# Gossip-Based Broadcast

João Leitão
INESC-ID / IST
jleitao@gsd.inesc-id.pt

José Pereira
University of Minho
jop@di.uminho.pt

Luís Rodrigues
INESC-ID / IST
ler@ist.utl.pt

**Abstract**

*Gossip, or epidemic, protocols have emerged as a powerful strategy to implement highly scalable and resilient reliable broadcast primitives on large scale peer-to-peer networks. Epidemic protocols are scalable because they distribute the load among all nodes in the system and resilient because they have an intrinsic level of redundancy that masks node and network failures. This chapter provides an introduction to gossip-based broadcast on large-scale unstructured peer-to-peer overlay networks: it surveys the main results in the field, discusses techniques to build and maintain the overlays that support efficient dissemination strategies, and provides an in-depth discussion and experimental evaluation of two concrete protocols, named HyParView and Plumtree.*

***The original publication is available at www.springerlink.com***

`http://www.springer.com/engineering/signals/book/978-0-387-09750-3`

## 1 Introduction

Many distributed systems benefit from the availability of a reliable broadcast service. Informally, reliable broadcast ensures that all correct participants receive all broadcast messages, even in the presence of network omissions or node failures. Gossip protocols have emerged as a highly scalable and resilient approach to implement reliable broadcast [1, 2, 3, 4]. In a typical gossip protocol, when a node wants to broadcast a message, it selects $t$ nodes from the system at random ($t$ 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 [4].

Gossip protocols are interesting because they are highly resilient (they have an intrinsic level of redundancy that allows them to mask node and network failures), scalable (they distribute the load among all nodes in the system), and simple to implement. In fact, gossip protocols have been proposed as a building block to solve several other problems in distributed systems such as: consistency management in replicated databases [5], failure detection [6], system monitoring [7], and distributed publish-subscribe brokers [2].

Strictly speaking, in order to select a gossip peer at random among all system nodes, each node would be required to know the entire system membership. Unfortunately, this solution is not scalable, not only due to the large number of nodes that may constitute the membership view, but also due to the cost of maintaining the complete membership up-to-date in face of the typical dynamics of large-scale systems. To overcome this problem, several gossip protocols rely on *partial views* [8, 2, 9, 10] instead of full membership information. A partial view is a small subset of the entire system membership. In each gossip step[1] a node extracts a random sample of peers as targets for gossip messages. The aim of a *membership service* (also called a peer sampling service [11]) is to maintain these partial views in such a way that a random selection of gossip peers from partial views approximates the random selection from the full membership.

Unfortunately, if each node has only a partial view, the system becomes more vulnerable to the effect of node failures. In particular, if a large number of nodes fail, the partial view of a node may include only failed nodes, therefore the network may become disconnected and several nodes may become isolated. Even if the network remains connected, if failed nodes are select as targets for gossip, specially in early gossip steps, the reliability of the gossip broadcast is severely disrupted. Also, the membership service may take a long time to restore partial views, with a negative impact on the reliability of all messages disseminated meanwhile. Thus, the quality of partial views is of paramount importance for the operation of gossip protocols.

A disadvantage of gossip-based broadcast protocols is that they incur in an excessive message overhead in order to enforce high reliability. This is not the case with structured broadcast protocols, such as the ones that rely on tree-construction but, on the other hand, structured protocols are very fragile in the presence of failures, lacking the natural resilience of epidemic protocols. A promising approach is to develop new broadcast primitives which combine gossip-based and tree-based approaches; in such a way one can benefit from the scalability and resilience of pure gossip-based and approximate the efficiency of the broadcast mechanism to that of tree-based solutions.

In this chapter we start by describing gossip protocols and their relation with unstructured overlay networks. We also discuss several strategies used in the development of gossip protocols and the characteristics of the unstructured overlay networks that are used to support those strategies. We then describe two recently proposed gossip-based protocols: a protocol to build an highly resilient unstructured overlay network, called HyParView, and a broadcast protocol that approaches the efficiency of structured tree-based protocols, called Plumtree. Experimental results show that when combined, these protocols allows one to build a broadcast primitive which is more resilient and more efficient than other gossip-based broadcast protocols.

## 2 Gossip-Based Broadcast

The basic idea behind gossip-based broadcast protocols is to have all participants in the protocol collaborating equally 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 detailed later in section 2.1). Upon receiving a message for the first time, a node repeats this process: by selecting $t$ gossip targets at random 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. The history of message identifiers may grow indefinitely during the execution of the protocol, unless some purging scheme is applied to garbage-collect obsolete entries. The challenge of selecting an adequate strategy to purge message histories is out of the scope of this chapter, and it has already been addressed by previous work, for instance in [12].

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 Parameters

Gossip protocols can be parameterized in order to better control their operation. The two most relevant parameters associated with the configuration of gossip protocols can be described as follows.

[1]We consider that a gossip step happens each time a node transmits (or retransmits) a broadcast message to a set of peers.

**Fanout:** This is the number of nodes that are selected as gossip targets by a node at each gossip step in order to retransmit the message. There is a trade-off associated with this parameter between desired reliability level and redundancy level of the protocol. High fanout values ensure higher levels of fault tolerance level (increasing the probability of reaching all nodes in the system, or in other words, of atomic delivery as defined in [4]) but also generate more 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 a value of zero - which is increased each time a node retransmits 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 set to some integer value (higher than 0), effectively limiting the maximum hops executed by each message in the overlay[2].

The reader should notice that by limiting the number of maximum gossip rounds one can also limit the maximum number of receivers. For instance, if the message is forwarded by flooding, in order to reach the entire system the maximum number of rounds must be set to a value equal or higher than the network diameter.

There is an inherent trade-off between reliability and redundancy level associated with the use of maximum number of rounds attribute. In unlimited mode (or configuring the *maximum rounds* parameter with high values) there is a higher probability of achieving atomic delivery but, on the other hand, more redundant messages are produced.

## 2.2 Strategies

We distinguish the following three basic approaches to implement a gossip protocol. These different strategies are distinguished by the way a message is disseminated to neighbors, and who initiates the gossip exchange: receiver or the sender.

**Eager push approach:** Nodes send the full payload to random selected peers as soon as they receive a message for the first time. This is an approach initiated by the sender.

**Pull approach:** Periodically, nodes query random selected peers for information about recently received or available messages. When they become aware of a message they did not received yet, they explicitly request to that neighbor the payload of that message. This is a strategy that works best as a complement to a best-effort broadcast mechanism (*i.e.* IP Multicast [13]).

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

The reader should notice that there is a trade-off when selecting eager push, pull, or lazy push strategies. Eager push strategies produce more redundant traffic but they also achieve lower latency than the remaining strategies (that require at least an extra round trip time to produce a delivery). From a latency perspective, lazy push gossip is very similar to pull gossip.

[2]Neighboring relations between nodes form an overlay network, as it will be discussed in Sect. 2.4.

Another important practical aspect to retain is that, contrary to pull/lazy push gossip, eager push gossip is not required 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.

These basic strategies can also be combined to develop more complex gossip strategies, that usually, try to benefit from the strong aspects of each individual strategy. As the reader can guess, the number of possible combinations is very high, as one can use different algorithms to switch from one strategy to the other. In the following, we depict two of these hybrid strategies.

**Eager push and pull approach:** Gossip is executed in two distinct phases. A first phase uses push gossip to disseminate a message in a best-effort manner to a maximum of participants (in this phase, the configuration of the push gossip can be somewhat conservative). A second phase of pull gossip is used to recover from omissions that may occur in the first phase. The idea is to lower the amount of redundancy of the gossip process, without decreasing its efficiency. However, the use of pull gossip for recovery increases the global delivery latency.

**Eager push and lazy push:** Gossip is executed by applying eager push to a sub-set of the gossip targets of each node. The selection of the sub-set of peers can be made using several strategies (examples can be found in [14] and [15]). Lazy push is used in the remaining gossip targets to recover from omissions and ensure the gossip properties.

## 2.3 Peer Sampling Service

A *peer sampling service* (as described in [11]) is a service that allows nodes to know a sub-set of the full group of nodes executing the protocol. The interface of this service is quite simple and may only export the following two methods:

**INIT():** This method initializes the service, if it has not been initialized before. If node $p$ returns from the INIT() call, there should be a non-zero probability of other participating nodes getting $p$ as a return value when calling the GETPEER() method.

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

The GETPEER() method is enough to support the operation 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 be redefined as:

**GETPEER($n, veto$):** Where $n$ is an integer greater than zero and *veto* is a node identifier. This method returns a list with, at most, $n$ identifiers of participating nodes that does not contain the *veto* 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 *veto* should be the identifier of the node who sent the message to the invoking node[3].

The semantics of GETPEER($n, veto$) can be informally described as: Return a sample of at most $n$ peer identifiers in the system which does not contain the identifier of either the invoking peer or the peer identified by *veto* (if *veto* is not *null*).

[3]When a node wishes to send a message by the first time, the *veto* argument takes the *null* value.

## 2.4 Partial View

A *partial view* is a set of node identifiers provided to each node. This set should be much smaller than the full system membership. The size constraint is related with scalability requirements. A viable strategy to promote scalability, if for the partial view size to grow logarithmic with relation to the total number of processes in the system. Typically, an identifier is a tuple $(ip : port)$ that allows a node to be reached. In some cases, such as in DHT's, the identifier is a random bit string which is used both to map the nodes in a specific position of the overlay and to support routing [16, 17].

A membership protocol is in charge of initializing and maintaining the partial views for 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. Furthermore, the new node has to build its own partial view, including identifiers of peers already in the system. Also, if a node fails or leaves the system, its identifier should be removed from all partial views as soon as possible.

Partial views 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 that contains the target node in his partial view.

The favored 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.

## 2.5 Strategies To Maintain Partial Views

We identify two main strategies that can be employed to maintain partial views, namely:

**Reactive strategy:** In this type of approach, a partial view is only updated in response to some external event that affects the overlay (*i.e.* a node joining or leaving the system). In stable conditions, partial views remains unaltered. Scamp [18, 19] is an example of a reactive algorithm[4].

**Cyclic strategy:** In this type of approach, a partial view is updated every $\Delta T$ 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 (which is further discussed in Section 2.8.1) is an example of a cyclic algorithm.

Reactive strategies usually rely on some failure detection mechanism to trigger the update of partial views when a node fails. If the failure detection mechanism is fast and accurate, reactive mechanisms can provide faster response to failures than cyclic approaches. This approach is also more efficient as it avoids the communication overhead required by the cyclic strategy to update views.

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, making this strategy more close, in nature, to the original intuition of gossip protocols.

## 2.6 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, that can be used to measure the quality of partial views, are intrinsically related with graph properties of the overlay they define. Some relevant properties are:

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

**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[5]. If this property is not met, isolated nodes will not be able to receive or send 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; it provides a measure of the reachability of a node in the overlay and also affects the probability and the maximum amount of redundant messages that *n* can receive. 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. As the reader could expect, the average path length is affected by the clustering coefficient. A high clustering coefficient will increase the average path length.[6]

**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. 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.

## 2.7 Performance Metrics

We now introduce three metrics that we will use to evaluate the performance of gossip-based broadcast protocols.

**Reliability:** Reliability is defined as the percentage of active nodes in a system which delivers a given broadcast message. A reliability value of 100% is indicative that the broadcast protocol was successful in delivering

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

[6]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.

a given message to all active nodes or, in other words, that the broadcast process resulted in an atomic broadcast as defined in [4].

As the reader should expect, the goal of a gossip-based broadcast protocol is to obtain a reliability of 100% despite network omissions or node failures.

**Relative Message Redundancy (RMR):** The relative message redundancy, or simply RMR, is a metric that captures the message overhead in a gossip-based broadcast protocol [14]. It is defined as:

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

where $m$ is the total number of payload messages exchanged during the broadcast process and $n$ is the total number of nodes that delivers the broadcasted message. This metric is only applicable when at least 2 nodes are able to deliver the message.

A RMR value of zero means that there is exactly one payload message exchanged 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 shows a poor network usage. Note that it is possible to achieve a very low RMR by failing to be reliable. Therefore, a broadcast protocol should aim to combine low RMR values with high reliability values. 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 the value of this tends to $fanout - 1$.

Control messages are not considered by this metric. The reason behind this decision is twofold: first, control messages are typically much smaller than payload messages and thus, they are not the main source of contribution to the exhaustion of network resources; secondly, control messages can usually be sent using delay and piggyback strategies, providing a better usage of the available network resources.

**Last Delivery Hop (LDH):** The last delivery hop, or simply LDH, is a metric which measures the round number of the last message which is delivered by a gossip-based broadcast protocol or, in other words, it measures the maximum number of hops performed by a message successfully delivered. Naturally, a gossip-based broadcast protocol should aim at minimizing this metric.

This metric in intuitively related with the diameter of the overlay network used to disseminate messages. Moreover, it can also provide some comparative measure relatively to the latency of a gossip-based broadcast protocol. When all links exhibit the same latency, the latency of a gossip broadcast is simply the last deliver hop multiplied by the *per hop* latency.

## 2.8 An Overview of Existing Protocols

Several gossip-based protocols have been proposed in the literature. In this section we describe some of these works, addressing three different types of protocols. We start with gossip based membership protocols, that build and maintain unstructured overlay network, which serve as substrate for the operation of gossip-based broadcast protocols. We then present a few gossip-based broadcast protocols.

We conclude by surveying application level multicast protocols that rely in some sort of tree-like structure to support data dissemination. Some of these protocols are not gossip-based broadcast schemes. However, they rely in peer-to-peer structured multicast solutions, being a complementary approach to gossip-based broadcast protocols.

### 2.8.1 Gossip-based Membership Protocols

**Scamp** Scamp [18, 19], is a reactive membership protocol that maintains two separate views, a *PartialView* from which nodes select their gossip targets, 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.

When a new 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 receiving 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 an 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. An unsubscribing node, say $n_u$, will send to some of its peers[7] in the *InView* a *replace request* containing an 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 un-subscription 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 (eventually) purge these identifiers from the *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 containing 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 expiration of the *lease time* of its last subscription. 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.

**Cyclon** Cyclon, is a cyclic membership protocol where nodes maintain a fixed length *partial view*. The size of the *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.

This protocol relies in a *shuffle* operation which is executed every $\Delta T$ 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

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

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.

The interested reader can refer to Chapter **??** where Cyclon is addressed in more detail.

### 2.8.2 Gossip-based Broadcast Protocols

**Bimodal Multicast** Bimodal multicast [1] was one of the pioneer works to combine tree-based best-effort multicast and gossip-based primitives to build reliable multicast in large scale distributed systems. The approach works as follows: in a first phase, broadcast messages are disseminated using the unreliable IP-Multicast [13] primitive; in a second phase, participants engage in gossip exchanges in order to mask omissions that may occur during the first phase. In the second phase, nodes periodically select a peer with whom they exchange a summary of (recently) received messages unique identifiers. After this exchange, nodes explicitly request messages that they miss.

This approach has two major drawbacks. One is that it depends on the availability of IP-multicast, which is not widely deployed in large-scale [20]. To overcome this limitation one could envision to replace IP-multicast by some application-level multicast protocol. Still, a second drawback persists: the approach requires the use of two distinct protocols and therefore, it may present undesired complexity. Moreover, the existence of two distinct protocols may limit the application of optimizations that are only possible if both phases are integrated in a single protocol. Due to these drawbacks, more recent work has favored the use of pure gossip approaches [18, 21].

**NeEM** NeEM, or Network Friendly Epidemic Multicast [22], is a gossip protocol that relies on the use of TCP to disseminate information across a gossip-based overlay. In NeEM, the use of TCP aims at eliminating correlated message losses due to network congestion. The authors show that improved gossip reliability can be achieved by leveraging on the flow control mechanisms of TCP.

NeEM controls buffer management directly, by disabling TCP buffers, and applies several purging strategies to discard messages when overflow occurs. This enables the gossip protocol to preserve throughput stability, even 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 gossip-based. The membership construction and maintenance is based on random walks in the overlay, with a probabilistically length dependent on a protocol configuration parameter $p$ whose value is fixed a priori. Random walks are used both when a node joins the overlay and in a cyclic manner, to advertise neighbors to random nodes. Nodes which receive these advertisements, will integrate received peers identifiers in their own partial views, replacing random peers if their views are full.

**CREW** CREW [8] 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 epidemic dissemination process. 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 by the protocol operation.

CREW uses an underlying membership service, also based on partial views, called Bounce. Bounce is briefly presented in [23] 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.

### 2.8.3 Structured Application-level Multicast

**Narada** Narada [24] is an efficient application-level multicast protocol based on dissemination trees that are constructed in two distinct steps. In the first step, the protocol creates and maintains a random and rich connected overlay (that the authors name *mesh*) which attempts to ensure that: (i) the quality of paths between any two nodes in the overlay is comparable to the quality of the unicast path between the same pair of nodes[8] and; (ii) each node has a limited number of neighbors. The overlay is self-organizing and self-improving, adapting itself to changing network conditions to preserve efficiency, by using a set of heuristics that add or remove links among 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.

Narada is targeted toward medium sized groups; all nodes maintain full membership lists and some additional control information for all other nodes, and consequently it does not scale to very large (and dynamic) 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).

**Scribe** Scribe [25, 26] is a scalable application-level multicast infrastructure built on top of a structured overlay network, named Pastry [16]. 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 both as *rendez-vous point* and as a root for the multicast tree. This node is selected, and can be found by other nodes, taking advantage of Pastry key-based resource location mechanism (using the multicast group name). The keys identifying rendez-vous points can be obtained locally at every node by applying a hash function to the multicast group name. 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 group, thus constructing a distribution tree.

Tree repair is performed as follows: Intermediate nodes periodically send HEARTBEAT messages to nodes which 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 route another JOIN message, that is used to set-up another route to the rendez-vous node, effectively healing 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 group by re-sending JOIN messages.

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

**MON** MON [27], 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[9]. 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 an 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

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

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

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 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, gaining one more parent node).

Because MON is aimed at supporting short-lived interactions, it is not required to maintain these structures for prolonged time. Therefore, it does not owns 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 ensures probabilistic atomic broadcast guarantees.

## 3 The HyParView Protocol

In this section we describe the *Hy*brid *Par*tial *View* membership protocol, or simply *HyParView*[9]. The motivation behind the design of HyParView is based on the following two observations:

- The fanout of a gossip protocol is constrained by the desired reliability and fault-tolerance levels of the protocol. When partial views are used, the quality of these views has an impact on the fanout required to achieve high reliability.
- High failure rates may have a strong impact on the quality of partial views and disrupt the broadcast protocol. Even if the membership protocol has healing properties, the reliability of broadcasted messages after heavy failures may be seriously affected. Therefore, gossip-based broadcast protocols would strongly benefit from membership protocols with fast healing properties.

Leveraging on the observations above, HyParView is based on the two following complementary techniques:

**Symmetric partial views:** which allow a node to have some control over its in-degree. Notice that, unlike other membership protocols based on partial views, this technique offers two interesting properties: *i)* it establishes an upper bound to the in-degree of every node, and *ii)* if every node in the system has a full partial view (*i.e.* if the out-degree of all nodes is equal to the size of partial views), then every node in the system is known by exactly the same amount of other nodes (*i.e.* every node has the same in-degree).

**Partial views with a size equal to the fanout $+1$:** which provide some determinism on the selection of gossip targets without changing the behavior of gossip protocols. This ensures that, as long as the overlay is connected, and every node has at least one correct node in its partial view, every node in the system will receive every disseminated message, as the dissemination process implicitly becomes a flooding in the overlay.

These techniques allow to use smaller fanout values while ensuring a high reliability. To reduce the time required to recover the accuracy of partial views, HyParView employs two additional techniques:

**Unreliable low-cost failure detector:** providing a timely detection of failed peers in the partial view. By immediately removing failed peers from the active view, one avoids to select such peers as gossip targets. Moreover, this allows a timely replacement of failed peers by correct ones, improving the overall connectivity of the overlay network.

**Maintain a backup partial view:** that is used to fill the main partial view (*e.g.* the partial view used to select gossip targets) when it contains free slots, for instance, after some peers had been detected as failed. This allows to improve the time required to recover the connectivity degree of the overlay network.

HyParView uses TCP as the unreliable low-cost failure detector. TCP is appropriate because, as described previously, we rely in symmetric partial views to establish deterministic communication patterns among peers. The use of TCP actually simplifies the implementation of these symmetric relations. Moreover, as initially noted and proposed in [22], TCP is a network friendly protocol which, unlike UDP, avoids the exhaustion of the network resources. Finally, TCP also allows to use more conservative fanout values, as the gossip protocol is no longer required to mask network omissions.

The use of a second and larger partial view, as a repository of backup nodes, also allows to address the fault-tolerance issues that arise due to the use of a small partial views. Although a smaller partial view increases the probability of a node becoming disconnected from the overlay due to faults, this limitation is circumvented by relying on the second partial view, which increases the probability of a node to contain information about correct nodes and quickly reconnect to the overlay.

Therefore, the HyParView protocol maintains two distinct views at each node. A small symmetric active view of size $fanout+1$ (one link from which a message is received and *fanout* links to which the message is relayed). A larger passive view, that ensures connectivity despite a large number of faults and that must be larger than $log(n)$. Note that the overhead of the passive view is small, as no TCP connections are kept open.

Active views of all nodes create an overlay that is used for message dissemination. Links in the overlay are symmetric: if node $q$ is in the active view of node $p$ then node $p$ is also in the active view of node $q$. As we have stated before, the architecture of HyParView assumes that nodes use TCP to exchange messages in the overlay. 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. When a node receives a message for the first time, it forwards the message to all nodes of its active view (except, obviously, to the node that has sent the message). Therefore, the gossip target selection is deterministic in the overlay. However, the overlay itself is created at random, using the gossip membership protocol.

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 fail (or leave). Notice that each node tests its entire active view every time it forwards a message. Therefore, the entire overlay is implicitly tested at every broadcast, which allows a very fast failure detection.

HyParView relies in a join mechanism based on fixed size random walks on the overlay. The join mechanism allows to add the identifier of a new node both to the active views and passive views of some random peers. This is achieved by adding the new node identifier: *i)* to the passive view of nodes located in a middle point of random walks; and *ii)* to the active view of nodes where random walks end.

Additionally, the passive view is maintained using a cyclic strategy. Periodically, each node performs a shuffle operation with one random node in the overlay network, found through a random walk, which will result in the update of both nodes passive views.

One interesting aspect of our shuffle mechanism is that the identifiers exchanged in the shuffle operation are not extracted only from the passive view: a node also sends its own identifier and some nodes collected from its active view to its neighbor. This increases the probability of having nodes that are *active* in the passive views and ensures that failed nodes are eventually expunged from all passive views.

## 4 Achieving Resilient Broadcast

The HyParView peer sampling service described in the previous section can be used to implement an efficient and resilient broadcast protocol just by relying on a eager push strategy to flood the messages in the overlay network defined by the active views. 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. As discussed before, this ensures that all nodes will receive every broadcast message as long as the membership protocol is able to maintain the overlay connected. The reader should notice that the algorithm can work in the "Infect and Die" model [28], as it only relays messages to other nodes upon the reception of each gossip message for the first time.

This strategy is only viable because HyParView maintains an active view that has a small degree[10]. The degree of the overlay will determine the relative message redundancy of the protocol in stable environment, for 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 tend to a value of 3.

The combination of flooding, generating some amount of message redundancy, with the mechanisms of HyParView, allows the protocol to: *i)* contrary to other gossip-based approaches, maintain a constant reliability of 100% in the presence of simultaneous node failures as high as 20%; *ii)* lower the number of hops required by the gossip-based broadcast protocol, to disseminate a message, to the lowest value allowed by the overlay diameter. This happens because when flooding the overlay, the protocol uses all available links between nodes, implicitly using all overlay shortest paths among peers to disseminate a message (independently of the sender).

Finally, 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.

## 5 Building Low Cost Spanning Trees

We now describe the Plumtree protocol[14], a gossip-based dissemination protocol that can exploit the special properties of the unstructured overlay network provided by HyParView, in order to efficiently build and maintain a highly resilient spanning tree, embedded in the original overlay, which allows to improve the cost of the gossip-based dissemination scheme.

The eager push strategy presented in Sect. 4 allows to obtain a high reliability while ensuring the smallest possible value of last delivery hop. Unfortunately in a stable environment it still produces a significant RMR (relative message redundancy) value[11]. An intuitive approach that could help to further reduce the RMR value is to use a structured overlay that establishes a multicast tree covering all nodes in the system. This is the motivation of this protocol, which was originally described in [14] and 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 that 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. The goal is to provide a tree construction mechanism 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. Additionally, this process should recover lost messages, increasing the overall reliability of the broadcast process.

[10] In fact, as stated before, HyParView was originally designed to support this specific strategy.

[11] 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.

Consider that the eager push strategy presented in Sect. 4 is used to flood the HyParView overlay from a single source. In stable conditions[12], it is likely that messages triggering deliveries to the application layer (*i.e.*, messages that are received at a node for the first time) are usually received through the same incoming link (*i.e.* from the same neighbor). Together, these links form a spanning tree that connects all nodes to a the source node (or root). All other links in the overlay are redundant, and are only required to mask node failures. Therefore redundant links can be *pruned* (removed) from the overlay for as long as no failure exists.

The basic idea behind the Plumtree protocol derives from this simple concept. The operation of Plumtree combines the basic flooding process with a *prune* process. Initially all links in a random (connected) overlay are considered as being part of the broadcast tree. Payload messages are only sent to those links (neighbors). Whenever a redundant (payload) message is received a PRUNE message is used to remove the link, used to transmit that message, from the tree.

Although this explain how tree construction may be performed, it does not address tree repair. Without additional measures, 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 technique based in a lazy push gossip strategy is employed. This technique enables nodes that do not receive some messages (due to a failure in the tree) to retrieve those messages from other neighbors and trigger the creation of new links to the spanning tree, reconnecting the tree. This procedure enables the whole spanning tree to be repaired. Thus, nodes are required to announce messages they receive through the links of the overlay that are not part of the spanning 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 new branch of the spanning tree. The interested reader can find more detail on the properties of this mechanism in [14].

Plumtree leverages in the reactive nature of the underlying peer sampling service, which allows to have a stable spanning tree structure in steady state. Moreover, to speed up the recovery process of the spanning tree, whenever a new node is added to the partial view of the underlying peer sampling service, the new link is immediately added to tree. Redundant branches that are created due to this procedure, are removed by the tree construction mechanism upon the dissemination of a broadcast message.

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

In summary, the Plumtree design is based on the following principles:

- It constructs a spanning tree on top of HyParView[13] that is optimized for systems with a single source. Nevertheless, the resulting tree is shared and can be used by any node to broadcast messages.

- The construction of the tree is based on the combination of an eager push algorithm and a pruning process. Because paths in the tree are selected using messages sent by a root node, our tree construction can be classified as a source-based strategy.

- The repair of the tree is based on 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 remaining links. Whenever a node requests a message it 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.

The tree built by Plumtree is optimized for a specific sender: the source of the first broadcast that is used to prune the redundant links from the original overlay in order to form the spanning tree. As a result, the overlay

[12]Stable conditions in this context concerns not only to no changes in the membership protocol, but also to a network where links have a low variance in behavior, such as latency or bandwidth.

[13]Although Plumtree was developed to leverage on the properties of HyParView, it is not limited to the use of this peer sampling service.

links that form the spanning tree are those that were faster to propagate the message disseminated by that specific node. 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 protocol needs to be executed.

## 6 Experimental Evaluation

### 6.1 Experimental Setting

All simulations described in the section were conducted using the PeerSim Simulator [29]. 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 protocols (those results are not presented here, as this is not the main focus of the chapter).

A modified version of Cyclon, named CyclonAcked, is also evaluated. This version adds a failure detection system to Cyclon, based on the exchange of explicitly acknowledgments during message dissemination. Thus, CyclonAcked is able, in less time, to detect failed nodes when it attempts to gossip to them and, therefore, to perform a faster removal of such elements from partial views, increasing the accuracy of the original Cyclon after failures. This benchmark is used to show that the benefits of the HyParView protocol are not only derived from the use of a reliable transport (and also, 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).

The eager gossip broadcast protocol was configured so that, when combined with HyParView, it implements the flood gossip strategy described previously, in Sect. 4. For fairness, the same configuration parameter was used in all membership protocols.

An implementation of the Plumtree protocol was also developed for the PeerSim simulator. This implementation relies in HyParView as the underlying membership protocol, as it was originally 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. For simplicity, and similarly to the Cyclon protocol, a single node as contact was used to create the HyParView overlay.

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. Each simulation cycle is composed, at most, of 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 only executes at a predefined cycle (or cycles) of a given simulation.

**Broadcast:** In this step a number of nodes[14] 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 correct nodes (*e.g.* nodes which have their internal state set to *Up* in the current simulation cycle).

**Membership:** In this step, the membership protocol executes any *cyclic* (*e.g.* periodic) 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).

Although most simulations follow the structure above, 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 is not required.

## 6.2 HyParView and Eager Push Strategy

As noted in Sect. 2.6, the overlays produced by membership protocols, or peer sampling services, should exhibit some relevant properties such as low clustering coefficient, small average shortest path, and balanced in-degree distribution, to contribute for a fast message dissemination and a high level of fault tolerance in gossip-based broadcast protocols. 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 1. Graph properties after stabilization**

Table 1 shows values concerning average clustering coefficient, average shortest path, and last delivery hop for all protocols after a period of stabilization of 50 membership cycles[15]. It can be seen that in terms of average clustering coefficient, HyParView achieves significantly lower values than Scamp or Cyclon. This is not surprising given that HyParView's active view is much smaller than other protocols partial views. Nevertheless, this feature is an important factor, which contributes to the high resilience that HyParView exhibits to node failures.

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

[15]In fact, this stabilization time is not required by Scamp, as it stabilizes immediately after the join period. HyParViews active view also stabilizes within 1 to 2 simulation cycles, but its passive view requires some rounds of membership in order to stabilize completely.

In terms of average shortest path, it is clear that HyParView falls behind Scamp and Cyclon. This is no surprise, as HyParView maintain a much 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, as can also be seen in Table 1.

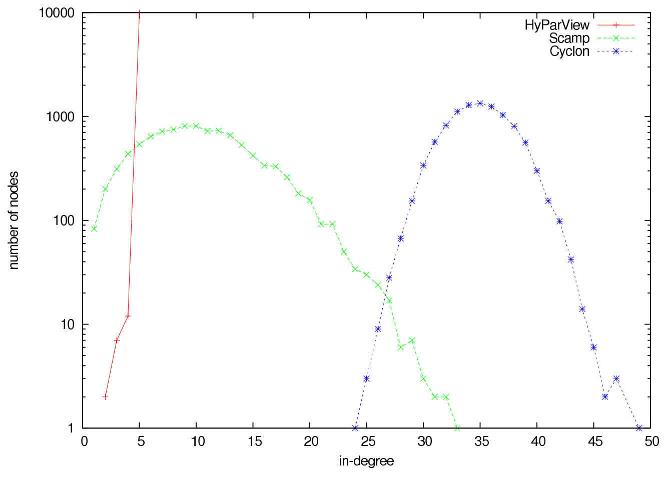

**Figure 1. In-degree distribution**

Figure 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 larger probability of receiving redundant messages, while other nodes have a very small probability of receiving messages even once. Naturally, nodes which are known by only a few other peers, have a smaller probability to be selected as gossip targets. Also, such 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 node is smaller. This is particularly 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 size (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[16], and that the number of nodes in these conditions is marginal (with only 2 in 10.000 nodes).

We now evaluate the impact of massive failures in the reliability of the eager push gossip broadcast, when combined with different membership protocols.

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. A measure of the reliability of 1.000 messages sent from random correct nodes was then extracted. All these messages were sent before the execution of any cycle of a membership protocols. 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

[16]Obviously, this also means that such nodes have an out-degree of 2.

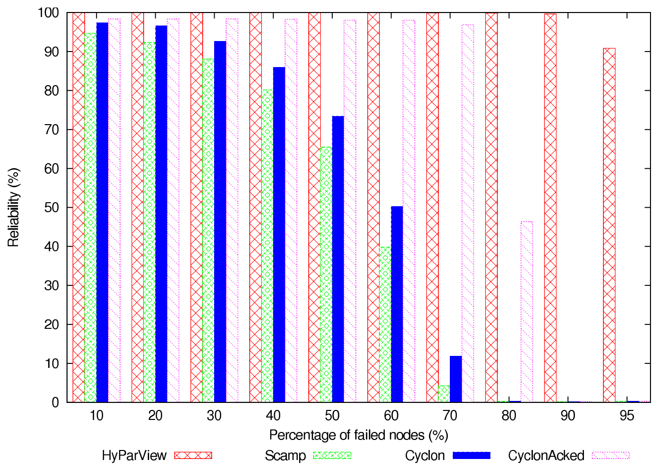

**Figure 2. Average reliability for 1000 messages**

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 is depicted in Fig. 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 90%. Both Scamp and Cyclon exhibit a constant reliability[17] for failure percentages as low as 10%, and their performance is significantly hampered with failure percentages above 40% (with reliability values below 50%). 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 behavior 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.

## 6.3 Plumtree

In this section, the Plumtree protocol is evaluated. In order to extract comparative figures, experimental results are shown together with those obtained by the eager push strategy and a pull based gossip strategy in the same scenarios. The Plumtree protocol performs better in scenarios with only one (fixed) sender. Nevertheless, the protocol was evaluated operating in two distinct modes:

**Single 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.

To evaluate the protocol in a stable environment, several 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 using one single node that serves as *contact* for all remaining nodes. No membership cycles are executed during the join process. Simulations are run for a total of 250 simulation cycles. The first 50 cycles of each run are used to ensure stabilization and hence, are 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 then evaluated for each message disseminated.

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

Simulations show that all the gossip-based broadcast protocols were able to achieve and maintain a reliability of 100% in stable conditions. This includes Plumtree operating in the two modes described above. This is expected, given the properties of the HyParView protocol, namely from the connectivity point of view: HyParView ensures the connectivity of the overlay, and all approaches (eager push and pull based strategies along with the tree based strategy) 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 serves as a validation for the design of the Plumtree protocol.

The relative message redundancy (RMR) value (as defined in Section 2.7) is of paramount importance when evaluating the Plumtree protocol, as this is the main metric aimed for optimization by this approach. Table 2 shows aggregated values of relative message redundancy obtained in our experiments for all gossip protocols. All protocols, in stable conditions, were able to maintain a constant value for RMR in the last 200 cycles of every simulation. 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 earlier in the chapter.

| | RMR |
|---|---|
| Eager | 3.00 |
| Pull | 0.00 |
| Plumtree (s-s) | 0.00 |
| Plumtree (m-s) | 0.00 |

**Table 2. Relative message redundancy in stable environment**

In Plumtree, after the stabilization of the protocol, payload messages are only propagated through links that belong to the spanning tree. The Plumtree algorithm eliminates redundant branches from the tree. Therefore, Plumtree does not generate any redundant message. As a result, the RMR value for the Plumtree protocol presents a constant value of 0.

The pull approach is also able to maintain a constant value of 0 for RMR. This is expected, as in a pull approach payload is only transmitted when explicitly requested by the receiver. Therefore, in a stable environment, a single payload message has to be transfered for each receiver. However, such an approach as some drawbacks as we show in the following paragraphs.

| | Payload | Control | Total |
|---|---|---|---|
| Eager | 39984.00 | 0.00 | 39984.00 |
| Pull | 9999.00 | 49994.33 | 59993.33 |
| Plumtree (s-s) | 9999.00 | 29987.33 | 39986.33 |
| Plumtree (m-s) | 9999.00 | 29990.00 | 39989.00 |

**Table 3. Number of messages received**

Table 3 shows the number of messages received by all gossip protocols after 150 cycles of stabilization. The extra control messages received by 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 broadcast message unique identifiers on a single IHAVE message. Notice that aggregation will not have a significant impact in reliability, as messages are still sent, 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 dissemination process.

Notice that the pull approach requires much more control messages than Plumtree. This is expected as the transmission of each payload message requires two distinct control messages. Additionally, several control messages are sent in the dissemination process which are redundant, similar to the payload messages sent by the eager push strategy.

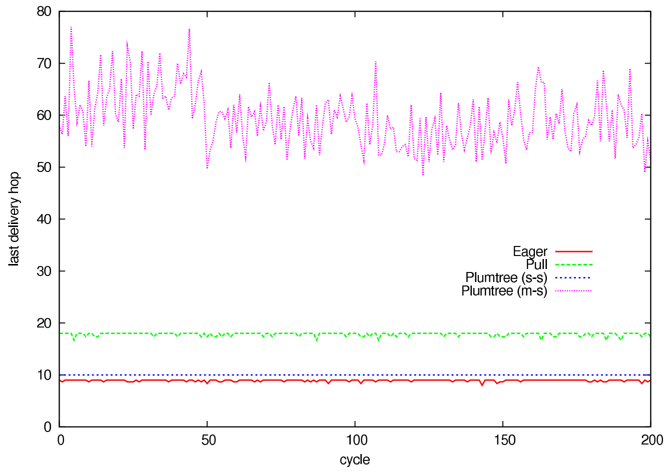

**Figure 3. Last delivery hop in stable environment**

Figure 3 presents the values for last delivery hop (LDH). The eager push protocol and Plumtree with a single sender offer the best performance. 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.

The pull approach doubles the value of LDH when compared with Plumtree and the eager protocol[18]. This happens because this strategy imposes additional delay to the dissemination of payload messages in the overlay. This will have a negative impact in the latency of the broadcast process.

With multiple senders the Plumtree LDH values are naturally higher. This happens because the spanning tree is optimized for the first source. Therefore, when messages are sent by nodes located at leaf positions of the tree, messages are required to perform additional hops in the overlay to reach all remaining nodes.

The interested reader may refer to [14], where an optimization to the Plumtree protocol described here is presented and evaluated. This optimization is able to lower the LDH value when the protocol operates in multiple senders mode. Moreover, the optimized protocol achieves a better distribution of the spanning tree branches across all nodes in the system.

## 7 Discussion and Future Directions

In this chapter we introduced some of the most relevant aspects of gossip-based broadcast protocols. Namely, we have described the basic principles of epidemic protocols, their typical parameters and operation modes. We have also described the main gossip strategies that can be found in literature, such as: eager push, lazy push, and pull; and discussed the associated trade-offs associated with the use of these techniques. To help the reader in assessing the behavior of different gossip-based broadcast solutions, we listed several metrics to evaluate the quality of the unstructured overlay networks created by peer sampling services, discussed their desirable values, and explained how they affect the epidemic dissemination process. We also offer a survey of some representative

[18]Notice that in these experiments we ignore the hop executed by the message explicitly requesting the transmission of payload.

protocols, including gossip-based membership protocols, gossip-based broadcast protocols, and for completeness, protocols based on the construction of spanning trees. Finally, we described and evaluated a family of concrete peer sampling and gossip-based broadcast protocols.

As future directions for research in this field, we consider that the body of work presented in this chapter can be improved in two distinct aspects:

- The algorithms presented in this chapter may be improved by considering the (possible) heterogeneity of nodes in the system. An obvious approach to achieve this is to use an adaptive fanout (and degree) for nodes. This value should be based on a function of the resources available at each node. By using such strategy, a node which is more powerful (*e.g.* which has more memory, or more bandwidth) would contribute more, not only to the maintenance of the unstructured overlay network, but also to the dissemination process (*e.g.* by sending more messages).

- Plumtree-like protocols could benefit from techniques to improved load distribution. For instance, when the current protocol operates in single sender mode, nodes which are located at the leaf positions of the spanning tree, do not contribute (actively) to the dissemination of messages payloads.

## References

[1] Birman, K., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal multicast. ACM Transactions on Computer Systems **17**(2) (1999)

[2] Eugster, P., Guerraoui, R., Handurukande, S., Kouznetsov, P., Kermarrec, A.M.: Lightweight probabilistic broadcast. ACM Trans. Comput. Syst. **21**(4) (2003) 341–374

[3] Hayden, M., Birman, K.: Probabilistic broadcast. Technical report, Ithaca, NY, USA (1996)

[4] Kermarrec, A.M., Massouli, L., Ganesh, A.: Probabilistic reliable dissemination in large-scale systems. IEEE Trans. Parallel Distrib. Syst. **14**(3) (2003) 248–258

[5] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, New York, NY, USA, ACM Press (1987) 1–12

[6] van Renesse, R., Minsky, Y., Hayden, M.: A gossip-style failure detection service. Technical Report TR98-1687, Dept. of Computer Science, Cornell University (1998)

[7] van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems **21**(2) (2003) 164–206

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

[9] Leito, J., Pereira, J., Rodrigues, L.: HyParView: A membership protocol for reliable gossip-based broadcast. In: DSN '07: Proc. of the 37th Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks, Edinburgh, UK, IEEE Computer Society (2007) 419–429

[10] Pereira, J., Rodrigues, L., Pinto, A., Oliveira, R.: Low-latency probabilistic broadcast in wide area networks. In: Proceedings of the 23th IEEE Symposium on Reliable Distributed Systems (SRDS'04), Florianopolis, Brazil (2004) 299–308

[11] Jelasity, M., Guerraoui, R., Kermarrec, A.M., van Steen, M.: 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, New York, NY, USA, Springer-Verlag New York, Inc. (2004) 79–98

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

[13] Deering, S., Cheriton, D.: Multicast routing in datagram internetworks and extended lans. ACM Trans. Comput. Syst. **8**(2) (1990) 85–110

[14] Leito, J., Pereira, J., Rodrigues, L.: Epidemic broadcast trees. In: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'2007), Beijing, China (2007) 301 – 310

[15] Carvalho, N., Pereira, J., Oliveira, R., Rodrigues, L.: Emergent structure in unstructured epidemic multicast. In: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Edinburgh, UK (2007) (to appear)

[16] Rowstron, A., Druschel, P.: 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, London, UK, Springer-Verlag (2001) 329–350

[17] Zhao, B., Kubiatowicz, J., Joseph, A.: Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley (2001)

[18] Ganesh, A., Kermarrec, A.M., L. Massouli e.: SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In: Networked Group Communication. (2001) 44–55

[19] Ganesh, A., Kermarrec, A.M., Massouli, L.: Peer-to-peer membership management for gossip-based protocols. IEEE Trans. Comput. **52**(2) (2003) 139–149

[20] Diot, C., Levine, B., Lyles, B., Kassem, H., Balensiefen, D.: Deployment issues for the IP multicast service and architecture. IEEE Network **14**(1) (2000) 78–88

[21] Voulgaris, S., Gavidia, D., van Steen, M.: Cyclon: Inexpensive membership management for unstructured p2p overlays. Journal of Network and Systems Management **13**(2) (2005) 197–217

[22] Pereira, J., Rodrigues, L., Monteiro, M.J., Oliveira, R., Kermarrec, A.M.: Neem: Network-friendly epidemic multicast. In: Proceedings of the 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03), Florence,Italy (2003) 15–24

[23] Deshpande, M., Xing, B., Lazardis, I., Hore, B., Venkatasubramanian, N., Mehrotra, S.: Crew: A gossip-based flash-dissemination system. Technical report, University of California (2005)

[24] Chu, Y.H., Rao, S., Seshan, S., Zhang, H.: A case for end system multicast. IEEE Journal on Selected Areas in Communications **20**(8) (2002) 1456–1471

[25] Rowstron, A., Kermarrec, A.M., Castro, M., Druschel, P.: SCRIBE: The design of a large-scale event notification infrastructure. In: Networked Group Communication. (2001) 30–43

[26] Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: SCRIBE: A large-scale and decentralized application-level multicast infrastructure. IEEE Journal on Selected Areas in communications (JSAC) **20**(8) (2002) 1489–1499

[27] Liang, J., Ko, S., Gupta, I., Nahrstedt, K.: MON: On-demand overlays for distributed system management. In: 2nd USENIX Workshop on Real, Large Distributed Systems (WORLDS'05). (2005)

[28] Eugster, P., Guerraoui, R., Kermarrec, A.M., Massoulie, L.: From Epidemics to Distributed Computing. IEEE Computer **37**(5) (2004) 60–67

[29] Jelasity, M., Montresor, A., Jesi, G.P., Voulgaris, S.: (The Peersim simulator) `http://peersim.sf.net`.