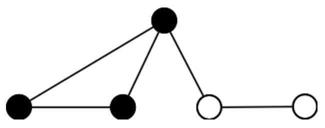
7.2.1 Why do we use $N-1$ instead of D?

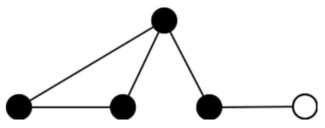
Suppose that the graph is always connected and dynamic, and that D (diameter of the graph) remains the same value throughout the execution of the algorithm, independently of the topology changes that the graph may suffer.

If the graph is static, we know that the number of steps needed for all nodes in the graph to become black coloured is D. Can we state the same for an always connected dynamic graph where D remains constant during the execution of the algorithm? The answer is: No. It can be proved with the following counter example, where D is equal to 3 throughout the execution of the algorithm:


(a) Initial Configuration


(b) After Round 1


(c) After Round 2


(d) After Round 3

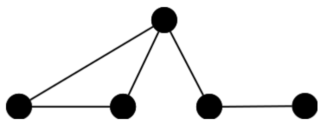

(e) After Round 4

Figure 2: Counterexample

In Figure 2, it can be observed that due to the dynamic nature of the graph the total number of steps needed to color the entire graph is 4 instead of 3, even though the diameter of the network remains unchanged throughout the execution of the algorithm. Note that a small change in the topology has been made in the first round (see Figure 2b) with respect to the *Initial Configuration*. Therefore, we conclude that at most $N-1$ rounds are needed to spread the information from one node to the rest of the nodes in a dynamic network.

7.3 Self-Organizing Wave Algorithm

A distributed algorithm is a *Wave* algorithm if the three following requirements are satisfied [24]:

- Termination: Each computation is finite.
- Decision: Each computation contains at least one decide event.
- Dependence: In each computation each decide event is casually preceded by an event in each process.

A wave algorithm exchanges a finite number of messages and then makes a decision, which depends on the events that take place in each process.

This algorithm is used a template for three other different algorithms that run concurrently in order to solve the session management and k-token dissemination problems. The other three algorithms are: *Maximum_ID*, *Session Verifier* and *Maximum Incarnation Number*.

The algorithm makes use of 5 main variables: N, X, x, $StabilityCounter$, and $ExistenceCounter$.

- N: It is an upper bound on the total number of nodes in the network.
- X: It represents a small x.
- x: A unique value composed of an incarnation number and a unique ID.
- $StabilityCounter$: It indicates whether X is stable or not.
- $ExistenceCounter$: It is used by the nodes to know whether the responsible for X exists in the system or not.

The reason why we make use of these two counters is because the nodes need to know whether a session is stable or not. In addition, they also need to know whether its creator exists in the system or not. The counters will provide such information, becoming this way in key elements for our algorithms.

7.3.1 Algorithm

Algorithm 1 Self-Organizing Wave (Template)

```
Require: $X_i$, $x_i$, $StabilityCounter_i$, and $ExistenceCounter_i$;
1: function Upon New_Round
2:     $StabilityCounter_i \leftarrow StabilityCounter_i + 1$
3:     $ExistenceCounter_i \leftarrow ExistenceCounter_i + 1$
4:     $StabilityCounter_i \leftarrow min(StabilityCounter_i, N)$
5:     $ExistenceCounter_i \leftarrow min(ExistenceCounter_i, N)$
6:     if $X_i = x_i$ then
7:         $ExistenceCounter_i \leftarrow 0$
8:     else
9:         if $ExistenceCounter_i = N$ then
10:            raise INEXISTENCE_INDICATION();
11:        end if
12:    end if
13:    SEND($X_i, StabilityCounter_i, ExistenceCounter_i$);
14: end function
```

```
15: function UPON RECEIVE($X_j$, $StabilityCounter_j$, $ExistenceCounter_j$)
16:     if $X_j = X_i$ then
17:         $StabilityCounter_i \leftarrow min(N, StabilityCounter_i, StabilityCounter_j)$  # Converge to the min
18:         $ExistenceCounter_i \leftarrow min(N - 1, ExistenceCounter_i, ExistenceCounter_j)$
19:     else
20:         raise DISAGREEMENT_INDICATION($X_j$, $StabilityCounter_j$, $ExistenceCounter_j$)
21:         Refresh()          # Set stability counter to 0
22:     end if
23: end function
24: function RESTART          # Reset the variables
25:     $X_i \leftarrow x_i$
26:     $StabilityCounter_i \leftarrow 0$
27:     $ExistenceCounter_i \leftarrow 0$
28: end function
29: function REFRESH
30:     $StabilityCounter_i \leftarrow 0$
31: end function
32: function ADOPT($\langle Incarnation, NodeID\rangle X$, $int\ StabilityCounter$, $int\ ExistenceCounter$))
33:     $X_i \leftarrow X$
34:     $StabilityCounter_i \leftarrow StabilityCounter$
35:     $ExistenceCounter_i \leftarrow ExistenceCounter$
36: end function
37: function ISSTABLE
38:     if $StabilityCounter_i = N$ and $ExistenceCounter_i < N$ then
39:         return $true$
40:     end if
41:     return $false$
42: end function
```

7.3.2 Properties

Property 1. In every round, both counters (*StabilityCounter* and *ExistenceCounter*) are increased by one, being N the maximum value they can reach.

Property 2. Every time a message, containing an $X_j \neq X_i$, is received, $StabilityCounter_i$ is set to 0.

Property 3. Every time a message, containing an $X_j = X_i$, is received,

$StabilityCounter_i$ converges to the minimum value out of N, *StabilityCounter*$_j$ and itself. And $ExistenceCounter_i$ is set to the minimum value out of $N-1$, $ExistenceCounter_j$ and itself.

Property 4. When $ExistenceCounter_i$ reaches value N, an non-existence indication is raised. This means that either X_i does not longer exist in the system or it has been disconnected from the network for at least one round.

Property 5. A node is considered to be in a stable configuration, when the stability counter is equal to N and the existence counter does not exceed $N-1$.

Lemma 7.2. *If there is a node in the network with different X than the rest of nodes, all nodes in the network will become aware of this in at most $N-1$ rounds. After $N-1$ rounds and while there exists at least one node with different X, the stability counters of all nodes will never reach value N.*

Proof. According to lemma 7.1, we know that the information spreads from one node to the whole network in at most $N-1$ rounds. Hence, if there is at least one node that has a different X, this will be known by the rest of the nodes in at most $N-1$ rounds. Looking at the neighbours, there are two different cases for any arbitrary configuration:

I Neighbour with different X. Suppose that p_i and p_j are neighbours throughout the execution of the algorithm and have different X ($Xi \neq Xj$). After one round, they will send to each other a message containing their X, and since they are not equal, the stability counter will be set to 0 according to Property 2. In the next round, the stability counter will be increased by one, but once they receive the message from the other node, they will set it again to 0. The same will happen in all successive rounds, therefore *StabilityCounter* will never reach N for none of these two nodes.

II Neighbour with same X. Assume we have the same scenario as in *(I)*, but this time there is one more node p_r, whose $X_r = X_i$. Even if p_r does not receive any message from p_j (it only receives it from p_i) throughout the execution of the algorithm, according to property 3, *StabilityCounter* will converge to the *min*, and thus it will adopt the value of $StabilityCounter_i$. Which will never be greater than 1 for this case. Therefore its *StabilityCounter* will never reach N.

□

7.4 Maximum_X Algorithm

Maximum_X algorithm is an algorithm that inherits from *Self-Organizing Wave* algorithm and its purpose is to continuously find the greater X among all nodes in the network. Moreover, the information given by this algorithm allows a node to know whether the node to which X belongs, is connected to the network or not.

7.4.1 Algorithm

Algorithm 2 Maximum_X

```
Require: $X_i$, $x_i$, $StabilityCounter_i$, and $ExistenceCounter_i$;
1: function Upon Event Inexistence_Indication
2:     Restart()
3: end function
4: function Upon Event Disagreement_Indication($X_j$, $StabilityCounter_j$,
   $ExistenceCounter_j$)
5:     if LexicographicalCompare($X_i, X_j$) = $GREATER$ then          # $X_j > X_i$
6:         Adopt($X_j, StabilityCounter_j, ExistenceCounter_j$)
7:     end if
8: end function
```

As it can be observed in the pseudo-code above, *Maximum_X* algorithm implements two event listeners (line 1 and 4). The variable x_i represents a value(e.g., numerical, character, etc.) and X_i represents the greatest lexicographical value seen by a node.

7.4.2 Properties

Property 1. Once an non-existence indication has been triggered, the counters are set to 0 and $X_i = x_i$.

Property 2. When a disagreement event is triggered and X_j is greater than X_i, then the node p_i adopts the values of counters of p_j and its X_j.

Lemma 7.3. *We can relate the black and white nodes lemma to our Maximum_X algorithm, where the node with greater x_i is the black node, and the rest are white nodes. Therefore, we assert that after at most $N - 1$ rounds $X_i = X_j$, $\forall$ p_i, p_j where $i \neq j$. In other words, after at most $N - 1$ rounds all nodes in the network have the same X.*

Lemma 7.4. *After at most $N-1$ rounds all the floating X(s)*[13] *will disappear from the system.*

Proof. According to Lemma 7.1, we know that the information spreads from one node to the whole network in at most $N-1$ rounds. There are two different cases for any arbitrary configuration:

I Suppose that $\forall\ p_i, \exists\ X_j, X_i$ s.t. $X_i > X_j$ and $\exists x_i = X_i$, but $\nexists\ x_j$ s.t. $x_j = X_j$. For this first case is trivial to see that since X_j is not the greatest value in the network, after at most $N-1$ rounds, when the node having its X equal to X_j, receive X_i, it will keep the one with greater value, in this case X_i, and thus, X_j will disappear from the system.

II Suppose that $\forall\ p_i, X_i = X$ and $\nexists\ x_i$ s.t. $x_i = X$. The *ExistenceCounter* of the nodes will be not equal to 0. 0, since the only one allowed to set this counter to 0 is p_i with $x_i = X$, which does not exist in the system any more. The existence counters of the nodes can only increase according to Property 1.1, thus after $N-1$ rounds, the existence counters of all nodes will reach the value N, which means, according to Property 2.1, the nodes have to reset their counters and set their $X_i = x_i$. Therefore, the floating value disappears from the network.

□

Corollary 7.5. *After at most $2(N-1)$ rounds, $\forall\ p_j, \exists\ x_i \geq x_j$, s.t. $X_i = x_i$, and $X_i = X_j$. And within N extra rounds, all nodes know that the rest of the nodes in the network agree on the same X_i , i.e. $\forall\ p_i, p_j$ where $i \neq j$, $StabilityCounter_i = StabilityCounter_j = N$.*

Proof. Suppose there is at least one node such that its X is floating, and X is greater than any x_i existing in the network. According to Lemma 7.4, within at most $N-1$ rounds, this value will disappear from the system. Once the floating X has vanished, the nodes will set $X_i = x_i$, and the *StabilityCounter* to 0. According to Lemma 7.3, within $N-1$ extra rounds, all nodes will have the same X. Now, in order for the nodes to know that the rest of the nodes have the same X, it is necessary that the *StabilityCounter* reaches N, which will happen within at most N rounds after all nodes have the same X. The reason why N rounds are enough, is because once all the nodes have the same X, the *StabilityCounter* will never be reset to 0 since all the nodes possess the same X, therefore the counter can only increase. According to the algorithm, each round *StabilityCounter* increases by 1,

[13]Floating X(s) are all the X_j whose x_j does no longer exist in the system.

hence, after N rounds its value will be equal to N. Consequently, within a total number of $3N - 2$ rounds all nodes know that they agree on the same X. □

7.4.3 Maximum_ID and Incarnation Number

The *Incarnation Number* and *Maximum_ID* algorithms are instantiations of the *Maximum_X* algorithm. The only difference between them are the values that x_i and X_i represent.

In the case of the *Incarnation Number* algorithm, the small x is a numerical value that represents the incarnation number and the big X represents the maximum incarnation number that all nodes will eventually agree on.

On the other hand, in the case of the *Maximum_ID* algorithm, the small x is a numerical or character value that represents a unique ID of a particular node and the big X represents the maximum ID that exists in the system and which all nodes will eventually agree on.

7.5 Session_Verifier Algorithm

The *Session_Verifier* algorithm is responsible for, as it can be deduced from its name, verifying two things: (a) The session is stable and (b) the node responsible for that session exists in the network.

7.5.1 Algorithm and Properties

Session_Verifier algorithm is simply an instantiation of the *Self-Organizing Wave algorithm* without any extension. Thus, it shares all its properties.

The variable x_i consists of an *incarnation number* and the ID of the node ($x_i = \langle IncarnationNumber, NodeID \rangle$). X_i represents the session that all nodes agree on.

As explained earlier, this algorithm only verifies that all nodes agree on the same session ($\langle IncarnationNumber, CreatorOfSessionID \rangle$). At the exact moment when there is a disagreement on the session, the stability is *broken* and this will spread to all nodes in the network. This leads to the following Corollary.

Corollary 7.6. *According to Lemma 7.2, if there is a node with different X_i, all nodes will notice it after at most $N - 1$ rounds.*

The reason for having this algorithm is to ensure that once a session has been created and becomes stable , this session will remain in a stable status, even though another eligible node with greater ID joins the network.

7.6 Session Manager Algorithm

The function of this algorithm is to unify all the previous mentioned algorithms, and make them work together in order to end up with a final algorithm that is able to switch from one session to another in case, for instance, the creator of the current session is down and all nodes agree on the same $MaximumID$. It is also the node responsible for increasing the incarnation number every time a change of session takes place.

7.6.1 Algorithm

Algorithm 3 Session Manager

```
Require: $Session, MaximumID, Incarnation$;
 1: function Upon End_Of_Round
 2:     if $Session.IsStable() = false$ and $MaximumID.IsStable = true$ and $Incarnation.IsStable() = true$ then
 3:         $Incarnation.x_i \leftarrow Incarnation.x_i + 1$
 4:         $Incarnation.X_i \leftarrow Incarnation.X_i + 1$
 5:         Session.Adopt($\langle Incarnation.X_i, MaximumID.X_i \rangle$, $MaximumID.StabilityCounter_i$, $MaximumID.ExistenceCounter_i$)
 6:     end if
 7: end function
```

The three variables required by the algorithm are instantiations of the previous algorithms. *Session* is an instantiation of the *Session_Verifier* algorithm. *MaximumID* is an instantiation of the *Maximum_ID* algorithm and *Incarnation* is an instantiation of the *Incarnation Number* algorithm.

The *Session Manager* algorithm is executed at the end of each round, and as shown in line 4 of the pseudo-code, it checks if there is a stable session, i.e. all nodes agree on the same session. In negative case, it tries to switch to the current $maximumID$. To do so, it also checks if all nodes agree on the same $maximumID$ and on the same incarnation number. In case these two verifications return true, then the current $maximumID$ is taken as a new session, and the incarnation number is increased by one.

Remark 7.7. *Once all nodes know that they agree on the same X_i, there is not agreement on the session and all nodes agree on the same incarnation number, then according to the session manager algorithm, X_i will be the session and will not change unless p_i goes down or a node with different $CreatorOfSessionID$ or incarnation number joins the network.*

7.7 Advantages and Disadvantages of the Algorithm

7.7.1 Advantages of Allowing One Session at a Time

The main benefit of using an algorithm that allows only one session in the system at a time is that the nodes only have to store the information associated to that session. Therefore, this algorithm is recommended to be used in systems where their devices have memory constrains.

The information associated to a session is all coded packets received associated to that session and the Gaussian Elimination Matrix generated to decode the tokens contained in those packets. If t is the number of coded tokens and l the size of a token, then the maximum size required to store the matrix will be multiple of: $t \times (t+l)$. Where $(t \times t)$ is the square matrix containing the coefficient part, and $(t \times l)$ refers to the right side of the matrix containing the tokens.

7.7.2 Disadvantages of Allowing One Session at a Time

During the analysis of the algorithm a series of scenarios for which this algorithm can become really inefficient have been discovered:

1. Session Agreement - $O(n^2)$. Suppose the following scenario: the incarnation number is stable and the node with the lowest ID in the network is the first reaching the minimum threshold of tokens, therefore it becomes an eligible node, and since there is no other session, it creates one. According to Corollary 7.5, we know that after $2(N-1)$ all nodes in the network will agree on the same session. But, now suppose that after $2(N-1)-1$ rounds, the node with the second lowest ID reaches the threshold and also becomes eligible and creates a session, avoiding that the previous session becomes stable. The same can happen with the node with the third lowest ID, and so forth. All nodes will agree on the same session in this really pessimistic and unlikely scenario after $O(n^2)$ rounds, which is the same time that the simple random forwarding algorithm needs to solve the problem.

2. Recovered node may disrupt all previous work. A less unlikely scenario is that a crashed node recovers after a while, having an incarnation number that does not correspond to the incarnation number agreed by the rest of the nodes in the network. This disagreement, will lead to a change of current session (if there exists one), and therefore, all the work down by the creator of that session and the rest of the nodes, in terms of amount of random linear combinations packets that have been broadcast, will be disrupted by the recently joint node.

Moreover, it can be the case that a node crashes while there exist a creator of a session sending random linear combinations of tokens. If the creator of the session terminates such session before the crashed node resumes, the crashed node will not be able to obtain any of the already sent information.

8 Algorithm Design: S Sessions at a Time

In this section, the design of the algorithm that solves the k-token dissemination and session management problems allowing the existence of S sessions in the system at the same time is introduced. The full version of the algorithm can be seen in the appendix. Here, only the most relevant and interesting parts of it are presented.

8.1 Lessons learned from previous approach

There are several points that needed to be taken into consideration before starting designing the second algorithm. Some of these points are:

- Minimize disruption. One of the things that need to be mitigated with respect to the previous approach is the capacity that a node has to disrupt all the work done by a previous node.
- Avoid indefinite session agreement. If a node becomes eligible and creates a session, then after at most $2(N-1)$ rounds, this node or another node will start with the network coding phase, independently of if one or more nodes have created a session.
- Extend the back-up property. In the previous approach, if the creator of a session fails, then it is checked if there is agreement on the Maximum ID node and in affirmative case, the Maximum ID becomes the new creator of a session. It would be interesting to extend this property, so instead of having only one back-up, it is possible to have L back-ups. Thereby, the algorithm will be able to tolerate until $L-1$ failures of session creators.
- Add possibility for a crashed-recovered node to recover some useful information. It is interesting if a node that has gone down and recovered within a limited period of time can have the chance to recover some of the coded tokens that have been sent while the node was absent.

8.2 Session Management: S Sessions at a Time

For this problem, instead of allowing only one session in the system, there can be up to $O(S)$ sessions existing in the system at the same time.

How these sessions must be handled is part of the new problem, as well as, how the algorithm can tolerate session failures. The final goal is that all nodes agree on the same set of sessions or at least in the same subset of sessions, so these sessions can become stable, and therefore the nodes responsible for those sessions can start broadcasting random linear combinations of tokens.

Before explaining the design of the algorithm, it is necessary to define the problem itself. These are some of the points that should be taken into account:

- A session becomes stable if all correct nodes agree on that session for at most $2N - 1$ consecutive rounds.
- Once a session expires all nodes must eventually notice it.
- A node may crash and recover without disrupting all the work done by the current nodes in the network.
- The system should tolerate $O(S)$ session failures.

The idea to improve fault tolerance is that it is possible to have up to $O(S)$ stable sessions at the same time, so if one of the stable sessions fails then another stable session will take over.

8.2.1 Failure-Free Legal Execution

As with the previous algorithm, in the following points, we explain how the algorithm should behave when no failures take place in the network.

- When a node, p_i, gathers the minimum number of tokens needed to start broadcasting random linear combinations of tokens, an abstract event is triggered. This event indicates that the node is ready to start sending coded tokens. Once this event is triggered, the node creates a session, s_i, and places it in a list of sessions, l_i that will be broadcast to all its neighbours.
- Once p_i broadcasts l_i, all its neighbours immediately receive it. The goal of broadcasting s_i is that all nodes eventually agree on s_i.
- While all nodes try to agree on one or more sessions, together with the list of sessions, plain-text tokens are sent.
- Once all nodes have agreed on s_i, that session becomes stable and the node responsible for s_i, i.e. p_i, will be able to start sending random linear combinations of tokens (As long as s_i is the first stable session on the list of sessions.).

- When all nodes have enough number of linear independent tokens in order to decode all tokens belonging to a particular session, s_i, p_i will stop maintaining s_i. Therefore, it will eventually become obsolete. Before being removed from the list of sessions of all nodes in the network together with its associated information, it will be kept for a certain period in the system. The reason why it is not directly removed is because we want the nodes to have a chance to recover or provide useful information in case an crash-resume failure occurs.

8.2.2 Non-Failure-Free Legal Execution

A node can be affected by three different types of failures: *crash-stop*, *crash-reboot* and *crash-resume*. Note that the behaviour is similar to the algorithm where only one session is allowed, but instead of having one principal session and a back-up session, we have a list of sessions of size S.

8.2.2.1 Crash-stop. Depending on the node affected by this failure, the failure will lead to two different consequences.

1. If the crashed node was not responsible for any session, the rest of the nodes will not even notice that another node has crashed.
2. However, if the crashed node, p_i, had created and broadcast a session s_i before crashing, all nodes will eventually notice that the node responsible for s_i is not longer in the system. Once the rest of the nodes realize that no node is maintaining session s_i, they will remove it from the main list of sessions, and placed temporally in another list. Eventually the session, together with its associated information, will be removed from the system.

8.2.2.2 Crash-reboot. As in the previous failure, if p_i was responsible for a session s_i that still exists in the system after crashing, this session will eventually disappear from the list of sessions of the rest of the nodes, but it will be kept for a while in another list.

8.2.2.3 Crash-resume. A node p_i may also crash and resume. That means that after crashing, the node may join the network with the same state as right before crashing. During the time that the node is absent, the rest of the nodes may do some progress (e.g., creating new sessions, removing all sessions, etc.). Depending on the progress made, the crashed-resumed node may recover all, partial or none of the tokens associated to the sessions.

- If p_i has an older *Incarnation number*, it will lead to a disagreement in the network and the calculation of the new *Incarnation number*.
- If p_i has the same *Incarnation number* as the rest of the nodes, nothing will happen.
- If p_i created a session s_i before crashing, once the node has resumed, it will keep maintaining the session until all correct nodes in the network can decode all tokens associated to that particular session.
- If p_i has in its list of sessions, sessions that are still maintained by other nodes in the network, p_i will have the chance to retrieve the tokens belonging to that sessions. This chance will depend on two factors: a) the nodes responsible for that sessions do not crash. And b) the nodes responsible for the sessions notice on time that p_i still needs to receive more packets in order to decode the tokens.

8.2.3 Managing S Sessions

In this approach, all nodes will possess a list of size S. The list is divided in three different lists of size S: *Old*, *Current* and *Future*.

- *Old list.* All sessions placed in this sub-list are either sessions whose creator is not longer in the system or sessions that have been finished because all nodes have the same information about the coded tokens or both of them[14].
- *Current list.* All sessions that belong to this sub-list are broadcast to the neighbours after the beginning of each round. Moreover, this list will indicate which kind of packet is transmitted, either a plain-text packet or a coded packet. If there is no stable session in this sub-list, then the content of the packet will be in plain-text.
- *Future list.* All sessions located in this sub-list are sessions which the node will potentially work with in the future. Every time a node reaches the threshold will firstly check if there is space in this list and if so, it will create and place the session in it.

When a node receives the sub-list of sessions, *CurrentList*, it will merge it with its *Current list*, and all the exceeding stable sessions will be merged with the sessions placed in *Future list.*

[14]It is possible that the node responsible for the session has crashed after having already started to send coded tokens.

All sessions placed in *Current list* that either their *ExistenceCounter* has reached value N (i.e. the node responsible for that session does not longer exists in the system) or their *Time To End* counter has value 0 (i.e all correct nodes in the network have the same knowledge about the coded tokens and they can not learn anything new), are merged with the sessions located in *Old list*.

8.3 Archives

An *archive* is associated to a session and consists of: the *Gaussian Elimination Matrix* and all the random linear combinations of tokens belonging to that session. Once a session is removed from all sub-lists, its archive is also removed.

In the first approach, the nodes only store the coded tokens that belonged to the current elected node and its respective *Gaussian Elimination Matrix*. Once all nodes are able to decode the tokens or all nodes agree on a new session, the archives belonging to the previous session are removed. However, in this second approach a node can store up to $O(S)$ archives, since it can have up to S sessions.

8.4 S Sessions Manager Algorithm

Algorithm 4 S Sessions Manager

```
Require: $session : \langle IncarnationNumber, ID \rangle$ , $FutureList[S] : \langle session \rangle$,
    $CurrentList[S] : \langle session \rangle$ , $OldList[S] : \langle session \rangle$;
 1: function UPON NEW_ROUND
 2:     if All nodes have same $GE\_Matrix$ from the same session $s_i$ then
 3:         Move $s_i$ from $CurrentList$ to $OldList$
 4:     end if
 5:     while $CurrentList$ is not $REPLETE$ and $FutureList$ is not $EMPTY$
    do
 6:         Move sessions from $FutureList$ to $CurrentList$
 7:     end while
 8:     for all sessions in $CurrentList$ do
 9:         UpdateCounters(session)
10:     end for
11:     if session $s_i$ from $CurrentList$ is in $OldList$ then
12:         Remove $s_i$ from $OldList$
13:     end if
14: end function
15: function UPON RECEIVE($list_j$)
16:     Merge($CurrentList$, $list_j$);
17:     Move to $FutureList$, sessions that are stable, have archives and that are
    no longer in $CurrentList$
18: end function
19: function UPDATECOUNTERS($S_i$, $StabilityCounter_i$, $ExistenceCounter_i$ )
20:     $StabilityCounter_i \leftarrow StabilityCounter_i + 1$
21:     $ExistenceCounter_i \leftarrow ExistenceCounter_i + 1$
22:     $StabilityCounter_i \leftarrow min(N, StabilityCounter_i)$          # Converge to
    the min
23:     $ExistenceCounter_i \leftarrow min(N, ExistenceCounter_i)$
24:     if ($S_i = s_i$) or ($S_i.ID$ is $= s_i.ID$ and there are archives for $S_i$) then
25:         $ExistenceCounter_i \leftarrow 0$
26:     else if $ExistenceCounter_i = N$ then          # If the session's owner has
    crashed
27:         Remove $S_i$ from $CurrentList$
28:         if there are archives for $S_i$ then
29:             Move $S_i$ to $OldList$
30:         end if
31:     end if
32: end function
```

8.5 Local Properties

Property 1. If all nodes agree on the same Gaussian Elimination Matrix from a session S_i and this session is placed in *CurrentList* then, it is moved from *CurrentList* to *OldList*.

Property 2. If any session in *CurrentList* is also located in *OldList*, then it is removed from *OldList*.

Property 3. If *OldList* is full and a new session is added, then the first session is removed from *OldList* along with its archives. Note that if the added session turns to be the first session, it is not even added to the list.

Property 4. If *FutureList* is full and a session is added to the list, the last session along with its archives is removed from the list. Note that if the added session turns to be the last session, it is not even added to the list.

Property 5. When a list, $list_j$, is received from a neighbour, P_j, this list is merged with *CurrentList*. All the stable sessions(with stored archives) that are no longer in the final list, are added to the *FutureList*.

Property 6. The existence counter is set to 0, if and only if, S_i corresponds to a current session s_i, or if there exists any archive belonging to S_i.

Property 7. If a session in *CurrentList* no longer exists, i.e. *Existence-Counter* is equal to N, then this session is removed from *CurrentList* and moved to *OldList* only if there are some archives belonging to the session.

Property 8. A session is removed from *OldList* when S newer sessions are added to the list.

Property 9. In every round if *CurrentList* is not replete and *FutureList* is not empty, sessions from *FutureList* are moved to *CurrentList* until *CurrentList* is replete or *FutureList* is empty.

Property 10. All the sessions stored in the lists are ordered according to their incarnation numbers (in case two sessions have the same incarnation number then the order will depend on the lexicographic value, where the session with greater ID is the first).

Lemma 8.1. *Once the creator of a session S_i, has removed this session from CurrentList and OldList, this session will eventually disappear from the lists of the rest of the nodes.*

Proof. Suppose that there is a node that introduces a session s_i in the system. This can occur when a node has gone down and then it recovers after a while. In addition, without loss of generality, suppose that this session is the first session among the already-existing sessions in *CurrentList*. The list of this node will be broadcast and, consequently, the session S_i. According to the Black and White lemma, this session will be received by all the nodes in at most $N - 1$ rounds. When the node with the same ID as the session, receives S_i, will see that S_i is no longer in *CurrentList* or *OldList*, therefore according to Property 6 it will not set its *ExistenceCounter* to 0, hence after at most $N - 1$ rounds all the *ExistenceCounter* of all nodes will reach value N. According to property 7, all nodes that do not have any archive belonging to session S_i will remove it from *CurrentList*. On the other hand, the nodes possessing archives that belong to this session will move S_i to *OldList*. Finally, according to Property 8, S_i will disappear from the system once all nodes that have S_i in their *OldList* store S newer sessions. □

Remark 8.2. *A session is removed from the lists in two possible ways: a) When the existence counter of a session reaches N and it does not have any archives belonging to it, then the session is immediately removed. b) In contrast, if there are archives belonging to the session, then the session is moved to OldList and according to property 8, eventually removed from this list too.*

Lemma 8.3. *A session can always be recovered as long as the creator of the session keeps the session in one of its lists. (and as long as it behaves as a correct node.)*

Proof. There can be three possible scenarios by looking at where the session is placed:

I The session is placed in *OldList*. When a node receives a list containing a session S_i of which that node is the creator, and that session is placed in *OldList*, according to Property 6, the *ExistenceCounter* of that session is set to 0. In addition, the session will be removed from *OldList* and will be treated as a current session with its stability counter set to 1. In case, the node that receives the list is not the creator of S_i , the only difference is that the node will not set the *ExistenceCounter* to 0.

II The session is placed in *CurrentList*. When a node receives a list containing a session S_i of which that node is the creator, and that session is placed in *CurrentList*, according to Property 6, the *ExistenceCounter* of that session is set to 0 and the *StabilityCounter* increased by one. In case, the node is not the creator both, the *ExistenceCounter* and the *StabilityCounter* are increased by one.

III The session is placed in *FutureList*. The session will eventually be moved to *CurrentList*.

□

Lemma 8.4. *There is no session in FutureList with lower identification than at least one session in CurrentList.*

Proof. Assume to the contrary that there is a session in *FutureList* with lower identification than at least one session in *CurrentList*. Note that there are two possible ways to add sessions to the *CurrentList*: by merging or by moving sessions from *FutureList* to *CurrentList*. According to property 10, all lists are ordered, therefore, when there is space in *CurrentList* for more sessions, the ones (if any) moved from *FutureList* to *CurrentList* are already ordered, hence all the remaining sessions(if any) in *FutureList* will have a greater identification. Contradiction!. On the other hand, when merging two lists, the resulting list will be also ordered and all the exceeding sessions will be placed in *FutureList*(if possible). Therefore in order to have a session in *CurrentList* with greater identification than at least one session in *FutureList* after merging, one of the exceeding session that was moved to *FutureList* had lower identification than at least another session that was placed in *CurrentList*, but since the sessions are ordered this leads to another contradiction! □

Lemma 8.5. *After merging two lists, it is not possible to have the same session in both CurrentList and FutureList.*

Proof. Assume to the contrary that after merging two lists there is a session S_i that is placed both in *CurrentList* and *FutureList*, that means that the received list, $list_j$, contained a session S_i which after the merging process was added to *CurrentList*. If this is the case, that means that before merging there were sessions in *CurrentList* that had a greater identification than S_i or that the union of *CurrentList* and $list_j$ did not exceed the length (S) of the list. For the first case, we refer to Lemma 8.4 that shows that this is not possible. For the second case, according to property 9, at the beginning of every if *CurrentList* is not replete and *FutureList* is not empty, sessions from *FutureList* are moved to *CurrentList* until *CurrentList* is replete or *FutureList* is empty, therefore S_i should have be placed before in *CurrentList* and removed from *FinalList*. Contradiction! □

8.6 Global Properties

Lemma 8.6. *Once a session, s_i has been created, this session or another session created during the interval, $N-1$, will become stable after at most $2(N-1)$ rounds after the moment it has been created. (As long as no creator of a session crashes during this period.)*

Proof. According to Lemma 7.1, we know that the information spreads from one node to the whole network in at most $N-1$ rounds. And there can be two different scenarios for any arbitrary configuration (supposing that the 3 lists are empty):

I Suppose that in the interval of $N-1$, apart from the creation of s_i, a number of R sessions have been created, where $R > S$. Since R is greater than the size of the list, some sessions must be dismissed. Now, suppose that s_i is removed because it has lower id than any of the R sessions. Thus, s_i will never become stable, but there will be at least a session s_r that after $N-1$ rounds is located in the list of every correct node and after N extra rounds will become stable. This is due to the fact that every time the nodes send the list, they agree at least on s_r and therefore, they increase by one its stability counter, which after N rounds reaches its maximum value, N. This indicates that the session has become stable.

II Suppose that in the interval of $N-1$, apart from the creation of s_i, a number of R sessions have been created, where $R > S$. But this time, s_i is allocated in the list because of its ID. Consequently, after $N-1$ rounds, all nodes in the network will possess s_i in their lists. And after N extra rounds, its stability counter will reach value N, indicating that this session is stable.

□

Lemma 8.7. *If both lists, Current and Future, are replete of stable sessions. The system can tolerate up to $2S-1$ failures, leading to a maximum period of $(2S-1)*(N-1)$ rounds without sending coded tokens.*

Proof. Suppose the creators, C_i, of the stable sessions, start crashing in the order their sessions are placed in the lists and with an interval of $N-1$ rounds. When the last session, s_{2S}, becomes the first element in the list, the node responsible for that session does not crash. For instance, the creator, c_1, of the first session, s_1, crashes. This will be noticed by the rest of the nodes within $N-1$ rounds. Once the rest of the nodes realize that the creator of s_1 is down, the session is removed from the list, $Current$. And the second session, s_2, becomes the first session in the list. But, it turns out that its creator, c_2, also crashes in that particular moment. Therefore, after $N - 1$ rounds all nodes will now that c_2 is also down, and so on. Since, the maximum number of sessions that can be located in $Current$ and $Future$ sub-lists is $(2S)$, c_{2S} will need to wait $(2S - 1) * (N - 1)$ rounds until it can start sending random linear combinations of tokens. □

9 Conclusions

The *greedy-forward* algorithm given in [15], solves the k-token dissemination problem by means of network coding in a more efficient way than using a simple *random forwarding* approach. In this thesis report, we present the *session management problem* and two algorithms to solve it. By solving such a problem, we can also resolve k-token dissemination problem, providing a robust solution that can deal with certain failures, such as, crash-stop, crash-reboot and crash-recovery.

Two different algorithms have been designed during the realization of the project. The first algorithm provides a realistic and robust solution for the k-token dissemination and session management problems for when one session at a time is allowed in the system. The main advantage of using the first algorithm is that it is ideal for systems with memory constrains. However, there are some scenarios for which this algorithm becomes very inefficient. Whereas the second presented algorithm, which can handle S sessions at a time, needs more memory resources. But at the same time, besides the properties of the first algorithm, it also has some extra features that makes it more robust and to be able to perform efficiently in scenarios where the first proposed algorithm does not.

Moreover, even thought, no implementation or test of the algorithms has been carried out, the algorithms described in this work are accompanied by several lemmas and their respective formal proofs, which show the correctness of the algorithms themselves.

References

[1] *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2009, San Francisco, California, USA, September 14-18, 2009.* IEEE Computer Society, 2009.

[2] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In Jennifer Welch, editor, *Distributed Computing*, volume 2180 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg, 2001.

[3] R. Ahlswede, Ning Cai, S.-Y.R. Li, and R.W. Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, 2000.

[4] A. Arora and M. Gouda. Distributed reset. *IEEE Trans. Comput.*, 43(9):1026–1038, September 1994.

[5] R. Baldoni, S. Bonomi, I. Chatzigiannakis, and G. Di Luna. Counting on anonymous dynamic networks through energy transfer. Technical Report 1, MIDLAB, 2013.

[6] Andrew Berns and Sukumar Ghosh. Dissecting self-* properties. In *SASO*, pages 10–19, 2009.

[7] S. Bhadra and S. Shakkottai. Looking at large networks: Coding vs. queueing. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–12, 2006.

[8] Ning Cai. *Network Coding Theory.* Now Publishers Inc, 2006.

[9] D. Conan, P. Sens, L. Arantes, and M. Bouillaguet. Failure, disconnection and partition detection in mobile environment. In *Network Computing and Applications, 2008. NCA '08. Seventh IEEE International Symposium on*, pages 119–127, 2008.

[10] Jean-Michel Couvreur, Nissim Francez, and Mohamed G. Gouda. Asynchronous unison (extended abstract). In *ICDCS'92*, pages 486–493, 1992.

[11] Shlomi Dolev, Ronen I. Kat, and Elad M. Schiller. When consensus meets self-stabilization. *J. Comput. Syst. Sci.*, 76(8):884–900, December 2010.

[12] Shlomi Dolev, Elad Schiller, and Jennifer L. Welch. Random walk for self-stabilizing group communication in ad hoc networks. *IEEE Transactions on Mobile Computing*, 5(7):893–905, July 2006.

[13] Christos Gkantsidis and Mitch Goldberg. Avalanche: File Swarming with Network Coding. `http://research.microsoft.com/en-us/projects/avalanche/`. [Online; accessed 19-April-2013].

[14] Bernhard Haeupler. Analyzing network coding gossip made easy. *CoRR*, abs/1010.0558, 2010.

[15] Bernhard Haeupler and David R.Karger. Faster information dissemination in dynamic networks via network coding. *CoRR*, abs/1104.2527, 2011.

[16] Ted Herman and Chen Zhang. Best paper: stabilizing clock synchronization for wireless sensor networks. In *Proceedings of the 8th international conference on Stabilization, safety, and security of distributed systems*, SSS'06, pages 335–349, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] Tracey Ho, R. Koetter, M. Medard, D.R. Karger, and M. Effros. The benefits of coding over routing in a randomized setting. In *Information Theory, 2003. Proceedings. IEEE International Symposium on*, pages 442–, 2003.

[18] I hong Hou, Yu en Tsai, Tarek F. Abdelzaher, and Indranil Gupta. Adapcode: Adaptive network coding for code updates. In *in Wireless Sensor Networks, in Proceedings of IEEE INFOCOM*, 2008.

[19] Sachin Katti, Hariharan Rahul, Wenjun Hu, Dina Katabi, Muriel Médard, and Jon Crowcroft. Xors in the air: practical wireless network coding. *SIGCOMM Comput. Commun. Rev.*, 36(4):243–254, August 2006.

[20] MinJi Kim, Jason Cloud, Ali ParandehGheibi, Leonardo Urbina, Kerim Fouli, Douglas J. Leith, and Muriel Médard. Network coded tcp (ctcp). *CoRR*, abs/1212.2291, 2012.

[21] Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the 42nd ACM symposium on Theory of computing*, STOC '10, pages 513–522, New York, NY, USA, 2010. ACM.

[22] Pierre Leone, Marina Papatriantafilou, EladM. Schiller, and Gongxi Zhu. Chameleon-mac: Adaptive and self- algorithms for media access control in mobile ad hoc networks. In Shlomi Dolev, Jorge Cobb, Michael Fischer, and Moti Yung, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6366 of *Lecture Notes in Computer Science*, pages 468–488. Springer Berlin Heidelberg, 2010.

[23] Andrew S. Tanenbaum. Network protocols. *ACM Comput. Surv.*, 13(4):453–489, December 1981.

[24] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000.

A Algorithm Design

Algorithm 5 S Sessions Token Dissemination Algorithm

```
Require: $session : \langle IncarnationNumber, ID \rangle$, $knownTokens[]$, $tokensToBeSent[]$, $codedPackets[]$, $storedSessions[]$, $GE\_Matrix[][][]$, $TTE$, $L\_TTE[]$, $sessionsManager$, $nodeID$;
 1: function UPON NEW_ROUND
 2:     if $tokensToBeSent.length() \geq M$ then
 3:         if $sessionsManager.Find(nodeID) = \perp$ then
 4:             $sessionsManager.CreateSession()$;
 5:         end if
 6:     else if $sessionsManager.Find(nodeID) \neq \perp$ then
 7:         $sessionsManager.StopMantainingSession()$;
 8:     end if
 9:     $session \leftarrow sessionsManager.GetStableSession()$;
10:     if $session = \perp$ then
11:         $TTE \leftarrow TTE - 1$;
12:         $TTE \leftarrow Max(0, TTE)$;
13:         if $TTE = 0$ then
14:             return $knownTokens[]$;
15:         else if elements in $tokensToBeSent > 0$ then
16:             $message \leftarrow ChooseTokens(MSG\_SIZE, TOKEN\_SIZE)$;
17:             $SEND(\langle PLAINTEXT, TTE, message, Hash(knownTokens) \rangle)$;
18:         end if
19:     else
20:         $index \leftarrow sessionsManager.Find(session)$;    # Try to find ”session” on the lists
21:         if $index = \perp$ then
22:             $storedSessions.Add(session)$;
23:             $index \leftarrow lastPosition$;
24:         end if
25:         $L\_TTE \leftarrow L\_TTE - 1$;
26:         $L\_TTE[index] \leftarrow Max(0, L\_TTE[index])$;
27:         if $L\_TTE[index] = 0$ then
28:             $decodedTokens \leftarrow Decode(index)$;
29:             $RemoveTokens(decodedTokens)$;
30:             $knownTokens \leftarrow knownTokens \cup decodedTokens$;
31:             $sessionsManager.Remove(CurrentList, session)$;
32:             $sessionsManager.Add(OldList, session)$;
33:         else
34:             $rlc \leftarrow [0...0]$;    # Set the Random Linear Combination to zeroed vector
```

```
            if $codedPackets[index]$ is empty and $session.ID = nodeID$ then
                $codedPackets[index] \leftarrow IndexTokens(M)$;
            end if
            for $i = 0$..number of elements in $codedPackets[index] - 1$ do
                $selection \leftarrow GenerateRandomNumber() \mod 2$;
                if $selection = 0$ then
                    $vector \leftarrow [0...0]$;
                else
                    $vector \leftarrow codedPackets[index][i]$;
                end if
                $rlc \leftarrow XOR(rlc, vector)$;
            end for
            $SEND(\langle CODED, L\_TTE, rlc, Hash(GE\_Matrix[index])\rangle)$;
        end if
    end if
end function
function UPON RECEIVE($session$, $packetType$, $TTE_j$, $msg$, $hash$ )
    if $packetType = PLAINTEXT$ then
        $tokens \leftarrow msg.getTokens(TOKEN\_SIZE)$;
        $tokensToBeSent \leftarrow tokensToBeSent \cup tokens$;
        $knownTokens \leftarrow knownTokens \cup tokens$;
        $myHash \leftarrow Hash(knownTokens)$;
        if $hash = myHash$ then
            $TTE_i \leftarrow Max(TTE_i, TTE_j)$;
        else
            $TTE_i \leftarrow N$;
        end if
    else if $packetType = CODED$ then
        $index \leftarrow Find(session)$;
        if $index = \perp$ then
            $storedSessions.Add(session)$;
            $index \leftarrow lastPosition$;
        end if
        $codedPackets[index] \leftarrow codedPackets[index] \cup msg$
        $GaussianElimination(GE\_Matrix[index], msg)$;
        $myNCHash \leftarrow Hash(GE\_Matrix[index])$;
        if $hash = myNCHash$ then
            $L\_TTE[index] \leftarrow Max(L\_TTE[index], L\_TTE_j)$;
        else
            $L\_TTE[index] \leftarrow N$
        end if
    end if
end function
```

Brief Explanation:

From *line 2* until *line 8*: If the minimum number of tokens needed to start the network coding phase is reached, then we check whether that nodes has already created a session or not. In negative case, a new session is created as long as there is space in the list. If the threshold is not reached, but there exists a session belonging to the node, we immediately stop maintaining the session.

From *line 9* until *line 49*: If there is no stable session, then a plain text packet is sent. Otherwise, we send a packet containing coded tokens associated to the stable session.

From *line 50* until *line 76*: When receiving a packet, the algorithm checks whether it is a plain-text packet or a coded packet.The tokens (coded or not coded) are stored, and the algorithm checks whether the receiver has the same information as the sender of the packet. In affirmative case, the *Time To End* counters are set to he maximum of both sender's and receiver's TTE, otherwise they are set to N.

Algorithm 6 *S* Sessions Manager

```
Require: $session : \langle IncarnationNumber, ID\rangle$, $FutureList[S] : \langle session\rangle$,
    $CurrentList[S] : \langle session\rangle$, $OldList[S] : \langle session\rangle$, $incarnation$;
 1: function Upon New_Round
 2:     while $CurrentList$ is not $REPLETE$ and $FutureList$ is not $EMPTY$ do
 3:         Move sessions from $FutureList$ to $CurrentList$;
 4:     end while
 5:     for all sessions$\langle S_i, StabilityCounter_i, ExistenceCounter_i\rangle$ in $CurrentList$ do
 6:         UpdateCounters($\langle S_i, StabilityCounter_i, ExistenceCounter_i\rangle$);
 7:     end for
 8:     if session $s_i$ from $CurrentList$ is in $OldList$ then
 9:         Remove $s_i$ from $OldList$;
10:     end if
11: end function
12: function Upon Receive($list_j$)
13:     Merge($CurrentList$, $list_j$);
14:     Move to $FutureList$, sessions that are stable, have archives and that are
    no longer in $CurrentList$;
15: end function
16: function UpdateCounters($\langle S_i$, $StabilityCounter_i$, $ExistenceCounter_i\rangle$)
17:     $StabilityCounter_i \leftarrow StabilityCounter_i + 1$;
18:     $ExistenceCounter_i \leftarrow ExistenceCounter_i + 1$;
19:     $StabilityCounter_i \leftarrow min(N, StabilityCounter_i)$;        # Converge to
    the min
20:     $ExistenceCounter_i \leftarrow min(N, ExistenceCounter_i)$;
21:     if ($S_i = s_i$) or ($S_i.ID$ is $= s_i.ID$ and there are archives for $S_i$) then
22:         $ExistenceCounter_i \leftarrow 0$;
23:     else if $ExistenceCounter_i = N$ then        # If the session's owner has
    crashed
24:         Remove $S_i$ from $CurrentList$;
25:         if there are archives for $S_i$ then
26:             Move $S_i$ to $OldList$;
27:         end if
28:     end if
29: end function
```

```
**function** CreateSession
    **if** $CurrentList$ is **not** REPLETE **and** Incarnation.IsStable() = true **then**
        $incarnation.i_i \leftarrow incarnation.i_i + 1$;
        $incarnation.I_i \leftarrow incarnation.I_i + 1$;
        $s_i \leftarrow \langle incarnation.I_i, node.ID \rangle$;
        Add($currentList, \langle s_i, 0, 0 \rangle$);    # Add session to correct position of the ordered Current list.
    **end if**
**end function**
**function** StopMantainingSession
    $s_i \leftarrow \perp$
**end function**
**function** GetStableSession
    **for** $i = 0$ to $CurrentList.length - 1$ **do**
        **if** $IsStable(CurrentList[i]) = true$ **then**
            **return** $CurrentList[i].S$;
        **end if**
    **end for**
    **return** $\perp$;
**end function**
**function** IsStable($\langle S, StabilityCounter, ExistenceCounter \rangle$)
    **if** $StabilityCounter = N$ **and** $ExistenceCounter < N$ **then**
        **return** $true$;
    **end if**
    **return** $false$;
**end function**
**function** CompareIncarnationNumbers($\langle IncarnationNumber\ a, IncarnationNumber\ b \rangle$)
    **if** $a = b$ **then**
        **return** $EQUAL$;
    **else if** ($a$ belongs to $(L-R, L]$ **and** $b$ belongs to $(L-R, L+R \mod (L+1)]$) **or** ($b$ belongs to $(L-R, L]$ **and** $a$ belongs to $(L-R, L+R \mod (L+1)]$) **then**
        **if** $(a + R \mod (L+1)) > (b + R \mod (L+1))$ **then**
            **return** $GREATER$;
        **else**
            **return** $LOWER$;
        **end if**
    **else**
        **if** $a > b$ **then**
            **return** $GREATER$;
        **else**
            **return** $LOWER$;
        **end if**
    **end if**
**end function**
```

Brief Explanation:

From *line 30* until *line 38*: A session is created as long as there is space in *CurrentList* and the incarnation number is stable. Once the session is created, the incarnation number is increased by one.

From *line 56* until *line 72*: When comparing sessions to be place it in the right position of the lists, both incarnation number and ID are checked. This method is used by the algorithm to compare the incarnation numbers of two sessions. I represents the maximum value an incarnation number can reach and S is the maximum number of sessions allowed in the system at the same time.

Algorithm 7 Main Algorithm

```
Require: $incarnationNumber : \langle I, StabilityCounter, ExistenceCounter \rangle$,
    $sessionsManager$, $tokenDisseminator$, $incarnation$;
 1: function UPON NEW_ROUND
 2:     $sessionsManager.NEW\_ROUND()$;
 3:     $incarnation.NEW\_ROUND()$;
 4:     $tokenDisseminator.NEW\_ROUND()$;
 5: end function
 6: function UPON RECEIVE($\langle sessions\_list_j$, $incarnationNumber_j$, $session_j$,
    $packetType_j$, $TTE_j$, $msg_j$, $hash_j \rangle$)
 7:     $incarnation.RECEIVE(\langle incarnationNumber_j \rangle)$;
 8:     $sessionsManager.RECEIVE(\langle sessions\_list_j \rangle)$;
 9:     $tokenDisseminator.RECEIVE(\langle session_j$, $packetType_j$, $TTE_j$, $msg_j$,
    $hash_j \rangle)$;
10: end function
```

Brief Explanation:

The variables *sessionsManager* and *tokenDisseminator* are instances of the *S Session Manager* and *S Session Token Dissemination* algorithms respectively. The variable *incarnation* is an instantiation of the *Maximum_X* algorithm presented in Section 6, where $x_i : \langle Integer \rangle$ that represents the incarnation number.