Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tomás Francisco Cruz Costa

Epidemic broadcast algorithms In a Byzantine environment

October 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tomás Francisco Cruz Costa

Epidemic broadcast algorithms In a Byzantine environment

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Professor Doutor José Orlando Roque Nascimento Pereira

October 2022

ACKNOWLEDGEMENTS

I would like to give my warmest thanks to my coordinator, Prof. José Pereira, for his guidance and dedication, whose experience and innovative thinking contributed a great deal to the quality of the presented work.

I would also like to thank my family and friends, who were always there for me, and that gave me the strength to endure this difficult task.

Finally, I would like to thank my girlfriend Susana, for the continuous support over the past year, both technical and emotional.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Assinado por: **Tomás Francisco Cruz Costa**
Num. de Identificação: 15639738
Data: 2022.10.31 13:37:42+00'00'

CHAVE MÓVEL

ABSTRACT

Peer-to-peer broadcasting algorithms are a scalable and cheap way of disseminating information to a large number of participants. However, most of these algorithms do not consider the possibility of some members acting in an unintended way, with malicious or selfish motives. In order to be useful in a real world scenario, these algorithms must be secure, robust and efficient, even in the presence of adversaries. This thesis presents an overview of the challenges that peer-to-peer broadcasting algorithms face, as well as some of the security mechanisms that can be employed to mitigate them. Thus, we present Bycast, a secure and efficient peer-to-peer broadcasting algorithm that is able to tolerate up to 45% of malicious nodes in the system. In order to achieve a high level of security, Bycast relies on strong membership integrity guarantees that make it harder for attackers to successfully compromise other nodes. In order to force nodes to cooperate, and contribute to the good performance of the system, Bycast employs an innovative auditing scheme that is able to detect nodes that are not cooperating with their resources, and evict them from the system.

KEYWORDS peer-to-peer, gossip, broadcasting, Byzantine.

RESUMO

Os algoritmos de transmissão “peer-to-peer” são uma maneira escalável e barata de disseminar informação para um grande número de participantes. No entanto, a maioria desses algoritmos não considera a possibilidade de alguns participantes agirem de forma imprevista face à especificação do algoritmo, quer por motivos maliciosos quer egoístas. Para serem úteis num cenário do mundo real, esses algoritmos devem ser seguros, robustos e eficientes, mesmo na presença de adversários. Esta tese proporciona uma visão geral dos desafios que os algoritmos de disseminação peer-to-peer enfrentam, bem como alguns dos mecanismos de segurança que podem ser utilizados para mitigá-los. Assim, apresentamos o Bycast, um algoritmo de transmissão peer-to-peer seguro e eficiente, capaz de tolerar até 45% de nodos maliciosos no sistema. Para alcançar um alto nível de segurança, o Bycast conta com fortes garantias de integridade no sistema de membership que torna mais difícil para os atacantes comprometerem outros nodos com sucesso. Para forçar os nodos a cooperarem e contribuir para o bom desempenho do sistema, o Bycast emprega um esquema de auditoria inovador que é capaz de detectar nodos que não estão a cooperar com os seus recursos de forma a removê-los do sistema.

PALAVRAS-CHAVE peer-to-peer, gossip, disseminação, Byzantino.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1

INTRODUCTION

For a long time, computer engineers have turned to nature in search of inspiration for new computing techniques by observing how naturally occurring phenomena behave, to solve complex problems. One of the areas that have emerged from those observations is epidemic-based or gossip algorithms, used in peer-to-peer communication. They resemble the spread of a virus in a biological community or the gossiping of a rumor in a social setting. Gossip algorithms can reliably disseminate information, even when there are node or link failures. They are popular as an effective solution for transmitting messages in large-scale systems, mainly peer-to-peer systems and other types of *ad hoc* networks Haas et al. (2006), but also, and more recently, they have gained relevance due to their usefulness in the sharing of information between nodes in many cryptocurrencies Yakovenko (2018), as well as their application in decentralized machine learning Hegedűs et al. (2019). This resilience comes however at a cost, in the form of message redundancy. To tackle this issue, some alternatives Carvalho et al. (2007); Leitao et al. (2007); Ferreira et al. (2010) have emerged, that extract a structure from the unstructured overlay and use it for a more efficient broadcast while relying on gossip to mask network and node failures. These algorithms, therefore, combine the strengths of gossip, namely their resilience, with the efficiency of structured approaches to provide a reliable broadcasting mechanism that is also efficient.

1.1 THE PROBLEM

Traditionally, gossip-based broadcast algorithms have focused on performance and reliability but in a non-Byzantine environment Birman et al. (1999); Leitao et al. (2007); Ferreira et al. (2010); Carvalho et al. (2007); Pereira et al. (2003). This means that every node follows the protocol in its entirety, without deviating from it either unintentionally or on purpose. This model not only does not take into account malicious deviations from the algorithm, such as malicious nodes who are trying to disrupt the normal behavior of the system, but also does not take into account arbitrary failures such as message corruption. These assumptions are quite optimistic when compared with what is observed in the real world. In reality, several factors can compromise the normal behavior of the algorithm, such as selfish nodes, who will try to deviate from the protocol to their advantage, and attackers who will actively try to compromise the security properties of the system, for example, by forging messages and impersonating other nodes. Some peer-to-peer systems such as Gnutella have been observed to have up to 70% of nodes who shared no files Saroiu et al. (2003), laying the burden on a small number of altruistic peers. These situations will cause a loss of performance on the system because it divides the overall computation by

fewer nodes. A secure broadcasting algorithm needs to be able to handle nodes who behave rationally and are not completely altruistic, either by tracking and evicting the nodes who refuse to cooperate with the system, or designing the protocol in such a way that rational nodes are actively discouraged from deviating from it. It is also essential that the protocol employs security mechanisms to prevent malicious nodes from compromising the experience of other nodes or the security properties of the overall system.

1.2 AIMS AND CONTRIBUTIONS

This dissertation aims to study the security challenges that peer-to-peer broadcasting algorithms face and create a peer-to-peer broadcasting algorithm that is resilient against a (bounded) number of malicious and rational nodes.

We introduce Bycast, a secure peer-to-peer broadcasting protocol that is capable of handling a significant portion of malicious and rational nodes in the system, while still maintaining a good performance. Bycast builds on Plumtree Leitao et al. (2007), by studying its vulnerabilities against a range of different attacks and adding mechanisms that will mitigate their effects. To deal with membership level attacks and decrease the efficiency of Sybil/collusion attacks, Bycast employs a new strategy when it comes to the node view calculation that allows for nodes to not only verify their neighborhood in the overlay but also the neighborhood of every other node in the system. Bycast also employs a decentralized auditing scheme that provides the ability to detect the nodes that are not contributing to the dissemination of information in the system. In order to test Bycast, a extensive evaluation process was created, with the purpose of corroborating Bycast's resiliency against malicious and rational behavior, as well as to evaluate the performance of the algorithm.

1.3 THESIS STRUCTURE

This document is structured in the following way:

- Chapter 2 presents the background on decentralized broadcasting algorithms, with a focus on gossip and tree-based approaches.
- Chapter 3 presents the threats that peer-to-peer broadcasting algorithms face, as well as an overview of some of the mechanisms that are used to mitigate them.
- Chapter 4 introduces Bycast. It starts with an approach for a fully decentralized membership service, and then a description of the semi-centralized membership service used by the protocol, as well as its broadcasting algorithm and security mechanisms.
- Chapter 5 presents simulation based evaluation process of Bycast.
- Chapter 6 presents the conclusions and future work.

2

BACKGROUND

2.1 BROADCASTING PROTOCOLS

The broadcast problem is described as an operation in which a message generated by a network node is sent to all other network nodes. Our definition considers that the source node can use the cooperation of intermediary nodes which allows for the cost of a broadcast operation to be shared across all nodes involved in the activity. Broadcasting protocols can be characterized by four main properties: whether they have a defined structure, their level of centralization, the amount of information they have of the system, and whether random decisions are incorporated in the algorithm.

2.1.1 *Structured vs Unstructured*

In a structured broadcasting algorithm, the overlay is organized into a set topology, either by a central authority or in a decentralized manner, in a way that will facilitate the efficient propagation of information. This approach usually results in better resource utilization, however, usually these algorithms are less reliable in a high churn setting, meaning an environment in which nodes enter and leave the system with a high frequency. In a structured broadcasting algorithm, usually the entrance or departure of a node needs to be accompanied by a repair mechanism that will ensure the maintenance of the structure of the overlay. On the other hand, unstructured approaches do not impose restrictions when it comes to the topology, and instead, nodes form connections between each other. This lack of restrictions mean these topologies are easier to build and to make more robust in a high churn scenario. The unstructured approach is less efficient, since there is no direct coordination between nodes. Each node needs to make decisions by itself, even though it can rely on information that was gathered from other nodes as well.

2.1.2 *Centralized vs Decentralized*

Broadcasting algorithms can be distinguished in their level of centralization. In centralized protocols, there is a main coordinator that makes the decisions regarding the algorithm for every node. This decision-making process can be assisted by gathering information from the nodes in the network. This architecture is usually associated with a concentration of the effort in the coordinator of the system which could result in latency increase and

the compromise of the scalability of the algorithm. Additionally, in the centralized approach the coordinator constitutes a single point of failure that can cause the failure of the whole system. The other alternative is a decentralized solution in which every node is responsible for its own decisions regarding the protocol. This approach does alleviate the risk of a denial-of-service of the system, by not relying on a central coordinator, as well as providing a better distribution of effort in the network, however it is hard to reach the levels of efficiency of a centralized approach, due to the inherent discordance of decentralized decision-making.

2.1.3 *Global vs local knowledge*

An algorithm is said to have global knowledge if the decision-making by the node uses information on the whole network, such as the available links, nodes positions and the overlay in general. This is a difficult property to achieve since nodes need to get information from all the other nodes. This may be viable in small networks, but gets harder to sustain as the overlay grows. Therefore there is another group of algorithms that rely on an incomplete amount of information regarding the system, or local knowledge. This information can for example be aggregated from communication with neighbors, and will be used by the process to make decisions regarding the routing of information.

2.1.4 *Deterministic vs Stochastic process*

It is also possible to distinguish algorithms in terms of how predictable they are. They can work with deterministic or stochastic approaches. When a process is said to be deterministic, no random decisions are made by the nodes. In contrast, a process can be considered stochastic if part of the decisions taken by the nodes are random. When a process is deterministic, multiple executions of the protocol in the same conditions will yield similar results while the results of the execution of stochastic processes will be unpredictable.

2.2 GOSSIP PROTOCOLS

Gossip algorithms are a subset of broadcasting algorithms that are inspired in the way rumors spread. They can usually be characterized as unstructured, decentralized, and stochastic algorithms that rely on local knowledge of the overlay. The algorithm relies on the cooperation of the peers in the system to forward the information to other peers, meaning that the original source of the information does not need to send the information, or even know, every node in the system. When a node wishes to broadcast a message *m*, it chooses *n* (parameter called *fanout*) nodes to whom it will send the message directly, and upon receiving the message *m*, each of those nodes will forward that message to *n* nodes chosen at random as well. When a node receives a duplicate message, which can happen frequently since nodes choose the *n* nodes randomly, it drops the message, so each node will only forward a message to *n* peers once for every message. Each message possesses as well a maximum hop count that registers the number of hops that it has traveled, and upon receiving a message, a node will discard it if the hop counter has reached the maximum (which is usually a tunable system wide parameter). It is important to note that the tuning of these two parameters is associated with a trade-off between fault tolerance and the overhead caused by the protocol. A smaller *fanout* value will mean that there is less bandwidth overhead, since a node forwards a message to less neighbors, however there is also less reliability, since the aspect that gives the algorithm its resilience is the fact that nodes receive the same message through different paths. Similarly, if the *max hops* value is very high, then the message will reach all the other nodes of the network with a higher probability, at a cost of a higher number of messages exchanged in the system. If the value is too low, then the number of messages exchanged will be lower but the probability of a node not receiving a message due to it being discarded is higher.

This algorithm combined with the tuning of parameters *fanout* and *max hops* can provide a reliable broadcasting mechanism even in the harshest of environments in a system with a considerable amount of faults. However, this resilience comes with a price. In order to achieve a high level of resilience, gossip protocols also have a high level of redundancy that allows for tolerance to node and network failures, which means the protocol is inefficient when it comes to bandwidth utilization.

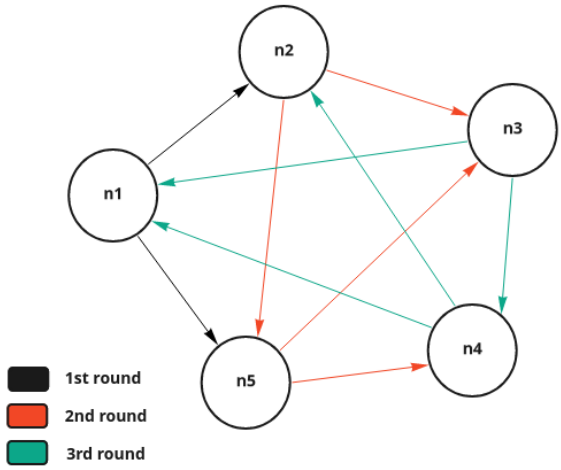

Figure 1: Gossip algorithm illustration in a round-based setting.

Figure 1 showcases a small example that illustrates the algorithm. Node *n1* starts by transmitting this message to two random members of the network, in this case *n2* and *n5*. After that, in round two, both *n2* and *n5* choose two random peers each, to which they will forward the message. In this case *n2* forwards the message to *n3* and *n5*, and *n5* forwards the message to *n3* and *n4*. Finally, *n3* and *n4* also choose two random members each to forward the message to, *n3* choosing *n1* and *n4*, and *n4* choosing *n1* and *n2*. Note that nodes can receive the message from different neighbors, and this is the main factor that contributes to gossip resiliency.

2.2.1 *Gossip strategies*

There are three main gossip strategies for message transmission in a gossip protocol:

- *Eager Push*: The traditional behavior of a gossip algorithm. Each node will forward a message to other nodes as soon as it receives it.
- *Pull*: Nodes do not immediately forward a message to other nodes. Instead, nodes query randomly selected peers for new messages and if they have new messages, then they will request that the message is forwarded to them.
- *Lazy push*: This mode of operation is similar to the *Eager Push* approach but instead of nodes directly forwarding a message to other peers, they forward only the message identifier. Upon receiving a message identifier, a node will make a pull request if the message is new, or discard the message if it is duplicate.

There is a trade-off between Push and Pull strategies. Push strategies achieve lower latency but at a cost of a higher level of redundancy. On the other hand, Pull strategies are more conservative in the way they operate and therefore have a higher latency but less overhead in terms of communication cost. These modes of operations,

combined with the configuration of the parameters which were discussed, namely *max_hops* and *fanout*, must be optimized for each use-case, taking in consideration the performance and reliability requirements of system.

2.2.2 *Bimodal Multicast*

Bimodal multicast Birman et al. (1999) was one of the first approaches for building a reliable multicast algorithm with gossip. It works with a two-phase mechanism. In the first phase, the protocol disseminates the messages using a simple but potentially unreliable tree-based multicast algorithm. In a second phase, participants engage in exchanges, in order to mask the omissions that occurred in the first phase. They do this by exchanging a summary of received messages identifiers. After this process, the nodes request the messages which they need. In the bimodal multicast algorithm, each peer stores and gossips the messages for a maximum number of rounds. When this limit is exceeded, for any given message, the actual message is purged.

One of the downsides from this approach is the complexity of the two-phase nature of the algorithm. Therefore a pure gossip algorithm could potentially achieve a better efficiency.

2.3 HYBRID STRATEGIES

Gossip strategies, specially the *Eager Push* approach, although resilient, have a considerable bandwidth utilization due to their redundancy. Therefore there have been algorithms Carvalho et al. (2007); Leitao et al. (2007) that have taken the approach of using an embedded structure that appears in the overlay in order to disseminate the information in a more efficient way, while still relying on gossip to provide a level of fault-tolerance that would not exist if only the structure was used. This approach combines the benefits of both approaches and minimizes their disadvantages. They are, however, not perfect. The conciliation of gossip and structured broadcast does mean that they get some of the advantages of pure gossip algorithms, like their resilience, and the advantages of structures broadcast, such as its efficiency, but it also means that these algorithms suffer with the disadvantages from both approaches as well, mainly some of the overhead of the gossip approach and some of the latency of the structured approach. It does, however, make the case for an overall lightweight set of algorithms that can provide a high level of resilience in an environment with failures and a latency that is very respectable.

2.3.1 *Plumtree*

The Plumtree algorithm Leitao et al. (2007) is a proposal to create a robust broadcasting solution that explores the inherent tradeo-ff between tree-based algorithms and epidemic approaches by joining the two in an hybrid solution. The way it works is by creating a broadcast tree embedded in a gossip-based overlay. This tree is used to broadcast the payload, and all the other links of the overlay are used in a *Lazy Push* mode, in order to mask failures in nodes or network links. The algorithm also maintains the tree in the sense that if a failure occurs in a node that is part of the broadcasting tree, then the algorithm will do the necessary adjustments until the tree is repaired.

Architecture

The Plumtree protocol is divided by two main functions:

- Tree construction: This component is in charge of generating the embedded broadcasting tree by selecting the links of the overlay that will be part of it.
- Tree repair: This component is in charge of repairing the tree whenever faults occur. This means that in case of a single or multiple failures, the protocol should make sure that the tree tree is repaired until it connects all nodes.

Tree construction

Each node maintains two sets of peers. A set of *Eager Push Peers* for which the node uses the *Eager Push* approach (sends the full message with payload), and a set of *Lazy Push Peers* for which a node uses the *Lazy Push* approach (sends only the identifier of a message). When a node receives a new message from a peer, it includes that link in its *Eager Push Peers*, which means it will be part of the embedded tree. This action will also assure that the link is bidirectional. When a duplicate message is received, its sender is moved from the *Eager Push Peers* set to the *Lazy Push Peers* set (because there was another faster node which provided the message before). A *PRUNE* message is also sent to the node so that it can also be aware that this link has been moved to the lazy push mode.

After the first broadcast is terminated, the construction of the embedded tree is finished, and the algorithm will begin to use both methods of operation, by sending *IHAVE* messages that contain only the id of a message in the *Lazy Push* links and *GOSSIP* messages that contain the payload in the *Eager Push* links, which are part of the embedded tree. The algorithm specifies that a scheduling policy for the *IHAVE* messages can be used, by aggregating multiple *IHAVE* messages and sending them in a single message. This scheduling policy does not have an impact in the algorithm's correctness as long as it assures that every *IHAVE* message is eventually scheduled for transmission.

Tree repair

Whenever there is a failure in a node or a link, at least one tree link is affected and with high probability there will be a partitioning of the tree (this does not happen only if the failure is in a leaf node). The way the algorithm deals with this situation is through the use of the *Lazy Push* messages which are exchanged in the links that are not part of the tree. Whenever a node receives an *IHAVE* message and notices that it has not received its corresponding *GOSSIP* message, it marks this message as missing, and starts a timer *t*. If the corresponding *GOSSIP* message is received before it expires, then this timer is canceled, and there will be no need to make modifications in the tree. However, if this timeout expires, then the node sends a *GRAFT* message to the node which sent the *IHAVE* message. This *GRAFT* message is part of the repair portion of the algorithm. Whenever a node sends a *GRAFT* message, it moves the destination node from *Lazy Push Peers* to *Eager Push Peers* as this link will now be part of the tree, and whenever a node received a *GRAFT* message, it moves the origin node

from *Lazy Push Peers* to *Eager Push Peers* and forwards the message which originated it. Therefore this *GRAFT* message has not only the purpose of triggering the transmission of the missing message but also adding the link to the embedded tree.

2.3.2 *Thicket*

Thicket Ferreira et al. (2010) is another hybrid algorithm that builds on *Plumtree* to assemble multiple trees on top of an unstructured overlay. Single tree based algorithms are very unbalanced when it comes to the effort that each node does in the system, since leaf nodes get to receive the information without having the burden of forwarding it to other peers, and therefore Thicket uses multiple trees in order to balance the amount of work each node does, by adjusting the number of trees where a node is an inner node and a leaf node. The remaining links are used with the purpose of ensuring the complete coverage of the spanning trees, as well as fault tolerance and load balancing. In order to ensure that most nodes are only interior in a single tree and to balance the load imposed on each participant, each node keeps an estimate of the forwarding load of its neighbors. Nodes aggregate this information by having their neighbors send them the number of nodes to which they forward messages in every tree. This strategy allows for a higher level of load balancing, when compared to the use of a single tree, but it assumes that nodes do not lie when they share the effort they are currently doing.

2.4 PEER-SAMPLING SERVICE

In decentralized broadcasting algorithms, every node must be provided with a view of the overlay. This view constitutes a set of peers that are part of the network and that the node will be able to directly exchange information with. The peer-sampling service is in charge of initializing and maintaining the views of the nodes, and also take into account the dynamicity of the system. When a node joins the overlay, the peer-sampling service should provide it a view of the system, consisting of a random set of peers, and when a node leaves it is also responsible for removing the leaving node from the views of every node containing it. Depending on the characteristics and needs of the application, this membership service can provide nodes with a global view of the system or a partial one. There are benefits and disadvantages to both approaches. Usually partial views are beneficial when it comes to performance and scalability, since in the global views case, memory requirements per member will grow linearly with the number of members in the system. Therefore a lot of algorithms rely on a partial view from the overlay. These are normally considerably smaller than the full system membership Leitão et al. (2010).

2.4.1 *Maintenance strategies*

There are two main strategies for the view maintenance mechanism:

- Reactive strategy: In this strategy, the local views from the overlay only change when a trigger event occurs in the overlay, like a node failing, joining or leaving. If the overlay conditions remain stable then the local views will remain stable as well.
- Cyclic strategy: In this strategy, the partial views are updated every specified unit of time. This is done by exchanging messages with one or multiple neighbors. Contrary to the reactive strategy, there are changes in the views even if the overlay is in a stable state.

2.4.2 *Properties*

A peer-sampling-service should guarantee a set of properties Ruiz and Bouvry (2015) that assure its efficiency. These properties can be seen as the set of characteristics of the graph that is defined by the views of all nodes.

- Connectivity: The overlay should guarantee that all nodes are connected. This means that at any time in the execution of the algorithm there should be a path between any two nodes in the overlay. This property is paramount in assuring the reliability of the broadcast, meaning that every node gets the broadcasted message.
- Degree distribution: The degree of a node can be interpreted as the number of neighbors that it is connected to. The out-degree is the number of nodes that are in his view and the in-degree is the number of nodes that have him in their views. Ideally the degree distribution would be uniform across all the nodes, in order to guarantee a higher level of load balance.

- Average path length: The average path length is the average of all shortest paths between two nodes where a shortest path between two nodes is the minimum number of edges that must be traveled to reach node *A* from node *B*. This number should be low so that the number of edges that a message needs to get to a node is as small as possible.
- Clustering coefficient: The clustering coefficient of a node is the ratio between that node's neighbors and the maximum possible number of edges between the node's neighbors. The higher the clustering coefficient the higher is the number of redundant messages received by nodes.
- Accuracy: Accuracy is the number of neighbors of that node that have not failed as a percentage of all the neighbors of the node. Ideally, peer-sampling services should provide a high accuracy, meaning that the vast majority of nodes in the views are working normally.

2.4.3 *Cyclon*

Cyclon is one example of a cyclic peer-sampling-service that relies on partial views. The size of the views is an algorithm parameter that can be tuned considering the number of nodes in the overlay and the level of fault-tolerance that is desired. The bigger the partial views are, the higher the level of fault-tolerance the algorithm will provide. *Cyclon* relies on a shuffle operation that is executed every round by every node. In this shuffle operation, the node selects the oldest node in its partial view and performs an exchange with that node. In this exchange each node provides to the other a sample of its partial view. This works as a failure detection mechanism as well, because if a node does not respond to a request, then it will be assumed that it failed. As it happens with many other peer-sampling-service's, a node who wishes to join the overlay must know another node that is already integrated in the network.

2.4.4 *HyParView*

HyParView is a peer-sampling service that relies on partial views instead of the complete membership information. This protocol relies on a hybrid utilization of the cyclic and reactive strategies. The cyclic strategy is used to maintain a bigger passive view that is used to assure the connectivity of the overlay even in the presence of faults. The reactive strategy is used to manage the smaller active views that will be used for message dissemination. The links in the overlay are symmetric which means if node A is in B's view, then node B is also in node A's view. This is important to assure that a node has control not only when it comes to its out degree but also to its in degree. A *TCP* connection is maintained between every link in the active views, and is used also as a failure detector. This overhead is not too big because the active views are small.

When a node wants to join the overlay, it needs to know another node that is already part of the overlay, and when a new node A contacts a node B, B inserts A in its active view, even if B's view is full (in that case it will drop a random node from its view to make room for A). After that, B will propagate a *FowardJoin* request using a random walk.

As mentioned before, the active view is managed using a reactive strategy. When a node notices that another node in its active view has failed (this can be achieved by using *TCP* as a reliable failure detector), it selects another node from its passive view to take its place. When a contact is made with this node, the node which initiated the contact provides a priority level. This priority level is high if node A does not have any other nodes in its view, or low, otherwise. When a node receives a connection request with high priority it accepts it with a probability of 1, even if it has to drop a random node from the view. Otherwise, if the request is low priority, a node will only accept it if it has free slots in its view. If node B denies node's A request, A will have to contact another node from his passive view.

The way the passive view is maintained is by using a shuffle operation. Periodically every node chooses a peer to perform a shuffle operation with. This shuffle is propagated using a random walk strategy. It consists of a set of nodes present in each-others active and passive views, that are traded between them. The conjugation of cyclic and reactive strategies combines the stability of reactive strategies with the resilience of cyclic protocols, and the result is a protocol that is able to preserve very high values of reliability, even with a percentage of failing nodes as high as 80%.

2.5 UNFAIRNESS IN BROADCASTING ALGORITHMS

In an ideal world, when considering a peer-to-peer broadcasting algorithm, each participant would contribute the same amount of resources and get the same quality or experience in return. However, in reality this does not always happen and some nodes of the system end up contributing with more resources than others. This can happen naturally as a result of the unbalanced nature of the algorithm itself but can also happen as a result of deliberate attempts by rational nodes to contribute less to the network by deviating from the protocol specification.

2.5.1 *Inherent Unbalance*

Broadcasting algorithms are not necessarily designed while considering load balancing requirements. One classic example of this are tree-based broadcasting algorithms. In tree-based algorithms, the leaves do not contribute to the propagation of the information, as shown in Figure 2, since they only have one neighbor, which is the one who sent them the information. This is the case in the Plumtree algorithm. The inner nodes do all the work of forwarding information to their downstream peers, while the leaf nodes get updates for free, i.e., without contributing back to the system.

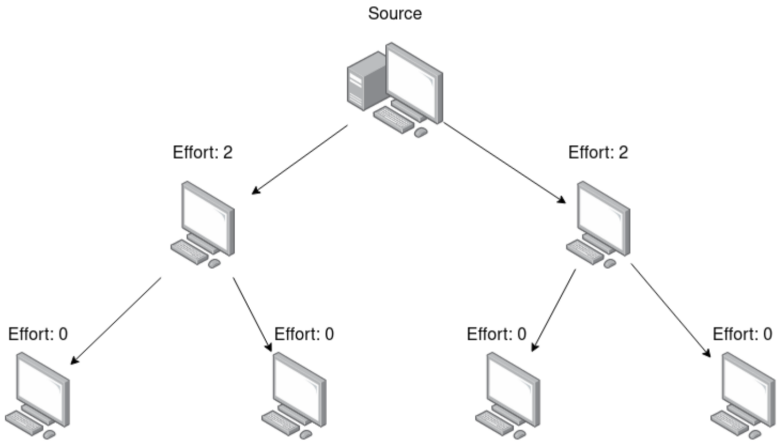

Figure 2: Representation of the effort in a broadcasting tree.

If we simulate the scenario of the Plumtree algorithm where a node is broadcasting information, we can observe in Table 1 that more than half the participants are leaf nodes, which means that more than half the nodes in the system do not contribute any resources to the system, leaving that burden in the inner nodes of the tree, who end up forwarding those messages to multiple neighbors. Furthermore, this situation worsens as we increase the degree of each node in the system, causing therefore a tradeoff between latency of the received packets and the percentage of leaf nodes in the tree. Another aspect that is worth noting in tree-based systems is that the nodes which are closer to the source receive the updates earlier than the ones that are further downstream. This situation can seem appealing at first, as it seems a natural penalization for the leaf nodes to get updates slower than the upstream nodes who are contributing with their resources for the algorithm, however, this is only true for the leaf nodes. For example, as shown in Figure 3, the nodes which are on the level immediately above the leaf nodes, (node B) get a latency that is almost similar to the leaf nodes (node C), however, they contribute with the same effort as the nodes that are on the first level of the tree (node A), receiving updates directly from the source.

Degree of nodes	Percentage of leaf nodes	Latency (in hops since source)
3	33%	4.8
4	49%	3.7
5	55%	3.0
6	61%	2.7
7	64%	2.5

Table 1: Percentage of leaf nodes and latency in hops from the source as the degree changes.

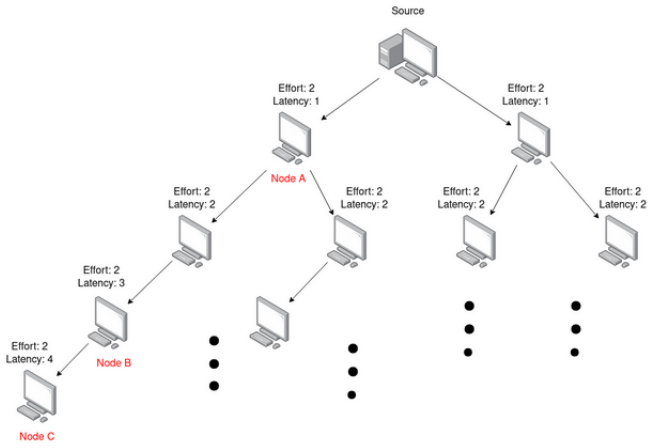

Figure 3: Unbalance of contribution in a broadcasting tree.

On the other hand, an example of an inherently balanced broadcasting protocol is a simple gossip algorithm. Due to the fact that every node chooses randomly the nodes to which they will forward the new updates every round, every node has the same chance of receiving the update early. If we consider the scenario of a standard gossip broadcast algorithm, where each of the nodes always forwards the information to a predetermined number of peers, every node will forward the same amount of messages, assuming they are all following the protocol to its extent.

2.5.2 *Selfish behavior*

The second reason that a particular peer-to-peer system might not be fair is due to the existence of selfish nodes who do not follow the protocol and instead try to minimize their work (bandwidth utilization) while maximizing their gain of the system (minimizing latency). These nodes can be described as free-riders, and their existence, as well as possible strategies to mitigate them, have been the target of a lot of research Zhang et al. (2009); Guerraoui et al. (2010); e Oliveira et al. (2013); Alotibi et al. (2019). By not contributing with resources to the system, free-riders cause the overall resource utilization from altruistic nodes to grow and also ultimately prevent nodes from receiving updates.

3

BAR MODEL

Most broadcasting algorithms and peer-sampling services consider a model where faults exist, meaning that nodes and links can fail, however they do not consider the possibility that nodes might deviate from the specification of the algorithm on purpose, if their interests do not align with the main goals from the system. This leaves algorithms vulnerable to unexpected behavior from nodes that can compromise the experience of other correct nodes, or in the worst case the security properties of the system as a whole.

One model that is more realistic in the sense that it captures the characteristics of a real life peer-to-peer network is the BAR model. The BAR model contemplates three types of nodes: Byzantine, Altruistic and Rational. They differ from each other due to their different goals in the system. Altruistic nodes are those that follow the protocol without deviating from it for any reason. Rational nodes deviate from the protocol when such action benefits them in any way, but have no inherent malice in their actions, meaning that they will not deviate from protocol unless they have something to gain from it. And finally, Byzantine nodes deviate from the algorithm specification arbitrarily, and this includes acting in such in a way that will cause the most harm to the other nodes. Note that Rational nodes do not necessarily need to be analyzed as a different category from Byzantine nodes. By behaving in a selfish way, they are damaging the experience for other nodes, and therefore that could be considered Byzantine behavior. The importance of distinguishing Byzantine from selfish nodes does not arise from the effect they have on the system, but rather their motivations. Since selfish nodes act in a way that would benefit themselves, the argument could be made that if selfish nodes could be punished and have a worse output from the system as a result of misbehaving, they would follow the protocol, since that would be the form of maximizing their output from the system Li et al. (2006); Zhang et al. (2009).

3.1 BYZANTINE NODES

Byzantine nodes are characterized as peers that can deviate indefinitely from the algorithm's specification. This behavior includes purposely acting in a way that will cause the deterioration of experience of other users. They are not necessarily interested in the broadcasted information, meaning that they are cannot be expected to operate in such a way that will maximize their gain from the system or even minimize their effort. This means that a Byzantine node can act in a way that will hurt its performance if it means that it will cause the most overall harm to the system, or to a target victim.

Byzantine nodes can furthermore join together, and collude, with the objective to cause even more damage to the system. This can further increase the difficulty of detecting and punishing these malicious nodes, since they can unite their efforts to hide their steps.

Byzantine nodes can perform a number of different attacks. These attacks range from attacks to the membership service, as well as to the broadcasting algorithm itself.

In order to analyse and study the effects that Byzantine nodes can have on distributed peer-to-peer networks, it is important to establish what the relevant security properties are Gheorghe et al. (2010). These properties can be intrinsic to the data that is exchanged, the nodes that are part of the network, or the network in general:

- **Authenticity** A node that receives a message should be able to verify the source of the message. This property allows, for example, for nodes to verify that the messages that they are receiving are authentic and originated from the intended source.
- **Non-repudiation** The property where a node who has sent a message is unable to deny he has sent it. This is important for example to proof that a node has misbehaved and should therefore be punished.
- **Integrity** A node that receives a message should be able to verify that the message has not been tampered with. A node should be able to detect modifications to messages originated from the source, or from its neighbors.
- **Confidentiality** The messages flowing in the system should only be visible and received by members of the system. The need for confidentiality is dependent on the nature of the system using the broadcasting algorithm.
- **Anonymity** The capacity for a node to remain undetected when it comes to its presence and actions in the system.
- **Access control** The ability to validate and verify the identity of the nodes that want to join the system. This property clashes with the anonymity, and that balance usually constitutes a trade-off that peer-to-peer systems must deal with, depending on their specific needs.
- **Availability** The ability of the overall system to be up and running in its normal state.

These malicious nodes can perform a range of different attacks, that try to compromise a number of these properties, from the membership service to the broadcasting algorithm. We describe each one in general and their particular effects in Plumtree.

3.1.1 *Content-poisoning attacks*

Content poisoning attacks break the condition of integrity and authenticity of the information transmitted in the system. They occur when an attacker creates false message updates, or modifies the existing ones, and forwards them to its peers. This attack has a cascading effect, since peers that receive the faulty update will also

forward it to their peers, therefore its consequences are not limited to the immediate neighbors of the attacker. The motivation for this behavior can range from censorship to pure malicious reasons and the degree of the consequences are influenced by the structure of the system, as well as the number of attackers.

In tree-based algorithms, the consequences from a content-poisoning attack heavily depend on the position of the malicious node in the broadcasting tree. If the attacker is a leaf or is positioned in the lower levels of the tree, the effect of the attack will be reduced since the faulty update will be forwarded to a low number of peers. On the other hand if the malicious node is positioned close to the source, then it will affect a higher number of peers. As we can see on Figure 4, if an attacker is positioned in the first level of the tree, (in this case assuming a binary tree), then it will have control over the updates of half the nodes in the system.

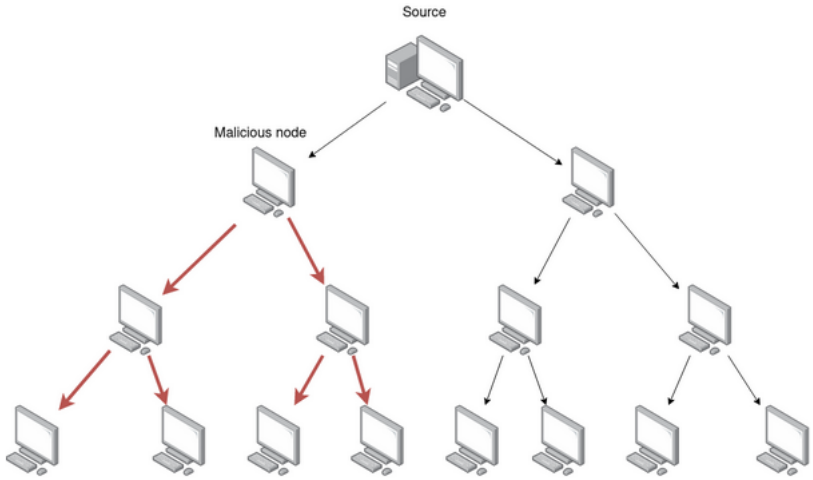

Figure 4: Effect of a malicious node on a broadcasting tree.

If we simulate a tree-based algorithm with 100 nodes, with an overlay of degree five, we can see that, when the attacker is situated in the upper levels of the tree, the number of nodes affected is substantially higher, because more nodes are downstream from it. As shown in Figure 5, on average, a malicious node provided malicious updates to around 20% of the nodes when it was positioned in the first level of the tree, and this reach decreased as the node was positioned in lower levels.

Furthermore, since the tree remains static unless it needs to be repaired (e.g., due to churn), it means that the affected nodes are always the same ones, namely the nodes that are downstream from the attacker. It is also important to note that these observations assume that the attacker does not have the ability to change his position on the broadcasting tree, and is only available to corrupt the peers that are downstream from it.

If we consider a traditional gossip algorithm, the effects of content poisoning are considerably different. Due to gossip's unstructured and random nature, the damage that a malicious node causes is more spread out through the whole network, and reaches on average a higher number of nodes. In our simulations, after the gossiping of 100 messages, 47% of the nodes had received at least one faulty update, however the global percentage of faulty updates received was only around 0,5% for each node. This means that in traditional gossip algorithms attackers are able to influence a bigger number of nodes, but contrary to the tree based approach, they influence a small percentage of updates for each of those nodes.

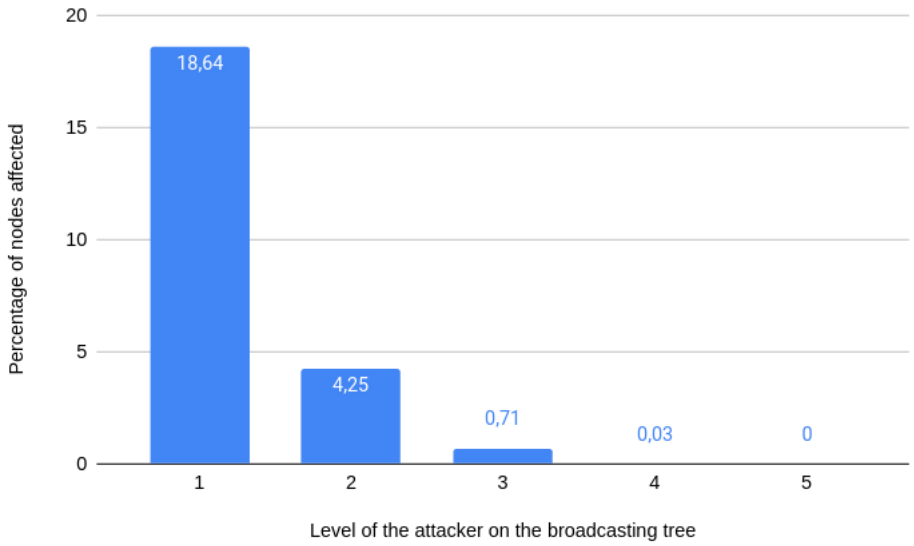

Figure 5: Percentage of affected nodes considering the level of the attacker in the tree.

When we consider Plumtree, due to the actual forwarding of information being done using a broadcasting tree, the effects of content-poisoning attacks are similar as the ones observed in traditional tree-based approaches. Since Plumtree did not have any security measures, a node which behaved in such a way would not be penalized, and the nodes that received the messages forwarded by him would not notice the attack since no checks are done.

Methods for mitigation of content-poisoning attacks

Mitigating content-poisoning attacks can be done in a variety of ways, and depends on the nature of the system. For example, if peers are able to distinguish legitimate updates from malicious updates, mitigating content-poisoning will be easier, since correct peers next to the attacker would immediately notice that the updates are faulty, and could take actions against the attacker, therefore canceling the cascading effect that was mentioned earlier. However, if peers are unable to distinguish faulty updates from truthful ones, it will be harder to stop the attack at the source, since the peers do not immediately recognize faulty updates, and even a correct peer could forward faulty updates.

One mechanism that would circumvent this issue is using the redundancy of updates that is typical of peer-to-peer systems to verify the authenticity of these updates. If a node received multiple instances of the same update from different neighbors, as it happens in push-based gossip algorithms, and they all matched when it comes to their content, then it would have a high degree of confidence that the update was authentic. If however the content of the update differed, then it would know that one of the neighbors is lying, and therefore could employ a mechanism to deal with the situation.

In the Plumtree algorithm, an update is received by a node from only one neighbor, however, the node also receives the alert that there is the new update (*IHAVE* message) from the neighbors whose links do not belong in the broadcasting tree. This means that if these *IHAVE* messages contained the hash of the update, for example,

a peer could use them as a source of confidence that the update it received is correct. This does however bring the disadvantage of a node not being able to deliver an update to the application before it received the corresponding *IHAVE* messages from its neighbors, to verify its integrity. One important aspect that should also be considered is how to deal with conflicting *IHAVE* messages. When this happens, the node knows with certainty that one of the messages is malicious but it does not know which one. The node could use a majority system to select the update which it considers truthful, but this means that it could deliver the wrong update in a situation where more than half of the *IHAVE* messages point to a wrong hash of the update.

SIGN ALL The most intuitive approach to combat content-poisoning attacks is to use asymmetric cryptography to guarantee the authenticity and integrity of the updates. In this scenario, the source calculates a signature of each update, by using its private key, and joins the signature to the update, while every node that receives an update verifies it by using the source's public key. This means that, assuming the cryptographic functions used are safe, an attacker cannot spoof an update to a neighbor, since they will not be able to sign it without the source's private key. This approach has the advantage of being secure, however there is a considerable overhead caused by the signing and verifying of each packet. For example, a source that is sending information at 3000 kbps bitrate, with an update size of 1024 bytes, would mean that the source would have to sign around 366 updates per second, and correspondingly, a receiver would have to verify 366 updates per second as well. Thus, we consider this approach too expensive to be viable.

LIST SIGNING One way of optimizing the "sign all" approach, would be for the source to calculate the hashes for a group of updates, and sign this checksum. The source would then forward this signed checksum first, followed by all the updates that were considered in it (unsigned). This means that the receiver will use this checksum to validate all the updates that are considered in it. One of the problems with this approach is that the source needs to delay the forwarding of messages until it has enough available to create the signed checksum. Meaning that a signed checksum of n messages, which will amortize the cost of a signature by those same n updates, will correspond to a delay of n messages in the source. Another problem with this approach is that if the receiving node receives the updates before receiving the signed digest in which that update is considered, it will not be able to verify the update until it receives the signed digest. One possible solution is to use a more resilient broadcast for these signed checksums such as flooding, since the benefit of the timely arrival of these signed checksums outweighs the added bandwidth usage of flooding these updates.

MERKLE-TREE CHAINING Merkle tree's chaining can be seen as a generalization of signing lists, and in this approach, the source builds an authentication tree that corresponds to a block of updates. Each leaf of the tree corresponds to an update, and the other nodes of the tree are built as the hash of the concatenation of their children. The hash of the source of the tree is then signed by the source, as illustrated on Figure 6, providing authentication guarantees to all the updates that belong to it. For receivers to verify an update, they need the signed top hash, the update position in the block, as well as all the neighbors nodes in the tree path to the source. This can mean that a node does not necessarily need access to the whole authentication tree to verify an update,

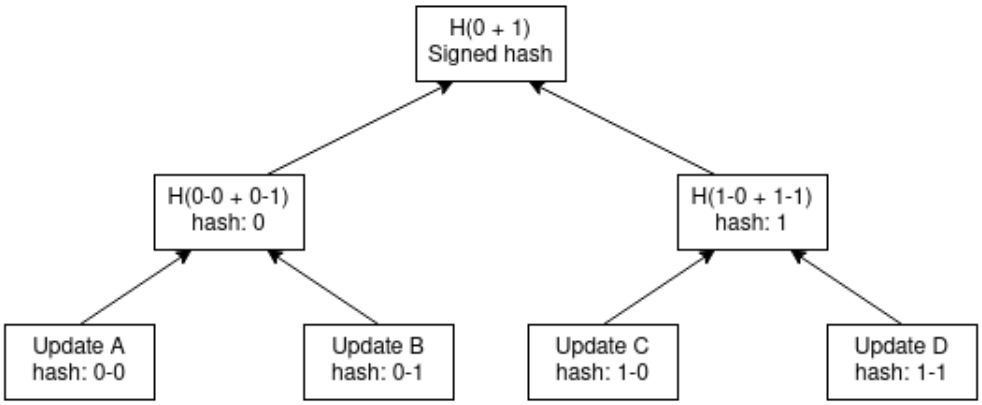

Figure 6: Generation of the Merkle-tree.

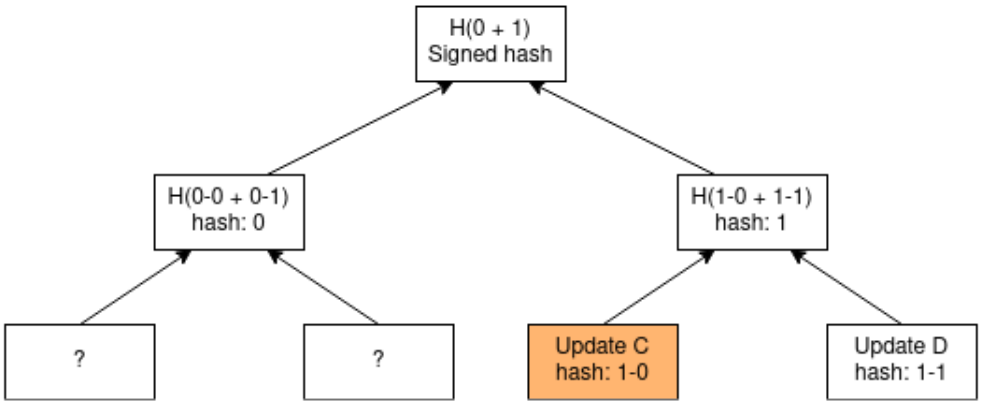

Figure 7: Verification of the Merkle-tree.

as depicted in Figure 7, at the cost of a slightly higher bandwidth overhead, as well as computation overhead. While in the list signing approach, the source must perform n hashes and 1 signature, with binary merkle-trees, the source must perform $2 * n - 1$ hashes and a signature. Similarly, the receiver must perform n hashes and 1 signature in the list signing approach while $2 * n - 1$ hashes and 1 signature must be performed by the receiver when Merkle-trees are being used.

HASH CHAINING Hash chaining is one of the approaches that enables the authentication of a continuous stream of packets, broadcasted over an unreliable medium to a group of receivers.

This form of amortizing digital signatures is different from the previous ones presented in the sense that a continuous stream of messages is broadcast by a sender. the authentication of the n^{th} message in the stream is achieved on the receipt of the $n + 1^{\text{th}}$ message. Thus, receivers use information in later packets to authenticate earlier packets, and this authentication can be traced back to the first message sent by the source, which was signed using conventional asymmetric cryptography.

TESLA Perrig et al. (2000, 2002) is a hash chaining algorithm that is based on this concept. It uses pseudo random functions and message authentication codes, or MACs, and it is based on the timed release of keys by

the sender. The sender starts by creating a signed hash of a generated key k without revealing it and transmits it to the rest of the network. The sender then generates a MAC that will authenticate the first packet using the key k. This authenticated packet will also carry a newly generated hash of a new key $k1$ that will authenticate the following packet. This way, the signature of the first hash of k will be the root of authentication for all the following packets. This approach allows authentication of all messages using only one digital signature for the first message, while relying on the utilization of hash functions and message authentication codes for the following. However this approach has some security vulnerabilities that must be considered. If an attacker is able to receive the packet P_{i+1} before his neighbor receives packet P_i, then he will be able to spoof the contents of the packet P_i to this neighbor, since the key that is used to authenticate packet P_i is disclosed in packet P_{i+1}.

3.1.2 *Sybil attack*

The Sybil attack Douceur (2002) is a type of attack on a peer-to-peer networks in which an attacker creates a large amount of pseudonymous identities, to increase its influence in the system and therefore subvert the system's behavior.

The consequences from a Sybil attack vary from system to system, depending on the characteristics of the environment and the security features it presents. The attacker can use this increased control over the system to subvert reputation systems, to deny service to some peers, subvert a voting result, etc. The Sybil attack can be thought of as a stepping stone to perform another, more disruptive attack, such as content poisoning. If an identity is able to control a large number of entities/peers of the system, the reach of the attack will be far superior compared to if the attack was performed by a single peer. Sybil attacks are usually performed at the membership level of a system, and are difficult to counter unless rigid mechanisms for identification assignment are used. The entrance barrier in a peer-to-peer network depends on how difficult it is to obtain a new identification Dinger and Hartenstein (2006). If nodes' identification can be easily spoofed and are not verifiable by other nodes, it is trivial for an attacker to create an arbitrary number of different identities. This process should also be safe and resilient, to avoid being attacked by malicious nodes. There are two main strategies when it comes to identifier assignment:

CENTRALIZED IDENTIFIER ASSIGNMENT Identifiers are issued by a centralized authority such as a CA, that signs id's and entrance requests. Relying on a centralized authority can help mitigate against Sybil attacks by, for example, employing identity validation measures such as identity cards, or add a cost to a node that wishes to enter the system like a small fee, to discourage and make Sybil attacks impractical.

One disadvantage of the centralized approach is that it constitutes a single point of failure because even if it is not crucial for the actual sharing of information, it is impossible for new nodes to join if it is unavailable. The centralized authority also has a computational overhead that is not amortized by all nodes, which can mean higher costs, and the need for additional infrastructure. In this approach, the way nodes would verify the authenticity of the id could be the traditional CA (Certificate Authority) scenario, in which the CA signs the certificate issued to a determined node with its private key and every node has the chance to verify it with the public key of the CA.

DISTRIBUTED ASSIGNMENT WITH EXTERNAL IDENTIFIERS We assume that all the peers have an external identifier, agnostic to the system, that we can use to deterministically generate the node id for the system. One example would be for the system id of a node be the hash of his IP address. In this scenario, the security of the identification process for our system is transferred to the security of global IP address issuance, which is ensured by the providers, sub-registries, etc, meaning that the level of security of the ids of the system is the level of security of the issuance of IP addresses. The advantage of this mechanism over a centralized authority is that there is not a single point of failure and there is no computational overhead. In contrast to central ID assignment, using external identities does not require an additional entity.

3.1.3 *Eclipse attacks*

An eclipse attack builds on the Sybil attack or collusion between different attackers to cause damage to other peers and the overall system. It happens when an attacker is able to surround a benign node or group of nodes, for example, by filling their view with attacker's identities, giving it full control of the benign peer's interactions with the system. With this control, the malicious node can for example starve the victims of messages from the system, or partition the overlay in a way that peers from one side cannot communicate with peers from the other.

As it happens in a Sybil attack, an eclipse attack is performed at the membership service level. If a malicious node is able to influence or control the view of the other nodes to the point where he can starve the victim's view of benign nodes, he will be able to successfully eclipse the victim from the rest of the system. The effectiveness of an eclipse attack depends on the nature of the membership service. Partial membership views are more vulnerable to this type of attack than full membership views, since it becomes easier to control a node's view when it is smaller and maintained using a fixed policy, which can usually be exploited. Another important aspect is whether the peer-sampling service is reactive or cyclic in its nature. By constantly changing the views from time to time, cyclic strategies are more naturally resilient to eclipse attacks, as it becomes harder for an attacker to encircle a benign node, since an attacker would have to actively prevent the peer from inserting a correct node into its view. One other aspect that affects the effectiveness of eclipse attacks is the method used to dissipate membership information among nodes. When push strategies are used, meaning that peers will take the initiative of sending their membership to other peers, the membership service is more vulnerable since an attacker can exploit this mechanism to target legitimate nodes with poisoned views. If the membership dissipation is mostly based on pull approaches, the legitimate node will have control on who to ask for the membership so an attacker does not have the same ability when it comes to targeting another peer.

Security measures to deal with this attack revolve around preventing a malicious node from influencing or taking control over correct nodes' views. One way this can be done is by putting some restrictions on the peer selection process so that contacted peers can verify that the contacting peer is acting according to the algorithm.

3.1.4 *Man-in-the-middle attack*

A man-in-the-middle attack is when an attacker secretly intercepts, relays and possibly alters the communications between two parties who believe that they are directly communicating with each other.

To pretend to be both parties, the attacker listens in on a communication session. An attacker uses a clever technique to place itself between two nodes that are exchanging data in a network so that the attacking host receives all data that should only travel between the two original hosts. If the attacker is passive, the attack may go undetected. The attacker in the active attack method has the option of altering the data being exchanged with the objective of, for example, performing a pollution attack, or affect the reputation of the node that it is spoofing. Most mechanisms that counter this attack revolve around providing authentication for the peers that are communicating. Considering Plumtree, the man-in-the-middle attack can be used by a malicious node to force it to be cut off from the broadcasting tree. If the malicious node, for example, forwards old messages to node A while impersonating node B, node A will detect this duplicate message and prune that link, as per Plumtree specification. If the malicious node does this for every link that node B is connected to, then it will be completely separated from the broadcasting tree. Fortunately, due to the tree repair mechanism already incorporated in Plumtree, consisting of the *IHAVE* and *GRAFT* messages, node B would be able to detect that it was cut off from the broadcasting tree, since it would receive *IHAVE* messages referring to an update while not receiving the update itself, and it add that link to the tree. This means an attacker would need to perform the attack repeatedly, to force the correct node to be constantly repairing. However, even if an attacker were to do that, it would not affect the reliability of the correct node.

Existing security measures that can be used to counter man-in-the-middle attacks revolve around providing some form of authentication between both parties, so that each party can be confident they are talking to the desired entity, and not an attacker impersonating said entity. The most intuitive mechanism would be for each node to have their own pair of keys, issued by a centralized authority, so that a node A could sign its messages to node B with its private key, and node B could verify these messages with node A's public key. The downside of this approach is that signing each message will add a significant overhead that is not be sustainable in real world scenarios.

One more efficient mechanism that can be used is a KEM (key encapsulation mechanism) where both symmetric and asymmetric encryption are used to ensure the authenticity of the exchanged communication between two nodes. In this mechanism, a node generates a session key, encrypts it with the other node's public key, so that the other node can decrypt it with its private key. After that, all communication between the nodes can be authenticated using a symmetric key mechanism such as a CMAC Dworkin (2005). In this case, asymmetric encryption is only used to verify and safely exchange the session key, and after that only symmetric key algorithms will be used, which are a lot faster. One downside of this approach is that even though every message is authenticated, there is no guarantee of non-repudiation for the exchanged messages. Meaning that node B knows for sure that node A sent that message because node A is the only one that knows the session key, but they cannot prove it since node B is not be able to prove that it did not generate that message itself. This could be an important factor when we consider peer-to-peer systems since nodes will be unable to prove the wrongdoing

of other nodes. Another important aspect of this mechanism is that it is only efficient in peer-to-peer systems where the views are not constantly changing. If the membership service on the basis of the peer-to-peer system is cyclic, meaning that views change from time to time, this would mean establishing a new connection with all the new nodes that are part of that new view and performing the symmetric key exchange. Depending on how often the nodes views' change, this could mean a considerable overhead impact.

3.2 RATIONAL NODES

In peer-to-peer systems, usually there is no central authority, and peers are organized and operate in a self-sufficient way Hoffman et al. (2009). This means that the performance of the system largely depends on the cooperation between peers. In an ideal world, all peers would follow the protocol without deviations, contributing for the optimum level of performance for the system. However, this assumption is rather optimistic, and in reality it is observable that there are nodes that act in a selfish way rather than contributing for the maximum system performance. These nodes are usually given the name of free-riders or rational nodes e Oliveira et al. (2013); Gheorghe et al. (2010). The goal of these nodes is to contribute with the least resources possible (e.g., bandwidth) to the system, while getting the best service possible. These nodes hurt the system by overloading the peers which are following the protocol and providing resources. This can in turn affect the performance of the overall system since overloaded nodes might not be able to keep up with demand. Furthermore, selfish they can adapt their behavior to avoid suspicion by other nodes even when there are security mechanisms in place, making it harder to detect and punish nodes that do not follow the specification of the algorithm. The consequence of free-riders in peer-to-peer broadcasting systems depends on the nature of the system itself. Strategies such as gossip that have a higher level of inherent fault-tolerance will be more resilient to free-rider behavior since it is masked by the redundancy associated with gossip algorithms. If we consider the simulation of a standard gossip algorithm with 100 nodes in which each altruistic node forwards the message to 5 other nodes, we can see in Figure 8 that the impact of the free-riders is very low on the resilience of the system up to 35% percent of rational nodes. The reason for this is that since every node chooses 5 random nodes to forward messages to, and every (correct node) forwards each message once, that means that on average, each node receives each message 5 times, from different sources.

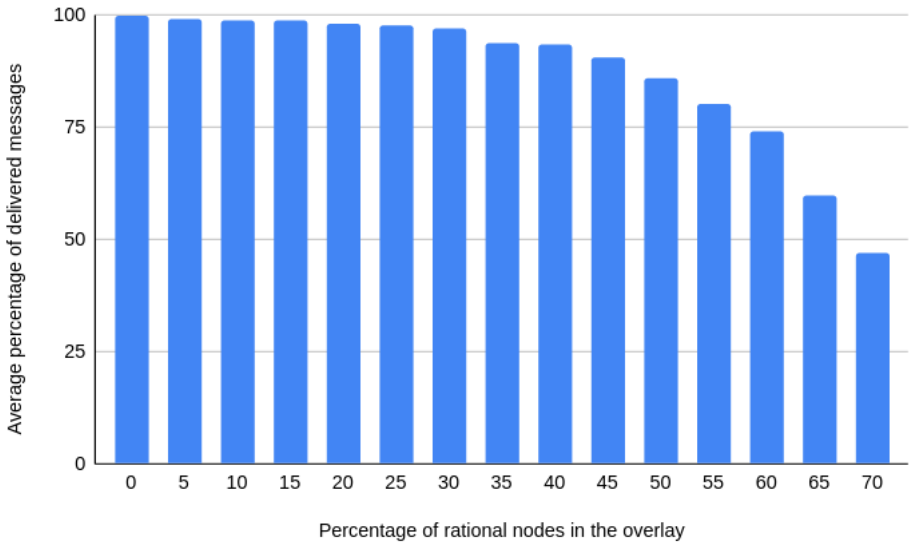

Figure 8: Resiliency (as a percentage of delivered messages) by percentage of free-riders in traditional gossip.

Another important aspect to consider is average last delivery hop for the received messages. As depicted in Figure 9, the increase in the percentage of rational nodes in the overlay leads to an increase of the latency. This happens because since the number of forwarders is decreased, it will take longer on average for the message to reach the nodes.

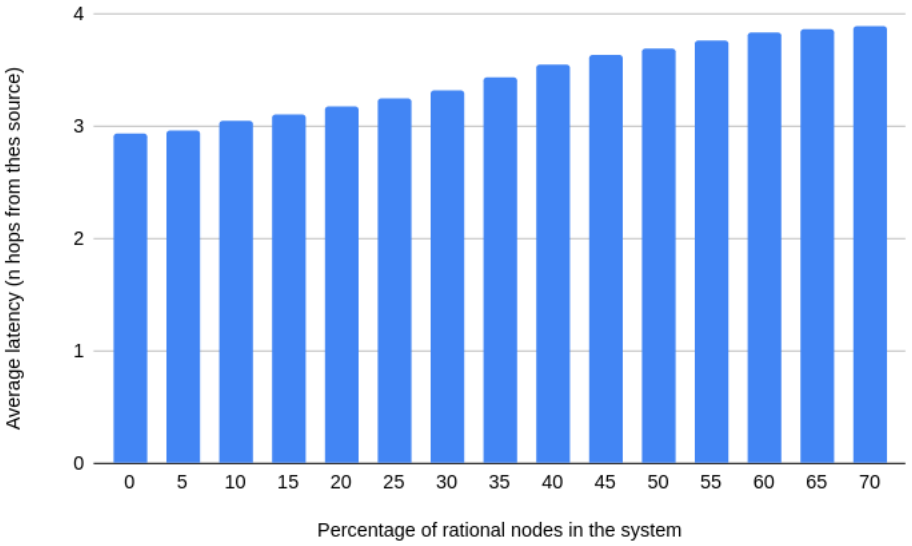

Figure 9: Difference in latency as a the percentage of free-riders grows.

In comparison, considering a traditional tree broadcasting algorithm on an overlay with 100 nodes, we can see, in Figure 10, that the impact of the Rational nodes on the resilience is considerably larger. The reason for this is because of the rigid structure and lack of redundancy of this approach. When a Rational node is an

inner node of the tree, all of the downstream peers from him will fail to receive the messages. As the amount of Rational nodes increases, the chance at least one of them is an upper level node will increase, leading to the quick decline of the resiliency of message delivery.

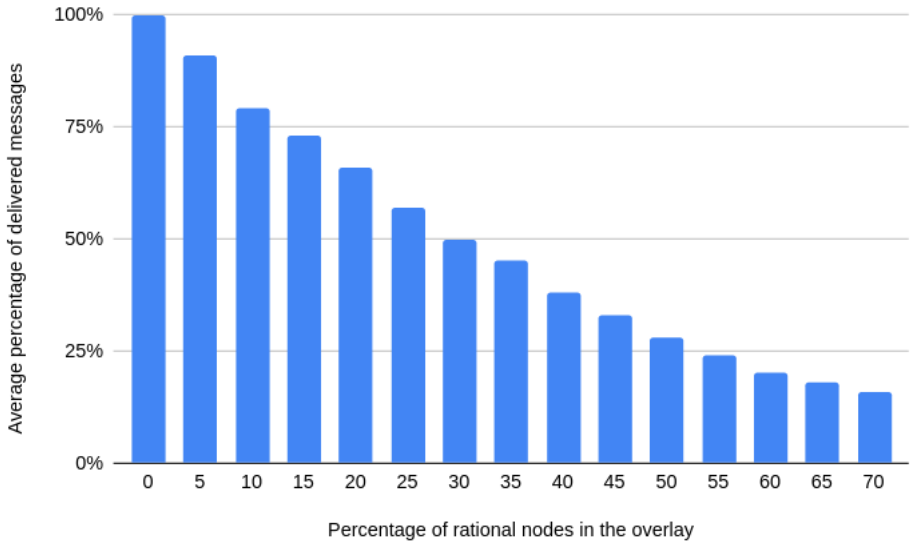

Figure 10: Resilience (as a percentage of delivered messages) by percentage of free-riders in a broadcasting tree.

In the traditional tree broadcast it does not make sense to measure the evolution of last delivery hop as a function of the percentage of free-riders since the nodes either receive the messages with the same latency (since the tree is static), or do not receive the messages at all, due to the fact of being downstream from a Rational node.

3.2.1 *Effects on Plumtree*

Plumtree has a natural resiliency to Rational nodes that do not forward messages. This is ensured by the tree repair mechanism that was originally created to handle faults. When a node does not receive a message through the broadcasting tree, due to its parent being a Rational node, it will receive an *IHAVE* message corresponding to that message through another neighbor and the node will then ask for the associated message, as well as change the links in the tree. This means that the reliability of the Plumtree algorithm even in the presence of Rational nodes remains very high unless the node is surrounded by Rational nodes, meaning that it could not receive the information of a new message through any channel.

Even though Plumtree is naturally resilient against Rational nodes, it is still important to create mechanisms to discourage nodes from being selfish. The first reason is that as the percentage of Rational nodes grows in the system, the average latency for the messages received by altruistic nodes also increases. Depending on the nature of the content being broadcasted, this increase in latency might make a difference between a good experience and a bad one. While in file-sharing peer-to-peer networks this increase in latency could be negligible,

the same cannot be said for streaming peer-to-peer networks where an increase in latency, even if small, might compromise the utility of the update received and lead to a worse user experience.

The other reason why it is important to discourage selfish behavior in Plumtree is that the higher the number of Rational nodes in the system is, the bigger the strain will be on altruistic nodes. This happens because that if a large percentage of nodes are unwilling to reciprocate with the forwarding of messages, that effort will fall upon the altruistic nodes who are following the protocol correctly. This situation is not only unfair but will also lead to a worse overall performance of the system since overloaded nodes might have trouble keeping up with the forwarding of messages to a high number of peers.

3.2.2 *Mitigating Rational behavior*

The task of mitigating free-riding behavior is not trivial. If it imposes a high overhead for the system, then it outweighs the damage that free-riders do to the system. Furthermore, the free-riding detection mechanism itself can be a target of malicious nodes, and therefore it must be resilient to manipulation to a certain extent.

Inherent generosity

The first mechanism that should be considered when it comes to mitigating Rational behavior is the existence of altruistic nodes that gain utility from the mere act of giving. The effect of these nodes in the system have an opposite effect to free-riders. They are willing to dedicate more resources to the system than they are consuming. The argument can be made that if the system model considers the existence of free-riders, then it should also consider the existence of altruistic nodes who are willing to provide help to their peers without an immediate reward.

Monetary payment scheme

Monetary payment schemes dictate that peers which consume resources from the system must pay the peers who provided those resources. Each node would start with a pre-determined balance. Every time a node forwards an update to another, they will a certain amount added to their balance. Alternatively, every time that a node receives an update from a peer, they will pay a certain amount to that peer. This scheme discourages peers from free-riding in the system. If a peer is not willing to forward its newly received updates to other peers, while still consuming and paying for updates from other neighbors, then their balance will decrease until it would reach 0, leaving the node unable to get further updates.

The first and most obvious problem that arises from this mechanism is the feasibility of implementing this virtual currency in the system. A node could, for example, tamper with its own balance and change it to a higher amount, or could also refuse to pay for the services of another peer. One solution for this problem would be in the form of a central authority which would be responsible for authorizing the payment between peers and also updating the balance of the peers. That way, the central authority could keep the balance of every peer, updating it at every transaction that would occur, and therefore be successful in preventing a peer from lying

about its own balance. There are, however, two main problems with this approach: The first one is that this central authority would constitute a single point of failure that could be the target of attackers. If this central authority was compromised, then it would be impossible to provide authenticity to transactions or the balances of the nodes, therefore blocking transactions altogether. The second problem is that depending on the number of nodes on the system, as well as the volume of transactions, the load on this entity would be substantial, therefore affecting not only operating costs but also compromising the scalability of the system.

One alternative to this centralized approach is Karma, an economic cooperation incentive system, that works by keeping track of the resource purchasing capability of each peer Vishnumurthy et al. (2003). This capability is presented in the form of a single scalar value, called karma, which captures the amount of resources that a peer has contributed and consumed, and represents the user's standing within the global system. Users are initially awarded a fixed amount of karma when they join the system. Karma uses groups of nodes, called bank-sets, to keep track of the karma belonging to the users. The karma balance is increased whenever the user contributes resources, and decreased whenever he consumes resources. A transaction will only be able to proceed if the consumer has enough karma for the resources involved. Thus, participants are ultimately forced to achieve parity between the resources they contribute and those they consume. The process of a payment between two nodes is the following:

- Node A sends Node B a signed message authorizing A's bank-set to transfer a certain amount to B
- Node B forwards this message sent by A to his own bank-set who in turn check the feasibility with A's bank-set
- If A's balance is enough to pay for that amount then that value is deducted from A's balance and added to B's balance

The integrity of a node's balance depends on the integrity of the corresponding bank-set. If an attacker is able to control a majority of a bank-set of a node, he will be able to tamper with the balance of that node. For big enough networks, it becomes infeasible for an attacker to do this. Considering an overlay of 10^6 nodes, if an attacker somehow gained access to 10% of the whole network, the odds of him controlling the majority of the bank-set of a node is less that $5.6 * 10^-10$.

This approach would be feasible in a system where the objects exchanged are of a larger size and such exchanges are initiated less often, such as a file-sharing service. However, for a broadcasting based algorithm such as Plumtree, it would be unreasonable to initiate the process of verifying the feasibility of an update forwarding between two nodes for every update.

Tit-for-Tat mechanism

The tit-for-tat mechanism, or direct-reciprocation mechanism, is an incentive approach that tries to mitigate the free-riding problem by demanding that exchanges of information are bilateral. This means that node A is only willing to trade updates with node B if node B also has something to offer. In this way, it would be impossible for nodes to act selfishly since they would not get any updates if they also refused to cooperate with other nodes.

At a first glance this mechanism seems flawless. If nodes can't get updates without also sharing their resources, then it's impossible for them to cheat the system. The problem with this mechanism is that the number of updates that each node has to offer needs to be extremely balanced, otherwise the nodes with less capabilities will starve Li et al. (2006). Even considering a naturally balanced broadcasting algorithm such as traditional pull-gossip, the reliability of the message delivery is impacted by the constraint of both nodes having valuable information to the other node in order to be successful in an exchange. When simulating the standard pull-gossip mechanism with an updated version implementing the tit-for-tat mechanism, where every round a node chooses a neighbor at random and trades the maximum amount of useful updates, we were able to observe that, while the traditional pull-gossip approach had a reliability of about 99% when it comes to message delivery, the version with the tit-for-tat mechanism got only a reliability of about 79%. Furthermore, the latency also deteriorates when it comes to the tit-for-tat approach, from about 2.0 hops in the standard version to around 2.8 hops in the tit-for-tat one.

One possible solution for this issue is to allow for exchanges of updates between nodes, even if one of the nodes does not have anything to offer, by demanding that the node without valuable updates to still make the same effort of the other node, sending dummy updates. These updates would not be useful for the node receiving them, and their only purpose is to force the node receiving the legitimate updates to also make an effort. The idea is that if the cost of sending these dummy updates is as high or slightly larger than the legitimate ones, then a Rational node will be encouraged to contribute with legitimate updates, eliminating free-riding behavior from the system. However, this approach presents two major problems. The first one is that this mechanism leads to a high level of bandwidth waste, since the nodes still need to make the effort of sending these dummy updates, and the other one is that there are no guarantees that a node would not provide legitimate updates only to then receive dummy updates from a node with no valuable information to them. If a node observes that its neighbor does not have any valuable information to them, then it still does not have any incentive to continue with the exchange.

When we consider an application of the tit-for-tat mechanism to Plumtree, it is clear that it would not be successful. Due to its tree-based nature, the information exchanged between peers is far from balanced, and therefore the amount of dummy updates sent by the leaf nodes, for example, would be enormous. Furthermore if we consider a scenario where only one source is streaming, this would mean that all the information flows in the broadcasting tree in only one direction, therefore originating one dummy update per legitimate update, doubling the amount of bandwidth used.

3.2.3 *Reputation systems*

In reputation based-systems, peers gain reputation by participating in exchanges of information with other nodes. This reputation can be calculated in a decentralized way, by the nodes that are part of the system, or by a centralized authority that keeps a registry for each node of the overlay. Centralized approaches are simpler to design and offer a higher degree of efficiency, since a peer will be provided with the reputation data that was collected from all the peers in the overlay. The main drawback of this approach is the fact that the central coordinator constitutes not only a single point of failure but also a bottleneck for large systems, since it will need

to receive and send reputation data to all the nodes in the overlay. In turn, decentralized approaches have the advantage of being more robust and scalable, since there is not a central coordinator does not oversee every reputation exchange, but there is a high message overhead that is necessary to maintain the reputation data.

In reputation systems, nodes can collect information about other nodes directly, by analyzing their past interactions with those nodes, or by collecting information about the participant from other peers. Nodes then use that information to compute a score for every peer on the overlay. With that score nodes decide to punish low-reputation nodes, therefore creating the incentive for nodes to cooperate with the system so that they can maintain a good reputation.

Even though indirect reputation schemes are more effective when it comes to the calculation of a node's reputation, since they can use not only their own experiences but also the experiences of other nodes in the overlay, they are also vulnerable to malicious behavior from nodes that wish to subvert the system. Attackers can, for example, target other nodes by reporting a low score for them, with the objective of getting them penalized. Attackers can also perform a "cleaning" attack by banding together and reporting a high score for each other, to increase their reputation. Therefore in order to be secure, reputation systems must take into account the possibility of false reports and employ security mechanisms that prevent malicious peers from influencing them.

3.3 BAR GOSSIP

BAR Gossip Li et al. (2006) is a peer-to-peer data streaming application that is capable of providing a low latency and stable throughput in the BAR model (Byzantine/Altruistic/Rational). To achieve this goal BAR Gossip uses a verifiable pseudo-random partner selection method that allows for the elimination of non-determinism in the algorithm execution but also keeping the advantages of traditional gossip. It also relies on a fair exchange primitive that entices the selfish nodes to collaborate with the algorithm.

The architecture of BAR Gossip relies on the assumption that randomness is inherently bad in algorithms that function in the Byzantine environment, because it gives a chance for Rational nodes to hide selfish actions as legitimate, non-deterministic behavior. Because of this, BAR Gossip uses verifiable pseudo-randomness to build a verifiable pseudo-random partner selection algorithm. This mechanism has the finality of removing the randomness aspect of the algorithm, in order to provide the possibility of verification when it comes to the partner selection, but maintains the unpredictability and rapid convergence of traditional gossip. This approach is joined by a fair exchange mechanism that encourages nodes to trade information with each other, therefore discouraging Rational nodes from free-riding in the system. The authors argue that enticing cooperation over short timescales is more effective and simpler than other approaches based on long term reputation.

3.3.1 *Partner selection*

In traditional gossip, a node periodically selects a partner to exchange information with, randomly. Also, the node which is contacted always accepts that request. This allows for a malicious node to choose the partner which he wants to contact and disguise that decision as random behavior, and compromise the resilience that

comes with gossip. Therefore, BAR Gossip makes nodes generate an unpredictable, deterministic seed for the pseudo-random generator and will provide all that information to the node that is chosen, so that he can verify the integrity of that process using the same pseudo-random generator with the corresponding seed.

3.3.2 *Balanced exchange*

Balanced exchange is the mechanism which is used by nodes to exchange updates. It works by having two nodes determining the largest number of new updates they can exchange while maintaining the trade equal. In each round, a node does both roles. It initiates an exchange with another client and responds to a balanced exchange request from another node. An exchange consists of four phases. The first phase consists of the partner selection, which is done using the partner selection algorithm. In the second phase the two nodes learn about each other's unexpired updates and determine the maximum number of updates that can be traded, as a one-for-one trade. In the third phase, the nodes encrypt the updates that were determined to be traded and send them to the other node, and finally, in the fourth phase, the nodes exchange the keys which are necessary to decrypt the updates. The reason why first the encrypted updates are exchanged and only after the keys are exchanged is to discourage Rational nodes from retaining from sending the updates after getting the other node's updates. This process can be seen as a Nash equilibrium and completing it is in the best interest of nodes that want to maximize their gains from the algorithm.

3.3.3 *Proof of misbehavior*

Proof of misbehavior is the mechanism which is used by BAR Gossip, to ensure that the nodes who send inconsistent messages risk eviction. One example of a proof of misbehavior is a client receiving updates that differ on what was agreed upon. A single client cannot have the authority to evict other nodes, and therefore BAR Gossip introduced a trusted agent of the broadcaster that audits possible POM's (proof of misbehavior). He does this by policing the system every round, and ordering random nodes to provide the POM's that they have gathered. If the node does not have a POM against another peer, it must send a dummy message. This is to discourage Rational nodes from lying to the auditor to save on bandwidth. If the auditor reaches the conclusion that a node has misbehaved, then it will inform the broadcaster by sending it a signed eviction notice, and the broadcaster will include all eviction notices in every update it broadcasts.

3.4 FIREFLIES

Fireflies Johansen et al. (2015) is a Byzantine resilient membership service that uses decentralized monitoring of the overlay in order to provide nodes with reasonably current views of the system. Fireflies is composed of three different sub-protocols. One pinging protocol which is used to detect failures of other nodes, a gossip algorithm that is used to propagate information between members in a bounded time, and a membership protocol that uses accusations and rebuttals in order to maintain the views.

In Fireflies, nodes monitor each other for failures, by using the pinging protocol. They are organized into rings, and their position in the ring is calculated with their identifier. On each ring, every node monitors their successor and if they detect that it has failed, they will issue an accusation and gossip it to other nodes. When a node receives an accusation for another node, it starts a timer with a predefined timeout. If that timeout expires, they remove the accused node from their view. A node who is incorrectly accused can also issue a note, or rebuttal to that accusation, so that the nodes in the system can cancel that timer and the node which was accused will therefore remain in the views from the nodes. In order to deal with Byzantine members that could issue incorrect accusations for other nodes, each node is able to invalidate a portion of the rings, so that accusations generated by predecessors of those rings will be ignored by the nodes in the system.

Byzantine members can disguise themselves as correct members by executing the protocol, or as stopped members by not executing at all, and so a correct member cannot determine which members are Byzantine unless they reveal themselves by sending messages that prove that they are not following the protocol. Fireflies also depends on a central authority to provide potential members with a certificate that will authenticate their identity as well as provide access control to the system if necessary.

3.5 BRAHMS

Brahms Bortnikov et al. (2009) is a peer-sampling service that is able to sample random nodes in large dynamic systems prone to Byzantine failures. It does this by using small local views, which means the maintenance overhead is kept small. Brahms provides independent uniform samples even in an environment with Byzantine failures, as opposed to traditional gossip-based membership services which only ensure that the average representation of nodes in local views is uniform.

The way Brahms achieves uniform independent samples is by introducing a sampler that is able to obtain independent uniform samples from a biased stream of identifiers. By using this sample to update part of the local view, Brahms avoids partitions and assures that long-standing nodes cannot be isolated from the overlay.

3.5.1 *Sampler*

The sampler uses min-wise independent permutations to uniformly sample elements from a data stream, even if this stream is biased. It can be implemented with a pseudo-random function. Consider for example a stream of id's a, b, a, a, c. If we feed this stream to the sampler it will compute the hash of each one and maintain the lowest found so far as the sample. The odds that the sample is a, b, or c is the same, meaning each of these id's hashes has the same probability of being the lowest, as illustrated in Figure 11. Brahms maintains a tuple of sampled elements in a vector of samplers, which are independent and work as a history sample used to update the views.

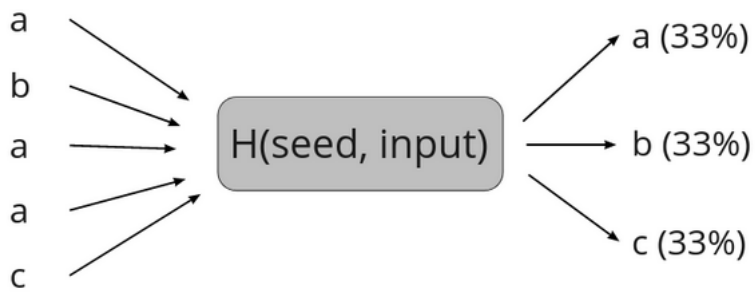

Figure 11: Sampler used by Brahms in order to achieve uniform independent samples.

3.5.2 *Id propagation*

The Brahms view is maintained by using a gossip algorithm. It uses two means for propagation: push and pull strategies. The push strategy consists in a node pushing its id to some other nodes, and the pull strategy consists in getting other node's views. These two methods are not enough to guarantee that the overlay is not partitioned in face of an attack, so Brahms uses history samples to update a small portion of the view. This guarantees that a node always has at least a small amount of correct nodes in its view.

3.6 SECURESTREAM

SecureStream Haridasan and van Renesse (2008); Haridasan and van Renesse (2006) is a peer-to-peer live-streaming system built to tolerate some kinds of malicious behavior. It uses the fireflies membership service as its secure peer-sampling service and counts on its resilience against membership level attacks. The way it deals with content pollution is by employing the list signing approach as a way to amortize the cost of digital signatures and avoid the large overhead that comes with the signing of every individual message. Its broadcasting medium is based on a pull approach, which is naturally resilient to some forms of abnormal behavior such as free-riding. To combat some forms of malicious behavior such as the refusal to forward messages and abundant request of messages by nodes, SecureStream employs an auditing mechanism that joins the efforts of a global auditor as well as the cooperation of the nodes of the overlay in order to detect and punish malicious nodes.

SecureStream opts to model selfish behavior as Byzantine, however there are advantages to doing it separately. The case can be made that it is more realistic to separate the two kinds of nodes, since Rational nodes will follow the protocol to its extend if it's the way they can get the maximum utility from the system while Byzantine nodes might cause harm to the system even if they impact their own experience in the process.

These mechanisms allow SecureStream to handle up to 25% of non-colluding Byzantine nodes in the system.

4

BYCAST

This chapter introduces Bycast, a Byzantine resilient peer-to-peer broadcasting algorithm that builds on Plumtree and its hybrid gossip philosophy. First, an approach for a secure decentralized peer-sampling service is presented. After that, an approach using a semi-centralized coordinator, that is used by Bycast, is presented. Next, the broadcasting algorithm is specified, including the modifications to Plumtree and the security mechanisms used to prevent man-in-the-middle and content-poisoning attacks. The free-riding detection is presented after that.

The security of a peer-to-peer broadcasting algorithm needs to be considered at two different levels. The peer-sampling service, and the broadcasting algorithm itself. One of these being compromised is enough for an attacker to impact the efficiency and reliability of the system.

4.1 MEMBERSHIP

The importance of securing the peer-sampling service against attacks has two major justifications. First, access control is fundamental for the security of a peer-to-peer system Gheorghe et al. (2010); Liu et al. (2008). If the creation of new identities is feasible for an attacker, then the system will be vulnerable to Sybil attacks. An attacker can simply create a number of identities that will allow them to get a bigger representation in the system, making it easier for them to influence the system. Making the task of creating new identities more difficult is the most effective way of handling these attacks. The other reason it is necessary to secure the peer-sampling service is because if an attacker can control its view or even more importantly, the view of other nodes, it will be easier for them to perform eclipse attacks, by partitioning one or more nodes from the rest of the overlay. It will also facilitate collusion attacks, since malicious nodes will be able to unite their efforts in an attempt to disrupt the system.

Considering these two aspects, as well as the requirements that the Plumtree algorithm imposes to the peer-sampling-service, the following properties must be assured by our the sampling service:

- Random uniform views: The views generated by the peer-sampling-service should be as close to random as possible. This will make it harder for an attacker to target a single node from the system since it would not be able to choose its own neighbors, the odds of the victim being in the attacker's view would be the same as for any other node, which considering a big enough overlay, would be quite low.

- Verifiable views: The views of a node should be verifiable by any other node in the system. If this is not true, then it would be possible for a malicious node to tamper with its view and target the desired victim, whether the victim really is on the attacker's original view or not.
- Balanced size views: The views for each node should be the same size. The justification of why this property is necessary is that it will allow for a more balanced work distribution between the nodes. If a correct node had a substantially bigger view then another, the effort performed by this node would be higher. This would not only mean an increase in the unfairness of the system but would also make it harder to detect free-riders in the system, since it would be possible for a free-rider to be perceived as just a correct node who happens to have a smaller view, and therefore dedicate less effort to the system.
- Stability: One of the requirements of the Plumtree protocol is that the views of the nodes are stable, due to its structured nature. If the views of the nodes change often, for example by using a cyclic peer-sampling service, then the broadcasting tree would be constantly repaired.
- Symmetric views: Another requirement of Plumtree is symmetry. Since the tree in the Plumtree algorithm is undirected, that means that if A has node B in its view, then node B also needs to have node A in its view as well.

4.1.1 *Fully decentralized approach*

The first approach at the creation of a peer-sampling service that would meet the list of requirements presented above as an adaptation of the Brahms Bortnikov et al. (2009) peer-sampling service to fit the needs of the Plumtree broadcasting algorithm. The original Brahms protocol guarantees already two of the requirements above without need for modification.

- Random Uniform Samples: The clever use of the sample mechanism by Brahms guarantees that every correct node that follows the protocol will have an approximately uniform sample of nodes in the system, therefore guaranteeing resilience against eclipse attacks by malicious nodes.
- Stability: The sample of each node will converge into a uniform sample of the overlay, and changes to the sample of a node will then only occur if a neighbor leaves the system.

However, Brahms fails to comply with the following requirements:

- Symmetric views: Since every node gets an independently generated random sample of the overlay, the samples of each node are not symmetric.
- Verifiable views: Brahms does not provide a mechanism for the verification of the samples generated by each node. This means that malicious nodes can tamper with their samples and add to them the victim node, or for example another malicious node which he wishes to collude with.

- Balanced size views: Every node in Brahms has the same number of nodes in its sample. However, since samples are generated randomly, the number of times each node appears in the sample of all the other nodes in the system will vary.

An adaptation of the Brahms protocol can be made for it to support symmetric views: Every time that a node A adds a node B to its sample, then it sends a message to node B, so that it will add node A to its in-sample. This way, both nodes are in each others' views.

For verifiable views, it is also possible to add a mechanism to Brahms that would allow for contacted nodes to check, to an extent, the authenticity of the sample of the node contacting it. Brahms specifies that each node possesses a list of samplers, instantiated as different pseudo-random functions to choose the lower node id generated by the function as the sampled id. One possible modification would be for a single well known pseudo-random function to be used for the list of samplers, but each one using a seed that is standardized for all nodes in the system. The reason for this change, is that in this way, a node receiving a message to add another node to its in-sample can calculate the the result of the hash of its own node id, using the contacting node's seeds, in order to judge the credibility of the connection.

As more and more node identifiers go through the sample of a node, the hash value of the sampled node identifier decreases (since the sampled id is the lowest hash value calculated so far). This means that considering a big enough overlay, it is reasonable to expect that the value of the sampled node should be below a certain threshold. The contacting node could compare the calculated hash with this threshold, and if it was below it, it would accept adding this node to its in-sample. If the result of the hash is higher than the threshold, then the node would not accept the connection. This mechanism excels at targeted attacks by malicious nodes. A malicious attacker will only be able to add its target to its view if the resulting hash of the node was below the threshold, which is not likely. This mechanism would also be efficient when it comes to preventing collusion attacks, however the effectiveness decreases rapidly as the number of colluding nodes grows. This is because as the number of attacker grows, the odds of at least two of them being able to generate a hash that is lower than the threshold is higher.

The value of the threshold represents a trade-off between the success in preventing colluding attacks, and the false positive rate. If the threshold is very low, then even correct nodes could have authentic samples that are higher than the threshold, meaning the nodes will not accept their connections, however, the success rate of targeted/collusion attacks is also very low since the odds of the targeted node generating a sample lower than the threshold will be very small. In the opposite case, when the threshold is very high, the false positive rate is low but the success rate for targeted/collusion attacks is higher since the odds of the target of an attacker generating a lower value than the threshold are higher. Simulating the Brahms protocol in an overlay of 100 nodes, each with a sample list of size 5, it is possible to see the change in the success rate for an attack and also the evolution of the false-positive rate. When the threshold is 1, meaning that the first byte of the hash generated for a node id has a value of equal or smaller than 1, (taking into account that the possibilities for the value of the first byte range from 0 to 255), the success rate of an attack is around 3%, however the false-positive rate is extremely high, at around 46%, as shown in Figure 12. This would mean that 46% of authentic connections would be perceived as malicious by the contacted node. As the value of the threshold increases, the success rate of the attack grows

as expected, reaching a percentage of around 30% when the threshold is 8. The false-positive rate decreases to a value of around 4%.

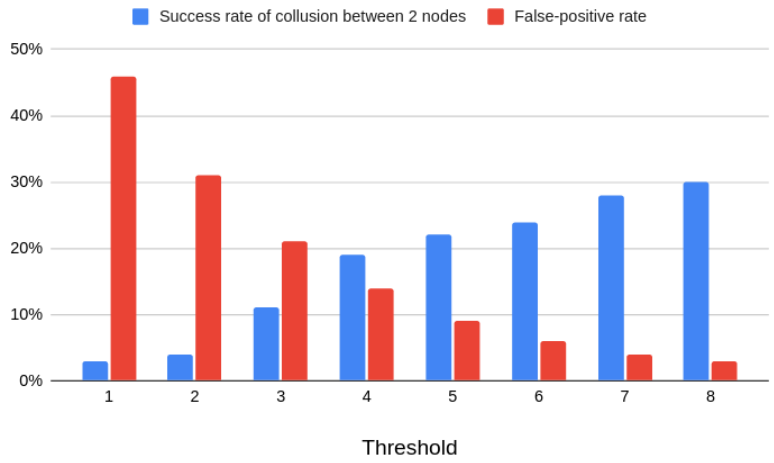

Figure 12: Success rate of collusion between two nodes and false positive rate as a function of the threshold.

Other factors that influence the effectiveness of this mechanism are the number of nodes in the system and also the degree of each node. As the number of nodes grows in the system, the odds of one of them generating a hash value lower than the threshold increase, meaning that the false-positive rate will decline. However, the odds of an attacker being successful in an attack will remain the same, since the threshold has not changed. Assuming an overlay with a degree of 5 for each node, where the threshold is static, and set at 4, it is possible to analyze the change in the success rate of attacks and false-positive rate while ranging the number of nodes in the system. The success rate remains constant at about 15% despite changes in the number of nodes, while the false-positive rate declines steadily from 38% in an overlay of 50 nodes to 2% when the number of nodes is 200, as shown in Figure 13.

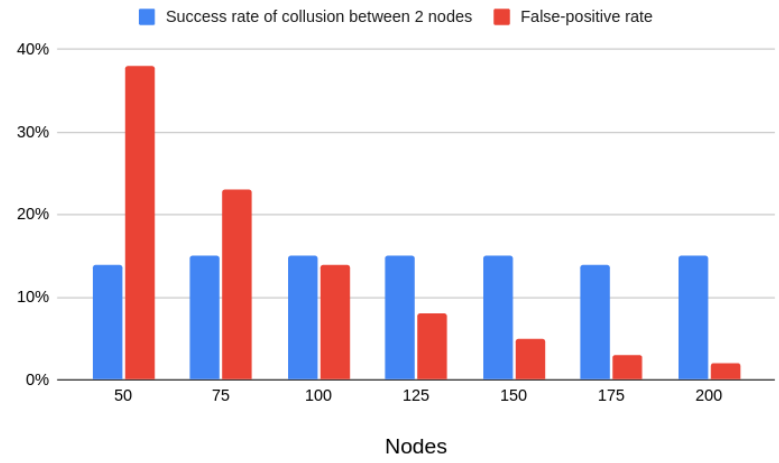

Figure 13: Success rate of collusion between two nodes and false-positive rate as the number of nodes grows on the overlay.

It is also important to note that as the number of nodes in the system grows, it is not expected that the degree of each node will remain the same. That would have consequences in terms of an increase in the latency for the broadcasting algorithm. Therefore it is also important to analyze the effects of a change in the degree in the verification mechanism. An increase in this degree will result in a higher number of samplers used by each node, which will in turn mean that the odds of an attacker being successful in an attack will grow as the sample also increases. This is because an attacker has more attempts when it comes to the possibility of the targeted node generating a sample that is lower than the threshold. Considering the previous simulation, but this time keeping the number of nodes static, at 100, and ranging the degree of a node between 2 and 10, the results are inverted. This time, the false-positive rate remains at a constant value, and the success rate of attacks grows as the degree increases, from 6% when the degree is 2, to around 23% when the degree is 8, as shown in Figure 14.

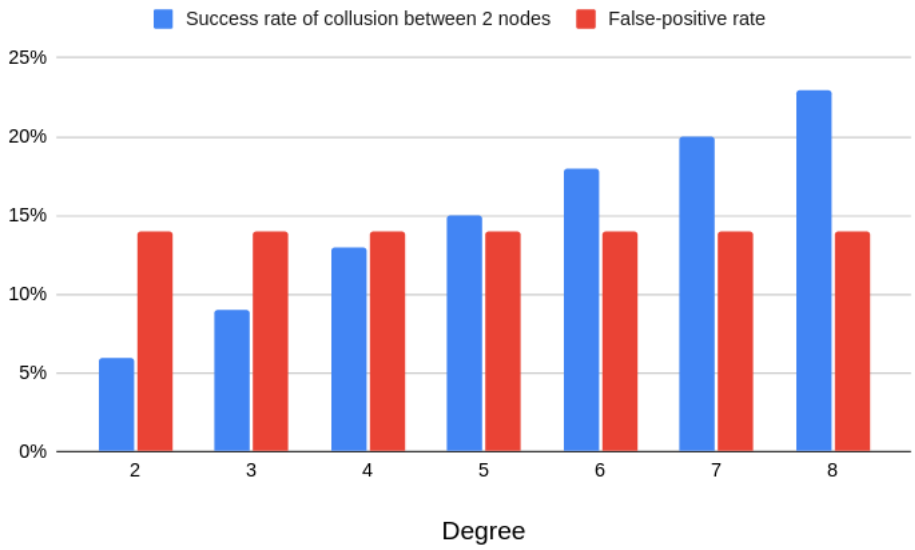

Figure 14: Success rate of collusion between two nodes and false positive rate as the degree of the overlay grows.

Considering both presented mechanisms that are used to modify Brahms into meeting the requirements for Bycast, only one requirement remains unchecked, the need for views of the same size. Even though the sample list is the same size for every node, since the samples are generated randomly, the in-samples will vary from node to node. Some nodes will probabilistically be contacted more than others, meaning that the overall view for each node will vary in size. For an overlay of 100 nodes, with a degree of 5, the ideal length for each node's view would be of about 10. The size of the out-sample would be fixed at 5 for every node, and the size of a node's in-sample would be of around 5 as well. However, after running preliminary tests, it was clear that there is a substantial deviation from the average by most nodes. Even though the mean of the view size is indeed of around 10, the standard deviation is very high, at around 2.3. This will increase the disparity in effort of each node, which will have a big impact on the efficiency of the free-riding detection mechanism. Trying to balance the views of each node would require increased complexity in the calculation of the views, as well their verification,

and for this reason combined with the fact that the verification mechanism for the views presents a high false-positive rate means that this peer-sampling service does not provide the guarantees necessary for the Bycast algorithm, and therefore another approach must be taken.

4.1.2 *Semi-centralized approach*

The difficulty in the conception of a peer-sampling service that fits the requirements that were designated for Bycast, comes from the inherent trade-off between the balance of the view's size and symmetry and their randomness guarantees. It is not reasonable to seek a well defined structure that is created randomly, in a decentralized way, and if rules and modifications are imposed in order to achieve this structure, then we relax the randomness guarantees and open the possibility for malicious nodes to take advantage of the algorithm that generates the overlay. However, the problem of generating random graphs with a fixed degree for each node in a centralized way has been studied extensively, and a number of fast algorithms exist for this use. An algorithm for generating random regular graphs is proposed in Kim and Vu (2003), with a time complexity of $O(n * d^2)$. The idea is to use this algorithm to generate the overlay that will be used by Bycast. Before broadcasting begins, a central coordinator generates a random seed that will be used as the seed for the graph generating algorithm, and distributes it to the nodes that want to take part of the system, along with the full membership of the nodes that will take part in it. Upon receiving this information, every node is able to run the algorithm for the view generation, meaning that every node has at its disposal the information of the view of every node in the system. This is the biggest advantage of this approach, since it will allow for nodes to verify incoming requests by other nodes, as well as prevent Sybil/eclipse attacks. This approach therefore satisfies every requirement that the Bycast algorithm needs from its membership service:

- Random uniform views: The graph generated using this algorithm is provably close to a uniform generator when the degrees are relatively small, meaning that the odds of two nodes being in each other's views will be the same for every two nodes in the system. This property is fundamental to ensure that malicious nodes cannot have an advantage when targeting or colluding with another node. Due to the deterministic nature of the view generation, a malicious node will be powerless to influence it. If an attacker tries to tamper with its view, for example by connecting to a victim node, the victim node can see that this connection is illegitimate and use this attempt as a proof of misbehavior of the connecting node.

- Symmetric views: The random regular graph that is generated using the algorithm provides an undirected graph, meaning that every edge in the graph is bidirectional, and for each two nodes in the system, if A is in B's view, then B is necessarily in A's view as well. No distributed mechanism is necessary to ensure the symmetry in the views, which is important to minimize performance issues, and could also be the target of malicious nodes as well.

- Balanced views: One of the parameters of the algorithm is the desired degree that will be satisfied by every node in the system, meaning that the views from every node will be the same size. This property

will greatly improve the inherent fairness of the system, which in turn will make it easier to discern free-riders from altruistic nodes.

- Stability: The views generated by this algorithm are static and will not change periodically, and therefore mechanism is necessary to deal with nodes entering and leaving the system.
- Verifiable views: The seed that will be used to generate the overlay is broadcasted by the central authority to every node in the system, and therefore, every node in the system is able to verify every other node's view.

JOINING NODES If the membership was open and completely dynamic, and nodes could join while the system was running, it would be necessary to recalculate the random regular graph, since it is not possible to adapt it into including a new peer without sacrificing some of the guarantees mentioned before, like the balanced view size and view verifiability. This limitation will be offset by employing a round mechanism that is used to mitigate this issue and also to provide further fairness to the effort performed by the nodes. Periodically the overlay is reconstructed by taking into account the nodes who left the system and also the new nodes who have asked to join the system since the beginning of the previous round. This means that the maximum time a node would have to wait to join the system would be the duration of one round, that is a parameter that could be configured by the central coordinator in line with the system's needs.

The process for a node joining the system starts with a node requesting a membership certificate from the central coordinator. This central coordinator might have different policies when it comes to access control depending on the application that is using the protocol and is out of scope of this work. After having its access granted by the central coordinator (Algorithm 1, lines: 1 - 6), the node will then wait until the next round of broadcasting begins.

Algorithm 1 Bycast coordinator algorithm

```
1:  upon event NodeJoining(id, PubKey) do
2:      assert id ∉ BannedNodes
3:      allowed ← AccessControl(id, PubKey)
4:      if allowed then
5:          MembershipNextRound ← MembershipNextRound ∪ {(id, PubKey)}
6:      end if

7:  upon event RoundStart() do
8:      R ← R + 1
9:      Membership[R] ← MembershipNextRound
10:     Degree[R] ← ChooseDegree()
11:     MembershipNextRound ← ∅
12:     Seeds[R] ← GenRandomSeed()
13:     LeavingNotices[R] ← ∅
14:     Views ← random_regular_graph(Membership[R], Degree, Seeds[R])
15:     PUBLISH(Membership[R], Seeds[R], Degree[R])

16: procedure BANNODE(node)
17:     BannedNodes ← BannedNodes ∪ {node}
18:     notice ← GenerateSignedNotice(node, CurrentTime())
19:     LeavingNotices[R] ← LeavingNotices[R] ∪ {notice}
20:     Views ← REPAIRVIEWGRAPH(notice)
21:     for each peer ∈ Views[node] do
22:         SEND(NODE_LEFT, peer, notice)
23:     end for
24: end procedure

25: upon event Receive(POM, node, proof) do
26:     if Valid(proof) then
27:         BANNODE(node)
28:     end if

29: upon event LeavingNode(node) do
30:     notice ← GenerateSignedNotice(node, CurrentTime())
31:     LeavingNotices[R] ← LeavingNotices[R] ∪ {notice}
32:     Views ← REPAIRVIEWGRAPH(notice)
33:     for each peer ∈ Views[node] do
34:         SEND(NODE_LEFT, peer, notice)
35:     end for
```

```
1:  upon event Receive(ACCUSATION, sender, accused_node) do
2:      made_accusations[R][sender] ← made_accusations[R][sender] + 1
3:      received_accusations[R][accused_node] ← received_accusations[R][accused_node] + 1
4:      sender_made ← 0
5:      accused_received ← 0
6:      for r ← (R − rounds_considered) to R do // Previous rounds that are considered
7:          sender_made ← sender_made + made_accusations[r][sender]
8:          accused_received ← accused_received + received_accusations[r][accused_node]
9:      end for
10:     total_connections ← Degree ∗ rounds_considered
11:     if sender_made > total_connections ∗ threshold_made_accusations then
12:         BANNODE(sender)
13:     end if
14:     if accused_received > total_connections ∗ threshold_received_accusations then
15:         BANNODE(accused)
16:     end if
```

LEAVING NODES Another situation that needs to be considered is the possibility of nodes leaving the system or failing in the middle of a round. Contrary to the entrance of nodes in the system, it is not feasible to prevent nodes from leaving in the middle of a round. When this happens, the overlay needs to be repaired, so that nodes that were in the neighborhood from the departed node can maintain their view size, and therefore not be negatively impacted by the departure of their neighbor. However, the repair mechanism must still assure that the repaired overlay is still verifiable by every node in the system. The mechanism works in the following way: When a node leaves the system, gracefully or by crashing, the central authority issues a departure notice, signed and with a timestamp (Algorithm 1, lines: 29 - 35). That notice is sent by the central coordinator to the departing node's immediate neighbors, in order to minimize the time that they have a smaller view size. That notice is then gossiped throughout the overlay by those nodes, so that every node can be aware of that departure. When a node receives a departure notice (Algorithm 2, lines: 29 - 37), they perform a repair operation to the overlay graph that consists in removing the node that left the system, as well as all its edges, and then pseudo-randomly creating links between the nodes that were left with a smaller view, therefore replenishing the degree of the affected nodes. The repair operation is not commutative, and this means that in order to assure the consistency of the views among all nodes, every node should apply the repair algorithm, provoked by different nodes leaving the system, in the same order. Therefore departure notices issued by the central coordinator carry a timestamp.

If multiple nodes leave the system or fail at the same time, then it is very likely that the order of the departure notices received by a node is not the correct one, and it is possible that it receives a notice with a timestamp that is smaller than the one they already applied to the graph. For this reason, nodes should keep the information of the latest changes to the overlay, including the departure notice, the edges that were removed from the overlay, and the edges that were added by the pseudo-random repair algorithm. With this information, nodes can rollback on the changes made to the graph and apply them in the correct order. Note that nodes only need to keep this information regarding the changes that are more recent, when, probabilistically, there is still the chance of an earlier departure notice arriving.

An optimization can also be made to this mechanism to lower the need for rolling back updates made to the overlay graph. If a node is not immediately affected by an update, meaning that node who left is not an immediate neighbor, they can restrain from updating the overlay graph until it is strictly necessary, for example in order to verify a connection between two nodes. This lazy strategy will lower the chances of nodes needing to perform rollbacks to the overlay, since they have more time to receive the updates.

Resiliency to Byzantine behavior

The biggest advantage of the semi-centralized approach is the resiliency to Sybil/eclipse attacks. Due to the deterministic generation of the overlay, the reach of a node's attack when it comes to membership is very limited, and the only way an attacker could subvert the security properties of the overlay is by controlling a big percentage of the overlay's nodes. For an attacker to partition a single node from the rest of the overlay, every peer in that node's view needs to be controlled by him, which is very unlikely, unless the attacker holds control over a very big portion of the overlay.

Algorithm 2 Round management

```
1:  upon event InitRound(R) do
2:      SeedOverlay, Members, Degree ← GetOverlayInfo(R)
3:      ViewGraph[R] ← random_regular_graph(Members, Degree, SeedOverlay)
4:      View ← ViewGraph[R][myself]
5:      LazyPushPeers ← ∅
6:      EagerPushPeers ← ∅
7:      missing ← ∅
8:      receivedMsgs ← ∅
9:      Digests ← ∅
10:     LeavingNotices[R] ← ∅
11:     received_pocs ← ∅
12:     sent_pocs ← ∅
13:     contribution_neighbors ← ∅
14:     contribution_myself ← ∅
15:     message_count[R] ← 0

16: upon event EndRound() do
17:     for each node ∈ contribution_neighbors do
18:         poc ← BuildPoc(node, contribution_neighbors[node])
19:         SignedPoc ← Sign(poc, PrivKey)
20:         sent_pocs[R] ← sent_pocs[R] ∪ SignedPoc
21:         SEND(POC, node, myself, SignedPoc)
22:     end for

23: upon event Receive(POC, sender, SignedPoc) do
24:     received_pocs[R] ← received_pocs[R] ∪ SignedPoc
25:     assert VERIFY(SignedPoc, PubKeySender)
26:     if Malicious(SignedPoc) then
27:         SEND(ACCUSATION, myself, sender)
28:     end if

29: upon event Receive(NODE_LEFT, notice) do
30:     assert VERIFY(notice, CoordinatorPubKey)
31:     if notice ∉ LeavingNotices[R] then
32:         LeavingNotices[R] ← LeavingNotices[R] ∪ notice
33:         View ← REPAIRVIEWGRAPH(notice)
34:         for each p ∈ View do
35:             SEND(NODE_LEFT, p, notice)
36:         end for
37:     end if
```

4.2 BROADCASTING ALGORITHM

The broadcasting algorithm used in Bycast (Algorithm 3) is inspired in Plumtree but with some modifications. Multiple trees are used for the broadcasting of information, to balance the distribution of work among all nodes in the overlay. In Plumtree a single tree is shared between all nodes, regardless of the number of sources of information in the overlay. As previously mentioned, this creates unfairness when it comes to the effort of each peer in the network, since leaf peers will not carry the burden of forwarding payload messages. Another shortcoming of the single-tree approach is that the interior nodes can be overburdened since the effort of propagation of the information in the overlay relies solely on them and is not evenly spread out among all nodes in the system. The utilization of multiple trees provides a more balanced effort distribution between the nodes of the system, which will be detrimental to the success of the free-rider detection mechanism employed. Multiple trees, one tree per source of information, will also mean that due to the way the Plumtree algorithm constructs the trees, each one of them will be optimized when it comes to latency, and the path between the source and each node is the fastest route between both nodes in the overlay. This will improve the latency of the message received by the nodes, which could be an important factor in latency-sensitive systems such as streaming peer-to-peer networks.

These improvements come with some costs. The first one is the necessity of each peer in the system having a different set of *Eager Push Peers* and *Lazy Push Peers* for each tree being used. This means that the memory utilization, contrary to the Plumtree algorithm, grows linearly as the size of the number of sources grows. However, the only information that is present in the set of *Eager Push Peers* and *Lazy Push Peers* is the id of the node, which means that the overhead should remain small for even considerably big overlays, especially considering the large amount of memory available even in small devices. The second disadvantage of this approach is the overhead of the tree construction and repair process. While in Plumtree only one tree has to be constructed, in Bycast this number is higher. In order to mitigate this overhead, the Plumtree algorithm was modified to include a new kind of message *BUILD*, that will be broadcasted by the nodes in the beginning of the round and will be used to construct the broadcasting trees. These *BUILD* messages do not carry a payload and therefore will result in the reduction of the overall bandwidth overhead provoked by the construction of the trees. This means that even though the number of messages will grow substantially, the bandwidth used by the algorithm will only grow a small amount. However, while in Plumtree when a node crashes or leaves it is only necessary to fix one broadcasting tree, in this case it will be necessary to fix all the broadcasting trees. This situation should not be problematic either, as the fixing algorithm relies on *GRAFT* and *PRUNE* messages that are very light due to not carrying a payload.

Algorithm 3 Dissemination algorithm

```
1:  procedure EAGERPUSH(m, mID, hop, sender, treeID)
2:      for each p ∈ EagerPushPeers[treeID] : p ≠ sender, p ≠ treeID do
3:          SEND(GOSSIP, p, m, mID, hop, myself, treeID)
4:          contribution_myself[sender] ← contribution_myself[sender] + 1
5:      end for
6:  end procedure

7:  procedure LAZYPUSH(m, mID, hop, sender, treeID)
8:      for each p ∈ LazyPushPeers[treeID] : p ≠ sender, p ≠ treeID do
9:          SEND(IHAVE, p, mID, hop, myself, treeID)
10:     end for
11: end procedure

12: upon event Broadcast(m_0, m_1, ..., m_n) do
13:     D ← ∅
14:     for k ← 0 to n do
15:         mID_k ← RandomID()
16:         mHASH_k ← hash(m_k)
17:         D ← D ∪ (mID_k, mHASH_k)
18:     end for
19:     SignedDigest ← Sign(D, PrivKey)
20:     for each p ∈ View do
21:         SEND(DIGEST, p, SignedDigest, myself)
22:     end for
23:     for k ← 0 to n do
24:         EAGERPUSH(m_n, mID_n, 0, myself, myself)
25:         LAZYPUSH(m_n, mID_n, 0, myself, myself)
26:         Deliver(m)
27:         receivedMsgs ← receivedMsgs ∪ {mID_n}
28:     end for

29: upon event Receive(DIGEST, SignedDigest, sender, t) do
30:     if Verify(SignedDigest, PubKey_t) then
31:         for each d in SignedDigest do
32:             Digests ← Digests ∪ d
33:         end for
34:     end if
```

```
1: upon event Receive(GOSSIP, m, mID, hop, sender, t) do
2:     assert sender ∈ View
3:     assert VERIFY(Digests, mID, m)
4:     if t ∉ EagerPushPeers then
5:         EagerPushPeers[t] ← View
6:         LazyPushPeers[t] ← ∅
7:         contribution_neighbors[sender] ← 0
8:     end if
9:     if mID ∉ receivedMsgs then
10:        Deliver(m)
11:        contribution_neighbors[sender] ← contribution_neighbor[sender] + 1
12:        message_count[R] ← message_count[R] + 1
13:        receivedMsgs ← receivedMsgs ∪ {mID}
14:        if ∃ (id, node, r, t) ∈ missing : id = mID then
15:            cancel Timer(mID)
16:            EAGERPUSH(m, mID, hop + 1, myself, t)
17:            LAZYPUSH(m, mID, hop + 1, myself, t)
18:            EagerPushPeers[t] ← EagerPushPeers[t] ∪ {sender}
19:            LazyPushPeers[t] ← LazyPushPeers[t]\{sender}
20:        else
21:            EagerPushPeers[t] ← EagerPushPeers[t]\{sender}
22:            LazyPushPeers[t] ← LazyPushPeers[t] ∪ {sender}
23:            SEND(PRUNE, sender, myself, t)
24:        end if
25:    end if

26: upon event Receive(PRUNE, sender, sender, t) do
27:    EagerPushPeers[t] ← EagerPushPeers[t]\{sender}
28:    LazyPushPeers[t] ← LazyPushPeers[t] ∪ {sender}

29: upon event Receive(IHAVE, mID, hop, sender, t) do
30:    if mID ∉ receivedMsgs then
31:        if ∄ Timer(id) : id = mID then
32:            setup Timer(mID, timeout_1)
33:        end if
34:        missing ← missing ∪ {(mID, sender, hop, t)}
35:    end if
```

1: **upon event** $Timer(mID)$ **do**
2: setup $Timer(mID, timeout_2)$
3: $(mID, node, hop, t) \leftarrow$ removeFirstAnnouncement($missing, mID$)
4: $EagerPushPeers[t] \leftarrow EagerPushPeers[t] \cup \{sender\}$
5: $LazyPushPeers[t] \leftarrow LazyPushPeers[t] \setminus \{sender\}$
6: SEND($GRAFT, node, mID, hop$, myself, t)

7: **upon event** Receive($GRAFT, mID, hop, sender, t$) **do**
8: $EagerPushPeers[t] \leftarrow EagerPushPeers[t] \cup \{sender\}$
9: $LazyPushPeers[t] \leftarrow LazyPushPeers[t] \setminus \{sender\}$
10: **if** $mID \in receivedMsgs$ **then**
11: SEND($GOSSIP, sender, m, mID, hop$, myself, t)
12: **end if**

4.2.1 *Content-poisoning prevention*

Plumtree is vulnerable to content-poisoning attacks. A malicious node could forward a modified payload and the receiving node would have no way of distinguishing it from a genuine payload. A mechanism should be employed so that every node is able to minimize the probability of delivering a faulty update. The mechanism that is employed by Bycast to mitigate this risk is the list signing mechanism that was presented in the previous chapter. The node broadcasting information computes the hash of a sequence of updates and stores them in a digest packet which will be signed (Algorithm 3, line 19). This information is then broadcasted to all peers, and they will verify it using the source's public key (Algorithm 3, line 3), indirectly providing authentication for the messages which were considered in the digest. While in traditional gossip algorithms, there would be a high risk that a node could receive a message before receiving the digest that will authenticate it, meaning that the node would only be able to deliver it when it finally received that digest, in Bycast that situation will be far less frequent since the same broadcasting tree is used by all the messages from the same source, meaning that the path they will follow will be the same in normal circumstances. The costs in terms of overhead are one signature generation by the number of packets contained in the digest at the source node and one signature verification per number of packets at the receiving node. This method also forces the source to buffer those packets at its side before it is able to create the digest that will authenticate them, but if messages are delivered in order through the broadcasting tree, it will cause no need to buffer messages at the destination.

4.2.2 *Man-in-the-middle attacks prevention*

If man-in–the-middle attacks are not prevented in Bycast, then the strong security properties that are provided by membership service are compromised. This happens because malicious nodes might be able to target their victims if they are able to spoof their neighbor's identities. Therefore some form of authentication is necessary between peers, to ensure that they are not being attacked by a malicious node.

The first and more obvious approach that could be taken in order to provide this necessary security property is to have every node sign every message it sends with its private key. Considering that the private key is only known by its corresponding node, this would allow for every destination node to verify the origin of the message, making it infeasible for an attacker to spoof another node's identity. The disadvantage of this approach is the immense overhead that would be provoked by having to sign and verify each message that is communicated by two peers, rendering it infeasible to be used.

The approach used by Bycast, and that minimizes the overhead caused to the system resembles the approach used by the TLS protocol Dierks and Rescorla (2008), consisting of an handshake that is used to authenticate both parties and to derive a symmetric key, that will be used for the encryption of all application data exchanged between the two nodes for the remainder of their connection. This allows each node to have confidence it communicating with the right entity and not an attacker that is impersonating that entity. Using this approach, however, means that non-repudiation will not be guaranteed for *GOSSIP* messages. This means that even though nodes would be able to detect content-poisoning attacks, for example, they would not be able to prove

that the attacker has performed them, since both nodes have access to the key that signs and verifies the messages exchanged between them.

4.3 FREERIDER DETECTION

In order to make sure that all the nodes in the overlay contribute with the propagation of messages, Bycast employs an auditing mechanism that will detect and punish nodes that do not contribute to the system (Algorithm 4).

At the end of each round, each node creates a POC (proof of contribution) (Algorithm 2, lines: 18, 21), which identifies itself, the number of messages that the neighbor sent him, and the id of the neighbor. This POC is signed by the node to ensure the authenticity and non-repudiation of the POC. The node repeats this process for every neighbor, and also receives its corresponding POCs from his own neighbors (Algorithm 2, lines: 23 - 28). The POCs received from neighbors during the last x rounds will be used by the node to prove that it has been contributing to the system when a node requests this evidence (Algorithm 4, lines: 3 - 4). This proof is necessary due to the fact that a node working less in a determined round does not necessarily mean that it is a free-rider. Even with the single tree per broadcaster mechanism, there is still a possibility that a correct node works significantly less than what would be the average in a round, therefore this mechanism is used to take into account not only the work that a node has done in the current round but all the work that the node has done in the last x rounds.

The way that nodes verify if a neighbor is a free-rider or not is by taking into account the POCs issued to him during the last x rounds and analyzing that information (Algorithm 4, lines: 5 - 23. If the overall average of messages sent per round is less than a configurable threshold (global to the system), then that is a proof of misbehavior and will be used to prove that the node is a free-rider. The more rounds of the past are considered, the more balanced this average will be to all altruistic nodes, therefore lowering the percentage of false-positives (altruistic nodes being identified as free-riders). However, taking into account a higher number of rounds will result in a higher overhead caused by the mechanism in terms of memory and bandwidth utilization, as well as computations necessary (to verify the signatures for each POC). Therefore, the number of rounds that are considered in the mechanism is a trade-off between performance and accuracy of the mechanism. Since this proof will only be necessary when a node is suspicious of its neighbor, for example, due to the fact that it is sending a lot more messages to it than it is receiving, this means that if the nodes behave in an altruistic way, the free-rider mechanism will not activate and the overhead caused by it will be only the issuance of the POCs at the end of the round.

The criteria to ask for suspicion from a neighbor is an important aspect to the system to tune, since it represents another trade-off between overhead and accuracy in detecting free-riders. If this policy is very relaxed, a lot of free-riders will not be detected, since their neighbors will not ask for their history as often. However, if the threshold is very high, the accuracy of the detection system will increase, but the amount of times that nodes ask for proof of an altruistic node will also increase. This does not mean the increase of the false-positive rate since these verifications will result in the exoneration of the altruistic nodes.

Algorithm 4 Freerider detection

1: **upon event** Suspicion(*neighbor*) **do**
2: SEND(*REQUEST_HISTORY*, *neighbor*)

3: **upon event** Receive(*REQUEST_HISTORY*, *sender* **do**
4: SEND(*RESPONSE_HISTORY*, *sender*, myself, *received_pocs*, *sent_pocs*)

5: **upon event** Receive(*RESPONSE_HISTORY*, *sender*, *received_pocs*, *sent_pocs*) **do**
6: **for each** $poc \in received\_pocs \cup sent\_pocs$ **do**
7: VERIFYPOC(*poc*) // Verifies signature
8: VERIFYVIEW(*poc*, $ViewGraph[poc.round]$, $LeavingNotices[poc.round]$) // Verifies if nodes were neighbors in that round
9: **end for**
10: $total\_message\_count \leftarrow 0$
11: $total\_contribution \leftarrow 0$
12: **for** $r \leftarrow (R - rounds\_considered)$ to R **do** // Previous rounds that are considered
13: **if** Sum($sent\_pocs[r]$) $\neq message\_count[r]$ **then** // number of messages exchanged in that round
14: SEND(*POM*, *coordinator*, *sender*, $sent\_pocs[r]$)
15: **end if**
16: **for each** $poc \in received\_pocs[r]$ **do**
17: $total\_contribution \leftarrow total\_contribution + poc.value$
18: **end for**
19: $total\_message\_count \leftarrow total\_message\_count + message\_count[r]$
20: **end for**
21: **if** $total\_contributions < total\_message\_count * effort\_threshold$ **then**
22: SEND(*POM*, *coordinator*, *sender*, *received_pocs*)
23: **end if**

4.3.1 *Attacks to the free-rider detection mechanism*

The free-riding detection mechanism is not free from attacks from malicious nodes and therefore it is necessary that the free-riding detection mechanism is itself resilient against misbehavior from nodes in the system. One way that malicious nodes can potentially attack the free-riding detection system is by issuing false POCs to their neighbors, with either higher or lower values than the real contribution from the neighbor. A malicious node can do this with the objective of interfering with other nodes, in order to harm their experience, and lead them to be classified as free-riders. These attacks can be done in a directed or undirected way, targeting one specific node or random nodes in general. If a node wishes to attack another specific node in the system, their impact will be mitigated by the inherent resiliency of the membership service. Due to the deterministic and verifiable way views are generated every round, a malicious node cannot force another peer to be in its view unless it naturally happens by chance. Also, because of the fact the observed contribution of a neighbor is the average of the last x rounds, the impact of a malicious POC sent by the malicious node to the neighbor will be reduced.

In order to deal with malicious nodes issuing illegitimate POCs, an accusation mechanism was put into place. When a node receives a POC with the incorrect value, it issues an accusation to that node, and sends it to the centralized coordinator (Algorithm 2, line: 27). If the percentage of accusations that a node receives over a period or rounds is higher than a determined threshold, he will be evicted from the system (Algorithm 1, line: 15). This threshold is not constant and will change from system to system. Registering received accusations by itself would open this mechanism up to malicious attacks. A malicious node could simply issue an accusation for every neighbor on every round, and this would lead to a significant increase in the eviction of correct nodes from the system. For this reason the central coordinator also registers the issuer of the accusation, and maintains the number of accusations that has been performed by them during the last rounds considered. If the number of accusations performed by a node goes over a certain threshold, the node will also be evicted from the system (Algorithm 1, line: 12).

Bycast employs a clever verification of a node's issued POCs to partially mitigate the effect of attacks to the free-rider detection mechanism. This verification is based on the observation that the overall number of received payload messages should be the same, or close, for every node. This means that the sum of the values of the issued POCs by a node after a round should be equal to the number of messages that were broadcasted during that round. This property will allow for Bycast to deal with rational collusion between nodes. Imagine that two colluding rational nodes happen to be in each other's views during a certain round. One attack that would allow them to save resources during that round would be issuing a POC with a high value to each other, without actually having sent the corresponding number of messages to the other. They would be capable of not contributing to any other neighbors without being detected by the free-riding detection mechanism because the POC they received from each other would indicate that the two nodes indeed contributed with a high number of messages during that round. By doing this verification, we are removing the incentive for rational nodes to collude amongst themselves. If rational node A sends an illegitimate POC to node B, for example saying that node B has contributed with *n* messages during that round, then node A will not have enough POC budget to send the right POCs to the neighbors that actually sent messages to node A. If node sends the right value in the

POC to these nodes then this verification will fail since the total sum of the issued POCs during that round will go over the correct number and the node will be evicted from the system. If instead node A tries to lie in the POC to the neighbors that actually sent messages, then those nodes will issue accusations, and that node will also be evicted from the system.

The only way for a node to be able to issue a spoofed POC to another rational node without being detected is by receiving less messages from correct nodes, in order to issue to them the right value in their POCs and use the remaining budget to the POC issued to the other rational node. However, this behavior goes against the supposition of a rational node's intention, that is achieving the best quality of service possible while saving as much as possible on resource utilization. Therefore we consider that Bycast actively discourages nodes from colluding with each other in order to save resources on the system.

4.4 DISCUSSION

While traditional peer-sampling services rely on partial views to allow for a better scalability of the protocol, there are a number of benefits that come from using full memberships when it comes to security. It is not clear for example how to provide the verifiability property of the views with a distributed algorithm without a full membership service, specially for broadcast algorithms that rely on *Eager Push* strategies such as the Plumtree algorithm. While in *Lazy Pull* strategies each node has more control over where it gets messages from, in *Eager Push* based strategies nodes do not choose who they get the updates from, making it harder to combat Byzantine behavior. The utilization of full memberships does however come at a cost that must be considered. The biggest disadvantage is the high utilization of memory. Memory requirements per member will grow linearly in the number of members. However, this is not necessarily a problem, specially for medium sized stable networks, of up to a few thousand nodes.

The utilization of a central coordinator does constitute a single point of failure, and could potentially make it more vulnerable to denial-of-service attacks than completely decentralized alternatives, and additional measures should be employed to protect the coordinator against targeted attacks, but we consider that protecting this central authority is out of the scope of this work.

Even though Bycast uses a centralized coordinator when it comes to the maintenance of the views and free-rider detection mechanism, the effort was made to structure its responsibilities in a way that would not undermine the scalability of the protocol. The central coordinator is responsible for two major tasks. The first one is the maintenance of the membership information for each round. This involves the issuance of certificates for the node that wishes to join the network on the next round, as well as the issuance of departure notices for the nodes that leave the system. Before a broadcasting round starts, the central coordinator will close the membership for that round, generate a seed that will be used to generate the overlay and publish this information. The duration of the rounds constitutes a trade-off between the attractiveness of the service and the overhead caused by the reshuffling of the overlay. If the rounds have a very big duration, nodes that are waiting to join the system in the next round might lose their interest, and the system will lose its appeal, but the reshuffling of the overlay will happen less quickly, meaning that its computation efforts will be amortized by through the bigger round. If in turn

the broadcasting rounds are too short, the likelihood of joining nodes losing interest is lower, but the reshuffling of the overlay will happen more often, meaning a higher computation effort.

The second major task of the coordinator is the collection of accusations generated by the nodes during the broadcasting rounds. This task is very lightweight, since the only verification that the central authority needs to do is to check if the corresponding nodes are above the accepted threshold when it comes to the made and received accusations, and issue and eviction notice if a node is breaking it.

5

EVALUATION

The evaluation process of Bycast has two major goals. The first is to test the resilience of the algorithm against Byzantine and rational behavior. A variety of tests were created to measure to what extent Byzantine nodes can compromise the experience of correct nodes and get them evicted from the system, as well as to test the efficiency of the employed security mechanisms when it comes to the detection of selfish nodes who do not contribute with their resources. The second major goal of this evaluation process is to evaluate the performance of Bycast when compared with the original Plumtree algorithm, taking into account the latency of message delivery and the total number of messages exchanged.

5.1 TESTBED

A simulator was implemented using the Python programming language, using an round-based strategy, to evaluate Bycast. The simulator works as follows: Every node in the overlay has an inbox and an outbox, and every iteration, the simulator processes every message in a node's inbox, and deposits every message that it wishes to send in its outbox. At the end of the round, those messages are delivered to the corresponding destinations. The simulator allows for the configuration of a number of general parameters that will be used to model the system. These parameters include:

- N: The total number of nodes that will be part of the broadcasting.
- BYZANTINE_N: The total number of nodes that will follow Byzantine behavior.
- RATIONAL_N: The total number of nodes that will follow rational behavior.
- D: The size of the views that each node should have.
- ITER: The number of iterations that the simulation should have, being that each iteration is defined as a simulation of one hop for every message traveling in the overlay.
- ROUND_SIZE: The size of each round in terms of iterations. Meaning the period in iterations before the overlay is reshuffled and the views of the nodes are recomputed.
- BROADCASTING_PLAN: The description of the iterations that each node will broadcast a message.

A total of three different types of nodes were modeled: Byzantine, rational and altruistic nodes. Byzantine nodes wish to disrupt the experience of one or more nodes, and are able to do this in two different ways: By lying in the POC (proof of contribution) that is delivered to its neighbors at the end of a round, or by generating false accusations directed at neighbors. Byzantine behavior is modeled in the simulator using two different parameters. The first is the *byzantine_coefficient*, which varies between 0 and 1, and specifies the ratio of neighbors that malicious nodes will attack, by generating false POCs. The second relevant simulation parameter is the *false_accusation_rate* which is also a value between 0 and 1, and specifies the percentage of times that a node will send an accusation to a neighbor, with the objective of getting them evicted. The decision on which neighbors Byzantine nodes will attack is performed randomly or in a pre-defined way depending on a configuration parameter, as to enable the testing of a large-scale sybil or collusion attack directed at a single node. The security mechanism that is used to detect malicious nodes also has two parameters that are considered in the simulation. These parameters are the *threshold_made_accusations* which are the max ratio of accusations (ratio between number of made accusations over the number of neighbors that the peer interacted with over the last x rounds) that a node can perform before being labeled as a malicious node and expelled from the system, and *threshold_received_accusations* which is the maximum ratio of accusations that a node can receive before being labeled as a malicious node and also expelled from the system.

Rational behavior is simulated by using the parameter *rational_coefficient*, and specifies the effort that each node will perform when it comes to the forwarding of messages. For example, if the *rational_coefficient* is set as 0.5, then selfish nodes will take part in the forwarding of messages for exactly half of the broadcasting trees, and will refuse to collaborate for the other half of the broadcasting trees. The way the node chooses in which broadcasting trees it will participate in the forwarding of messages is uniformly at random. The parameters that are used to configure the free-riding detection mechanism are the following: *suspicion_threshold* specifies the minimum contribution of a node before it triggers a suspicion. If a node contributes to a neighbor less than this parameter during a round, he will request the history of the node so that it can verify that the node is not free-riding. The second parameter is the *freerider_threshold*, which specifies the minimum effort (in terms contribution over what would be expected) that a node can do before being flagged as a free-rider and expelled from the system. Finally, the parameter *rounds_freerider_detector* specifies the number of previous rounds that will be used in order to verify the effort of a node.

The random regular graphs that will be used as the overlay for every round are generated with the *networkx* Hagberg et al. (2008) library. It allows for the generation of random regular graphs by specifying the degree of each node and the total number of nodes in the graph. It further allows for the utilization of a provided seed which is necessary in order to create the graph in a deterministic and verifiable way.

A network of 100 nodes with a degree of 5 was used for all the tests, except when otherwise specified. The choice of a smaller overlay size was made to allow performing the testing scenarios with higher range of configurations, such as the percentage of malicious/rational nodes, as well as to repeat the tests a higher number of times, in order to have more confidence in the results. Each result represents the average of 10 simulation runs. For the Byzantine resiliency tests, a false accusation rate of 0.5 was used, the threshold for made accusations

was set at 0.4 and the threshold for received accusations was set at 0.5. The threshold for the minimum effort of a node was set at 0.5, meaning nodes have to contribute with at least 50% of the messages they receive.

5.2 RESULTS

5.2.1 *Resiliency to Byzantine behavior*

We start by studying the resiliency of the Bycast protocol in the presence of malicious attacks.

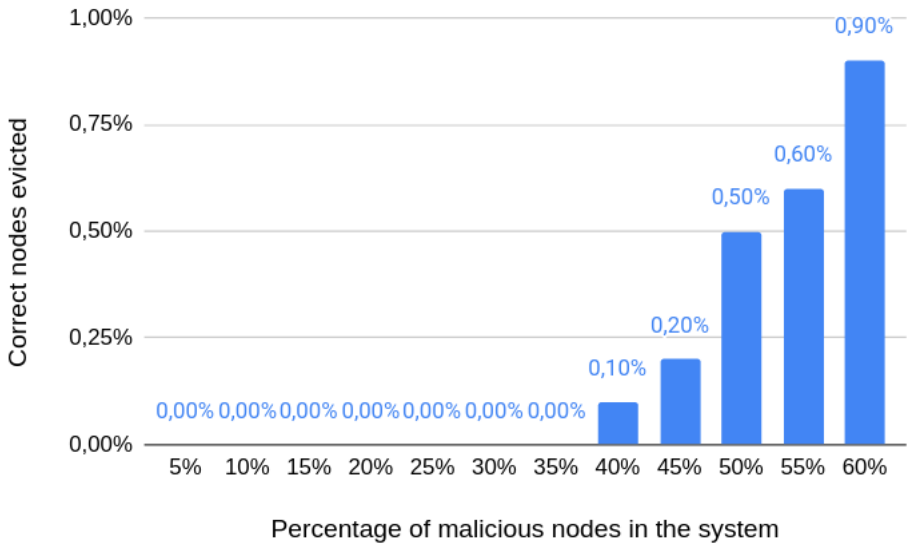

Figure 15: Percentage of correct nodes evicted from the system as the percentage of malicious nodes in the overlay grows - Illegitimate POCs.

Figure 15 depicts the percentage of correct nodes that were wrongfully evicted from the system as the result of illegimitate POCs that were issued by Byzantine nodes. In this case, each malicious node picks, every round, two random neighbors. It will send a POC with the value of 0 to one of them, and the sum of the expected values for both to the other. This strategy will allow for the malicious node to still have the expected overall sum of the issued POCs, meaning this situation will not result in the immediate eviction of the malicious node. Also, since the threshold for received accusations was set at 0.5, and the node will only receive accusations from these two neighbors out of 5 total, their rate of received accusations is below the threshold, meaning that this strategy is able to avoid detection and eviction of the malicious node. Even as the percentage of Byzantine nodes in the overlay grows, it is possible to see that the reach of the malicious behavior is limited, with less than 1% false-positive rate with up to 60% of malicious nodes in the system. The reason for the low efficiency of the attack is firstly justified by the utilization of multiple rounds in the free-rider detector mechanism. This means that even if a node happens to be neighbors with multiple malicious nodes in a round, and receive a POC with the value 0 from each of them, this situation will be amortized by the other rounds that are also considered, minimizing the impact from the current one. Another important factor that needs to be taken into consideration is that in this

scenario malicious nodes are not colluding among themselves, and therefore the probability of a correct node being targeted by many malicious nodes in the same round is decreased, contributing to the low rate of eviction of correct nodes, even when the percentage of malicious nodes in the system is high.

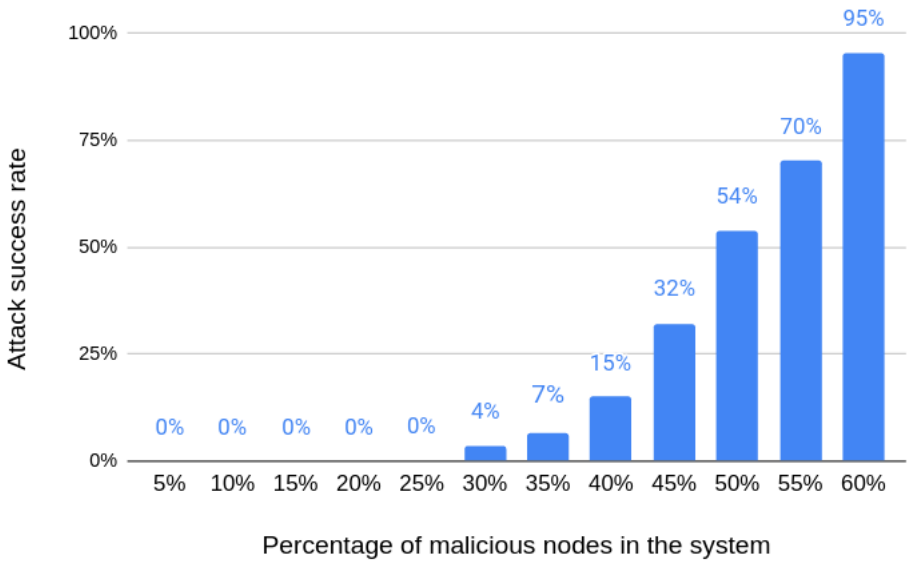

Figure 16: Percentage of correct nodes evicted from the system as the percentage of colluding malicious nodes in the overlay grows - Illegitimate POCs.

The next scenario that was tested, illustrated in Figure 16, was the possibility of an attacker controlling a number of different peers, and having the possibility to join their efforts in order to cause harm in the overlay. In this case, the attacker targets a single node, by sending it false POCs, with the value of 0, in order to make that victim node be considered a free-rider and therefore be evicted from the overlay. When the attacker held less of 25% of the overlay under their influence, they were unable to successfully evict a correct node. This is due to the fact that the nodes under his control were not in the view of the victim node enough times throughout the rounds in order to be able to send the false POCs and bring the average reported effort of the node below the free-rider threshold of 0.5.

As the attacker influences a higher portion of the overlay, however, the ratio of times that the attacker was successful also grew, due to the fact that the number of times that one of the malicious nodes was in the view of the attacker increased. When the attacker controlled 40% of the overlay, they were able to make a targeted correct node be evicted 15% of the times, and this probability grew to 95% of the times, as the attacker held 60% of the nodes under his influence.

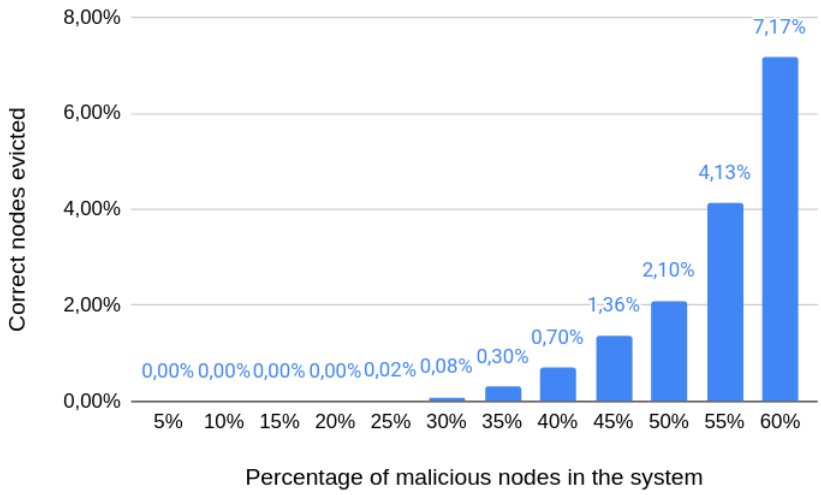

Figure 17: Percentage of correct nodes evicted from the system as the percentage of malicious nodes in the overlay grows - Illegitimate accusations.

When we consider the second scenario of Byzantine behavior, the generation of false accusations by malicious nodes, we can observe in Figure 17 that, just like it happened in the first attack, the eviction of correct nodes is negligible even when 40% of the overlay nodes are performing the attack. The reason for the low impact of this type of behavior is in part explained by the randomness of the overlay. Due to the utilization of random regular graphs, the effects of this malicious behavior will be amortized by all nodes equally, therefore reducing the effect on each one individually. Also, since the centralized authority also registers the node which made the accusation, if malicious nodes excessively try to harm other nodes, by increasing the amount of accusations they send to their neighbors, that will result in their eviction from the system. If in turn the malicious nodes reduce their false accusation rate, the negative impact on the system will be even smaller.

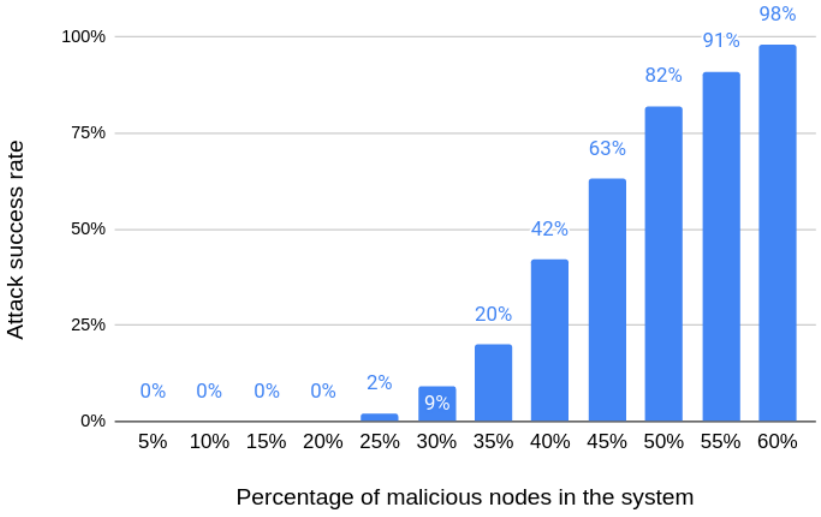

Figure 18: Percentage of correct nodes evicted from the system as the percentage of colluding malicious nodes in the overlay grows - Illegitimate accusations.

When the malicious nodes collude with each other to find a specific victim, however, the efficiency of the attack, just like it happened in the first scenario, increases. We can observe in Figure 18 that when 25% of the nodes in the overlay were malicious and colluded with each other, they were successful in their attack to the victim node only 2% of the time. In turn, when 30% of the nodes were malicious colluders, that percentage rose to 8,5%. The probability of the success grows fast as the percentage of malicious colluders increases, and this is due to the fact that the chance of the victim node having malicious nodes in its view grows, which will mean it will receive more accusations.

The rise of efficiency when nodes collude among themselves is because the false accusations are more spread out through correct nodes when the malicious nodes are not colluding, since even if two malicious peers are in the neighborhood of a correct victim, they might not both choose to attack it, however if the malicious nodes are colluding, every time a victim has a malicious node in its view, it will receive an accusation from that node.

5.2.2 *Resiliency against rational nodes*

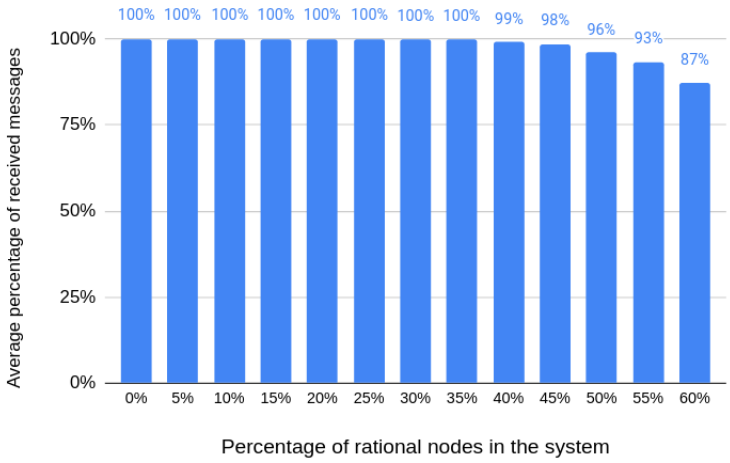

Figure 19: Resiliency of Bycast (average of the percentage of total messages received) as the percentage of free-riders in the system grows.

Figure 19 shows the reliability of the algorithm, as the average percentage of the total messages that each node received, for Bycast. For up to 35% of the entire overlay acting in a rational way, the reliability is close to 100%. The reason why this happens is due to the fact that for the reliability to decrease it is necessary that the graph that includes the nodes that are willing to forward messages for a certain broadcasting tree is not connected, meaning that there will be at least one node that is surrounded in its view by neighbors who refuse to forward him messages for that broadcasting tree. For low enough percentages of rational nodes in the system this situation is extremely unlikely. As the percentage of rational nodes increases, however, the probability of this situation happening increases rapidly, meaning that the overall average reliability will also decrease substantially.

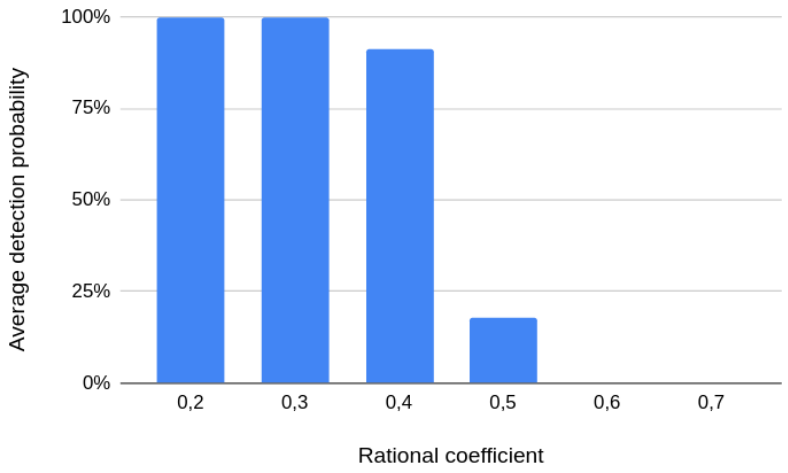

Figure 20: Observed probability of detection of a free-rider as the threshold_coefficient varies.

In order to try to avoid suspicion, rational nodes can adjust their effort, by for example selecting a percentage of the existing broadcasting trees which they will participate in the forwarding of information. The more a rational node tries to reduce the forwarding of information, the easier it will be for them to be detected. In our simulations, as shown in Figure 21, a node that made 20% of the effort of what would be expected from a correct node was detected 100% of the time, this probability lowered to around 91% when the node worked 40% from what was expected, and 20% when he contributed with 50% of what was expected. In our simulations, any node that made an effort of 60% or more of what would be expected would be able to be undetected. One argument that can be made is that even a probability of detection of 20% could be enough to deter a rational node from behaving in a selfish way, due to the fear of being evicted, so it is reasonable to assume that the free-rider detection mechanism is successful, for this configuration parameters, at convincing rational nodes to contribute with at least 50% of what would be expected from a correct node.

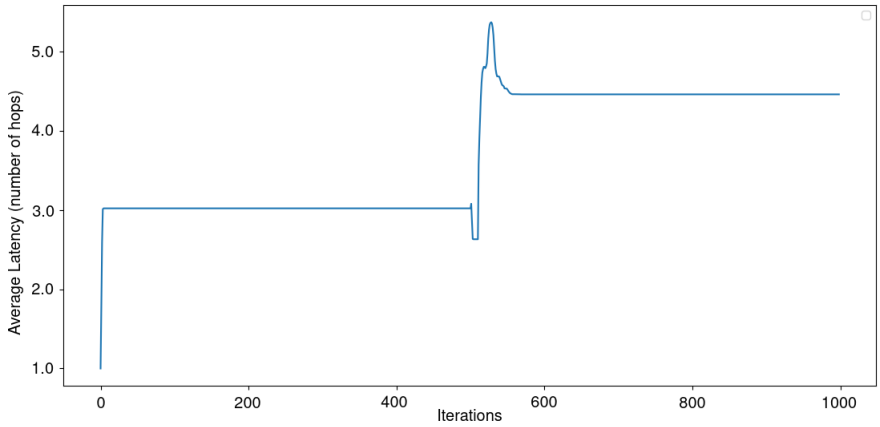

Figure 21: Variation in average latency during a broadcasting round, when 50% of the nodes stop forwarding messages at the middle of the round.

Figure 21 shows the variation in latency during a broadcasting round (in hops from the source). At the middle of the round, half the nodes, selected at random, start free-riding and therefore stop forwarding messages. It is possible to see that even in a scenario where a huge restructuring of the broadcasting trees is caused, when half of the peers stop forwarding messages, the latency has a spike of about 4 hops, during a few rounds, and soon stabilizes again, this time to a slightly higher value of about 3.5 hops. The reason that the average latency is slightly higher than before the nodes start free-riding, is because the fact that half the nodes stop forwarding messages will cause the broadcasting trees to decrease in degree and increase in depth.

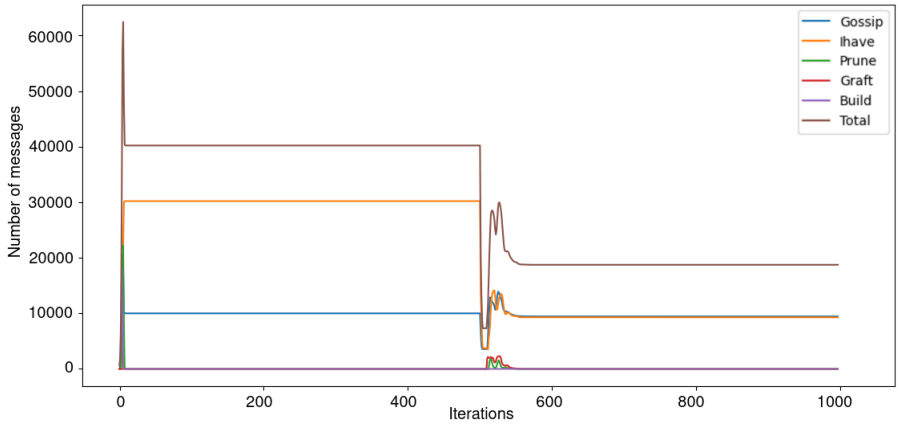

Figure 22: Variation in the number of messages exchanged during a broadcasting round, when 50% of the nodes stop forwarding messages at the middle of the round.

Figure 22 shows the same scenario as above, but this time measuring the number of messages exchanged during the round. At the beginning of the round there is a spike in the number of messages exchanged, and this is due to the process of the construction of the broadcasting trees. As the tree construction process finishes, the total number of messages stabilizes. During this time, the vast majority of messages exchanged are *IHAVE* messages, since the messages which carry the payload (*GOSSIP*) messages are only flowing through the broadcasting trees. Due to the stability in the network, during this time there are not any *PRUNE* or *GRAFT* messages being exchanged. When half the nodes suddenly stop forwarding messages and therefore cease participating in the broadcasting, the tree repair portion of the broadcasting algorithm is triggered, which justified the increase in both *GRAFT* and *PRUNE* messages. After the tree repair process, it is possible to observe that the number of *GOSSIP* messages remained the same as before the disruption however the number of *IHAVE* messages decreased. This is due to the fact that the trees will be modified in a way that will still reach every node (unless the node is partitioned from the overlay). The decrease of *IHAVE* messages is due to the lower volume of links that are being used to forward messages.

5.2.3 *Performance of Bycast*

Nodes	Degree	Plumtree average latency	Bycast average latency
100	5	8.00	3.02
1000	10	14.33	3.28
5000	15	20.03	3.48

Table 2: Latency as hops from the source comparison between Bycast and Plumtree.

One of the advantages of the multi-tree approach is that every broadcasting tree is optimized for its source, and that will provide vast improvements in latency over one shared tree among all broadcasters. As shown in Table 2, in an overlay of 100 nodes, with the degree of 5, the average latency for a receival of a message was of 8 hops in Plumtree, however it was only of 3,02 in Bycast. As the overlay and degree grows, the observed benefits also increase. With an overlay of 1000 nodes with a degree of 10, the Plumtree average latency was of 14 hops, while the Bycast latency was of only 3,28, almost 4 times smaller. For an overlay of 5000 nodes, with a degree of 15, the Plumtree latency was of 20 hops while the Bycast latency was of 3,48 hops. Depending on the context that the broadcasting algorithm is used, for example in video streaming, this gain in latency could translate into a much better experience for the end user.

Message type	Plumtree	Bycast
Prune	302	30200
Build	-	30200
Gossip	2000302	2000000
Ihave	6039698	6009800
Total	8040302	8070200

Table 3: Total number of messages exchanged in a broadcasting round.

When it comes to the volume of messages that are exchanged in the overlay, Bycast presents a slight overhead since it utilizes different broadcasting trees for each broadcaster, as depicted in Table 3. When a round starts, in Plumtree, only one tree needs to be built, by using the Prune messages to move the links that will not be part of the tree into *Lazy Push* mode. As Bycast uses one tree per broadcaster, this construction penalty will happen for each one of the trees, justifying the increase in *PRUNE* messages on the Bycast side. However, while Plumtree uses Gossip messages that will trigger the Prune messages and construction of the tree, in a flooding strategy, in Bycast the first message that is broadcasted by each node is a “Build” message that does not contain a payload. Therefore the total cost in terms of message count for the construction of the tree in Plumtree was 302 Gossip messages and 302 Prune messages, in Bycast the cost was of 30200 Prune messages and 30200 Build messages. While the difference is considerable, the fact that Bycast used messages without payload in the tree construction process will mean that the bandwidth utilization of Bycast will be efficient. Apart from the

tree construction process, the number of messages for Plumtree and Bycast are identical, which means that the difference in terms of message count between Plumtree and Bycast will be smaller in percentage if the rounds are bigger. In this case, considering a round in which each node broadcasted 200 messages, the total overhead in terms of message count for Bycast was of only 0,3% when compared with Plumtree.

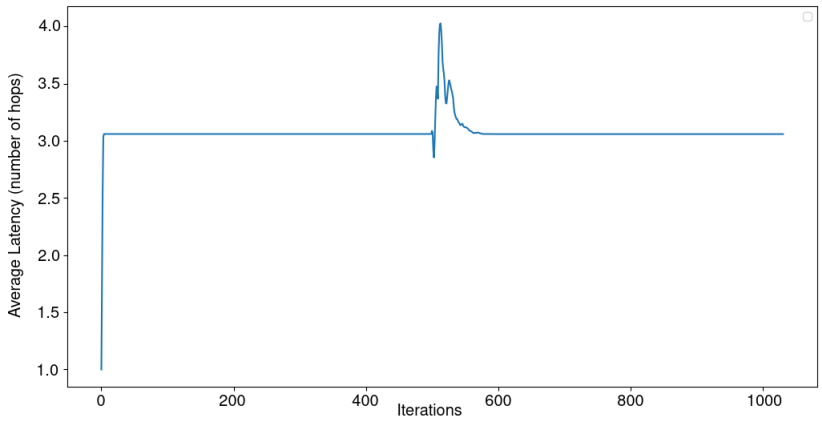

Figure 23: Variation in average latency during a broadcasting round, when 50% of the nodes leave the system at the same time.

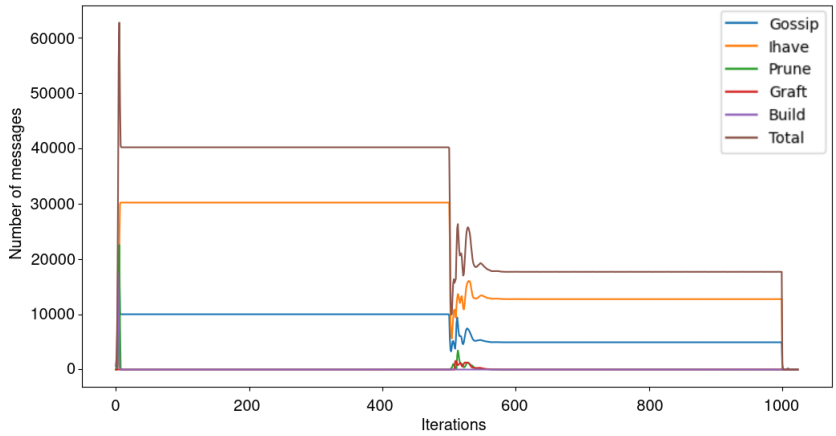

Figure 24: Variation in the number of messages exchanged during a broadcasting round, when 50% of the nodes leave the system at the same time.

Figure 23 shows the number of messages exchanged throughout a round in a scenario where 50% of the overlay nodes leave the system in the middle of a broadcasting round. It is possible to see a spike in the latency, from around 3.1 hops to 4.0 hops. Due to the repair algorithm for the overlay, and also the repair algorithm for broadcasting trees, this increase in latency is temporary and decreases back to normal after the overlays are reconfigured and the trees are repaired. When it comes to the number of exchanged messages, shown in Figure 24, it is possible to see that when the nodes leave the system, a considerable amount of *GRAFT* and *PRUNE*

messages are triggered in order to fix the broadcasting trees. This situation explains the instability of the number of *GOSSIP* and *IHAVE* messages. After the overlay and broadcasting trees are repaired, the number of *PRUNE* and *GRAFT* messages go back to zero. The reason why the overall number of messages exchanged after this event is significantly lower is purely due to the lower number of nodes in the system.

Nodes	Degree	Time (in milliseconds)
100	5	0.0218
500	8	0.3411
1000	10	0.948

Table 4: Random regular graph algorithm measured time.

Nodes need to get the information from the views of every node in the system, for the current round, but also the number of previous rounds that are considered by the free-rider detection mechanism, to provide verifiability of the views, as well as to be able to verify the authenticity of the histories and POCs. Two approaches that can be taken to get this information. The first one is for each node to keep in memory the graph that represents the connections of every node in the overlay. This would mean that whenever a node needed to verify the legitimacy of a POC for example, the node would check the graph to verify that the corresponding nodes were indeed in each other's views on that round. This approach has the disadvantage of occupying a large amount of memory, especially for bigger overlays. If we consider an overlay of 1000 nodes, with view sizes of 10, and if we consider that the total amount of rounds that are considered by the free-rider detection system is 5, then the amount of memory that a node would spend on membership information would be of 136kB (assuming node id's of 8 bytes and utilization of adjacency lists).

An alternative around this memory intensive approach is for the node to run the algorithm that generates the view graph using the seed of the corresponding round, that was provided by the central authority. This way, whenever a node needs to verify a POC, it can simply generate the graph corresponding to the round of the POC, and then verify that the connection is indeed authentic, and then dispose of the graph when it is no longer necessary. In order to verify the feasibility of this approach the algorithm of the view generation was implemented in the C++ language, and tested using a modern laptop. As shown in Table 4, even for overlays with up to a thousand nodes, the cost of the algorithm in terms of computation effort is very low, and would therefore enable this strategy. For even an overlay with a 1000 nodes and a degree of 10, it took a little less of a millisecond to perform the generation of the view graph. A hybrid approach would also further optimize the trade-off between memory utilization and computation effort spent on generating the view graph, and that is a subject that could be studied in future work.

6

CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

Peer-to-peer broadcasting algorithms are appealing due to their scalability potential without a significant increase in costs and loss of performance. They use the resources from the members that are part of the network in order to achieve their goal. However, most existing algorithms consider a model where all the nodes in the system are fully cooperative, and therefore are open to exploitation when used in the real world. In reality, malicious nodes can try to deteriorate the experience of other nodes or the system in general, and rational nodes seek to maximize their utility from the system without contributing with their resources.

Bycast aims at creating a secure, reliable broadcasting algorithm that does not compromise performance to reach a high level of security. It achieves this goal by employing a rigid structure to the overlay whose verifiability prevents exploitation by malicious nodes, while not incurring in a big cost to the nodes of the system. The conscious decision was made of making some strategic compromises, for example when it comes to the use of a semi-closed full membership, and the consequent higher memory utilization. Bycast also uses a semi-centralized coordinator, to guarantee strong security properties that allow for it to handle a great amount of malicious and rational nodes, while still providing good reliability and latency guarantees. To achieve a high level of resiliency without a significant increase in the bandwidth utilization, Bycast uses an adaptation of the Plumtree protocol as its broadcasting algorithm. It combines the natural resiliency of gossip algorithms with a low overhead typical of tree-based approaches. Furthermore, the use of multiple trees increases the fairness of the system, which facilitates the detection of free-riding behavior. Bycast employs a hybrid auditing system where nodes monitor the contribution of their neighbors. If a node contributes with less than a specified threshold then it will face consequences such as eviction from the system. This auditing scheme is also resilient to attacks, by employing an accusation system in order to detect maliciouus nodes that try to compromise the free-rider detection system. The central coordinator is responsible by gathering accusations and issuing eviction notices when it determines that a node has acted in an illegitimate way.

Due to the strong security properties, such as view verifiability, Bycast is able to handle up to 25% of colluding byzantine nodes in the network, and up to 40% of non-colluding byzantine nodes. Bycast is also able to handle up to 35% of free-riding nodes without it reflecting on its reliability, with 50% of free-riding nodes in the overlay resulting in a reliability of over 95%. We believe even with the trade-offs that were made to reach this level of

security, Bycast is be able to scale up to a few thousands of nodes. For bigger overlays, the main bottlenecks would be firstly the existence of the centralized coordinator that manages access control and the departure of nodes from the overlay, and also the memory utilization for each node. It is not clear if it is possible to achieve a high level of security while not relying on the utilization of full memberships without employing expensive security mechanisms like distributed consensus. Fireflies and BAR Gossip, for example, two of the most well known algorithms in this area, rely on full memberships available to the nodes in order to ensure their security properties.

In sum, Bycast takes a more conservative approach when it comes to the membership service properties, that allows it to reach a high level of resiliency to byzantine behavior, while not compromising its scalability and performance.

6.2 FUTURE WORK

In order to increase the degree of confidence of its security measure, Bycast can be tested in more complex and realistic scenarios. Some of these scenarios include the heterogeneity of the system's nodes when it comes to the available resources and connection speed, and different strategies that malicious nodes can choose in order to compromise the system. It would also be beneficial for Bycast to be implemented and tested in a real world scenario, as opposed to simulated using a round-based approach. A real life test-bed would corroborate the security mechanisms with a higher degree of confidence than a simulation.

This thesis presents two approaches when it comes to the verification of the views. Generating the overlay graph from the seed every time a connection needs to be checked or maintaining the information of the overlay graph constantly in memory. That trade-off should be studied in order to find the optimal strategy between memory utilization and processing overhead.

BIBLIOGRAPHY

Basmah Alotibi, Noof Alarifi, Majid Abdulghani, and Lina Altoaimy. Overcoming free-riding behavior in peer-to-peer networks using points system approach. *Procedia Computer Science*, 151:1060–1065, 2019. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2019.04.150. URL https://www.sciencedirect.com/science/article/pii/S1877050919306179. The 10th International Conference on Ambient Systems, Networks and Technologies (ANT 2019) / The 2nd International Conference on Emerging Data and Industry 4.0 (EDI40 2019) / Affiliated Workshops.

Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, may 1999. ISSN 0734-2071. doi: 10.1145/312203.312207. URL https://doi.org/10.1145/312203.312207.

Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. *Computer Networks*, 53(13):2340–2359, 2009. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2009.03.008. URL https://www.sciencedirect.com/science/article/pii/S1389128609001182. Gossiping in Distributed Systems.

Nuno Carvalho, Jose Pereira, Rui Oliveira, and Luis Rodrigues. Emergent structure in unstructured epidemic multicast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 481–490. IEEE, 2007.

T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. URL http://www.ietf.org/rfc/rfc5246.txt. Updated by RFCs 5746, 5878, 6176.

J. Dinger and H. Hartenstein. Defending the sybil attack in p2p networks: taxonomy, challenges, and a proposal for self-registration. In *First International Conference on Availability, Reliability and Security (ARES'06)*, pages 8 pp.–763, 2006. doi: 10.1109/ARES.2006.45.

John R. Douceur. The sybil attack. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, pages 251–260, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45748-0.

Morris J. Dworkin. Sp 800-38b. recommendation for block cipher modes of operation: The cmac mode for authentication. Technical report, Gaithersburg, MD, USA, 2005.

João F. A. e Oliveira, Ítalo Cunha, Eliseu C. Miguel, Marcus V. M. Rocha, Alex B. Vieira, and Sérgio V. A. Campos. Can peer-to-peer live streaming systems coexist with free riders? In *IEEE P2P 2013 Proceedings*, pages 1–5, 2013. doi: 10.1109/P2P.2013.6688712.

Mario Ferreira, Joao Leitao, and Luis Rodrigues. Thicket: A protocol for building and maintaining multiple trees in a p2p overlay. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, page 293–302, USA, 2010. IEEE Computer Society. ISBN 9780769542508. doi: 10.1109/SRDS.2010.19. URL https://doi.org/10.1109/SRDS.2010.19.

Gabriela Gheorghe, Renato Lo Cigno, and Alberto Montresor. Security and privacy issues in p2p streaming systems: A survey. *Peer-to-Peer Networking and Applications*, 4:75–91, 06 2010. doi: 10.1007/s12083-010-0070-6.

Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod, and Swagatika Prusty. Lifting: Lightweight freerider-tracking in gossip. In Indranil Gupta and Cecilia Mascolo, editors, *Middleware 2010*, pages 313–333, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16955-7.

Zygmunt J. Haas, Joseph Y. Halpern, and Li Li. Gossip-based ad hoc routing. *IEEE/ACM Trans. Netw.*, 14(3): 479–491, jun 2006. ISSN 1063-6692. doi: 10.1109/TNET.2006.876186. URL https://doi.org/10.1109/TNET.2006.876186.

Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

M. Haridasan and R. van Renesse. Defense against intrusion in a live streaming multicast system. In *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*, pages 185–192, 2006. doi: 10.1109/P2P.2006.15.

Maya Haridasan and Robbert van Renesse. Securestream: An intrusion-tolerant protocol for live-streaming dissemination. *Computer Communications*, 31(3):563–575, 2008. ISSN 0140-3664. doi: https://doi.org/10.1016/j.comcom.2007.08.028. URL https://www.sciencedirect.com/science/article/pii/S0140366407003180. Special Issue: Disruptive networking with peer-to-peer systems.

István Hegedűs, Gábor Danner, and Márk Jelasity. Gossip learning as a decentralized alternative to federated learning. In José Pereira and Laura Ricci, editors, *Distributed Applications and Interoperable Systems*, pages 74–90, Cham, 2019. Springer International Publishing. ISBN 978-3-030-22496-7.

Kevin Hoffman, David Zage, and Cristina Nita-Rotaru. A survey of attack and defense techniques for reputation systems. *ACM Comput. Surv.*, 42(1), dec 2009. ISSN 0360-0300. doi: 10.1145/1592451.1592452. URL https://doi.org/10.1145/1592451.1592452.

Håvard D. Johansen, Robbert Van Renesse, Ymir Vigfusson, and Dag Johansen. Fireflies: A secure and scalable membership and gossip service. *ACM Trans. Comput. Syst.*, 33(2), may 2015. ISSN 0734-2071. doi: 10.1145/2701418. URL https://doi.org/10.1145/2701418.

Jeong Han Kim and Van H. Vu. Generating random regular graphs. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, page 213–222, New York, NY, USA, 2003. Association

for Computing Machinery. ISBN 1581136749. doi: 10.1145/780542.780576. URL https://doi.org/10.1145/780542.780576.

Joao Leitao, Jose Pereira, and Luis Rodrigues. Epidemic broadcast trees. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 301–310. IEEE, 2007.

João Leitão, José Pereira, and Luís Rodrigues. *Gossip-Based Broadcast*, pages 831–860. 10 2010. doi: 10.1007/978-0-387-09751-0_29.

Harry C Li, Allen Clement, Edmund L Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. Bar gossip. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 191–204, 2006.

Yong Liu, Yang Guo, and Chao Liang. A survey on peer-to-peer video streaming systems. *Peer-to-Peer Networking and Applications*, 1(1):18–28, 2008. doi: 10.1007/s12083-007-0006-y.

J. Pereira, U. do Minho, L. Rodrigues, U. de Lisboa, M.J. Monteiro, R. Oliveira, and A.-M. Kermarrec. Neem: network-friendly epidemic multicast. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, pages 15–24, 2003. doi: 10.1109/RELDIS.2003.1238051.

A. Perrig, R. Canetti, J.D. Tygar, and Dawn Song. Efficient authentication and signing of multicast streams over lossy channels. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 56–73, 2000. doi: 10.1109/SECPRI.2000.848446.

Adrian Perrig, Ran Canetti, J. Tygar, and Dawn Song. The tesla broadcast authentication protocol. *RSA CryptoBytes*, 5, 11 2002. doi: 10.1007/978-1-4615-0229-6_3.

Patricia Ruiz and Pascal Bouvry. Survey on broadcast algorithms for mobile ad hoc networks. *ACM Comput. Surv.*, 48(1), jul 2015. ISSN 0360-0300. doi: 10.1145/2786005. URL https://doi.org/10.1145/2786005.

Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems*, 08 2003. doi: 10.1007/s00530-003-0088-1.

Vivek Vishnumurthy, Sangeeth Ch, and Emin Sirer. Karma : A secure economic framework for peer-to-peer resource sharing. 06 2003.

Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.

Kan Zhang, Nick Antonopoulos, and Zaigham Mahmood. A review of incentive mechanism in peer-to-peer systems. In *2009 First International Conference on Advances in P2P Systems*, pages 45–50, 2009. doi: 10.1109/AP2PS.2009.15.