UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Gossip-based broadcast protocols

João Carlos Antunes Leitão

MESTRADO EM ENGENHARIA INFORMÁTICA

May 2007

Gossip-based broadcast protocols

João Carlos Antunes Leitão

Dissertação submetida para obtenção do grau de
MESTRE EM ENGENHARIA INFORMÁTICA

pela

FACULDADE DE CIÊNCIAS DA UNIVERSIDADE DE LISBOA

DEPARTAMENTO DE INFORMÁTICA

May 2007

This work was partially suported by FCT with the co-funding of FEDER through
the project P-SON: Probabilistically-Structured Overlay Networks (POSC/EIA/60941/2004).

Aos meus avós: Deolinda e Manuel,
à minha Mãe e ao meu Irmão.

Acknowledgements

I begin these acknowledgements with the most sincere thanks to my advisor, Professor Luís Rodrigues. His constant dedication and support during the realization of this work were essential to its success. Working with him during my master, allowed me to gain new insight not only in computer science but also in other fields of expertise. I am very grateful for the opportunity to work with him.

I also want to thank José Orlando Pereira. Because of his vision and insight this work was able to attain its current level of quality.

My thanks are also extended to the LaSIGE, the Department of Informatics of the Faculty of Sciences of the University of Lisbon, and its members for the conditions they provided to support my work. In particular, I wish to thank every member of the DIALNP research group and everybody in the "famous" laboratory room 6.3.33 for their constant support, help, ideas, and the great work (and fun) environment they provided. Working with these people has been a real pleasure that I will never forget.

My friends, which fortunately are many, also have been a constant presence in my life, and they all have, in some way, contributed to this work. To all of them I extend my thanks, and in particular I have to say how very grateful I am for the reviews, suggestions and constant support of Inês Fragata, Ricardo Graça and Liliana Rosa.

Finally, my thanks to all my family. In particular to my brother Paulo Leitão who believed in me from the first day.

Abstract

Gossip, or epidemic, protocols have emerged as a powerful strategy to implement highly scalable and resilient reliable broadcast primitives. Due to scalability reasons, each participant in a gossip protocol maintains only a partial view of the system, from which they select peers to perform gossip exchanges. On the other hand the natural redundancy of gossip protocols makes them less efficient than other approaches that rely in some sort of structured overlay network.

The thesis addresses gossip protocols and the problem of building partial views to support their operation. For that purpose, the thesis presents and evaluates a new scalable membership protocol, which is called HyParView, that provides a number of properties, such as degree distribution, accuracy and clustering coefficient, that are highly useful to the construction of efficient gossip protocols.

The thesis also introduce two new gossip protocols, based on HyParView, that provide high reliability with small message redundancy. One is an eager push gossip protocol while the other is a tree based gossip broadcast protocol. Simulations results show that, in comparison with other existing protocols, HyParView-based gossip protocols not only provide better reliability but also support higher percentages of node failures, and are able to recover faster from these failures.

Keywords: membership protocols, gossip protocols, reliable broadcast, fault tolerance

Resumo

Os protocolos de rumor (gossip), também chamados de epidémicos, emergiram recentemente como uma estratégia viável para a concretização de primitivas de difusão altamente escaláveis e resilientes. Por maior capacidade de escala, cada participante num protocolo de rumor mantêm apenas uma vista parcial de todo o sistema, a partir da qual efectua a selecção dos nós com os quais realiza troca de rumores. Por outro lado, a redundância natural destes protocolos tornam-nos menos eficientes do que outras abordagens que se baseiam na utilização de redes sobrepostas com estrutura.

Esta tese aborda protocolos de disseminação epidémica e o problema da construção de vistas parciais para suportar a sua operação. Com esse fim, a tese apresenta e avalia um novo protocolo escalável de filiação denominado HyParView, que oferece várias propriedades, como a distribuição de grau, exactidão e coeficiente de agrupamento, que são bastante úteis na construção de protocolos de disseminação epidémica eficientes.

Esta tese introduz também dois novos protocolos de disseminação epidémica baseados no HyParView, que oferecem elevada confiabilidade produzindo um número reduzido de mensagens redundantes. Um destes protocolos baseia-se na utilização de "eager push" enquanto que o outro baseia-se na utilização de uma árvore de disseminação epidémica. Resultados obtidos através de simulações mostram que, quando comparado com outros protocolos existentes, os protocolos de disseminação epidémica baseados no HyParView, não só conseguem garantir melhores valores de confiabilidade mas também exibem um tempo de recuperação às falhas inferior.

Palavras Chave: protocolos de filiação, protocolos epidémicos, broadcast confiável, tolerância a faltas

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

A gossip, or epidemic, broadcast protocol is a protocol that operates as follows. When a node wants to broadcast a message, it selects t nodes from the system at random (this is a configuration parameter called *fanout*) and sends the message to them; upon receiving a message for the first time, each node repeats this procedure (Kermarrec *et al.*, 2003). Gossip protocols are an interesting approach because they are highly resilient (these protocols have an intrinsic level of redundancy that allows them to mask node and network failures) and distribute the load among all nodes in the system.

Ideally, one would like to have each participant to select gossip targets at random from the entire system membership. Unfortunately, this is not a scalable solution, not only due to the high memory costs associated with maintaining full membership information about all nodes participating in the protocol, but also due to the cost of ensuring that such information is up-to-date.

To overcome this scalability problem, several existing protocols rely on a *partial view*, instead of full membership information. A partial view is a small subset of the entire system membership, from which nodes can select peers to whom relay gossip messages. This solution resolves the scalability issues, but it also makes the system more vulnerable to the effects of nodes failures (for instance, by increasing the chance of having the system partitioned.). If partial views are carefully constructed, gossip protocols may be used to implement highly scalable and resilient reliable broadcast primitives.

On the other hand, gossip based broadcast protocols are less efficient than other approaches that rely on some sort of structured overlay to disseminate information, as the intrinsic redundancy of gossip protocols produces more network traffic, which might exhaust network capacity, making any sort of broadcast impossible. This is the price to pay in order to avoid the high cost and additional complexity of construction, and also the time costs for repair, such structured overlays.

Motivation

The work presented on the thesis is motivated by the following observations:

- The fanout of a gossip protocol is constrained by the target reliability level and the desired fault tolerance of the protocol. When partial views are used, the quality of these views has an impact on the fanout required to achieve high reliability[1].

- High failure rates may have a strong impact on the quality of partial views. Even if the membership protocol has healing properties, the reliability of message broadcasts after heavy failures may be seriously affected.

- Structured approaches to reliable broadcast use less network resources, by avoiding redundant messages. If some structure can be extracted from the normal operation of the gossip protocol some resource consumption gains can be achieved without the maintenance associated with pure structured approaches.

Contributions

The primary goals of this work are to design and implement: *i)* a membership service for gossip-based reliable broadcast, and *ii)* gossip strategies that can be combined with such a service in order to provide high values of message delivery (aiming at 100% delivery) even in scenarios where large number of nodes fail simultaneously.

[1]A precise definition of reliability is given in Section 2.3.

In detail, the contributions of the thesis can be enumerated as follows:

- A novel, highly scalable, gossip based membership protocol which is based on two distinct partial view, for different purposes, and that are maintained by different strategies. This membership protocol, named HyParView, is able to sustain large rates of node failures while ensuring high reliability.
- An eager push gossip protocol, developed to leverage on HyParView's properties, that can ensure high reliability values and fast message dissemination, while using smaller fanout values than other existing protocols.
- A tree based gossip protocol that, combined with HyParView, is able to provide as much reliability as the flood strategy without generating large amounts of redundant messages. This strategy combines eager push and lazy push gossip approaches, to explicitly produce a fault tolerant spanning tree.

Thesis Structure

The rest of this thesis is structured as follows:

Chapter 2 presents related work, addressing topics such as gossip protocols, existing membership protocols, and application level multicast.

Chapter 3 starts with the description of the HyParView membership protocol then, based on this membership protocol, two distinct gossip protocols are proposed, namely a flood gossip protocol and a tree based gossip protocol.

Chapter 4 shows an extensive evaluation of the previous protocols based on simulations.

Chapter 5 presents the conclusions and future work.

Chapter 2

Related Work

The thesis addresses gossip-based broadcast protocols and the underlying membership protocols required for their operation. This chapter introduces fundamental concepts, starting with a brief explanation of gossip protocols in general. The concepts of *peer sampling service* and *partial view* are then introduced followed by the definition of a number of metrics which are used in the evaluation of the overlay networks established by these partial views. Next, some specific metrics to evaluate the performance of gossip protocols are introduced. Some techniques used in the creation and maintenance of overlay spanning trees used in application-level multicast are presented. This chapter concludes with a brief overview of existing membership services for gossip-based broadcast protocols and a description of existing solutions for application-level multicast.

2.1 Gossip Protocols

This section introduces gossip protocols and some strategies used in the implementation of these protocols.

2.1.1 Gossip Overview

The initial inspiration to gossip protocols comes from sociology - by the observation of how gossips spreads in a community - and biology - by the observation of how diseases spreads over a population - the last justifies the designation of "*epidemic protocols*", another name by which this class of protocols is also known.

Gossip protocols have been proposed as a building block to solve various problems in distributed systems namely: consistency management in replicated databases (Demers *et al.*, 1987), failure detection (Renesse *et al.*, 1998), publish-subscribe (Eugster *et al.*, 2003) and application level reliable broadcast (Ganesh *et al.*, 2001; Voulgaris *et al.*, 2005).

The basic idea behind gossip is to have all participants in the protocol to collaborate, in the same manner, to disseminate information. To this end, when a node wishes to send a broadcast message, it selects t nodes at random - its *gossip targets* - and sends the message to them (t is a typical configuration parameter called *fanout*, which is explained later in section 2.1.2). Upon receiving a message for the first time, a node repeats this process (selecting t gossip targets and forwarding the message to them).

If a node receives the same message twice - which is possible, as each node selects its gossip targets in an independent way (without being aware of gossip targets selected by other nodes) - it simply discards the message. To allow this, each node has to keep track of which messages it has already seen and delivered. Without purging, this set of message identifiers may grow continually during the execution of the protocol. The problem of purging message histories is out of the scope of this thesis; it has been addressed previously, for instance in Koldehofe (2003).

The simple operation model of gossip protocols not only provides high scalability but also, a high level of fault tolerance, as its intrinsic redundancy is able to mask network omissions and also node failures.

2.1.2 Parameters

There are two important parameters associated with the configuration of gossip protocols:

Fanout: This is the number of nodes that are selected as gossip targets by a node for each message that is received by the first time. There is a trade-off associated with this parameter between desired fault tolerance / reliability level and redundancy level of the protocol. High fanout values guarantee a major fault tolerance level and probability of atomic delivery but it also generates an increasing redundant network traffic.

Maximum rounds: This is the maximum number of times a given gossip message is retransmitted by nodes. Each message is transmitted with a *round value* - initially with value zero - which is increased each time a node retransmit the message. Nodes will only retransmit a message if its *round value* is smaller than the *maximum rounds* parameter.

A gossip protocol can operate in one of the two following modes:

- Unlimited mode: In this mode of operation the parameter maximum rounds is undefined, and there is no specific limit to the number of retransmissions executed to each gossip message.
- Limited mode: In this mode of operation the parameter maximum rounds is defined with a value above 0, effectively limiting the maximum hops executed by each message in the overlay[1].

There is an inherent a trade-off between reliability and redundancy level associated with the use of this attribute. In unlimited mode (or configuring the *maximum rounds* parameter with high values) there is a major probability to achieve atomic delivery (as defined in Kermarrec *et al.* (2003)), on the other hand, there will be more redundant messages produced.

2.1.3 Strategies

We distinguish the following four approaches to implement a gossip protocol:

Eager push approach: Nodes send messages to random selected peers as soon as they receive them for the first time.

[1]Neighboring relations between nodes form an overlay network, as it will be explained later, in Section 2.2.2.

Pull approach: Periodically, nodes query random selected peers for information about recently received messages. When they receive information about a message they did not received yet, they explicitly request to that neighbor the message. This is a strategy that works better when combined with some best-effort broadcast mechanism (*i.e.* IP Multicast (Deering & Cheriton, 1990)).

Lazy push approach: When a node receives a message for the first time, it gossips only the message identifier (*i.e.* for instance, the hash of the message) and not the full payload. If peers receive an identifier of a message they have not received, they make an explicit pull request.

Hybrid approach: Gossip is executed in two distinct phases. A first phase uses push gossip to disseminate a message in a best-effort manner. A second phase of pull gossip is used in order to recover from omissions produced in the first phase.

There is also a trade-off between eager push and pull strategies. Eager push strategies produce more redundant traffic but they also achieve lower latency than pull strategies, as pull strategies require at least an extra round trip time to produce a delivery. Lazy push gossip is very similar to pull gossip in the sense that it also requires at least an extra round trip time to achieve a message delivery. This approach differs from pull gossip in the sense that the dissemination process is started by the "sender" node whereas, in pull gossip, the dissemination process is started by the receiver.

One other aspect to retain is that eager push gossip does not require, contrary to pull/lazy push gossip, to maintain copies of delivered messages for later retransmission upon request. Hence, pull/lazy push gossip approaches are more demanding in terms of memory usage at each node.

2.2 Membership

We now introduce a number of concepts relevant in the context of membership protocols.

2.2.1 Peer Sampling Service

A *peer sampling service* (which was introduced in Jelasity *et al.* (2004)) is a abstract service that allows nodes, executing a gossip protocol, to obtain a subset from the full group of nodes executing the protocol.

The proposed interface of this service is quite simple and is only composed by the following two methods:

init(): This method initializes the service if it has not been initialized before. Note that, although the specific procedure for this method is implementation dependent, it should, at least, ensure that the probability of other participating nodes selecting the identifier of the node that called the init method, as a return value of the `getPeer()` method is greater than 0.

getPeer(): This method returns the identifier of a participating node, as long as there exists more than one node executing the service. The node returned should be selected at random across nodes that have called the `init()` method, although the specific qualities of this randomness (*i.e.* correlation with returned identifiers from previous call of this method) are implementation dependent.

The `getPeer()` method is enough to support the requirements of any gossip protocol - as a node can call repeatedly this method if it requires more than one peer - however, in practice, this method can (and should) be redefined as:

getPeer($n, peer$): Where n is an integer greater than zero and *peer* is a node identifier. This method returns a list with, at most, n identifiers of participating nodes that does not contain the *peer* identifier nor the identifier of the invoking node. This method should be called with n equal to the fanout used by the gossip protocol and *peer* should be the identifier of the node who sent the message to the invoking node[1].

[1]When a node wishes to send a message by the first time, the *node* argument should take a *null* value.

2.2.2 Partial View

A *partial view* is a set of node identifiers maintained locally at each node. This set should be a much smaller than the full system membership information; the size constraint is related with scalability requirements, that should be, ideally, of logarithmic size with the number of processes in the system. Typically, an identifier is a tuple $(ip : port)$ that allows a node to be reached.

A membership protocol is in charge of initializing and maintaining the partial views at each node in face of dynamic changes in the system membership. For instance, when a new node joins the system, its identifier should be added to the partial view of (some) other nodes and it will have to create its own partial view, including identifiers of nodes already in the system. On the other hand, if a node fails or leaves the system, its identifier should be removed from all partial views as soon as possible.

Partial view establish *neighboring* associations among nodes. Therefore, partial views define an overlay network or, in other words, partial views establish an oriented graph that captures the neighbor relation between all the nodes executing the protocol. In this graph, nodes are represented by a vertex while a neighbor relation is represented by an arc originating from the node who contains the target node in his partial view.

One possible implementation of a peer sampling service is to use a membership service that maintains a partial view of participating nodes at each node. The selection of nodes to serve as gossip target is then performed locally using the partial view.

2.2.3 Strategies To Maintain Partial Views

There are two main strategies that can be used to maintain partial views, namely:

Reactive strategy: In this type of approach, a partial view only changes in response to some external event that affects the overlay (*i.e.* a node joining or leaving the system). In stable conditions, partial view remains unaltered. Scamp (Ganesh *et al.*, 2001, 2003) is an example of such an algorithm[1].

[1]To be precise, Scamp is not purely reactive as it includes a lease mechanism that forces nodes to periodically rejoin.

Cyclic strategy: In this type of approach, a partial view is updated every ΔT time units, as a result of some periodic process that usually involves the exchange of information with one or more neighbors. Therefore, a partial view may be updated even if the global system membership is stable. Cyclon (Stavrou *et al.*, 2002; Voulgaris *et al.*, 2005) is an example of such an algorithm.

Reactive strategies usually rely on some failure detection mechanism to trigger the update of partial views when a node leaves the system. If the failure detection mechanism is fast and accurate, reactive mechanisms can provide faster response to failures than cyclic approaches. On the other hand, a cyclic strategy allows each node to select a wide range of distinct nodes as gossip targets for different messages even in stable conditions, as the elements of each partial view are continually changing.

2.2.4 Partial View Properties

In order to be useful, namely to support fast message dissemination and high level of fault tolerance to node failures, partial views must own a number of important properties. These properties are intrinsically related with graph properties of the overlay defined by the partial view of all nodes and are also used to measure the quality of these partial views. Some of the most important properties are:

Connectivity The overlay defined by the partial views should be connected. To consider an overlay as connected, there should be at least one path from each node to all other nodes[1]. If this property is not met, isolated nodes will not receive broadcast messages.

Degree Distribution In an undirected graph, the degree of a node is simply the number of edges of the node. Given that partial views define a directed graph, it is important to distinguish *in-degree* from *out-degree* of a node. The in-degree of a node n is the number of nodes that have n's identifier in their partial view;

[1]Obviously, if the graph is directed, the path between nodes have to respect the direction of arcs.

it provides a measure of the reachability of a node in the overlay. The out-degree of a node n is the number of nodes in n's partial view; it is a measure of the node contribution to the membership protocol and consequently a measure of the importance of that node to maintain the overlay.

If the probability of failure is uniformly distributed in the node space, for improved fault-tolerance both the in-degree and out-degree should be evenly distributed across all nodes executing the membership protocol.

Average Path Length A path between two nodes in the overlay is the set of edges that a message has to cross from one node to the other. The average path length is the average of all shortest paths between all pair of nodes in the overlay. This property is closely related to the overlay diameter. To ensure the efficiency of the overlay for information dissemination, it is essential to enforce low values of the average path length, as this value is related to the time (and number of hops in the overlay) a message will require to reach all nodes.[1]

Clustering Coefficient The clustering coefficient of a node is the number of edges between that node's neighbors divided by the maximum possible number of edges across those neighbors. This metric indicates a density of neighbor relations across the neighbors of a given node, having it's value between 0 and 1. The clustering coefficient of a graph is the average of clustering coefficients across all nodes. This property has a high impact on the number of redundant messages received by nodes when disseminating data, where a high value to clustering coefficient will produce more redundant messages. It also has an impact in the fault-tolerant properties of the graph, given that areas of the graph that exhibit high values of clustering will more easily be isolated from the rest of the graph.

Accuracy Accuracy of a node is defined as the number of neighbors of that node that have not failed divided by the total number of neighbors of that node. The accuracy of a graph is the average of the accuracy of all correct nodes.

[1]The reader should notice that this property is only meaningful if the property of connectivity is met. If the overlay is not connected then at least one node in unreachable which translates into a infinite shortest path between all other nodes and that node.

Accuracy has high impact in the overall reliability of any dissemination protocol using an underlying membership protocol to select its gossip targets. If the graph accuracy values are low, the number of failed nodes selected as gossip targets will be higher, which, in turn, can disrupt the gossip process. To avoid this,higher fanout values must be used to mask the selection of failed nodes.

2.3 Gossip Metrics

It is essential to define a set of metrics to be used in order to evaluate the performance of gossip protocols. Some of the metrics used in this thesis are defined as follows:

Reliability Gossip reliability is defined as the percentage of active nodes that deliver a gossip broadcast. A reliability of 100% means that the protocol was able to deliver a given message to all active nodes or, in other words, that the message resulted in an atomic broadcast as defined in Kermarrec *et al.* (2003).

Relative Message Redundancy (RMR) This metric measures the messages overhead in a gossip protocol. It is defined as:

$$\left(\frac{m}{n-1}\right) - 1$$

where m is the total number of payload messages exchanged during the broadcast procedure and n is the total number of nodes that received that broadcast. This metric is only applicable when at least 2 nodes receive the message.

A RMR value of zero means that there is exactly one payload message exchange for each node in the system, which is clearly the optimal value. By opposition, high values of RMR are indicative of a broadcast strategy that promotes a poor network usage. Note that it is possible to achieve a very low RMR by failing to be reliable. Thus the aim is to combine low RMR values with high reliability. Furthermore, RMR values are only comparable for protocols that exhibit similar reliability. Finally, note that in pure gossip approaches, RMR is closely related with the protocol fanout, as it tends to fanout -1.

Control messages are not considered by this metric, as they are typically much smaller than payload messages hence, they are not the main source of contribution to the exhaustion of network resources. Moreover, these messages can be sent using piggyback strategies providing a better usage of the network.

Last Delivery Hop (LDH) The last delivery hop is the round number of the last message that is delivered by a gossip protocol or, in other words, is the maximum number of hops that a message must be forwarded in the overlay that causes a message delivery. This metric has a close relation with the diameter of the overlay used to disseminate messages, and it also gives some insight on the latency of a gossip protocol.

The reader should notice that, if all links between nodes were to exhibit the same latency, the latency of a gossip broadcast transmission would simply be the last deliver hop multiplied by the *per hop* latency.

2.4 Application-level Multicast

Application-level multicast (Chu *et al.*, 2000) appears as an alternative to IP Multicast (Deering & Cheriton, 1990), to circumvent the deployment problems of IP multicast in the internet structure (Diot *et al.*, 2000).

Several application-level multicast solutions have been proposed, such as those presented in Ratnasamy *et al.* (2001), Rowstron *et al.* (2001) or Zhuang *et al.* (2001). Usually, these solutions try to produce distribution structures like trees that have a performance comparable to that of IP Multicast (using low level metric such as, for instance, latency or physical link stress.).

Reduce end-to-end latency or physical link stress are not main goals of gossip, or of the work presented in this thesis, nevertheless one has to consider that the use of distribution trees allows a protocol to broadcast a message to large group of participants without generating the excessive redundancy in network traffic that may be produced when using gossip strategies.

Unfortunately, the overhead of building a distribution tree is usually very high. Also, these protocols usually exhibit problems when facing node failures, as the tree become disconnected and has to be repaired, which might exhibit a big complexity and large overheads. Until the tree is repaired, the messages can not be sent to participants in a reliable manner, impairing the reliability of the broadcast protocol. This is even more noticeable when massive failures occur in these systems; in these scenarios the only solution might be to rebuild, from scratch, the multicast tree.

2.4.1 Tree Construction

The key aspect of these protocols, is that nodes self-organize in a tree structure so that each node knows exactly to whom it has to forward messages.

In order to build these trees, state has to be set-up at nodes from the root of the tree to all receivers. There are two main strategies to accomplish this:

Receiver-based strategy: In this type of approach, the receiver sends a special message to the root of the tree. This message is used to set-up a path between the receiver and sender while it traverses the network. The process of adding a new member is complete as soon as the message reaches the root tree or any node that already maintains state concerning the tree.

This process allows for a faster node integration in the tree, as it does not always require that all nodes contact the root. On the other hand it might not select the best path between the root and the receiver if the capacity of links are not symmetric.

Scribe (Castro *et al.*, 2002; Rowstron *et al.*, 2001) is an example of an application-level multicast protocol that employs this strategy.

Source-based strategy: In this type of approach, the tree is constructed by selecting a path from the root of the tree to all individual receivers. The process is usually initiated when the root node receives a request from a specific receiver. Subsequently, the path between the nodes is then set up by routing a special message from the root to the receiver.

This strategy will choose the best path between the root and the receiver, but it has a additional cost in the set-up process, as it requires messages to travel from the receiver to the root and back from the root to the receiver.

Also the root is a bottleneck, as all receivers have to contact it in order to join the tree.

Bayeux (Zhuang *et al.*, 2001) is an example of an application-level multicast protocol that employs this strategy.

2.4.2 Tree Repairing

When a node fails, the tree becomes disconnected. The number of nodes that effectively become disconnected from the source will depend on the distance of the failing node to the root (intuitively, if a tree has n elements, a degree of d, and is balanced, the failure of a node that is connected to the root would leave approximately n/d nodes disconnected from the root).

Given that failures disconnect the tree, it is of paramount importance to have some process to repair it. As in the process of construction of the tree, there are two strategies to address this problem, that can be described as follows:

Receiver-based strategy: In this type of approach, each receiver is responsible to detect the failure of its parent, and to initiate actions to rebuild the tree when this happens. This is a technique used in Scribe.

Source-based strategy: In this type of approach, each node is responsible to detect the failure of its children, and when this happens it should take measures to link the orphan nodes to himself. This solution is not used very often as it requires each node to have full topology information on the tree. Nevertheless, Bayeux employs this technique.

2.5 Existing Protocols

In this section, some existing protocols are presented. First SCAMP and CYCLON are introduced: these are pure membership protocols that rely on partial views. Each is representative of a different strategy to maintain these partial views. Next NEEM and CREW are briefly introduced. Both are gossip protocols that use TCP connections to better disseminate information.

The section concludes with the introduction of some application level multicast protocols, and finally the MON, a system that produces *on-demand* overlay structures, is depicted.

2.5.1 Scamp

Scamp (Ganesh *et al.*, 2001, 2003), is a reactive membership protocol that maintains two separate views, a *PartialView* from which nodes select their targets to gossip messages, and a *InView* with nodes from which they receive gossip messages. One interesting aspect of this protocol is that the *PartialView* does not have a fixed size, it grows to values that are distributed around $\log n$, where n is the total number of nodes executing the protocol, without n being known by any node executing the protocol.

When one node wishes to join the overlay, it has to know a node that already belongs to the overlay, to which it sends a *new subscription request*. Upon reception of this request, a node forwards it to all neighbors that belong to its *PartialView* in the form of a *forwarded subscription request*; it also creates c additional copies of this *forwarded subscription request* that are forwarded to c random neighbors from the *PartialView*; c is a configuration parameter that is related with the level of fault tolerance supported by this protocol as it will affect the global distribution of degree (in-degree and out-degree) values across the overlay. Higher values of c will produce overlays in which nodes have, on average, higher degrees. In turn, this will also impact network usage, as well as other graph properties.

Upon receiving a *forwarded subscription request* a node integrates the new member in its local *PartialView* (if the node is not already present) with a probability p, where p is equal to $1/(1 + sizeof(PartialView))$. If the node does not integrate the new member, it forwards the request to a random neighbor in his own *PartialView*. To avoid these messages to be forwarded an infinite number of times, which is more probable when the number of nodes in the overlay is small, there is a upper limit to the number of times a node can forward the same message. When this limit is reached the message is simply dropped.

The *InView* is used when a node wishes to leave the overlay. In this case an unsubscribing node, say n_u, will send to some of it peers[1] in the *InView* a *replace request* containing a element from its *PartialView*, say n_p. The node that receives this request will replace in its *Partial View* the identifier of n_u with the received identifier n_p. To the remaining nodes in its *InView*, n_u will simply send a request asking them to remove its own identifier from their *PartialView*.

In order to recover from node isolation, this algorithm uses a mechanism in which nodes periodically send heartbeat messages to all members of their *PartialView*. If a node does not receive a heartbeat for a long time, it assumes that it has become isolated, and it sends a *new subscription request* to a random node in his own *PartialView*, in order to rejoin the overlay.

When a node fails ($i.e$ leaves the system without executing the unsubscription procedure), its identifier will remain in the *PartialViews* of some correct nodes, which means that it can still be selected by those nodes as a gossip target. In order to purge this identifiers from *PartialViews* of correct nodes, Scamp relies in a *lease mechanism*. When a node joins the overlay, its subscription has a finite lifetime which is called its *lease time*. When the lease of a node subscription expires, all peers having that node identifier in their *PartialView* should delete it. Each node is responsible to rejoin the overlay through a *new subscription request* sent to a random peer in its *PartialView* before the *lease time* of its last subscription expires. The *lease time* of each subscription might be set individually by each node (sending information relative to it in the *new subscription* request), or be enforced through a global configuration parameter that affects all nodes.

2.5.2 Cyclon

Cyclon (Voulgaris *et al.*, 2005), is a cyclic membership protocol where nodes maintain a fixed length *partial view*. The size of *partial view* is a protocol parameter: it takes into account the maximum number of nodes that are expected to participate in the protocol and the desired level of fault-tolerance (in the sense

[1]The number of peers who receive a *replace request* is $sizeof(InView) - c$ this is related with the overlay desired average degree.

that the bigger the *partial views* are, the smaller is the probability of the overlay to become partitioned, specially by having single isolated nodes).

This protocol relies in a *shuffle* operation which is executed every ΔT time units by every node. Basically, to execute a shuffle operation, a node selects the "oldest" node in its partial view and performs an exchange with that node. In the exchange, the node provides to its peer a sample of its partial view and, symmetrically, collects a sample of its peer's partial view. If the selected node does not reply to the shuffle request, the originator of the shuffle will assume that the selected node has failed, and removes its identifier from it's own partial view. The authors show that this behavior generates an overlay with similar properties to those of random graphs.

This protocol requires each node identifier, in partial views, to have an age value associated with it. The age value is increased for all node identifiers in the partial view at the beginning of each shuffle operation. Furthermore, since shuffle targets are selected according to their age, this protocol eliminates failed nodes identifiers from partial views in a bounded time.

As in Scamp, a node that wishes to join the overlay must know another node that already belongs to the overlay. The join operation is based on fixed length random walks on the overlay. The join process ensures that, if there are no message losses nor node failures, the in-degree of all nodes will remain unchanged. Additionally, the partial view of the new node will exhibit the same properties of the partial views of all other nodes in the overlay.

2.5.3 NeEM

NeEM, or Network Friendly Epidemic Multicast (Pereira *et al.*, 2003), is a gossip protocol that relies on the use of TCP to disseminate information across the overlay. In NeEM, the use of TCP is motivated by the desire to eliminate correlated message losses due to network congestion. The authors show that better gossip reliability can be achieved by leveraging on the flow control mechanisms of TCP.

NeEM applies buffer management techniques directly itself (by disabling TCP buffers) using several purging strategies to discard messages on overflow. This enables the gossip protocol to preserve throughput stability even at times when the network became congested and also avoids inter-blocking of nodes, due to exhaustion of TCP reception buffers.

NeEM uses its own (partial view) membership service, which is also maintained through gossip. This membership service is based on random walks in the overlay, with a probabilistically length dependent on a value p that is fixed and is a protocol parameter. Random walks are used when a node joins the overlay and also in a cyclic manner, to "advertise" neighbors to random nodes.

2.5.4 CREW

CREW (Deshpande *et al.*, 2006), is a gossip protocol for flash dissemination, *i.e.* fast simultaneous download of files by a large number of destinations using a combination of pull and push gossip. It uses TCP connections to implicitly estimate available bandwidth thus optimizing the fanout of the gossip procedure.

CREW uses an underlying membership service, also based on partial views, called Bounce. Bounce is briefly presented in Deshpande *et al.* (2005) where the authors claim, based on experimental results, that the use of the overlay produced by Bounce is equivalent to the selection of nodes uniformly at random, from all nodes in the system. Bounce relies in random walks to establish neighbor relations between nodes. Random walks are probabilistically terminated according to a certain probability p that depends on the degree of the receiving node, a random factor and finally, to avoid infinite sizes random walks in the overlay, the length of the actual random walk.

Unfortunately a full specification of the Bounce protocol is not available, nor a full evaluation of the protocol has been published.

The emphasis of CREW is on optimizing latency, mainly by improving concurrent pulling from multiple sources. A key feature is to maintain a cache of open connections to peers discovered using a random walk protocol, to avoid the latency of opening a TCP connection when a new peer is required.

2.5.5 Narada

The Narada protocol (Chu *et al.*, 2002), is used to support efficient application-level multicast, relying in dissemination trees that are produced in two distinct steps.

In the first step the protocol creates and maintains a random and rich connected overlay (that the authors name *mesh*) that try to ensures that quality[1] of paths between any two nodes in the overlay is comparable to the quality of the unicast path between that pair of nodes, and that each node has a limited number of neighbors.

Also, the overlay is self-organizing and self-improving, and it try to be as efficient as possible and adapt itself to network conditions, by using a set of heuristics that adds or removes links between nodes.

In a second step, the overlay is used to create several multicast trees rooted at each source. To this end a distance vector algorithm is run on top of the overlay. Nodes that wish to join a multicast group explicitly select their parents among their neighbors using information from the routing algorithm.

Unfortunately, Narada is targeted toward medium sized groups; all nodes maintain full membership list and some additional control information for all other nodes, and consequently it can't scale to very large systems. Also the normal dynamics of the algorithm may partition the overlay. Affecting the global reliability of the system, until the protocol is able to repair the overlay. The authors do not explicitly show results concerning the effect of failures in the reliability of the multicast.

2.5.6 Bayeux

Bayeux (Zhuang *et al.*, 2001), it is a source-specific, application-level multicast system that leverages in Tapestry (Zhao *et al.*, 2001), a wide-area location and routing architecture that also maintains an overlay network. Bayeux uses a *source-based* approach to set-up and tear down distribution trees that work as follows:

When a node wishes to join a multicast group, or in other words, a distribution tree, it must know the root of that group and send a JOIN message to that node. Upon receiving a JOIN request, a source node uses Tapestry to route a TREE

[1] In this context, quality refers to application dependent metrics such as latency or bandwidth.

message to the new node. As the JOIN message is routed along the Tapestry overlay, it is used to set-up state on nodes to explicitly create a distribution tree. There are also similar LEAVE and PRUNE messages, that are used in the same way to remove state from nodes when a receiver wishes to leave a multicast group.

Although Bayeux is fault-tolerant, as it take-in the fault-tolerance nature of the underlying Tapestry, it requires that root nodes maintain information concerning all receiving nodes, also root nodes are single point of failure and a bottleneck, as all messages that are broadcasted on the distribution tree must pass through them. The authors propose a replication scheme to compensate for this, never the less this implies that Bayeux will not scale properly in very large systems with several thousands of receivers.

There is also a lack of experimental results concerning the effect of failures (and massive node failures) on the reliability of the dissemination scheme.

2.5.7 Scribe

Scribe (Castro *et al.*, 2002; Rowstron *et al.*, 2001) is a scalable application-level multicast infrastructure built on top of Pastry (Rowstron & Druschel, 2001).

Scribe supports multicast groups with multiple senders. It constructs a distribution tree for each group, by using a receiver-based strategy and leveraging in Pastry as follows:

Each multicast group has a node that serves as *rendez-vous point*. This node is selected, and can be found by other nodes, using the multicast group name, and taking advantage of Pastry resource location mechanism. This node will serve as a root for the multicast tree. When a node wishes to join a multicast group it uses Pastry to route a JOIN message to the rendez-vous point. This message is used to set-up state in the intermediate nodes along the route, concerning the specific multicast message, thus constructing a distribution tree.

Repairing the tree is done by using a similar strategy as follows: Intermediate nodes periodically send HEARTBEAT messages to nodes they have registered as being their children. A node will suspect that its parent node has failed when it stops receiving HEARTBEAT messages from it. In this case, the node uses Pastry to send another JOIN message, that is used to set-up another route to the node, recovering the tree structure.

All state concerning multicast trees is maintained using a soft state approach. Therefore, nodes have to periodically refresh their interest in belonging to a multicast route by resending JOIN messages.

Although Scribe is fault-tolerant and it provides a mechanism to handle root failures, it only provides best-effort guarantees. The authors argue that strong reliability and also order guarantees are only required for some applications, and that those properties can be easily provided on top of Scribe.

2.5.8 MON

MON (Liang *et al.*, 2005), which stands for Management Overlay Network, is a system designed to facilitate the management of large distributed applications and is currently deployed in the PlanetLab testbed[1].

MON builds on-demand overlay structures that are used by users to issue a set of *instant management commands* or distribute software across a large set of nodes. To that end it uses a random overlay network based in *partial views* that is maintained by a *cyclic* approach. It supports the construction of both tree structures and directed acyclic graphs structures.

A tree is always rooted at a external entity (named the MON client). To build the tree the MON client sends a SESSION message to a nearby MON node. A node that receives a SESSION message for the first time reply with a SESSIONOK and becomes a child node of the SESSION sender. It then sends k SESSION messages to random nodes from its partial view. A node that receives a SESSION message for a second time simply sends a PRUNE message to its originator. Hence the tree is constructed using the combination of a sender-based strategy with a gossip strategy, where k is the gossip fanout value.

To build a directed acyclic graph, where a node can have more than one parent, MON employs the same algorithm with the following modifications in order to avoid cycles: Each node has a *level* value, where the level at the root is 1 and the level of other nodes is 1 plus the level of their (first) parent. When a

[1]http://planet-lab.org/

node receives a second Session message, that also carries the level value of the node who sent it, a node can accept the message, and reply with a SessionOK message, if the level value in the Session is smaller than its own (therefore, it gains one more parent node).

Because MON is aimed at supporting short-lived interactions hence, it does not require to maintain these structures for prolonged time, therefore, it does not have any repair mechanism to cope with failures. Also, it only gives probabilistic coverage of all nodes, as the gossip strategy used to disseminate the Session message only gives probabilistic atomic broadcast guarantees.

2.6 Summary

This chapter introduced gossip protocols and some fundamental concepts, which will be central in the following discussion presented in the thesis.

It introduces an abstract service: *peer sampling service* and also defines the *partial view* concept. In the following chapter a new membership protocol - HyParView - that implements the peer sample service abstraction based on partial views will be presented.

Some metrics, used in the evaluation of overlay networks established by partial views, were explained and specific metrics for evaluating gossip strategies were also proposed. The chapter follows by presenting some existing gossip-based membership and application-level multicast protocols.

In the following chapter a novel gossip-based membership protocol - HyParView - is presented as well as two gossip strategies that can be used in combination with HyParView. Metrics presented in the chapter will be used in in Chapter 4 for the evaluation of the HyParView protocol and both gossip strategies.

Chapter 3

Gossip-based Broadcast Systems

This chapter presents the main contributions of the thesis. It starts with presenting a generic architecture of a gossip-based system. It shows how the specific components described in the thesis fit into that generic architecture and how they interact.

If follows with the presentation of HyParView, a membership protocol for gossip-based reliable multicast. The rationale behind the design of the protocol is presented as well as the description of the protocol in some detail. Pseudo code is depicted that illustrates some specific details of this protocol.

This chapter concludes with the presentation of two distinct gossip strategies, namely the eager push strategy and the tree strategy. Both strategies can be independently used with the HyParView membership protocol to obtain different trade-offs between reliability and efficiency in message broadcast.

3.1 Gossip-based System Architecture

Gossip protocols are a middleware component that is usually implemented between the application layer and the transport layer. Figure 3.1 shows a simple view of a gossip-based system. A gossip protocol has two main components, depicted in Figure 3.2, that can be described as follows:

Gossip strategy: It is the component that controls the message flow in the gossip protocol. It selects which messages are delivered to the above applica-

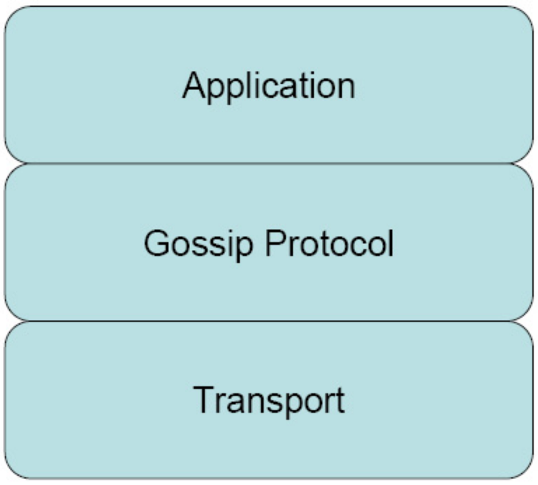


Figure 3.1: Generic gossip-based system architecture

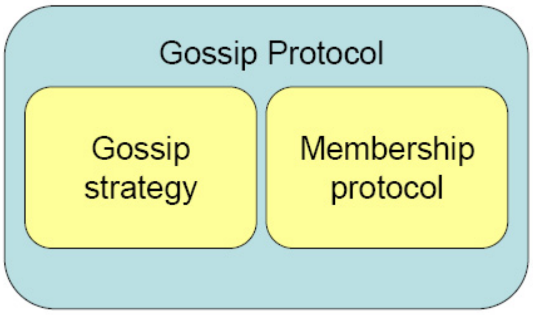


Figure 3.2: Components of a gossip protocol

tion and which messages are retransmitted to other nodes. This component should also determine which gossip mode to use (either eager push, pull, lazy push or hybrid modes, as seen in section 2.1.3) when sending messages to other nodes. If the gossip strategy requires the use of a pull, lazy push or hybrid mode, it also maintains a message repository to enable it to send messages when it receives explicit payload message requests from neighbors.

A gossip strategy may be *topology aware* or *topology independent*, in the sense that it might require to keep a track of neighbors maintained by the membership protocol or not. This will have implications in the interface used to obtain information about other peers, a issue that will be further addressed later in Section 3.1.2.

Membership protocol: It is the component that maintains state concerning other nodes participating in the gossip protocols, it implements the abstraction of a *peer sampling service* (as described in section 2.2.1). The main goal of this component is to provide to the gossip strategy component a sample of other peers from the system, to whom gossip messages may be sent.

3.1.1 Proposed Gossip-based System Architecture

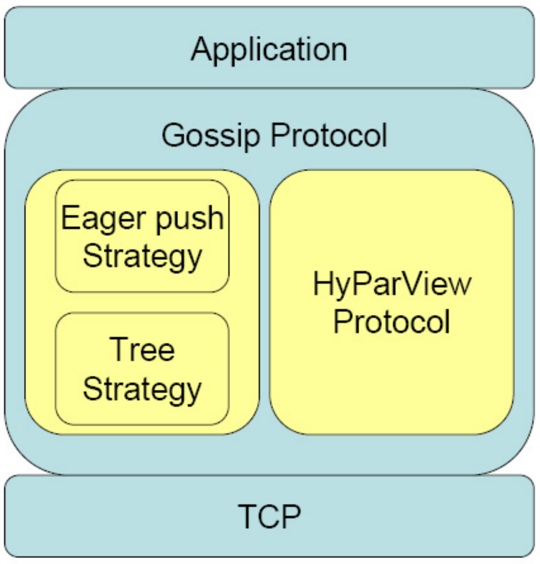


Figure 3.3: Specific gossip-based system architecture

Figure 3.3 illustrates how the specific components developed in the context of this work fit in generic gossip-based architecture. It also shows that we have selected TCP as the transport layer of choice for the operation of our protocols. This choice is justified below.

The components can be briefly described as follows:

HyParView protocol is a novel gossip-based membership protocol. It was developed to sustain high level of node failures, while ensuring connectivity of the overlay that is implicitly created by the neighbor relations between nodes. It also ensures that the overlay as a set of other desirable properties as listed in section 2.2.4.

Eager push algorithm is a topology independent gossip strategy which was devised to obtain a high reliability and low latency by leveraging on the special properties of HyParView.

Tree algorithm is a topology aware gossip strategy that reduces the message redundancy produced on the overlay, this is accomplished by creating a tree like structure across nodes.

These components will be described in detail in following sections of this chapter. They have been designed to operate on top of TCP. We selected TCP because it helps to maintain the symmetry in partial views of the membership protocol as well as enables the protocol to have a network friendly behavior, as the flow control mechanisms of TCP will avoid that the gossip protocol exhausts network resources. TCP also provides an unreliable failure detector service which is the basis for the reactive strategy employed in the maintenance of active views of the HyParView protocol (this will be further addressed later, in Section 3.2.). The advantages of TCP will became even more clearer as each component is described in more detail.

3.1.2 Components Interactions

The interaction between all components of the system is depicted in figure 3.4. These interactions are based on the interface exported by each component.

An `Init` call is used by the application to initialize the HyParView protocol. It also uses a `Broadcast` call to the gossip strategy component when it wishes to send a message to all nodes in the system. The application layer should also export a `Deliver` up-call. This call is used by the gossip strategy when a broadcast message is received by the first time. Notice that the gossip strategy

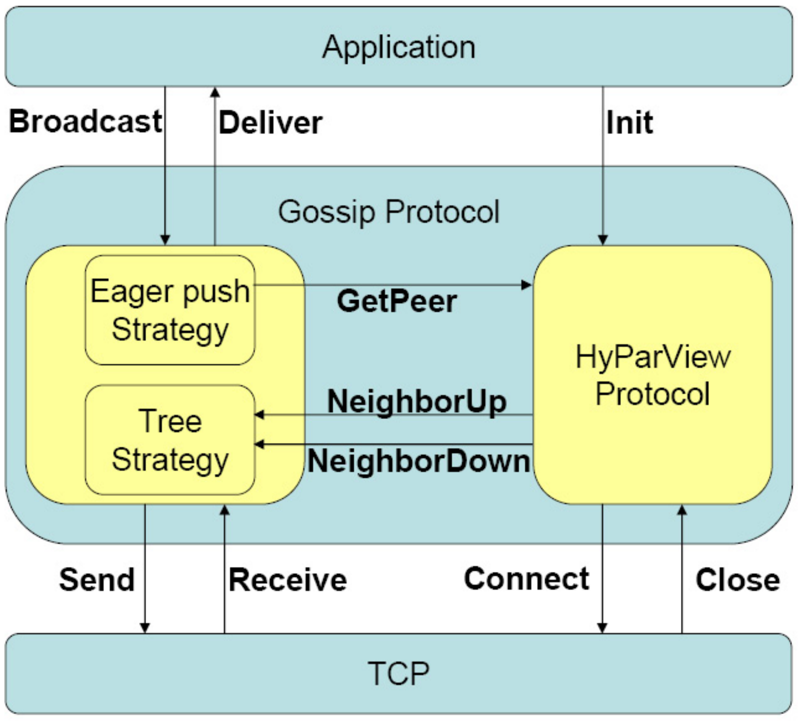


Figure 3.4: Interactions between components of the system

component should not deliver the same message to the application layer more than once.

Gossip strategies have two distinct ways to interact with the HyParView protocol. If the gossip strategy is topology independent (like the eager push protocol), it simply uses the `GetPeer` call. The signature of this call has been discussed in section 2.2.1. If the gossip strategy is topology aware, it requires to be informed whenever a change in the neighbors maintained by the HyParView protocol happens. To this end, the gossip strategy should support two callback methods, `NeighborUp` and `NeighborDown`, that are used by the HyParView protocol to notify it whenever a node is inserted or removed from its active view [1].

Finally, the HyParView protocol is responsible for handling all TCP connections by using the CONNECT method and handling all `Close` notifications. The

[1]In fact, the tree strategy gossip protocol also makes use of the `GetPeer` call. This happens in order to support a broader set of membership protocols.

gossip strategy component is responsible to use the `Send` primitive and handle the `Receive`[1] callback of TCP to handle messages received at each node.

3.2 HyParView

3.2.1 Rationale

As stated in Chapter 1, one of the main motivations of this work is to obtain high values of reliability using a small fanout value (*i.e.* in the order of $\log(n)$, where n is the total number of nodes), while supporting high number of nodes failures, maintaining the level of broadcast reliability as high as possible.

There are two intuitive arguments that explain why a small fanout value does not offer high level of reliability in simple eager push gossip when using previous membership protocols:

1. If a small fanout is used, the random selection of nodes allows the existence of runs where some nodes in the system are never selected as gossip targets.

2. When using partial views instead of global membership information, there are (typically) no assurances that each node is known by the same amount of peers in the overlay (in other words, there are no assurances that every node in the system has the same in-degree).

Notice that the combination of the two phenomena is particularly negative, because nodes which are less popular in the system (*i.e.* which have a smaller in-degree) will have less probability of being selected as gossip targets and consequently will never receive some gossip messages, which in turn will affect the global reliability of the gossip protocol.

In order to solve the first problem, one might rely on a deterministic algorithm, that each node should apply in order to select gossip targets each time it broadcasts or relays a message. This algorithm should ensure that every node in the system is selected at least once, as a gossip target, by another node. Ideally

[1]To be precise, HyParView also uses the TCP layer to send and receive messages. For simplicity these interactions were not represented in Figure 3.4.

it should ensure that all nodes send the same number of messages (for load distribution and fairness) and that, in a stable environment (*e.g.* without any node failure or message omission), every node receives each message the same number of times.

The simplest deterministic algorithm consists in having each node to select all nodes in its partial view as gossip targets. This ensures that, if all nodes in the system have a in-degree value above 0, all nodes will be selected, at least once, as a gossip target, as long as the overlay is connected.

To allow the selection of all nodes in the partial view, the partial views size must be at most t, where t is the fanout value used by the above gossip protocol. This may be a problem when one wants to use a small fanout, as the fault tolerance level of the overlay produced by small partial views is considerably lower. For instance, a high percentage of nodes might easily became disconnected (*i.e.* with a in-degree equal to 0) in the presence of node failures.

To ensure that nodes do not became disconnected as a result of node failures in the overlay, each node must have knowledge of more peers than those in its partial view. This can be achieved if each node maintains a second, larger, partial view as a backup set of nodes. The size of the backup view can be set taking in account memory constraints and the desired level of fault tolerance, and it should be greater than $log(n)$ to ensure, with high probability, the connectivity of the overlay in faulty scenarios[1].

The above solution does not completely address the second problem, as there are still no guarantees that every node will have the same in-degree. Failure to satisfy this property has implications on the resilience of the gossip protocol in the presence of node failures or network omissions, making some nodes - the ones with smaller in-degree - more susceptible to be affected by these failures.

To improve in-degree distribution, and also allow each node to know and have some measure of direct control over its own in-degree value, one might use a symmetric membership. If all nodes in the system use partial views with the same size, then all nodes will, eventually, converge to the same in-degree value, as each node will try to fill its own partial view.

[1]See, for instance, the results published in Eugster *et al.* (2004).

When using symmetric partial views, nodes will always receive gossip messages from peers belonging to their local partial view. To allow the use of a fanout of t without sending the gossip message back to the same node from which the message was received for the first time - which is clearly a redundant message that will never result in a delivery - partial views should have a size of $t + 1$.

This model is compatible with the optimized interface procedure for a *peer sampling service* that was defined in section 2.2.1.

3.2.2 Algorithm

3.2.2.1 Overview

The *Hy*brid *Par*tial *View*, or simply, HyParView protocol maintains two distinct views at each node. A small active view of size fanout+1. A larger passive view, that ensures connectivity despite a large number of faults and must be larger than $log(n)$. Note that the overhead of the passive view is minimal, as no connections are kept open.

The active views of all nodes create an overlay that is used for message dissemination. Links in the overlay are symmetric. This means that if node q is in the active view of node p then node p is also in the active view of node q. This architecture assumes that nodes use a reliable transport protocol to broadcast messages in the overlay. In practice, this means that each node keeps an open TCP connection to every other node in its active view. This is feasible because the active view is very small, thus the extra overhead produced by TCP is not high enough to become a problem. When a node receives a message for the first time, it broadcasts the message to all nodes of its active view (except, obviously, to the node that has sent the message), this operation is equivalent to use a set of nodes as gossip targets obtained by calling the `getPeer`$(n, peer)$ method of the *peer sampling service*. Therefore, the gossip target selection is deterministic in the overlay. However, the overlay itself is created at random, using the gossip membership protocol described in this section.

A reactive strategy is used to maintain the active view. Nodes can be added to the active view when they join the system. Also, nodes are removed from the active view when they are suspected as failed, by leveraging on TCP as an unreliable failure detector. TCP is said to function as an unreliable failure detector because it can generate false positives (*e.g.* when the network becomes suddenly congested). Also the use of TCP simplifies the task of ensuring the symmetry property of active views.

The reader should notice that, as each node tests its entire active view every time it forwards a message. Therefore, the entire broadcast overlay is implicitly tested at every broadcast, which allows a very fast failure detection.

HyParView does not owns an explicit *leave mechanism*, because the overlay is able to react fast enough to node failures. Hence when a node wishes to leave the system, it can simply be treated as if the node has simply failed.

In addition to the active view, each node maintains a larger passive view. The passive view is not used for message dissemination. Instead, the goal of the passive view is to maintain a repository of nodes that can be used to replace failed members of the active view.

The passive view is maintained using a cyclic strategy. Periodically, each node performs a shuffle operation with one random node in the overlay in order to update its passive view.

One interesting aspect, of the shuffle mechanism of HyParView, is that the identifiers that are exchanged in a shuffle operation are not only from the passive view: a node also sends its own identifier and some nodes collected from its active view to its peer. Because there are stronger guarantees of the correctness of nodes in the active view than the passive view. By shuffling nodes from the active view there is a increase in the probability of having nodes that are correct in the passive views which also ensures that failed nodes are eventually expunged from all passive views. This will be further addressed later, in Section 3.2.2.4.

3.2.2.2 Join Mechanism

When a node wishes to join the overlay, it must know another node that already belongs to the overlay. That node is called the *contact node*. There are several ways to learn about the contact node, for instance, members of the overlay could be announced through a set of well known servers, however this is not in the scope of this thesis and so will not be further addressed here.

In order to join the overlay, a new node n establishes a TCP connection to the contact node c and sends to c a JOIN request. A node that receives a JOIN request will start by adding the new node to its active view, even if it has to drop a random node from it, in order to create a space in its active view. In this case a DISCONNECT notification is sent to the dropped node. The effect of the DISCONNECT message is described later in the Chapter and depicted in Algorithm 2.

The contact node c will then send to all other nodes in its active view a FORWARDJOIN request containing the new node identifier. The FORWARDJOIN request is then propagated in the overlay using a random walk. Associated to the join procedure, there are two configuration parameters, named *Active Random Walk Length*, that specifies the maximum number of hops a FORWARDJOIN request is propagated in the overlay, and *Passive Random Walk Length*, that specifies at which point in the walk the new node identifier is inserted in a passive view. To use these parameters, the FORWARDJOIN request carries a "time to live" field that is initially set to *Active Random Walk Length* and decreased at every hop.

When a node p receives a FORWARDJOIN, it performs the following steps in sequence:

1. If the time to live is equal to zero *or* if the number of nodes in p's active view is equal to one[1], it will add the new node to its active view. This step is performed even if a random node must be dropped from the active view and inserted into the passive view. In the later case, the node being ejected from the active view receives a DISCONNECT notification.

2. If the time to live is equal to *Passive Random Walk Length*, p will insert the new node into its passive view.

3. The time to live field is decremented.

[1]Considering that active views are symmetric, if p's active view only contains one node it must be the identifier of the node who sent the FORWARDJOIN to p, hence p is unable to further propagate the message on the overlay and should accept it.

Algorithm 1: Join mechanism

```
Data:
myself: the identifier of the local node
activeView: a node active partial view
passiveView: a node passive view
contactNode: a node already present in the overlay
newNode: the node joining the overlay
ARWL: Active random walk length
PRWL: Passive random walk length
upon init do
    Send(JOIN, contactNode, myself);

upon Receive(JOIN, newNode) do
    call addNodeActiveView(newNode)
    foreach $n \in$ activeView and $n \neq$ newNode do
        Send(FORWARDJOIN, $n$, newNode, ARWL, myself)

upon Receive(FORWARDJOIN, newNode, timeToLive, sender) do
    if timeToLive== 0||#activeView== 1 then
        call addNodeActiveView(newNode)
    else
        if timeToLive==PRWL then
            call addNodePassiveView(newNode)
        $n \longleftarrow n \in$ activeView and $n \neq$ sender
        Send(FORWARDJOIN, $n$, newNode, timeToLive-1, myself)
```

4. If, at this point, n has not been inserted in p's active view, p will forward the request to a random node in its active view (different from the one from which the request was received).

Algorithm 1 depicts the pseudo-code for the join operation.

3.2.2.3 Active View Management

The active view is managed using a reactive strategy. When a node p suspects that one of the nodes present in its active view has failed (by either disconnecting or blocking), it selects a random node q from its passive view and attempts to establish a TCP connection with q. If the connection fails to establish, node q is considered failed and removed from p's passive view; another node q' is selected at random from the passive view and a new attempt is made. The procedure is repeated until a TCP connection is established with success.

When the connection is established with success, p sends to q a NEIGHBOR request with its own identifier and a priority level. The priority level of the request may take two values, depending on the number of nodes present in the active view of p: if p has no elements in its active view the priority is *high*; the priority is *low* otherwise.

A node q that receives a high priority NEIGHBOR request will always accept the request, even if it has to drop a random member from its active view (again, the member that is dropped will receive a DISCONNECT notification and will be added to q's passive view). If a node q receives a low priority NEIGHBOR request, it will only accept the request if it has a free slot in its active view, otherwise it will refuse the request.

The rationale behind this priority values is simple, if a node p does not have any element in its active view it is disconnected from the overlay, meaning that he can not send nor receive any broadcast message. Because of this it has priority to establish a neighbor relation with q , even if some node n has to be dropped from the active view of q, as there are good changes that n might have some other nodes in its active view, meaning that it will not became disconnected from the overlay[1].

If the node q accepts the NEIGHBOR request, p will remove q's identifier from its passive view and add it to the active view. If q rejects the NEIGHBOR request, p will select a new node from its passive view and repeat the whole procedure.

3.2.2.4 Passive View Management

The passive view is maintained using a cyclic strategy. Periodically, each node perform a shuffle operation with one other node at random. The purpose of the shuffle operation is to update the passive views of the nodes involved in the exchange, and eventually expunge some failed nodes from it, increasing the passive view accuracy.

The node p that initiates the exchange creates an exchange list with the following contents: p's own identifier, k_a nodes from its active view and k_p nodes from its passive view (where k_a and k_p are protocol parameters). It then sends

[1]Even if a node q drops from the active view a node n, that only had q in its active view, n will always be able to rejoin the overlay. Notice that, in the worst case scenario, n will only have q at his passive view, because it received a DISCONNECT message from q, hence it will be able to contact q issuing a NEIGHBOR request with high priority which q will have to accept reconnecting n to the overlay.

the list in a SHUFFLE request to a random neighbor of its active view. SHUFFLE requests are propagated using a random walk and have an associated "time to live", just like the FORWARDJOIN requests (during all experiments executed so far with this protocol, the value of this "time to live" was configured with same value used in the *Passive Random Walk Length* parameter discussed in section 3.2.2.2).

A node q that receives a SHUFFLE request will first decrease its time to live. If the time to live of the message is greater than zero and the number of nodes in q's active view is greater than 1, the node will select a random node from its active view, different from the one he received this shuffle message from, and simply forwards the SHUFFLE request. Otherwise, node q accepts the SHUFFLE request and send back, using a temporary TCP connection, a SHUFFLEREPLY message that includes a list with a number of nodes selected at random from q's passive view, equal to the number of nodes received in the SHUFFLE request.

Then, both nodes integrate the elements they received in the SHUFFLE/ SHUFFLEREPLY message into their passive views (naturally, they exclude their own identifier and nodes that are part of the active or passive views). If the passive view is full, nodes have to remove other nodes to free space in order to include the received ones. Nodes attempt first to remove identifiers that they have sent to their peers and, if no such identifiers remains, they simply drop a random element from their passive view.

3.2.2.5 View Update Procedures

Algorithm 2 depicts some basic manipulation primitives used to change contents of the passive and active views. The important aspect to retain from these primitives, is that nodes can be moved from the passive view to the active view in order to assure a full active view, or in reaction to node failures. Nodes can also be moved from the active view to the passive view whenever a correct node has to be removed from the active view. Note that since links are symmetric, by removing a node p from the active view of node q, q creates a "empty slot" in p's active view. By adding p to its passive view, node q increases the probability of shuffling p with other nodes and, subsequently, having p be target of NEIGHBOR requests.

Algorithm 2: View manipulation primitives

```
Data:
activeView: a node active partial view
passiveView: a node passive view

procedure dropRandomElementFromActiveView do
    $n \longleftarrow n \in$ activeView
    Send(DISCONNECT, $n$, myself)
    activeView $\longleftarrow$ activeView $\setminus \{n\}$
    passiveView $\longleftarrow$ passiveView $\cup \{n\}$

procedure addNodeActiveView(node) do
    if node $\neq$ myself and node $\notin$ activeView then
        if isfull(activeView) then
            call dropRandomElementFromActiveView
        activeView $\longleftarrow$ activeView $\cup$ node

procedure addNodePassiveView(node) do
    if node $\neq$ myself and node $\notin$ activeView and node $\notin$ passiveView then
        if isfull(passiveView) then
            $n \longleftarrow n \in$ passiveView
            passiveView $\longleftarrow$ passiveView $\setminus \{n\}$
        passiveView $\longleftarrow$ passiveView $\cup$ node

upon Receive(DISCONNECT, peer) do
    if peer $\in$ activeView then
        activeView $\longleftarrow$ activeView $\setminus$ {peer}
        call addNodePassiveView(peer)
```

3.2.2.6 Interaction With TCP Flow Control

The use of TCP could cause the whole system to block in the presence of slow nodes. This happens because any node that is slow in consuming messages, will force its neighbors to block due to the flow control mechanisms of TCP (Stevens, 1997).

All nodes that became blocked will, in turn, became unable to consume the messages they receive. Consequently, all their neighbors will also block when trying to send messages to them and, eventually, this effect will spread to all nodes in the overlay in an epidemic manner.

To avoid this phenomenon, one can rely in a variation of the technique proposed in Pereira *et al.* (2003). The technique works as follows:

All nodes would buffer, at the application layer, the messages to be sent to other nodes, in dedicated buffers. An independent buffer is maintained for each neighbor in the active view. Furthermore, TCP is invoked using non-blocking primitives. When a application buffer for a given neighbor becomes congested, two different approaches can be employed:

1. The slow neighbor is expelled from the active view, without being inserted on the passive view.

2. The node will drop some selected messages from the buffer. This selection can be based upon any of the purging strategies presented in Pereira *et al.* (2003).

With both approaches, the blocking of the entire overlay is avoided.

3.3 Eager Push Strategy

3.3.1 Rationale

The idea behind the eager push strategy is simply to flood broadcast messages through the overlay network. This ensures that all nodes will receive broadcast messages as long as the membership protocol is able to maintain the overlay connected.

This strategy is only viable because HyParView produces an active view that has a small degree[1]. The degree of the overlay will determine the relative message redundancy of the protocol in stable environment, for an instance, if the overlay has a degree of 5, the fanout of the protocol will be 4 (because the overlay has symmetric links), hence the relative message redundancy expected by this gossip strategy in a stable environment will be a value close to 3.

The combination of flooding with some amount of message redundancy allows the protocol to completely mask failures of nodes better than other existing gossip approaches (*e.g.* maintaining a constant reliability of 100%) for massive node failures as high as 20%.

Furthermore, this strategy ensures that the max hop of delivery is as low as the overlay diameter allows. Because all links between the nodes are used, it ensures that all shortest path between nodes are used to disseminate messages (independently of the sender).

[1]In fact, and as it was hinted in section 3.2.1, HyParView was originally designed to support this specific strategy.

Another point that favors this strategy is it simplicity. It is based on a pure eager push gossip approach, hence it does not have to buffer messages it delivers and, as it is topology independent, it does not require the maintenance of complex state related with its neighbors.

3.3.2 Algorithm

This strategy is implemented by a simple eager push gossip protocol and is depicted in Algorithm 3.

Algorithm 3: Eager push protocol

```
Data:
myself: the identifier of the local node
receivedMsgs: a list of received messages identifiers
f: the fanout value

upon event Broadcast(m) do
    mID ⟵ hash(m + myself)
    peerList ⟵ getPeer(f,null)
    foreach p ∈ peerList do
        trigger Send(GOSSIP, p, m, mID, myself)
    trigger Deliver(m)
    receivedMsgs ⟵ receivedMsgs ∪ {mID}

upon event Receive(GOSSIP, m, mID, sender) do
    if mID ∉ receivedMsgs then
        receivedMsgs ⟵ receivedMsgs ∪ {mID}
        trigger Deliver(m)
        peerList ⟵ getPeer(f,sender)
        foreach p ∈ peerList do
            trigger Send(GOSSIP, p, m, mID, myself)
```

This algorithm ensures that all links in the overlay are used at least once by leveraging on the semantics of the `getPeer()` call of the HyParView protocol.

The reader should notice that the algorithm can work in the "Infect and Die" model (Eugster *et al.*, 2004) as it only relays messages to other nodes upon the reception of each gossip message for the first time.

3.4 Tree Strategy

3.4.1 Rationale

The eager push strategy presented above allows to obtain a high reliability while ensuring the smallest possible value of last delivery hop. Unfortunately in stable environment it still produces a significant RMR (relative message redundancy) value[1]. An intuitive approach that could help to mitigate this is is to use a structured overlay that establishes a multicast tree covering all nodes in the system. To achieve this, we created a new gossip protocol that was named *p*ush-*l*azy-p*u*sh *m*ulticast *tree* or simply *Plumtree*.

Plumtree has two main components, each one answers a specific challenge of a fault-tolerance broadcast scheme which employs spanning trees. These can be defined as follows:

Tree construction This component is in charge of selecting which links of the random overlay network will be used to forward the message payload using an eager push strategy. We aim at a tree construction mechanisms that is as simple as possible, with minimal overhead in terms of control messages.

Tree repair This component is in charge of repairing the tree when failures occur. The process should ensure that, despite failures, all nodes remain covered by the spanning tree. therefore, it should be able to detect and heal partitions of the tree. The overhead imposed by this operation should also be as low as possible.

Several broadcast applications only need to have one sender while supporting a large number of receivers. This is the case of *news dissemination services*, where a news source wants to provide information to a set of users, *software update systems* where a software provider wants to push new software releases into a large set of stations or *live video broadcast* where a source is sending a streaming of video to a set of viewers.

[1]The reader should notice that the RMR value is still lower than those obtained with other protocols that must use higher values of fanout to ensure a (probabilistic) high level of node coverage.

Because of this, several applications require a single spanning tree optimized to deliver messages from one specific source node. The source node should, evidently, be the root of the spanning tree.

Several application-level multicast protocols (*e.g.* Zhuang *et al.* (2001) or Liang *et al.* (2005)) build a tree by flooding the network with a special message that is sent from a content distribution node, *e.g.* a TREE message. Whenever this message is sent through a link, that link is marked by the sender as being a *branch* on the multicast tree. Special PRUNE messages are then used to remove redundant branches from the multicast tree. A similar strategy can be used to create a spanning tree on top of the random overlay maintained by the active views of HyParView, this would work as follows:

The eager push algorithm presented in the last section already floods a (somewhat) stable random overlay, using the active view of the HyParView protocol. Gossip messages are sent through all links in the overlay. An intuitive remark is that, in stable conditions[1], messages that generate a delivery to the above application layer (*i.e.* that are received at a node for the first time) which are sent by the same source node, usually are received by nodes through the same overlay link (*i.e.* from the same neighbor). Together these links form a spanning tree that connects all nodes to a given source node (or root). All other links in the overlay are redundant, and are only required to cover for node failures, therefore redundant links can be *pruned* (removed) from the overlay as long as no node failures happen.

The basic idea behind the Plumtree protocol presented here, comes from this simple concept. The operation of Plumtree combines the basic flooding process with a *prune* process. Some links between nodes are marked as being part of the broadcast tree and payload messages are only sent to those links (neighbors). Initially all links in a random (connected) overlay are considered as being part of the broadcast tree. Then whenever a message is received for the second time a PRUNE message is used to remove the link, used to transmit the redundant message, from the tree.

[1]Stable conditions in this context concerns not only no changes in the membership protocol, but also a network which presents a low variance.

Although this covers the first operation presented before, it does not addresses the second one. Using this approach, a single node failure is able to partition the spanning tree, disconnecting a large set of nodes from the source.

To solve the challenge of repairing the spanning tree a lazy push gossip strategy is employed. This strategy will enable nodes that do not receive some messages, because they have become disconnected from the sender, to retrieve those messages from neighbors whom received them and, at the same time, to add new links to the spanning tree therefore, reconnecting themselves to the sender. This simple operation enables the whole spanning tree to be repaired. To support this process, nodes also announce messages they receive through the links of the overlay that are not part of the broadcast tree by sending IHave messages. Whenever a node requests a message from a neighbor, by sending a Graft message, the link between those nodes becomes a branch of the spanning tree.

The spanning tree is constructed with a node serving as root, hence it is optimized, at least in terms of last delivery hop (latency), to messages that are sent by that node. But considering that the links on the overlay are symmetric, any node can use the same tree structure to broadcast his own message, although this will result in sub-optimal routing, in terms of last delivery hop.

3.4.2 Algorithm

3.4.2.1 Overview

Briefly, the Plumtree protocol has the following relevant aspects:

- It constructs a spanning tree on top of HyParView[1] that is optimized for systems with a single sender. Nevertheless, it can be used to broadcast messages from any node on the system.

- The construction of the tree is based on the combination of an eager push algorithm and a pruning process. Because the paths in the tree are selected using messages that are sent from a given root node, it can be said that the tree construction uses a source-based strategy.

[1]Although Plumtree was developed to leverage on the properties of HyParView, it is not limited to the use of this peer sampling service. This is discussed further ahead in the thesis.

Algorithm 4: Internal data structure

```
Data:
myself: the identifier of the local node
receivedMsgs: a list of received messages identifiers
f: the push fanout value
eagerPushPeers: a list of neighbors whom links form the spanning tree
lazyPushPeers: a list of neighbors whom links does not belong to the spanning tree
lazyQueue: a list of tuples {mID,node,round}
```

- The repair of the tree is based in a lazy push gossip approach. In addition to forwarding the payload through the links that form the spanning tree, nodes also send IHave messages on the other links. Whenever a node requests a message he has missed from a neighbor, a new branch is added to the tree. Because the repair process is controlled by the receiver, it can be said that the tree repairing uses a receiver-based strategy.

- Several IHave announcements may be aggregated into a single control message to avoid excessive control traffic in the network. This can be achieved by applying a scheduling policy that can be designed by taking into consideration application specific requirements.

3.4.2.2 Additional Data Structures

The Plumtree protocol has to maintain a more complex internal state than the eager push gossip strategy. This is partially due to its nature being a topology aware gossip strategy, it has to keep information concerning its neighbors. Additionally, due to the use of an hybrid eager push/lazy push gossip approach, it has to keep track of information concerning all IHave messages received, as well as internal timers to trigger the request of missed messages from its neighbors.

Algorithm 4 shows the required data structures kept by the Plumtree protocol. The *eagerPushPeers* and *lazyPushPeers* are sets that maintain information concerning the node's neighbors. A nodes neighbor must be, and can only be, at one of these sets. Hence these sets have the following properties:

$$eagerPushPeers \cap lazyPushPeers = \emptyset$$

$$eagerPushPeers \cup lazyPushPeers = active\ view$$

The *lazyQueue* set contains a list of received IHave messages. For each IHave message, there is information stored concerning its sender, the advertised gossip message identifier (*mID*) and the round value, which gives an indication of the distance, in hops, to the source of the gossip message.

For simplicity, the set that contains received messages to support the recovery mechanism and the specific method to clean up that set have been omitted from the algorithms that will be presented next.

3.4.2.3 Peer Sampling Service And Initialization

Although Plumtree design was motivated by the special characteristics of the HyParView protocol, it can be used with other membership protocols that also create and maintain a random overlay network. However the overlay network maintained by these peer sampling services should present some essential properties that must be ensured at all times. Those properties can be described as follows.

Connectivity: The overlay should be connected, despite failures that might occur. This has two implications. Firstly, all nodes should have in their partial views, at least, another correct node. Secondly, all nodes should be in the partial view of, at least, a correct node.

Scalable: The Plumtree protocol is aimed toward the support of large distributed applications. Therefore, the peer sampling service should be able to operate correctly in such large systems (*e.g.* with more than 10.000 nodes).

Reactive membership: The stability of the spanning tree structure depends on the stability of the partial views maintained by the peer sampling service. When a node is added or removed to the partial view of a given node, it might produce changes in the links used for the spanning tree. This changes may not be desirable hence, the peer sampling service should employ a reactive strategy that maintains the same elements in partial views when operating in steady-state.

In addition to these properties, that are fundamental to the correct operation of Plumtree, the peer sampling service may also exhibit a set of other desirable properties, in the sense that they improve the operation of the protocol. One such property is to maintain symmetric partial views. If the links that form the spanning tree are symmetric, then the tree may be shared by multiple sources. Symmetric partial views render the task of creating bidirectional trees easier, and reduce the amount of peers that each node has to maintain.

To support a larger group of peer sampling service, in the initialization step of Plumtree, we use the `getPeer()` generic interface of the service to obtain a sample of, at most, f neighbors (where f is the eager push fanout value) that are used to initialize the *eagerPushPeers* set. The reader should notice that this is not expected from a gossip strategy that is topology aware and also it is not required when using a membership service that maintains such a small partial view as the HyParView protocol. However this might be useful when using membership protocols that maintain larger passive views, to limit the number of eager push neighbors each node has to maintain. Another aspect of this, is that with these protocols, the coverage of the spanning tree will be probabilistic, and dependent of the fanout value select, as specified in Eugster *et al.* (2004).

3.4.2.4 Tree Construction Process

After the initialization of the *eagerPushPeers* set described above, nodes construct the spanning tree by moving neighbors from *eagerPushPeers* to *lazyPushPeers*, in such a way that, after the protocol evolves, the overlay defined by the first set becomes a tree. When a node receives a message from the first time it includes the sender in the set of *eagerPushPeers* (Algorithm 5, lines: 24–33). This ensures that the link from the sender to the node is bidirectional and belongs to the broadcast tree. When a duplicate is received, its sender is moved to the *lazyPushPeers* (Algorithm 5, lines: 34–37). Furthermore, a PRUNE message is sent to that sender such that, in response, it also moves the link to the *lazyPushPeers* (Algorithm 5, lines: 38–40). This procedure ensures that, when the first broadcast is terminated, a tree has been created.

Algorithm 5: Spanning tree construction algorithm

```
procedure dispatch do
    announcements ⟵ policy (lazyQueue) //set of IHAVE messages
    trigger Send(announcements)
    lazyQueue ⟵ lazyQueue \announcements

procedure EagerPush (m, mID, round, sender) do
    foreach p ∈ eagerPushPeers: p ≠sender do
        trigger Send(GOSSIP, p, m, mID, round, myself)

procedure LazyPush (m, mID, round, sender) do
    foreach p ∈ lazyPushPeers: p ≠sender do
        lazyQueue ⟵ (textscIHave(p, m, mID, round, myself)
    call dispatch()

upon event Init do
    eagerPushPeers ⟵ getPeer(f)
    lazyPushPeers ⟵ ∅
    lazyQueue ⟵ ∅
    missing ⟵ ∅
    receivedMsgs ⟵ ∅

upon event Broadcast(m) do
    mID ⟵ hash(m+myself)
    call EagerPush (m, mID, 0, myself)
    call lazyPush (m, mID, 0, myself)
    trigger Deliver(m)
    receivedMsgs ⟵ receivedMsgs ∪ {mID}

upon event Receive(GOSSIP, m, mID, round, sender) do
    if mID ∉ receivedMsgs then
        trigger Deliver(m)
        receivedMsgs ⟵ receivedMsgs ∪ {mID}
        if ∃ (id,node,r) ∈ missing :id=mID then
            cancel Timer(mID)
        call EagerPush (m, mID, round+1, myself)
        call lazyPush (m, mID, round+1, myself)
        eagerPushPeers ⟵ eagerPushPeers ∪ {sender}
        lazyPushPeers ⟵ lazyPushPeers \ {sender}
        call Optimize (m, mID, round, sender) // optional
    else
        eagerPushPeers ⟵ eagerPushPeers \ {sender}
        lazyPushPeers ⟵ lazyPushPeers ∪ {sender}
        trigger Send(PRUNE, sender, myself)

upon event Receive(PRUNE, sender) do
        eagerPushPeers ⟵ eagerPushPeers \ {sender}
        lazyPushPeers ⟵ lazyPushPeers ∪ {sender}
```

One interesting aspect of this process is that, assuming a stable network (*i.e.* with constant load), it will tend to generate a spanning tree that minimizes the message latency (as it only keeps the path that generates the first message reception at each node).

As soon as nodes are added to the *lazyPushPeers* set, messages start being propagated using both eager and lazy push. Lazy push is implemented by sending IHave messages, that only contain the broadcast message ID, to all *lazyPushPeers* (Algorithm 5, lines: 5–7). Note however that, to reduce the amount of control traffic, IHave messages do not need to be sent immediately. A scheduling policy is used to piggyback multiple IHave announcements in a single control message. The only requirement for the scheduling policy for IHave messages is that every IHave message is eventually scheduled for transmission.

3.4.2.5 Announcement Policy

In the evaluation of Plumtree (which will be presented in the following chapter), and for the sake of simplicity of the experimental model, the announcement policy employed was the simplest one. This policy selects all pending IHave messages in the *lazyQueue* whenever the `Dispatch` procedure of the protocol is called (Algorithm 5, line 11) and immediately send them. This does not take any advantage of aggregating these messages. On the other hand this strategy allows to minimize the latency of the protocol.

3.4.2.6 Fault Tolerance And Tree Repair

The tree repair process is based on a lazy push gossip strategy.

When a failure occurs, at least one tree branch is affected. Therefore, eager push is not enough to ensure message delivered in face of failures. The lazy push messages exchanged through the remaining nodes of the gossip overlay are used both to recover missing messages but also to provide a quick mechanisms to heal the multicast tree.

When a node receives a IHave message, it simply marks the corresponding message as missing (Algorithm 6, lines: 1–15). It then starts a timer, with a predefined timeout value, and waits for the missing message to be received via

Algorithm 6: Spanning tree repair algorithm

```
upon event Receive(IHave, mID, round, sender) do
    if mID ∉ receivedMsgs do
        if ∄ Timer(id): id=mID do
            setup Timer(mID, timeout$_1$)
        missing ⟵ missing ∪ {(mID,sender,round)}

upon event Timer(mID) do
    setup Timer(mID, timeout$_2$)
    (mID,node,round) ⟵ removeFirstAnnouncement(missing, mID)
    eagerPushPeers ⟵ eagerPushPeers ∪ {node}
    lazyPushPeers ⟵ lazyPushPeers \ {node}
    trigger Send(Graft,node,mID,round,myself)

upon event Receive(Graft, mID, round, sender) do
    eagerPushPeers ⟵ eagerPushPeers ∪ {sender}
    lazyPushPeers ⟵ lazyPushPeers \ {sender}
    if mID ∈ receivedMsgs do
        trigger Send(Gossip, sender, m, mID, round, myself)
```

eager push before the timer expires. The timeout value is a protocol parameter that should be configured considering the diameter of the overlay and a target maximum recovery latency, defined by the application requirements.

When the timer expires at a given node, that node selects the first IHave announcement it has received for the missing message. It then sends a Graft message to the source of that IHave announcement (Algorithm 6, lines: 6–11). The Graft message has a dual purpose. In first place, it triggers the transmission of the missing message payload. In second place, it adds the corresponding link to the broadcast tree, healing it (Algorithm 6, lines: 12–16). The reader should notice that when a Graft message is sent, another timer is started to expire after a certain timeout, to ensure that the message will be requested to another neighbor if it is not received meanwhile. This second timeout value should be smaller that the first, in the order of an average round trip time to a neighbor.

Note that several nodes may become disconnected due to a single failure, hence it is possible that several nodes will try to heal the spanning tree degenerating into a structure that has cycles. This is not a problem however, as the natural process used to build the tree will remove any redundant branches produced during this process by sending Prune messages (*i.e.*, when a message is received by a node more than once).

Algorithm 7: Overlay network change handlers

```
upon event NeighborDown(node) do
    eagerPushPeers ⟵ eagerPushPeers \ {node}
    lazyPushPeers ⟵ lazyPushPeers \ {node}
    foreach (i,n,r) ∈missing:n=node do
        missing ⟵ missing \ {(i,n,r)}

upon event NeighborUp(node) do
    eagerPushPeers ⟵ eagerPushPeers ∪ {node}
```

3.4.2.7 Dynamic Membership

We now describe how Plumtree reacts to changes in the gossip overlay. These changes are notified by the peer sampling service using the `NeighborDown` and `NeighborUp` primitives. When a neighbor is detected to leave the overlay, it is simple removed from either the *eagerPushPeers* set or the *lazyPushPeers* set. Furthermore, the record of IHAVE messages sent from failed members is deleted from the missing history (Algorithm 7, lines: 1–5). When a new member is detected, it is simply added to the set of eagerPushPeers, *i.e.*, it is considered as a candidate to become part of the spanning tree (Algorithm 7, lines: 6–7).

An interesting aspect of the repair process is that, when "sub-trees" are generated, due to changes on the global membership, it is only required that one of the disconnected nodes receive an IHAVE message, to reconnect all those nodes to the root node (repairing the whole spanning tree). This is enough to heal the spanning tree as long as only a reduced number of nodes fail, generating disconnected "sub-trees". When larger numbers of nodes fail it is more probable to have single nodes isolated from the tree. In such scenarios the time required to repair the tree might be too large. To speedup the healing process, we take benefit of the healing properties of the peer sampling service. As soon has the peer sampling service integrates a disconnected node in the partial view of another member, it generates a *NeighborUp* notification. This notification immediately puts back the disconnected member in the broadcast tree.

3.4.2.8 Sender-Based Versus Shared Trees

The tree built by Plumtree is optimized for a specific sender: the source of the first broadcast that is used to move nodes from the *eagerPushPeers* set to the *lazyPushPeers* set. In a network with multiple senders, Plumtree can be used in two distinct manners.

- For optimal latency, a distinct instance of Plumtree may be used for each different sender. This however, requires an instance of the Plumtree state to be maintained for each sender-based tree, with the associated memory and signaling overhead.
- Alternatively, a single shared Plumtree instance may be used for multiple senders. Clearly, the last delivery hop value may be sub-optimal for all senders except the one whose original broadcast created the tree. On the other hand, a single instance of the Plumtree protocols needs to be executed.

Later, in the next Chapter, results will be depicted that shows the Plumtree performance for a single sender and for multiple senders using a shared tree, this will allow the reader to better assess the trade-offs involved.

3.4.3 Optimization

The spanning tree produced by the algorithm is mainly defined by the path followed by the first broadcast message exchanged in the system. Therefore, the protocol does not take advantage of eventual new, and best, paths that can appear in the overlay, as a result of the addition of new nodes/links. Moreover, the repair process is influenced by the policy used to scheduled IHave messages. This two factors may have a negative impact in the Last Delivery Hop value exhibit by the algorithm as the system evolves.

3.4.3.1 Rationale

The main goal of the optimization is to allow nodes to change their upstream neighbors in order to lower the distance between them and the sender node (or sender nodes).

Algorithm 8: Optimization

```
procedure Optimization(mID, round, sender) do
    if $\exists$ (id,node,r) $\in$ missing: id=mID then
        if $r < round \land round - r >= threshold$ then
            trigger Send(GRAFT, node, null, r, myself)
            trigger Send(PRUNE, sender, myself)
            eagerPushPeers $\longleftarrow$ eagerPushPeers $\setminus$ {sender}
            lazyPushPeers $\longleftarrow$ lazyPushPeers $\cup$ {sender}
            lazyPushPeers $\longleftarrow$ lazyPushPeers $\setminus$ {node}
            eagerPushPeers $\longleftarrow$ eagerPushPeers $\cup$ {node}
```

An intuitive approach to evaluate the relative distance, in terms of hops in the random overlay, is to compare the round value in Gossip and IHave messages that a given node receives from neighbors. The round number is incremented in one unit each time a message is further propagated in the overlay, meaning that this value is an accurate measure of number of hops to the original sender of the message.

The key idea is to compare the distance of two neighbors to the sender of a given gossip message. If the round value is significantly lower in the IHave message, the node should perform an action to exchange the senders of those messages in its local view of the spanning tree. This is achieved by removing the sender of the Gossip message from the *eagerPushPeers* set and replace it with the sender of the IHave message.

Notice that the round value should be significantly lower and not simply lower. This is motivated by the following two reasons:

1. Nodes that are closer to the sender would, eventually, become overloaded with neighbors establishing branches with them. Because of this the load of relaying messages would be too much concentrated in those nodes.

2. It could be difficult to have a stable spanning tree structure. Stability would be impossible to obtain in scenarios with multiple senders, as nodes would be constantly changing their upstream members to minimize the distance between them and the sender of the last broadcast message.

3.4.3.2 Algorithm

Algorithm 8 depicts the optimization procedure developed to overcome the limitations explained above. The optimization requires a new parameter called *threshold* which is the minimum difference (in terms of number of hops) between a given payload message and any IHave message received concerning that same payload message, received by the same node, in order to trigger the optimization behavior.

The procedure in itself is very simple. A node that triggers a optimization will simply send 2 messages. Firstly it will send a Graft message to the sender of the IHave message, in order to establish the link to that element as a link of the spanning tree. The reader should notice that the identifier of the payload message in this request is set to *null*, this happen because the node which performs the optimization already has the payload message, and it notifies the receiver of the request, that no transmission of any payload is required. Secondly it will send a Prune message to the sender of the original payload message, to remove its link from the spanning tree. The node also updates its *eagerPushPeers* and *lazyPushPeers* sets to reflect the change in the spanning tree structure.

This strategy will ensure that the number of links on the spanning tree is constant and that all nodes remain connected. It only try to select links for the spanning tree which are closer to the sender of the last broadcast message. This optimization is also very important in systems where each participant broadcasts messages in the overlay in bursts. As it allows the spanning tree to optimize itself for each sender in turn. Results that show this will be presented in Chapter 4.

3.5 Summary

This chapter presented a generic architecture for a gossip-based system and depicted the components developed and how they fit into that architecture. The HyParView protocol, and two distinct gossip strategies, a eager push strategy and a tree strategy, were introduced. The introduction of these protocols began with the intuition behind their design followed by their detailed specification, witch was presented with pseudo code when convenient. In the following chapter

HyParView and both gossip strategies are going to be evaluated through simulation.

Chapter 4

Evaluation

This chapter presents the experimental evaluation of the protocols proposed in the previous chapter, namely: *i*) the HyParView membership protocol, *ii*) the eager push gossip protocol and; *iii*) the tree based gossip protocol. It begins by describing the experimental setting used in the evaluation, followed by the configuration parameters used for each protocol. Then we present and discuss simulation results which allow to evaluate the performance of the protocols in different scenarios of execution.

4.1 Experimental Setting

All simulations were conducted using the PeerSim Simulator[1]. In order to get comparative figures, both HyParView, Cyclon and Scamp were implemented in this simulator. In order to validate the implementations of Cyclon and Scamp, results obtained with the PeerSim Simulator were compared with published results for these systems (these simulations are omitted from the thesis, as they do not add to assess the merit to this work).

We have also implemented a modified version of Cyclon, that we named CyclonAcked. This version adds a failure detection system to Cyclon, based on the exchange of explicitly acknowledgments during message dissemination. Thus, CyclonAcked is able to detect a failed node when it attempts to gossip to it and,

[1]Available at: http://peersim.sourceforge.net/

therefore, is able to remove failed members from partial views, increasing the accuracy of these views. This benchmark is used to show that the benefits of HyParViews approach are not derived only from the use of a reliable transport (as an unreliable failure detector), but also from the clever use of two separate partial views.

An eager push gossip broadcast protocol for PeerSim was also implemented. This gossip protocol is able to use any of the membership protocols referred above as a *peer sampling service* (by means of the `GetPeer()` method). It operates in unlimited gossip mode, such that the global reliability is not affected by the configuration of the *maximum rounds* parameter (as shown in section 2.1.2).

Our eager gossip broadcast protocol was configured so that when combined with HyParView it implements the flood gossip strategy described in Section 3.3. For fairness, the same configuration parameter was used with all membership protocols.

Implementations of the basic Plumtree protocol, and its optimized version, were also developed for the PeerSim simulator. These implementations use HyParView as the underlying membership protocol, as they were designed to leverage on its properties, such as the symmetry and stability of the active view.

In all simulations, the overlay was created by having nodes join the network one by one, without running any membership rounds in between. Cyclon was initiated by having a single node to serve as contact point for all join requests. Scamp was initiated by using a random node already in the overlay as the contact point. These are the configurations that provide the best results for each of these protocols. HyParView achieves similar results with either method (the simulations use the same procedure as in Cyclon).

All simulations conducted in the PeerSim simulator used its cycle based engine. Each simulation is composed of a sequence of cycles, which begin at cycle 0, and have, at most, the following steps:

Failure: In this step, some number (or a percentage) of nodes are marked as failed (*e.g* their internal state is set to *Down*). This step might not be required for all simulations and it usually is only executed at a predefined cycle (or cycles) of the simulation.

Broadcast: In this step a number of nodes[1] send a broadcast message. The step is only terminated when there are no more messages in transit in the overlay (either gossip messages or any gossip strategy specific control messages).

Data retrieval: In this step, performance information is retrieved by inspecting the internal state of protocols and components of all active nodes.

Membership: In this step, the membership protocol executes any *cyclic* step. It is also in this step that any membership protocol that uses TCP becomes aware of failures that might have happened in the last *failure* step.

Clean up: All data stored on nodes concerning the broadcast of messages or any information concerning the state of the overlay in the current cycle is erased (*e.g.* temporary state that is maintained at nodes or specific components of the simulation is deleted).

The reader should notice that, although most simulations follow this structure, some of them are conducted without performing all these steps. For instance, when testing the behavior of a protocol in a stable environment (*i.e.* without the presence of node failures), the failure step is not required. Also, when the goal of the simulation is to evaluate properties of the overlay network, the broadcast step does not happen.

4.2 Experimental Parameters

All experiments were conducted in a network of 10.000 nodes and results show an aggregation from 3 independent runs of each experiment. Furthermore, membership protocols and gossip based broadcast protocols were configured as follows:

- In HyParView, the active membership size was set to 5, and passive membership's size to 30. *Active Random Walk Length* parameter was set to 6 and the *Passive Random Walk Length* was set to 3. In each shuffle message, $k_p = 4$ elements (at most) were sent from the passive view, while $k_a = 3$

[1]The nodes that send broadcast messages can be selected at random or be predefined.

elements (at most) were sent from the active view. The total size of shuffle messages is 8, as nodes also send their own identifier in each shuffle message.

- Cyclon protocol was configured with partial views of 35 elements (this is the sum of HyParView's active and passive view sizes). Shuffle message lengths were set to 14 and the time to live to random walks in the overlay was configured to 5.
- Scamp was configured with parameter c - the parameter that is related with fault tolerance of the protocol - to 4. The reason behind the selected value to this parameter was because it generated partial views which size's where distributed around a middle point of 34, which is as near as we could be from the value used in other protocols.
- The eager push gossip broadcast protocol was configured with a fanout value of 4. The reader should notice that, when combined with HyParView which was configured with active views size of 5, the protocol implements a flood gossip strategy.
- Both the basic Plumtree protocol and its optimization protocols were also configured with the fanout value set to 4.
- The optimized Plumtree protocol was configured with a threshold value of 7. This value was selected because it is close to the last delivery hop observed when using HyParView with the flood gossip strategy. Preliminary experimental work also showed that this value provided the best results[1].

4.3 HyParView And Eager Push Strategy

4.3.1 Graph Properties

As noted in Section 2.2.4, the overlays produced by membership protocols should exhibit some good properties such as low clustering coefficient, small average

[1]These experiments are not shown in the thesis as they are not relevant to the main goals of this work.

shortest path, and balanced in-degree distribution, to allow a fast message dissemination and a high level of fault tolerance. In this section it is shown how the simulated protocols perform regarding these metrics.

	Average clustering coefficient	Average shortest path	Last delivery hop
Cyclon	0.006836	2.60426	10.6
Scamp	0.022476	3.35398	14.1
HyParView	0.000920	6.38542	9.0

Table 4.1: Graph properties after stabilization

Table 4.1 shows values to average clustering coefficient and average shortest path for all protocols after a period of stabilization of 50 membership cycles[1]. It can be seen that in terms of average clustering coefficient, HyParView achieves significantly lower values than Scamp or Cyclon, which is expected considering that HyParView's active view is much smaller than other protocols partial views. Nevertheless, this is an important factor, that explains the high resilience that HyParView exhibits to node failures.

In terms of average shortest path, it is clear that HyParView falls behind Scamp and Cyclon. This is no surprise, as HyParView only maintain a smaller active view, which limits the number of distinct paths that exist across all nodes. Fortunately, this has no impact on the latency of the gossip protocol. The small level of global clustering, and the fact that all existing paths between nodes are used to disseminate every message, makes a HyParView based gossip protocol to deliver messages with a smaller number of hops than the other protocols. This is depicted in Table 4.1, on the last delivery hop column.

Figure 4.1 shows the in-degree distribution of all nodes in the overlay after the same stabilization period. Cyclon and Scamp have an in-degree distribution across a wide range of values, which means that some nodes are extremely popular on the overlay, while other nodes are almost totally unknown. As stated before, because of this distribution, some nodes on the overlay have greater probability

[1]In fact, this stabilization time is not required by Scamp, as it stabilizes immediately after the join period, HyParViews active view also stabilizes immediately but its passive view requires some rounds of membership to stabilize completely.

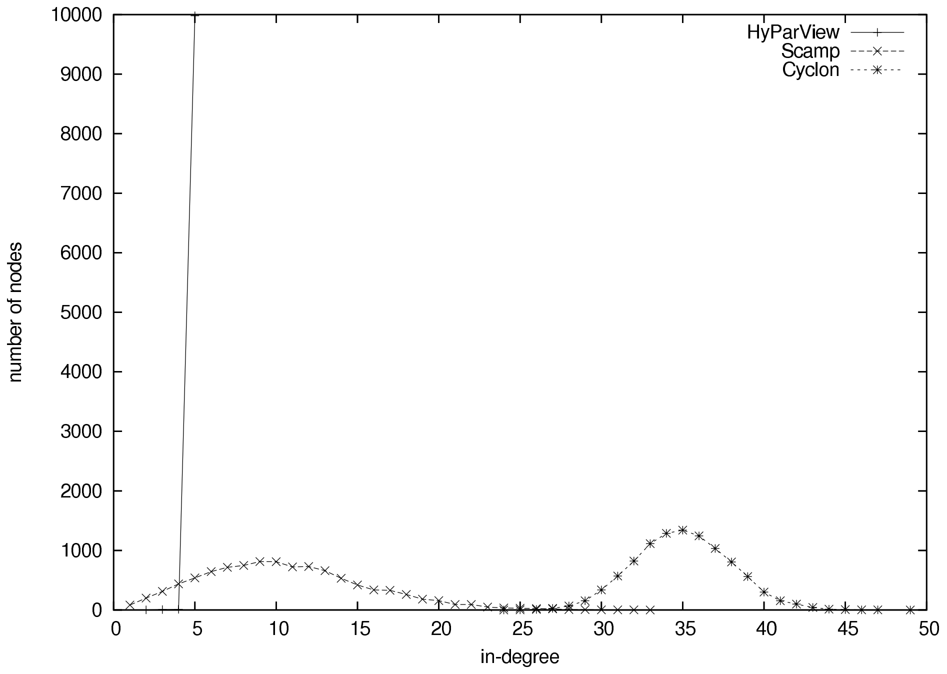


Figure 4.1: In-degree distribution

to receive redundant messages, while other nodes have a very small probability to receive messages even once. Notice that, as some nodes are known by few other neighbors, they have a smaller probability to be selected as gossip targets. Also these nodes have an increased probability to become disconnected from the overlay, as the number of nodes that are required to fail in order to disconnect the network is smaller. This is specially obvious in Scamp, where some nodes are only known by one other node.

Due to HyParViews symmetric active view, almost all nodes in the overlay are known by the maximum amount of peers possible, which is the active view length (5). This means that all nodes, with high probability, will receive each message exactly the same amount of times. Also, there is little probability for any node not to receive a message at least once. Finally, notice that nodes who have the smallest in-degree have at least 2 neighbors, and that the number of nodes in these conditions is marginal (with only 1 or 2 nodes).

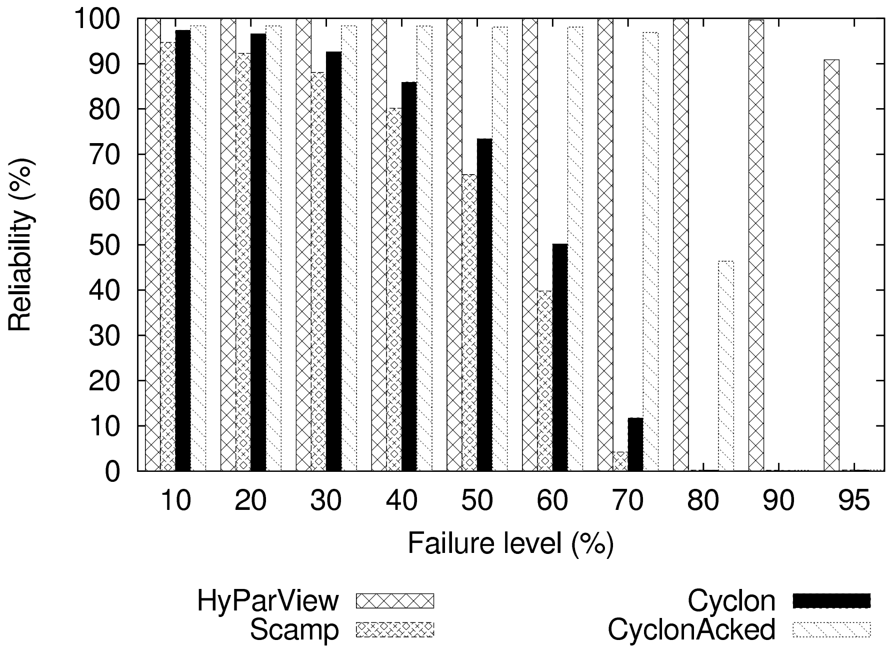


Figure 4.2: Average reliability for 1000 messages

4.3.2 Effect Of Failures

In this section, we evaluate the impact of massive failures in the reliability of the eager gossip message broadcast, when different membership protocols are used.

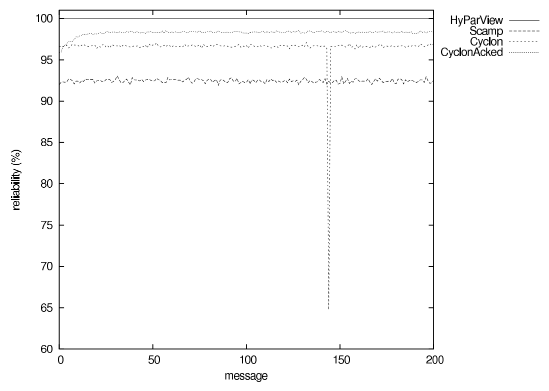
In each experiment, all nodes join the overlay after which they execute 50 cycles of membership protocol to guarantee stabilization. After the stabilization period, failures at random are induced in a percentage of all nodes in the system, ranging from 10% to 95% of node failure. Measure of the reliability of 1.000 messages sent from random correct nodes was then taken. All these messages were sent before the execution of another cycle of the membership protocol. However, the membership protocols still execute all reactive steps; in particular, they can exclude a node from their partial views if the node is detected to be failed. The rationale for this setting is that the interval of the cyclic behavior of the membership protocols is often long enough to allow thousands of messages to be exchanged; the goal here is to focus on the impact of failures in the reliability of these specific broadcasts.

The average reliability for these runs of 1.000 messages is depicted in Figure 4.2. As it can be seen, massive percentage of failures have almost no visible impact on HyParView below the threshold of 90%. Even for failure rates as high as 95%, HyParView still manages to maintain a reliability value in the order of deliveries to 90% of the active processes. Both Scamp and Cyclon exhibit a constant reliability[1] for failure percentages as low as 10%, and their performance is significantly hampered with failure percentages above 40% (with reliabilities below 50% of nodes). On the other hand, CyclonAcked manages to offer a competitive performance. Although the reliability is not as high as with HyParView, it manages to keep high reliabilities for percentage of failures up to 70%. This behaviour highlights the importance of fast failure detection in gossip protocols and shows the benefits that come from the use of TCP as an unreliable failure detector.

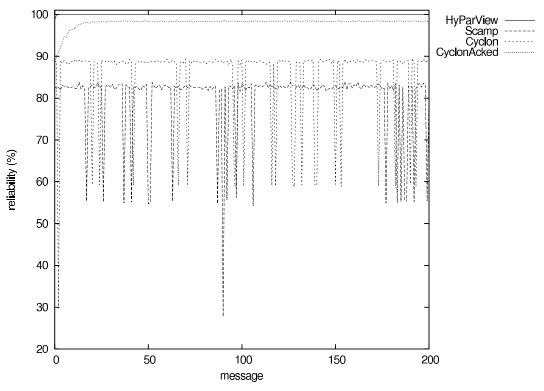
Figures 5a-5f shows the evolution of reliability with each message sent, after the failures, for different failure percentages. In all figures, HyParView is the line that offers better and faster recovery usually near the 100%. Next appear CyclonAcked, Cyclon and finally Scamp in this order for every failure level depicted. Above 80% failures all these lines appear close to the value of 0%.

From the figures, it is clear that HyParView recovers almost immediately from the failures. This is due to the fact that all members of the active views are tested in a single broadcast. Basic Cyclon/ Scamp gossip protocols, as they do not use a reliable transport protocol, are unable to recover until the membership protocol is executed again. In order to maintain reliability under massive percentage of failures, they would have to be configured with very high fanout values (which is a cost inefficient strategy in steady state). The figures also show that by adding acknowledgments to the Cyclon based gossip protocol, CyclonAcked recovers a high reliability after a small number of message exchanges (approximately 25). Note that, in CyclonAcked, a node is only tested when it is selected at random as a gossip target. However, for percentage of failures in the order of 80%, CyclonAcked is unable to regain the reliability levels as HyParView. This is due to the following phenomenon: given that the Cyclon overlay is asymmetric, some nodes may have outgoing links and no incoming link; therefore, some nodes are

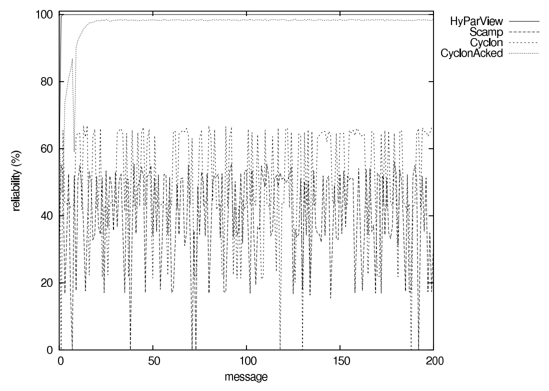
[1]Although their reliability is unable to reach 100% with a fanout of 4.



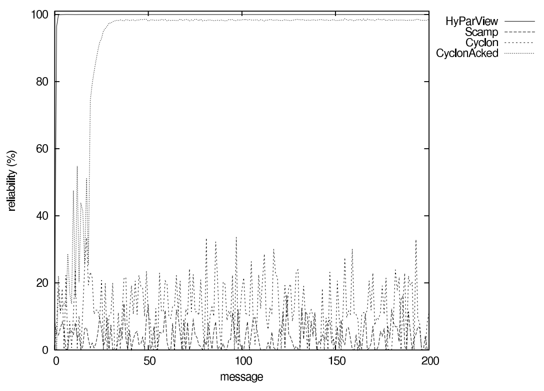
(a) 20%



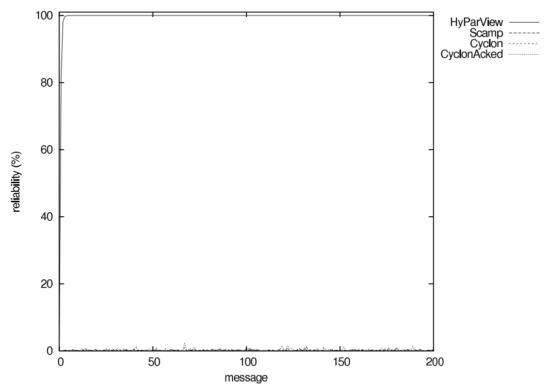
(b) 40%



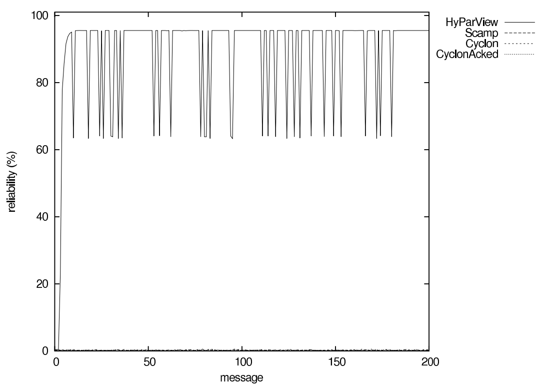
(c) 60%



(d) 70%



(e) 80%



(f) 95%

Figure 4.3: Reliability after failures

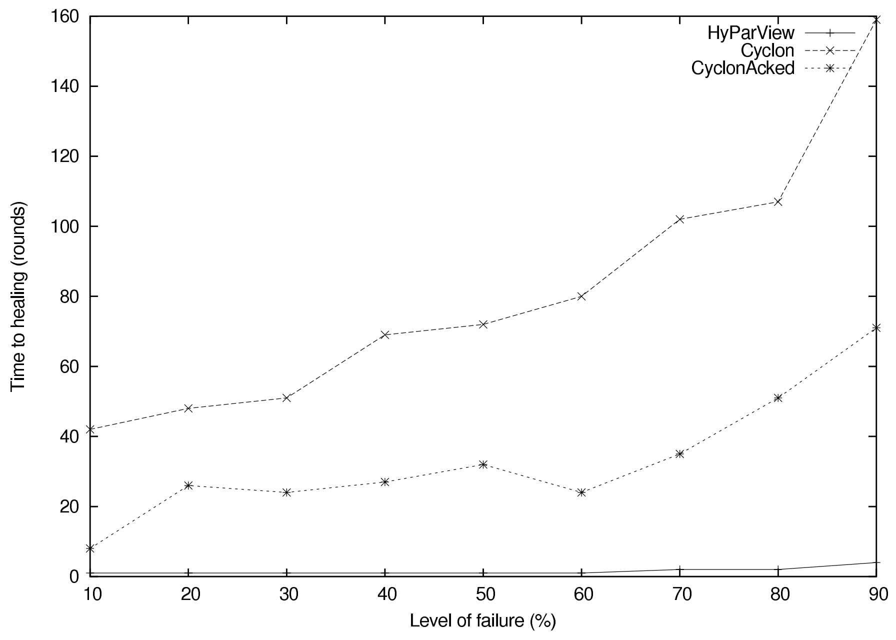


Figure 4.4: Membership convergence

still able to broadcast messages but are unable to receive any message. On the other hand, in HyParView, the active membership is symmetric, which means that if a node is able to reach another correct node in the overlay, it is necessarily reachable by messages sent by other nodes hence, it is easier to maintain the overlay connected. This feature and a very low clustering coefficient (see Section 2.2.4) explains the high resilience of HyParView.

4.3.3 Healing Time

Figure 4.4 shows how many membership cycles are required to achieve the same reliability in the message dissemination after a massive node failure (for different percentage of node failures). These results were obtained as follows: in each simulation, after the stabilization period, failures are induced. Subsequently, multiple membership protocol cycles are executed. In each cycle, 10 random nodes are selected to execute a broadcast. Then the average reliability of these messages were calculated, and the number of cycles required for each protocol to regain a reliability equal or bigger than the one exhibit by that same protocol before failures were induced was counted[1].

As expected, after the results presented before, HyParView recovers in few rounds for all percentages below 80%, usually in only 1 to 2 cycles. Cyclon requires a significant number of membership cycles, that grows almost linearly with the percentage of failed nodes to achieve this goal. Values for Scamp are not presented, as the total time for Scamp to regain it's levels of reliability depends on the *Lease Time* (as explained in section 2.5.1), which is typically high enough to preserve some stability in the membership protocol.

4.4 Plumtree

In this section the Plumtree gossip protocol is evaluated. This evaluation covers the basic tree algorithm (as depicted in section 3.4.2) and also its optimization (as shown in section 3.4.3.2). In order to have comparative figures, experimental results of these strategies are shown together with those obtained by the eager push strategy in the same scenarios.

The Plumtree protocol performs better in scenarios with only one (fixed) sender. Nevertheless, this protocol, and its optimization, were evaluated in two distinct modes of operation:

One sender (labeled as *s–s*) mode in which a single node is the source of all broadcast messages.

Multiple senders (labeled as *m–s*) mode in which each broadcast message is sent by a random correct node.

4.4.1 Stable Environment

To evaluate the protocols in a stable environment simulations were conducted in the following manner: First all nodes join the overlay by using the HyParView join mechanism. As in the evaluation of the HyParView protocol, this is achieved by

[1]This constraint was relaxed for 90% failure percentage as all protocols were unable to regain their reliability level. Nevertheless all protocols were able to stabilize with reliability values slightly below the values exhibited before the failure. For this reason, the number of cycles to stabilization were counted instead.

using one single node that serves as *contact* for all other nodes. No membership cycles are executed during the join process. Simulations are run for a total of 250 cycles. The first 50 cycles of each run are used to ensure the stabilization of the simulation and hence, are usually not depicted in the results. In each simulation cycle, in the broadcast step, a single node sends a message. The reliability, last delivery hop and relative message redundancy of the broadcast protocols are evaluated for each message disseminated.

4.4.1.1 Reliability

Simulations show that all gossip protocols achieve and maintain a reliability of 100% in stable conditions. This is expected, given the properties, specially from the connectivity point of view, of the HyParView protocol. HyParView ensures the connectivity of the overlay, and both approaches (either the eager push based or the tree based) ensure that all nodes in the overlay receive each broadcast message as long as the overlay remains connected. Still, measurement of reliability in stable conditions serve as a validation for the design of the Plumtree protocol.

4.4.1.2 Relative Message Redundancy

The RMR value (relative message redundancy, as defined in Section 2.3) is of paramount importance when evaluating the Plumtree protocol, as this is the metric aimed for optimization by this approach.

Figure 4.5 shows aggregated values of relative message redundancy for all gossip protocols in the last 200 cycles of several simulations. Notice that, as expected, the eager protocol has a relative message redundancy close to 3. This derives directly from the fanout value used (4), as explained in Section 2.3.

In Plumtree, after the stabilization of the protocol, payload messages are only propagated through links that belong to the spanning tree. The natural evolution of Plumtree (and its optimization) ensures that the tree does not contains redundant branches. Therefore, it does not generate any redundant message. Because of this, the RMR value for the Plumtree protocol and its optimization is constantly 0 and is not visible on the graph.

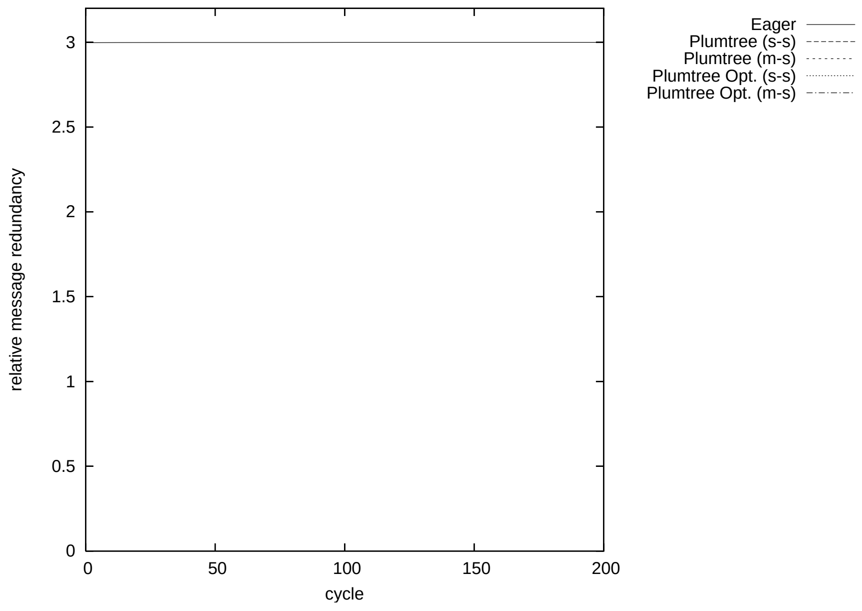


Figure 4.5: Relative message redundancy in stable environment

	Payload	Control	Total
Eager	39984.00	0.00	39984.00
Plumtree (s-s)	9999.00	29987.33	39986.33
Plumtree (m-s)	9999.00	29990.00	39989.00
Plumtree Opt. (s-s)	9999.00	29989.33	39988.33
Plumtree Opt. (m-s)	9999.00	38976.00	48975.00

Table 4.2: Number of messages received

Table 4.2 shows the number of messages received by each strategy 150 cycles after the start of the simulation. The extra control messages received when using the Plumtree are essentially due to IHave messages. However, the reader should consider that: *i*) usually IHave messages are smaller than payload message, hence these messages will contribute less to the exhaustion of network resources and *ii*) IHave messages may be aggregated, by delaying the transmission of these messages and sending several message identifiers in a single IHave message. Notice that aggregation will not have a significant impact in reliability, as messages are sent anyway only with a delay. Therefore, aggregation would only affect the time required to repair the spanning tree after failures, and the overall latency of the system.

The reader should also notice that the Plumtree protocol optimization with multiple senders generates close to 10.000 extra control messages than the same protocol with a single sender or than the basic Plumtree protocol with multiple senders. This represents a 22.5% increase in signaling cost. This happens due to the following phenomena. Because each message is sent by a different node, the protocol will trigger many times its optimization routine. This requires the exchange of 2 extra messages with neighbors, which explains the higher amount of control messages in this case.

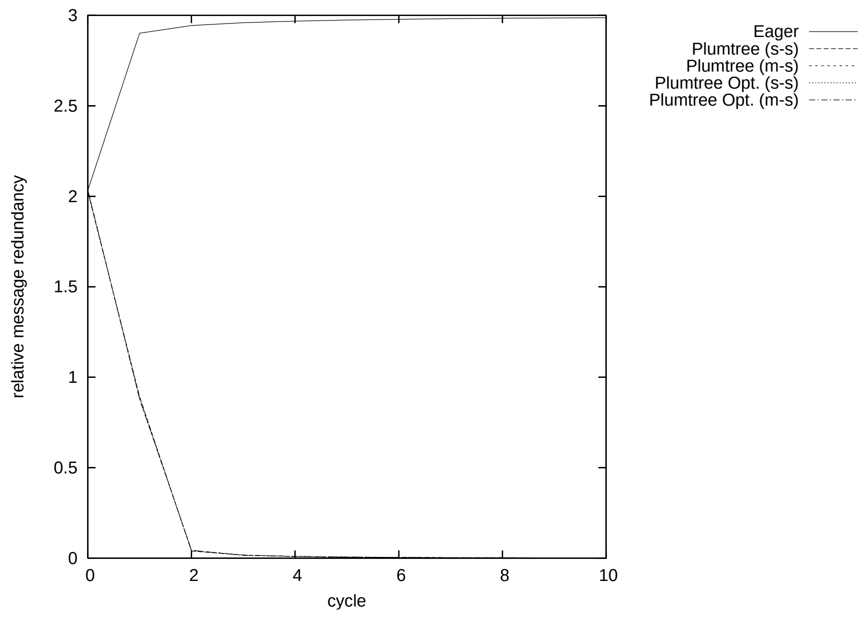


Figure 4.6: Relative message redundancy during bootstrap process

The Plumtree protocol relies on a prune technique to construct the spanning tree. Because of this, it is expected that during the dissemination of the first broadcast message a number of extra messages is produced. In order to evaluate the overhead of the tree building process, the relative message redundancy for the first 10 cycles of simulation are presented in Figure 4.6. All tree based approaches have similar values, and all of them only take 2 membership cycles to stabilize. It's also shown that only the first message broadcast produces a visible overhead concerning messages exchanged in the overlay.

The relative message redundancy value of 2 observed for all instances of the Plumtree protocol in the cycle 0 can be explained as follows: When the first message is broadcast in the system, each node has all its neighbors in the *eagerPushPeers* set, therefore, the behavior of all nodes degenerate into a flood approach. This is the reason why the RMR value for both the Plumtree protocol and the eager push protocol, is the same is cycle 0.

The reason why the RMR value is sightly above 0 in cycle 1 is due to a different reason. In the membership step of the first cycle, the HyParView protocol uses information from passive views to fill the active views of several nodes. This generates several `NeighborUp` notifications that are received by Plumtree; these neighbors are added to the *eagerPushPeers* set in each node. This process generates some redundant messages in this cycle. These messages however, serve to connect nodes to the spanning tree and moreover, are used to optimize the tree in a scenario with a single sender. This happens because a node which receive the same payload message from two links in the overlay, will only keep the link from which it received the message for the first time.

4.4.1.3 Last Delivery Hop

Figure 4.7 presents the values for LDH (last delivery hop, as defined in Section 2.3) for all protocols. The eager protocol and Plumtree with a single sender offer the best performance. Notice that the eager protocol uses all available links to disseminate messages, which ensures that all shortest paths of the overlay are used. This shows that with a single sender Plumtree is able to select the links that provide faster delivery.

With multiple senders the basic Plumtree protocol values are very high. This happens because the spanning tree is optimized to the node who broadcasts the first message. Therefore, when messages are sent by nodes located at leaf positions of the tree, they require to perform more hops in the overlay to reach all other nodes. On the other hand, the optimization of the protocol is able to lower significantly the value of LDH. The reader should notice that because the optimization triggers for different senders, in fact it will better distribute the links that form the spanning tree through the overlay, effectively removing the bias of the tree to the sender of the first message.

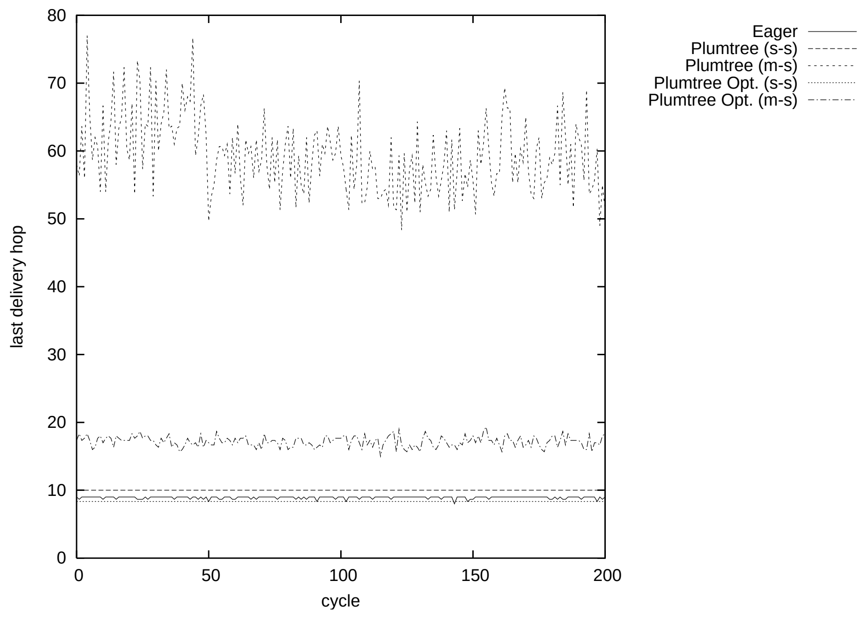


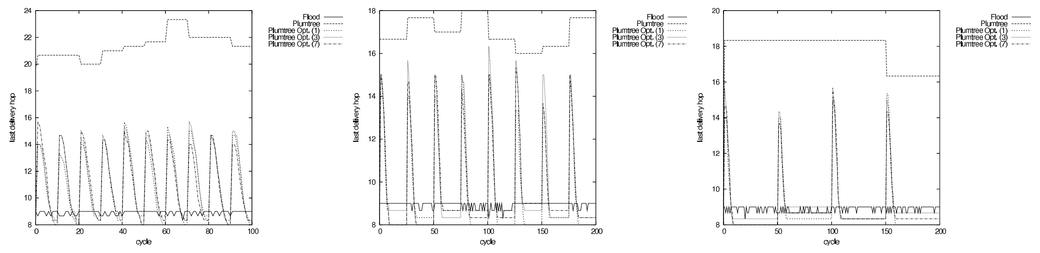
Figure 4.7: Last delivery hop in stable environment

4.4.2 Effect Of Bursty Behavior

Many systems exhibit a behavior where nodes communicate through bursts of messages. We assume that in such a system, nodes will broadcast a number of messages, and while a node is broadcasting its messages, all other are only receiving. One example are conferencing systems with floor control (Dommel & Garcia-Luna-Aceves, 1997).

To evaluate the behavior of Plumtree in such an environment we run experiments where we use multiple senders but, where the source remains the same for a certain number of cycles. As in the previous experiments, in each simulation cycle only one node broadcasts a message. The number of simulation cycles where the source remains unchanged is therefore the length of the message burst used by nodes. Experiments were conducted for 3 distinct burst lengths namely: 10 message burst, 25 message burst and 50 message burst.

All experiments were run in stable environments (*i.e.* no node failures where induced in the system). The reader should notice that, because this experiments were conducted in stable environments, results obtained for reliability and RMR are similar to those presented above. Therefore those results will not be depicted here. Each experiment uses 3 different values for the *threshold* parameter of the Plumtree optimization. Values used were 1, 3 and 7, in order to show the impact of this parameter in the time required by the protocol to obtain the best performance in terms of LDH.



(a) 10 messages burst (b) 25 messages burst (c) 50 messages burst

Figure 4.8: Last delivery hop with bursts of messages

Figure 4.8 shows the evolution of the LDH value and for each protocol for each burst size. Because the source is the same for a number of messages, the optimization is able to improve the spanning tree.

First, as expected, the Plumtree optimization is able to improve the performance to values that match the values obtained by the eager push strategy. Moreover, the number of messages[1] required to ensure that that protocol converges to the (best) optimized tree for a sender is always the same and it is independent of the *threshold* parameter. The number of messages required is 8, and this value is dependent of the overlay diameter which is also 8.

One could think that smaller *threshold* parameter would reduce the number of messages required. This however does not happen due to the following phenomena. When a source starts sending its burst, the tree has been optimized for other source. The optimization propagates trough the tree in a cascade, because nodes send IHave messages with the round value of the payload message they received first, even if the node took measures to improve its distance in the spanning tree to the source. Therefore, with each message sent, a new set of nodes become aware that some of its neighbors have optimized their distance to the sender. In

[1]Remember that in each simulation cycle, only a single message is sent therefore, the number of simulation cycles are equal to the number of messages sent.

fact, the optimization of the spanning tree is propagated in the random overlay as an epidemic process, whereas each round of the process is equivalent to each message sent by the same source.

This raises one interesting observation, the Plumtree optimization will only be effective for message bursts whose length is larger than the diameter of the random overlay. Moreover, the gain obtained by the optimized Plumtree protocol is proportional to the length of message bursts sent by nodes.

Finally, the reader should notice that the optimized Plumtree protocol is able to obtain better values for LHD than the eager push protocol, what at a first glance should be impossible, as this strategy floods the overlay and therefore should ensure the smallest possible values for LDH. The explanation for this comes from the following phenomena. In fact, and because the eager push protocol uses a fixed fanout, that protocol uses all links available in the overlay except one, which is directly connected to the sender of each message. Remember that the overlay maintained by HyParView has a degree of fanout+1. On the other hand, Plumtree only employs the fanout value to initialize its *eagerPushPeers* set. Due to `NeighborUp` notifications the number of nodes in this set might rise to a maximum of fanout+1 peers. In practice, the source of a message employs an eager push strategy in all its links. Notice that the use of an extra link which is connected directly to the source will have a high probability to improve the LDH in one unit. Which explains the better values obtained by the Plumtree protocol.

4.4.3 Effect Of Failures

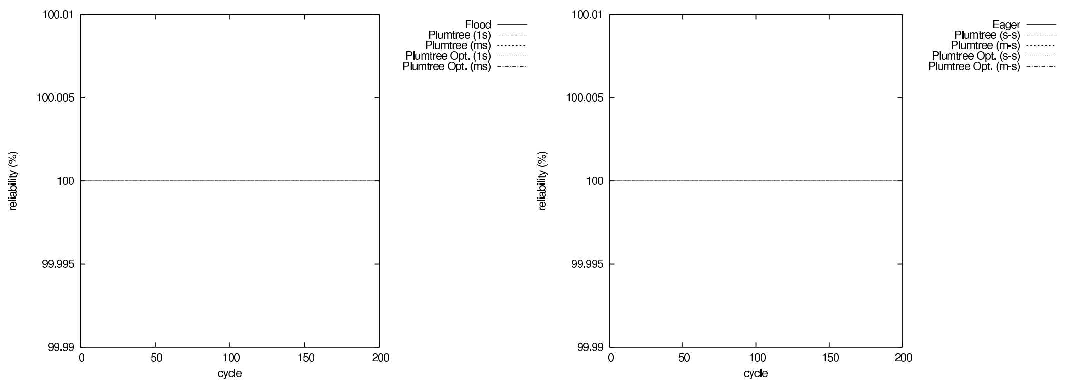
In this section the properties of gossip strategies are tested in faulty scenarios where nodes can crash. Two distinct scenarios were experimented.

In the first scenario, small number of node failures were induced for 100 consecutive cycles. These experiments aim to show the impact of a very unstable environment in the properties of each gossip strategy. In the second scenario the impact of massive failures (as evaluated ins Section 4.3.2) was tested. The aim of these experiments is to show the implications of tree based gossip strategies in scenarios were large percentages, ranging from 10% to 95% of all nodes, can fail simultaneoulsy (*e.g.* on the event of a natural catastrophy like an earthquake or a tsunami).

4.4.3.1 Sequential Failures

In this scenario, simulations were conducted in the following way: In each simulation cycle one message was broadcasted by one node. The first 50 cycles were used to ensure stabilization of the experiment. After the stabilization period, a constant failure rate was induced in the system for 100 simulation cycles. After that, the simulations continued for more 100 cycles.

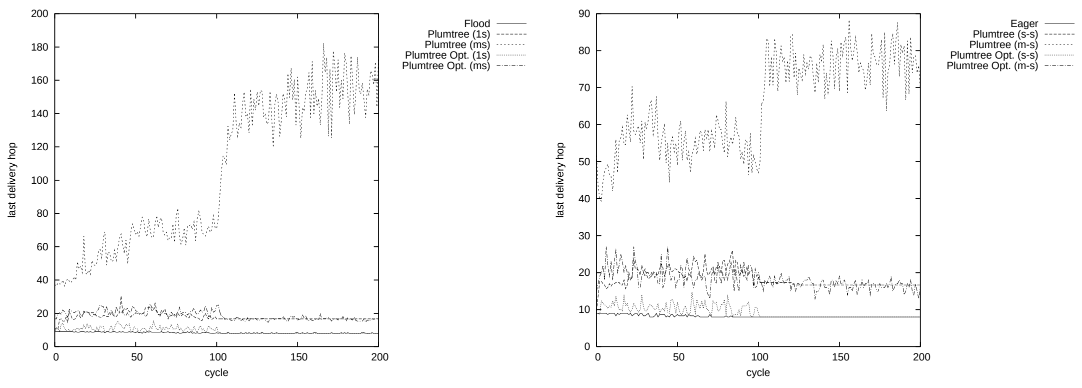
Two distinct failure rates were used: 25 and 50 nodes per cycle. The nodes that fail are selected at random, except when running the one sender mode of operation, where the sender never fails. The reader should notice that, in the multiple sender mode of operation, this exception does not exist, and the sender of the first broadcast message (the node that served as root in the construction of the spanning tree) can also fail. This also shows that the tree based gossip strategy can tolerate failures of the root node of the spanning tree.



(a) 25 nodes per cycle (b) 50 nodes per cycle

Figure 4.9: Reliability with sequential failures

Figures 4.9a and 4.9b show reliability for each gossip strategy in the last 200 cycles of simulation for both failures rates (4.9a for a failure rate of 25 nodes per cycle and 4.9b for 50 nodes per cycle). For both failure rates, all protocols are able to maintain a reliability of 100%. This comes from the natural resilence of HyParView for such small failures.



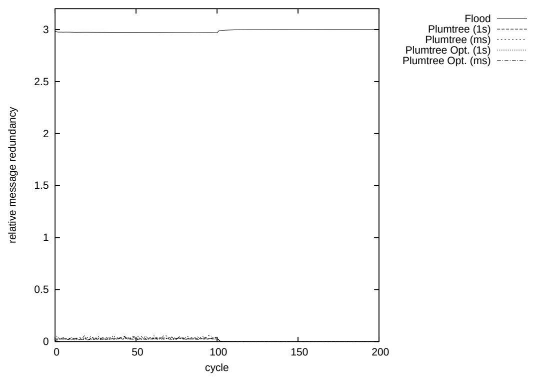
(a) 25 nodes per cycle (b) 50 nodes per cycle

Figure 4.10: Last delivery hop with sequential failures

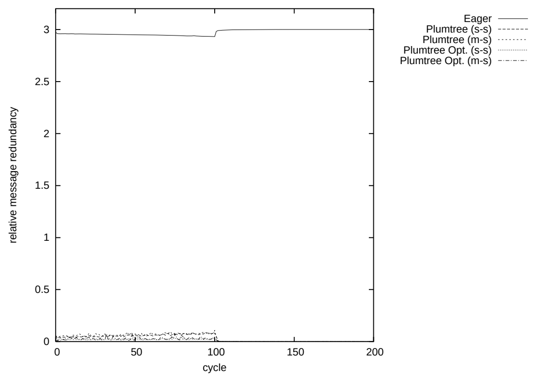
.

The LDH value of all protocols is slightly affected by failures. Figure 4.10 presents the LDH for the same 200 simulation cycles for both failure rates. It shows that LDH values become unstable for all protocols in cycles where failures are induced. This is expected, as these failures may remove links that were part of the optimal paths between nodes. Results also show that, when failures are no longer induced, all protocols are able to regain a more constant value. Also the original Plumtree protocol with multiple senders is the case where the impact of failures is more visible, as the LDH value is significantly lower during this period. This happens because of the following phenomena. The addition of new links as a reaction to failures will produce some redundant paths in the spanning tree. This allows the protocol to optimize the tree by selecting links which reduce the distance to the sender of the first message broadcasted hence, reducing the LDH for those messages.

Finally, Figure 4.11 shows the RMR values for all protocols. The eager protocol presents values slightly lower than 3 in the first 100 cycles while the Plumtree protocol (in all cases) presents values slightly above 0. Notice that failures will remove some links from the overlay therefore, there will be less payload messages being received by nodes with the eager protocol, on the other hand, the membership protocol will add new links to replace the lost ones. This will trigger `NeighborUp` events in some nodes, which will add new neighbors to their



(a) 25 nodes per cycle



(b) 50 nodes per cycle

Figure 4.11: Relative message redundancy with sequential failures

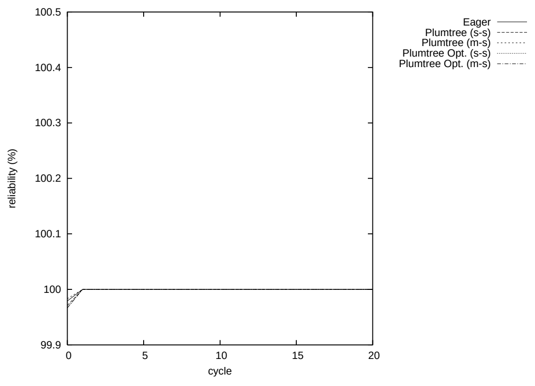
LazyPushPeers set hence, more payload messages will be sent and therefore, the number of redundant payload messages received by nodes will increase.

4.4.3.2 Massive Failures

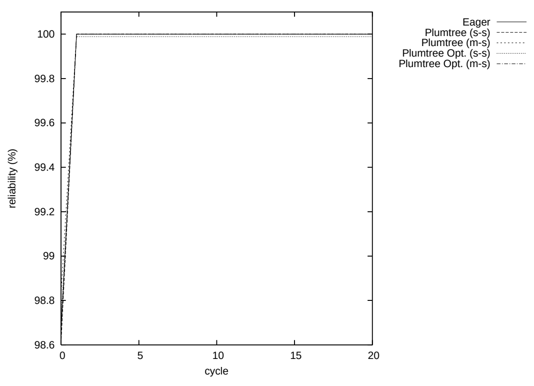
In this section the effect of massive failure for all gossip strategies is evaluated. This evaluation was conducted by using a similar technique as the one employed in Section 4.3.2 and can be summarized as follows: After a stabilization period of 50 cycles in each simulation, node failures are induced at random with different percentages of the total number of nodes in the system, ranging from 10% to 95% in the failure step. After that 200 simulation cycles were executed. In each cycle, in the broadcast step, one node broadcasts a message. Values concerning reliability, last delivery hop and relative message redundancy are again evaluated and presented next.

Figures 4.12a-4.12f show the evolution of global reliability for each gossip strategy in the 10 cycles after failures were induced[1]. It shows that all strategies are able to regain their reliability values of 100% after a small amount of membership cycles, which is expected, as this resilience to failures and fast healing capacity comes from the use of the HyParView protocol as the underlying peer sampling service, by all these gossip protocols.

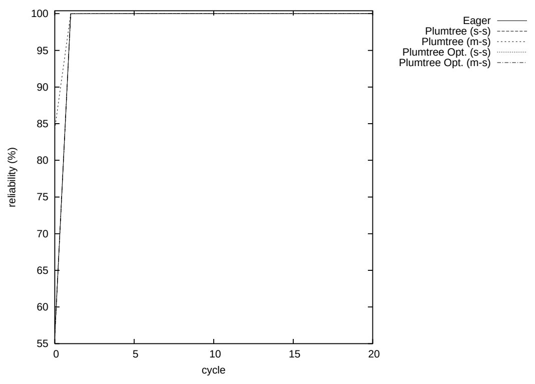
[1]Only 10 cycles are depicted instead of 200 because the reliability values stabilized hence, there was no significance to the values.



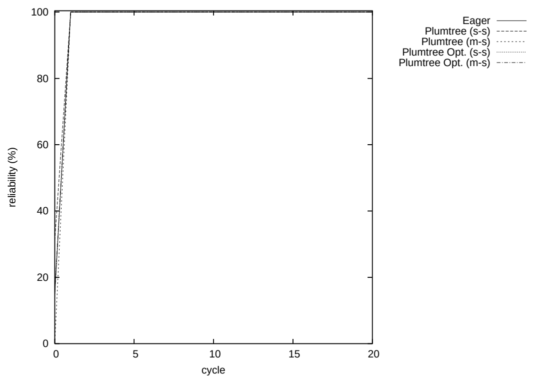
(a) 20%



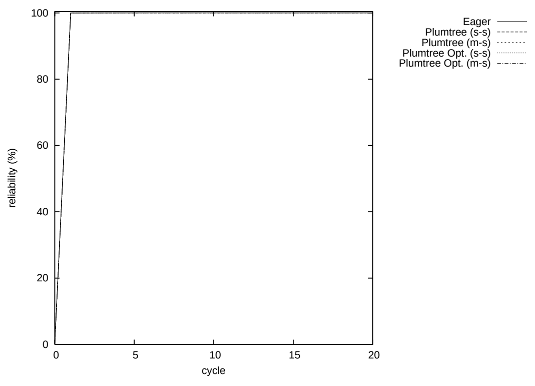
(b) 40%



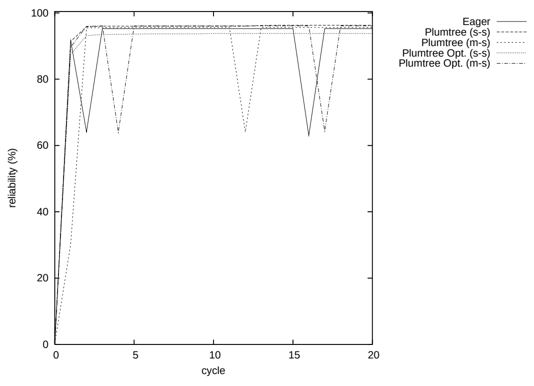
(c) 60%



(d) 70%



(e) 80%



(f) 95%

Figure 4.12: Reliability after failures

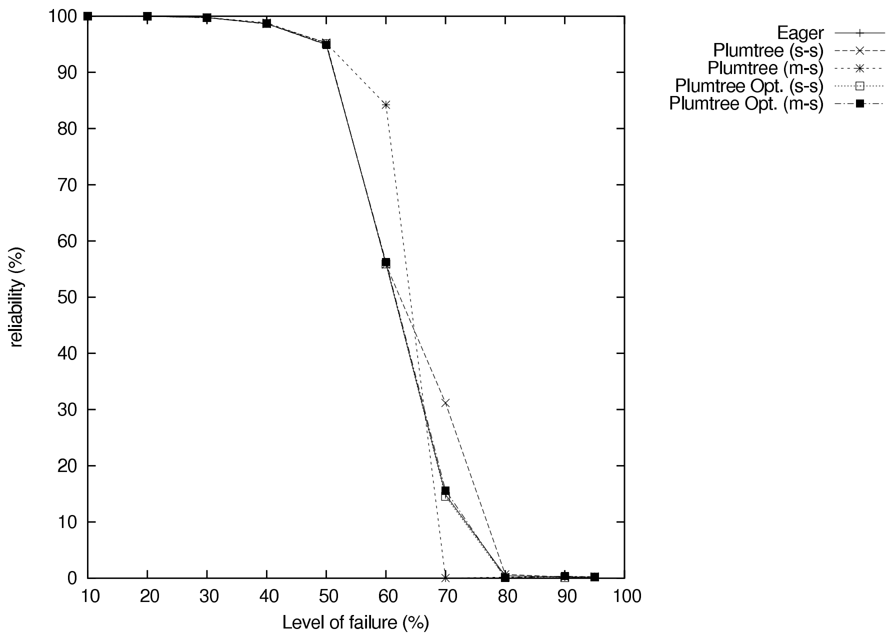


Figure 4.13: Reliability of gossip immediately after failures

Figure 4.13 depicts the reliability of each protocol immediately after failures. The reader should notice that for failure percentages above 70%, reliability drops to values close to 0 for all protocols. This happens because the overlay becomes disconnected after such large percentage of node failures.

4.4.4 Healing Time

Similar to the results presented in Section 4.3.3, Figure 4.14 depicts the number of simulation cycles required by the each gossip strategy to regain its reliability level after a massive node failure (for different percentages of node failures). The results were obtained by running simulations in which, after a stabilization period, failures were induced. The simulation then continues running, and in each cycle one node sends a broadcast message. The number of cycles to the broadcast protocol to regain a reliability value equal or higher than the one exhibited before the failures were counted[1].

[1]Similar to what was done in section 4.3.3, these constraints were relaxed for the values concerning 90% of failures

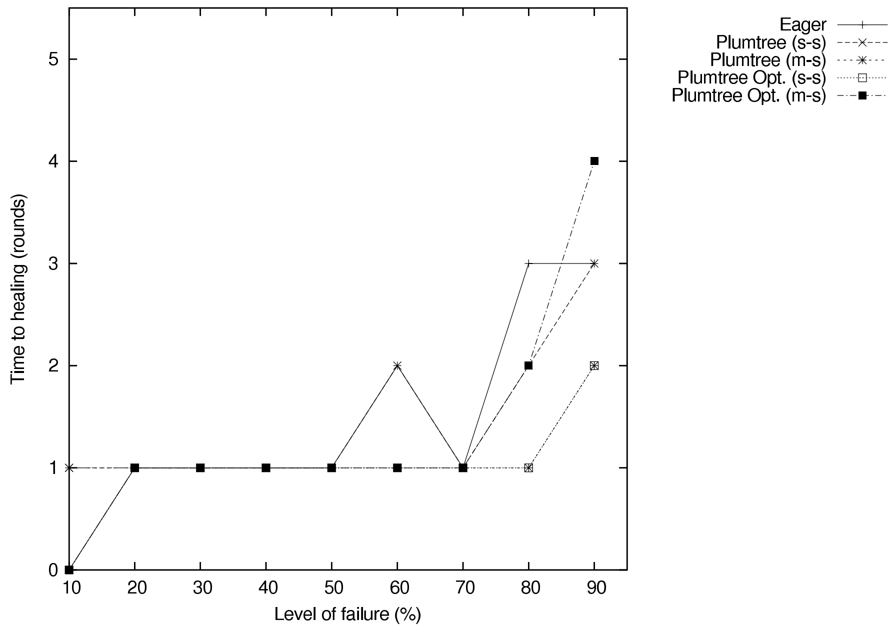
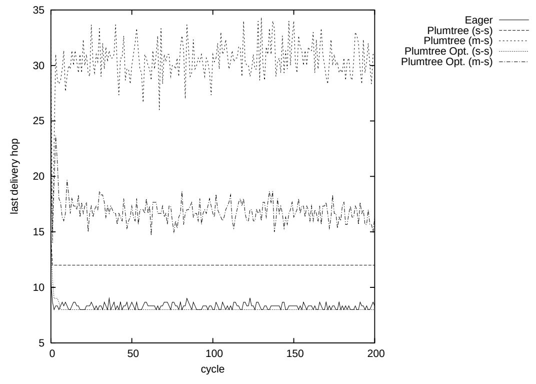


Figure 4.14: Healing time

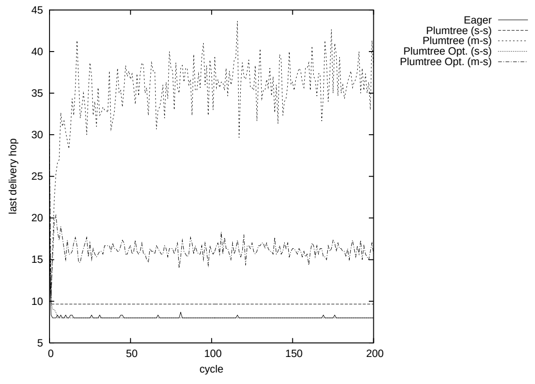
The time to regain reliability, in number of simulation cycles, is not significantly different. This clearly shows that Plumtree retains the reliability of the eager push gossip protocol, given that the spanning tree embedding is only used to select which links are used for eager/lazy push.

Figures 4.15a-4.15f show the LDH for all protocols after failures for 6 different failure percentages. As expected, it confirms the results presented above. All protocols are able to maintain a, somewhat, constant value for LDH. Whereas the eager protocol and Plumtree (and its optimization) with a single sender have the best performance, followed by the optimized Plumtree with multiple senders and finally the original Plumtree with multiple senders.

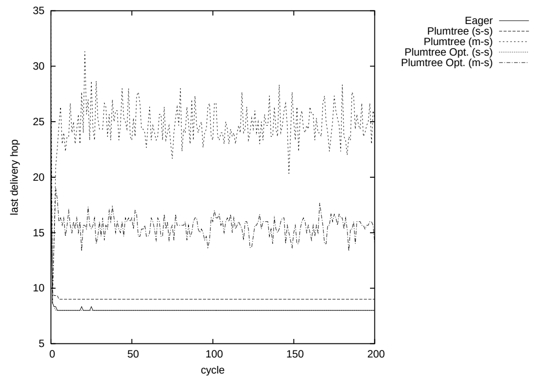
Finally the RMR value exhibited by each gossip strategy is also evaluated in faulty scenarios. The results are depicted in Figures 4.16a-4.16f. The important aspect to retain from these figures is that all protocols are able to regain their RMR levels before failures in only a couple of cycles. After failures all protocols exhibit a low level of redundancy. For all failure percentages, all versions of the Plumtree protocol show an increase of redundant messages after failures. This is due to the transmission of extra payload messages as a result of the healing



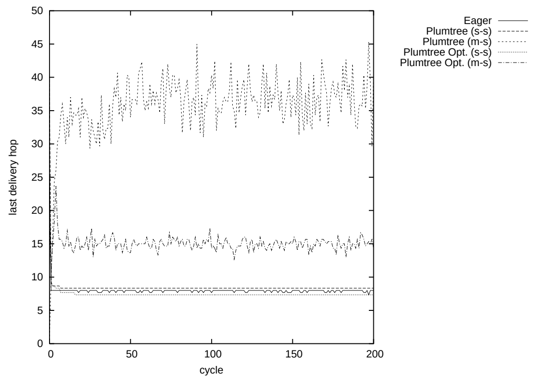
(a) 20%



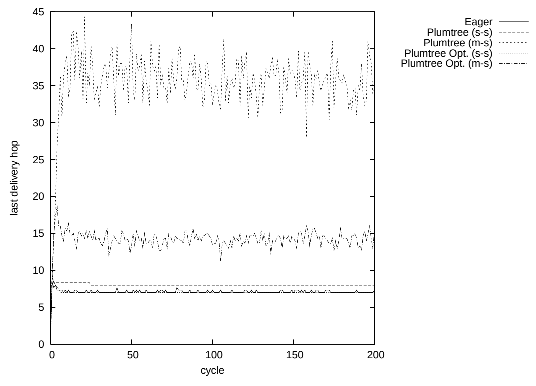
(b) 40%



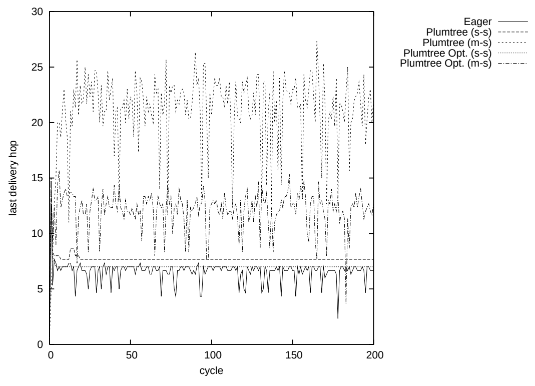
(c) 60%



(d) 70%

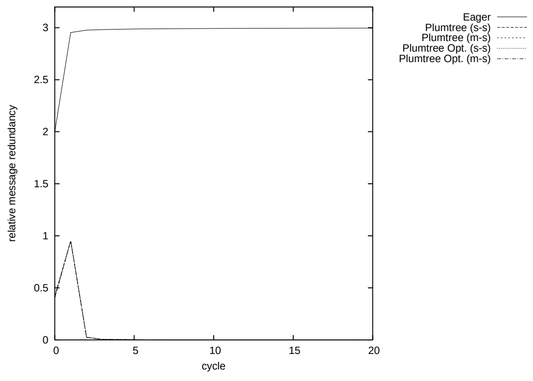


(e) 80%

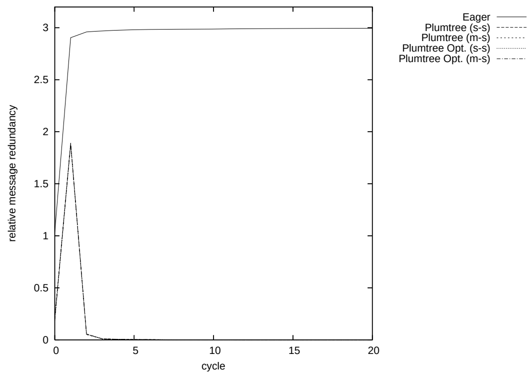


(f) 95%

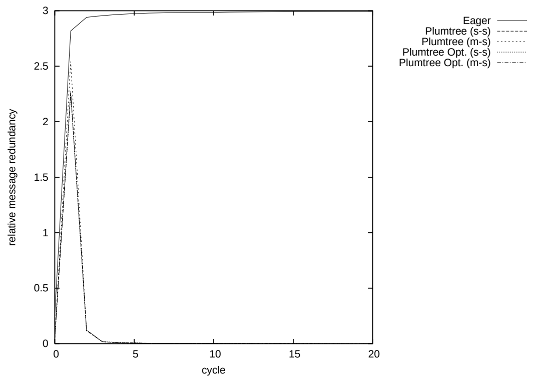
Figure 4.15: Last delivery hop after failures



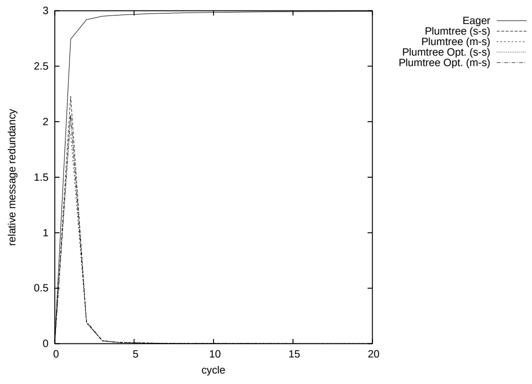
(a) 20%



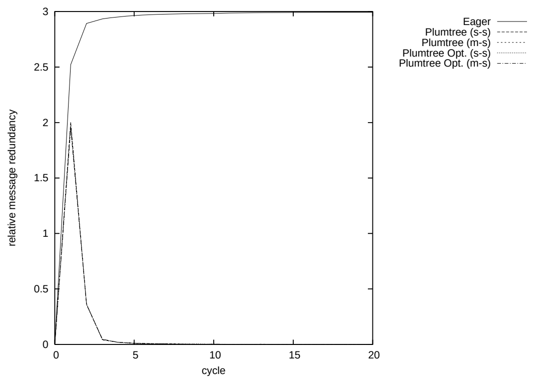
(b) 40%



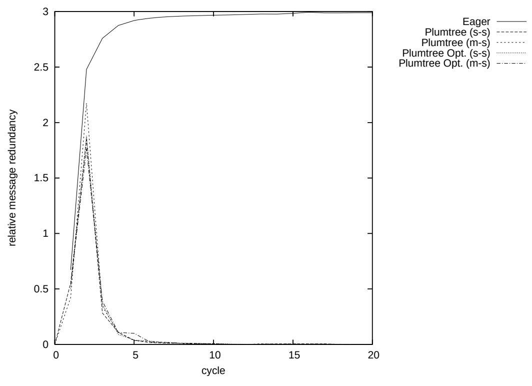
(c) 60%



(d) 70%



(e) 80%



(f) 95%

Figure 4.16: Relative message redundancy after failures

process of the overlay, which adds new links to compensate those lost due to failures. The same effect justifies the rise of the RMR value, after failures, for the eager protocol.

4.5 Summary

This chapter presented the evaluation results, obtained through simulation, of the HyParView protocol and all gossip strategies developed. All these strategies use HyParView as underlying membership protocol. Evaluation covered reliability, last delivery hop and relative message redundancy of these gossip strategies in stable environments and also in two distinct faulty scenarios.

The next chapter presents final remarks concerning the protocols developed and the experimental results presented in this chapter, and will also point directions for future work.

Chapter 5

Conclusion And Future Work

5.1 Conclusions

Gossip protocols are appealing because they work on overlays that have very small maintenance cost. On the other hand, a broadcast protocol based on a minimum cost spanning tree may be very effective in stable networks, but may be impractical to use on systems with large number of nodes where failures are common (due to the cost of reconfiguring the tree). Therefore, gossip protocols seem obvious candidates to applications that require extremely high resilience to failures of large percentage of nodes. Such massive failures can happen due to attacks (for instance, a worm that shuts down all the machines of a particular make) or in catastrophic natural disasters (such as earthquakes).

In the thesis two distinct gossip strategies were presented, which combine deterministic and gossip approaches. This is achieved by using an overlay topology that is created by a probabilistic (partial) membership protocol, and then use different deterministic broadcast strategies on top of this overlay. The membership protocol, called HyParView, maintains a small active view and a larger passive view for fault-tolerance. It was shown that the HyParView protocol, when combined with eager push gossip protocol, allows the broadcast protocol, to provide and preserve very high values of reliability, with a small fanout, in faulty scenarios where the percentage of failed nodes can be as high as 80%.

It is possible to extract the following lessons from the results presented in the thesis. To start with, the speed of failure detection is of paramount importance to sustain high reliability in the presence of massive percentage of faults. A gossip strategy that relies on the use of a reliable transport over stable overlay (built using a probabilistic membership protocol), offers the best performance possible in this regard, given that the reliable transport also serves as an unreliable failure detector. Also, by using all the links of the overlay, it is possible to aim at 100% reliability as long as the overlay remains connected. Furthermore, it allows the use of smaller fanout values than protocols that have to mask failures and network omissions with the redundancy of gossip. The use of small fanout values is what makes possible to use all the links of the overlay with small overhead. Additionally, the maintenance of a passive view, with candidates to replace failed nodes in the active view, offers high resilience to massive failures. Therefore, the use of an hybrid approach that contains a small active view and a larger (low cost) passive view offers a better resilience and better resource usage than using a single (large) view with a higher fanout.

The thesis also presented two gossip strategies that leverage on the properties of HyParView.

- An eager push gossip strategy (which was used when evaluating HyParView performance against other membership protocols), that works by flooding the overlay maintained by HyParView's (small) active view.

- A tree based gossip protocol, which we named Plumtree, was also introduced. This in a rather more complex strategy that relies in the combination of eager push and lazy push approaches. The algorithm explicitly builds a spanning tree (*i.e.* the protocol maintains state concerning the structure of the tree) which enables it to apply eager push of payload messages in the tree branches without generating any redundant message reception. A lazy push gossip phase is used to efficiently detect nodes that have become disconnected from the spanning tree due to node failures. This phase automatically repairs the tree, while ensuring that nodes receive as many broadcast messages as possible.

- An optimization of the Plumtree protocol was also suggested. This optimization is able to make adjustments on the spanning tree structure in order to ensure a constant number of hops required to disseminate messages to all participants, which is achieved by sacrificing a bit of the stability in the structure of the tree.

Experimental results show that, the embedded spanning tree employed by Plumtree, on top of a low cost random overlay network, allows to disseminate messages in a reliable manner with considerably less traffic on the overlay than a simple gossip protocol. Moreover, by exploiting links of the random overlay that are not part of the spanning tree, one can efficiently detect partitions and repair the tree. One interesting aspect of the Plumtree approach, is that it can be easily used to provide optimized results for a small number of source nodes, by maintaining state for one spanning tree for each source. This is feasible as our approach does not require the maintenance of complex state.

In terms of latency, the Plumtree protocol is more effective for building sender based trees but, with the optimizations proposed, it can also be used to support shared trees, with a penalty in terms of overall system latency which presents twice that value. This can be achieved by relaxing the constraints on the stability of the spanning tree. In such a way, one can improve the latency of the system and provide better results when the spanning tree is shared by several nodes to disseminate messages. It was also showed that the same strategy used to detect and repair the spanning tree can easily be extended to optimize the tree for these conditions. We defend that this is of paramount importance in order to avoid the negative impact in terms of latency when sharing the tree but also to support communication models which are based on message bursts from single nodes, with several senders.

Concerning the application of these gossip strategies, one can conclude that, for large systems with high safety requirements or with low latency requirements as for an instance, a national coordination response system for emergencies, the eager push gossip strategy presents itself as the best approach, as it gives the best results concerning these specific requirements. For non-critical systems, as for instance a large-scale news or software update system, which do not present such demanding safety requirements, but where a high reliability is still expected while consuming as few resources as possible (notice that these should be *background* applications), the tree gossip strategy is possibly the better approach to use.

5.2 Future Work

The HyParView membership protocol, and both gossip strategies, presented in the thesis should be further experimented and evaluated in more complex and realistic scenarios. Further evaluation should specially focus on the impact of the underlying network topology and its implications on the performance of these protocols. For instance, it is of interesting to understand the amount of overhead imposed by the use of TCP (instead of UDP), as well as to evaluate the latency of broadcast protocols that rely on HyParView and gossip strategies presented here. To assess these aspects, different techniques and platforms from the ones employed in the thesis, need to be used.

At the time of the writing of this thesis, a java[1] implementation of HyParView, and of the broadcast protocols, are being developed and validated. These implementations, will be used to conduct further experiments with two additional testbeds. First, the ModelNet large-scale emulation infrastructure[2] (Vahdat *et al.*, 2002), that will enable us to run simulations in realistic large-scale network topologies with large numbers of nodes. The PlanetLab testbed[3] (Chun *et al.*, 2003) will also be employed in future experiments, as it will provide information and insight on a real world application of these protocols.

HyParView is a novel membership protocol whose properties should be further analysed from a theoretical point of view. For instance, the relation between the size of passive views and the resilience of the membership protocol to node failures must be more clearly defined.

Other gossip strategies may provide interesting results if combined with HyParView. For instance, recent work presented in Carvalho *et al.* (2007), tries to improve broadcast protocols based on gossip, by reducing unnecessary redundant messages, using an approach that also combines eager push and lazy push gossip and information about the execution environment of each node that is obtained by special components named *monitors*. The authors show that their approach enables them to produce probabilistic emergent data paths (*i.e.* without explicitly coordinating nodes to organize themselves in such a structure), that resembles

[1]http://java.sun.com
[2]http://modelnet.ucsd.edu/
[3]http://www.planet-lab.org/

an optimized spanning tree, where more powerful nodes contribute more to the information dissemination. It would be interesting to evaluate the implications of combining this strategy with the HyParView membership protocol. Hopefully, this will result in a gossip based gossip protocol that exhibits high reliability using a small fanout, being highly resilient to node failures while reducing redundant traffic in the network.

Finally, it would also be interesting to experiment an eager push gossip on top of HyParViews with adaptive fanout, by taking into account the heterogeneity of nodes, in order to maximize the use of available resources, like bandwidth. To do this, and maintain the deterministic selection of gossip targets, nodes would also require to adapt their degree (and in-degree), which might prove an effective approach to produce optimized and adaptive emergent overlays.

Bibliography

CARVALHO, N., PEREIRA, J., OLIVEIRA, R. & RODRIGUES, L. (2007). Emergent structure in unstructured epidemic multicast. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, (to appear), Edinburgh, UK. 86

CASTRO, M., DRUSCHEL, P., KERMARREC, A. & ROWSTRON, A. (2002). SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, **20**, 1489–1499. 15, 22

CHU, Y.H., RAO, S.G. & ZHANG, H. (2000). A case for end system multicast. In *Measurement and Modeling of Computer Systems*, 1–12. 14

CHU, Y.H., RAO, S., SESHAN, S. & ZHANG, H. (2002). A case for end system multicast. *IEEE Journal on Selected Areas in Communications*, **20**, 1456–1471. 20

CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M. & BOWMAN, M. (2003). Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, **33**, 3–12. 86

DEERING, S.E. & CHERITON, D.R. (1990). Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, **8**, 85–110. 8, 14

DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D. & TERRY, D. (1987). Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth* *annual ACM Symposium on Principles of distributed computing*, 1–12, ACM Press, New York, NY, USA. 6

Deshpande, M., Xing, B., Lazardis, I., Hore, B., Venkatasubramanian, N. & Mehrotra, S. (2005). Crew: A gossip-based flash-dissemination system. 20

Deshpande, M., Xing, B., Lazardis, I., Hore, B., Venkatasubramanian, N. & Mehrotra, S. (2006). Crew: A gossip-based flash-dissemination system. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, 45, IEEE Computer Society, Washington, DC, USA. 20

Diot, C., Levine, B.N., Lyles, B., Kassem, H. & Balensiefen, D. (2000). Deployment issues for the IP multicast service and architecture. *IEEE Network*, **14**, 78–88. 14

Dommel, H.P. & Garcia-Luna-Aceves, J.J. (1997). Floor control for multimedia conferencing and collaboration. *Multimedia Syst.*, **5**, 23–38. 70

Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kouznetsov, P. & Kermarrec, A.M. (2003). Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, **21**, 341–374. 6

Eugster, P.T., Guerraoui, R., Kermarrec, A.M. & Massoulie, L. (2004). From Epidemics to Distributed Computing. *IEEE Computer*, **37**, 60–67. 31, 40, 46

Ganesh, A.J., Kermarrec, A.M. & Massoulie, L. (2001). SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Networked Group Communication*, 44–55. 6, 10, 17

Ganesh, A.J., Kermarrec, A.M. & Massouli&#233;, L. (2003). Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.*, **52**, 139–149. 10, 17

JELASITY, M., GUERRAOUI, R., KERMARREC, A.M. & VAN STEEN, M. (2004). The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, 79–98, Springer-Verlag New York, Inc., New York, NY, USA. 9

KERMARREC, A.M., MASSOULI&#233;, L. & GANESH, A.J. (2003). Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. Parallel Distrib. Syst.*, **14**, 248–258. 1, 7, 13

KOLDEHOFE, B. (2003). Buffer management in probabilistic peer-to-peer communication protocols. In *Proceedings of the 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, 76–87, Florence,Italy. 6

LIANG, J., KO, S.Y., GUPTA, I. & NAHRSTEDT, K. (2005). MON: On-demand overlays for distributed system management. In *2nd USENIX Workshop on Real, Large Distributed Systems (WORLDS'05)*. 23, 42

PEREIRA, J., RODRIGUES, L., MONTEIRO, M.J., OLIVEIRA, R. & KERMARREC, A.M. (2003). Neem: Network-friendly epidemic multicast. In *Proceedings of the 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, 15–24, Florence,Italy. 19, 38, 39

RATNASAMY, S., HANDLEY, M., KARP, R.M. & SHENKER, S. (2001). Application-level multicast using content-addressable networks. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, 14–29, Springer-Verlag, London, UK. 14

RENESSE, R.V., MINSKY, Y. & HAYDEN, M. (1998). A gossip-style failure detection service. Tech. Rep. TR98-1687, Dept. of Computer Science, Cornell University. 6

ROWSTRON, A.I.T. & DRUSCHEL, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, 329–350, Springer-Verlag, London, UK. 22

Rowstron, A.I.T., Kermarrec, A.M., Castro, M. & Druschel, P. (2001). SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, 30–43. 14, 15, 22

Stavrou, A., Rubenstein, D. & Sahu, S. (2002). A lightweight, robust p2p system to handle flash crowds. Tech. Rep. EE020321-1, Columbia University, New York, NY. 11

Stevens, W. (1997). RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Status: PROPOSED STANDARD. 38

Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostic, D., Chase, J. & Becker, D. (2002). Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, **36**, 271–284. 86

Voulgaris, S., Gavidia, D. & Steen, M. (2005). Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, **13**, 197–217. 6, 11, 18

Zhao, B.Y., Kubiatowicz, J.D. & Joseph, A.D. (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, UC Berkeley. 21

Zhuang, S.Q., Zhao, B.Y., Joseph, A.D., Katz, R.H. & Kubiatowicz, J.D. (2001). Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of NOSSDAV*. 14, 16, 21, 42