Technical Report RT/28/2010

Thicket: A Protocol for Building and Maintaining Multiple Trees in a P2P Overlay

Mário Ferreira
INESC-ID / IST
mvvf@gsd.inesc-id.pt

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

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

May 2010

Abstract

One way to efficiently disseminate information in a P2P overlay is to rely on a spanning tree. However, in a tree, interior nodes support a much higher load than leaf nodes. Also, the failure of a single node can break the tree, impairing the reliability of the dissemination protocol. These problems can be addressed by using multiple trees, such that each node is interior in just a few trees and a leaf node in the remaining; the multiple trees approach allows to achieve load distribution and also to send redundant information for fault-tolerance. This paper proposes Thicket, a decentralized algorithm to efficiently build and maintain such multiple trees over a single unstructured overlay network. The algorithm has been implemented and is extensively evaluated using simulation in a P2P overlay with 10.000 nodes.

Sections of this technical report were published in the proceedings of SRDS 2010.

Thicket: A Protocol for Building and Maintaining Multiple Trees in a P2P Overlay

Mário Ferreira
INESC-ID / IST
mvvf@gsd.inesc-id.pt

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

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

Abstract

One way to efficiently disseminate information in a P2P overlay is to rely on a spanning tree. However, in a tree, interior nodes support a much higher load than leaf nodes. Also, the failure of a single node can break the tree, impairing the reliability of the dissemination protocol. These problems can be addressed by using multiple trees, such that each node is interior in just a few trees and a leaf node in the remaining; the multiple trees approach allows to achieve load distribution and also to send redundant information for fault-tolerance. This paper proposes Thicket, a decentralized algorithm to efficiently build and maintain such multiple trees over a single unstructured overlay network. The algorithm has been implemented and is extensively evaluated using simulation in a P2P overlay with 10.000 nodes.

1 Introduction

Mechanisms to support the dissemination of information in a reliable and efficient manner, to a very large number of participants, are extremely relevant for a wide range of applications, ranging from large scale monitoring and control infrastructures [15] to live video streaming and IP Television (IPTV) services [9].

This paper addresses the described problem by proposing a peer-to-peer dissemination mechanism that relies on the cooperation among all participants (as opposed to solutions that assume the availability of an underlying IP-multicast service). The peer-to-peer approach has already proved successfully in circumventing the difficulties faced when attempting to deploy global IP-multicast support [5, 6].

More precisely, we aim at mechanisms that allow to build multiple trees on top of an unstructured overlay, connecting a data source and a large number of recipients. Tree-based solutions are appealing because they promote an efficient usage of available resources, namely by avoiding the redundancy of approaches such as flooding or gossip. However, in a tree, interior nodes support a much higher load than leaf nodes. Also, the failure of a single interior node is able to break the tree compromising the reliability of the dissemination protocol. These problems can be addressed by using multiple trees, such that each node is interior in just one, or few, trees and a leaf node in the remaining; multiple trees allow to achieve load distribution and also to send (controlled amounts of redundant) information on different trees for fault-tolerance. The introduced redundant data is useful in applications such as live streaming, for instance by leveraging on network coding techniques it is possible to split the original data stream in several slices and send these slices through different trees. These slices might encode enough redundancy such that if a node temporarily misses messages from one of the trees, it is still able to decode the original stream using the remaining slices received from the remaining trees.

This paper motivates, describes, and evaluates Thicket, a novel decentralized algorithm to efficiently build and maintain such multiple trees over a single overlay network. As it will become clearer in the following section, Thicket addresses a relatively unexplored region of the design space, by building multiple trees in a decentralized manner, on top of an *unstructured* overlay. Unstructured peer-to-peer overlays are more robust to system dynamics than structured solutions, such as Distributed Hash Tables (DHTs), as they pose much less constraints of the overlay topology. Thicket has been implemented and has been extensively evaluated using simulation in a P2P overlay with 10.000 nodes.

The remaining of this paper is organized as follows. Section 2 motivates our work by making a survey on competing approaches and by discussing their advantages and limitations. Then, to illustrate the challenges in our design, in Section 3, we demonstrate the limitations of some naive "nuts and bolt's" approaches to the problem. Thicket is presented and described in detail in Section 4, while Section 5 provides experimental results. Finally, Section 6 concludes the paper.

2 Related Work

There are mainly three basic approaches for achieving large-scale information dissemination in peer-to-peer systems: the *gossip* approach, the *tree* approach, and the *embedded tree* approach:

- The gossip approach consists in letting the source select f peers at random from the system (this is a configuration parameter called *fanout*) and sending the message to them. Upon the reception of a message for the first time, each node simply repeats this procedure. This approach (illustrated by protocols such as[2] or [7]) is simple, highly scalable, and robust. Unfortunately, gossip protocols are not resource efficient, as their robustness derives from a significant amount of redundancy.

- The tree approach consists in having participants coordinating among themselves in order to build an overlay with the topology of a fault-tolerant tree. An example of this approach can be found in [8]. The main advantage of a tree approach is resource efficiency, as the topology avoids unnecessary redundancy in the dissemination process. Unfortunately, a tree is hard to maintain in face of high dynamics, therefore this solution is not efficient for very large systems subject to churn (*i.e.* constant filiation changes due to concurrent node departure and arrival).

- The embedded tree approach consists of using efficient mechanisms to build an embedded tree over an existing overlay [13, 18]. The overlay maintenance is delegated to some existing protocol.

In this paper *we are interested in the embedded tree approach*, as these solutions typically are able to combine the best features from the pure gossip and the pure tree approaches (as detailed in [13]).

Note that the embedded tree approach can be applied both to structured and unstructured overlays. An example of the former is Scribe [18], that builds trees on top of the Pastry DHT [17]; examples of the later can be found in [13, 1]. Solutions based on unstructured overlays are more appealing as they have the potential to be more robust in face of system dynamics: since unstructured overlays pose less constraints on the topology, they can be repaired faster than structured overlays.

Tree based solutions can be classified in single-tree or multiple-tree solutions. Single tree solutions are naturally simpler but have two main problems: they promote an unbalanced resource usage among peers (nodes that are interior to the tree consume resources to forward data while leaf nodes only receive data); they also suffer from temporary disruptions when one interior node fails and the tree needs to be repaired. Multiple-tree solutions, as the name implies, rely on several trees connecting the same set of participants. Trees are built in such a way that a node is only interior in one or a small subset of all trees and a leaf node in all the remaining. This approach provides load-balancing, as all nodes contribute with their resources (e.g. bandwidth) to forward data. Furthermore, by sending redundant information in some trees (for instance by using network coding techniques[4]), it is possible to achieve higher fault-tolerance: since the failure of a node only disrupts the tree where it acts as an interior node, receivers are still able to operate using the data received from the remaining trees.

Therefore, in this paper *we are interested in approaches that build multiple-trees*. These approaches can be further classified according to the type of algorithm that is used to build the set of trees. Centralized algorithms rely on some specialized nodes, that have a global knowledge of the topology, to build the trees. Note that,

	Centralized	Decentralized	
		structured overlay	unstructured overlay
Single tree	Bayeux [20]	Scribe [18]	Mon [15], Plumtree [13]
Multiple tree	CoopNet [16]	Splitstream [3]	Chunkyspread [19], *THICKET*

Table 1. Thicket in the design space

even a centralized solution is not trivial, as the problem of optimal tree construction is NP-hard [11]. Centralized approaches have little practical interest for very-large scale systems, as they are not scalable and it is hard to make them fault-tolerant.

Therefore, *we are interested in decentralized approaches*. The most relevant examples of a decentralized approach are SplitStream [3] and Chunkyspread [19].

SlipStream leverages on a variant of Scribe to build multiple disjoint spanning trees over the Pastry [17] DHT. Similar to our work, the authors strive to build trees in which a node is interior in a single tree. Additionally the authors propose a scheme that allows nodes to control their degree in the tree where they are interior (*i.e.*, controlling the forwarding load of each node) according to their capacities. Unlike our work, the authors rely on a DHT; nodes are interior in a single tree by design, as each tree is rooted in nodes with identifiers having distinct prefixes. Notice that the overhead of maintaining a DHT is far superior than maintaining an unstructured overlay network. Additionally, the unstructured overlay can potentially recover from failures faster than Pastry: in Pastry a crashed node cannot be replaced by any given node, only nodes with the "right" identifier (accordingly to the DHT organization logic) can be employed for this task. Moreover, the scheme employed by the authors to enforce maximum degree on interior nodes may result in several peers becoming disconnected from the tree with a negative impact on the reliability of the data dissemination protocol. SplitStream also requires additional links between peers in addition to the ones provided by Pastry, which results in additional overhead.

Chunkyspread [19] is a protocol that builds and maintains several spanning trees on top of an unstructured overlay network, while trying to limit the load and degree of nodes accordingly to their capacities. However, Chunkyspread mechanism does not attempt to control the number of trees where a node is interior. This results in trees that are not independent among themselves *i.e.*, where nodes can act as an interior node in several trees. This is clearly an undesirable property from the reliability point of view. In fact, we demonstrate in Section 5 that independent trees are extremely relevant in scenarios where nodes can fail.

In summary, we aim at designing a solution that combines the following features: i) It embeds trees in a peer-to-peer overlay, as this offers a good trade-off between efficiency and robustness; ii) is fully decentralized; iii) is able to build multiple-tree that have few interior nodes in common; and iv) can operate on top of unstructured overlays.

Table 1 illustrates several relevant combinations in the design space for the problem we are addressing, providing some notable examples of solutions for each region. Thicket is the first protocol that not only exploits a relatively unexplored fraction of the design space, that owns several advantages as described above, but also does so while promoting the construction of spanning trees where nodes act as interior mostly in a single tree, contributing to improve the reliability of broadcast schemes.

3 Some Naive Approaches

As noted in the previous section, our goal is to design a decentralized algorithm for building t trees on top of an unstructured overlay. At first sight such a goal may appear to be easy to achieve. In particular, it is tempting to consider an algorithm that is a trivial extension to previous work, namely, the following two alternative solutions appear as natural candidates:

- Since previous work has shown how to build multiple trees on top of a structured overlay, one may consider to use a similar approach on the unstructured overlay. In particular, one could select t proxies of the root at random (for instance, by doing a random walk from the source node), and then build a different tree rooted at each of these proxies. This approach can also be seen as a simplified version of the Chunkyspread protocol. We have named this approach the *Naive Unstructured spliTStream*, or simply, NUTS.

- Since previous work has shown how to build a single tree, in a decentralized manner, on top of an unstructured network, one may also consider the simple solution that consists in running such algorithm t times, *i.e.*, creating t different unstructured overlays and embedding a different tree over each one of these overlays. The intuition is that the inherent randomization in the construction of the unstructured overlays (and of the embedded trees) would be enough to create trees with enough diversity. We have named this approach *Basic multiple OverLay-TreeS*, or simply, BOLTS.

We have implemented these two basic "nuts and bolts" strategies to assess how good they perform in practice. We analyze their resulting performance to extract some guidelines for the design of Thicket.

For these experiments we have used HyParView [14] to build the overlay network. HyParView is a protocol for building unstructured overlays that has the feature of balancing both the in- and out-degrees of nodes in the overlay. Therefore, the topology created by HyParView approximates a random regular graph. This is beneficial to our goals, because it makes load balancing easier. For building the trees we have used the Plumtree protocol [13] . Plumtree embeds a tree in topologies such as the ones created by HyParView.

In order to experiment the NUTS approach, we have constructed a single HyParView overlay and used Plumtree to create t trees rooted at random nodes in the overlay. To experiment the BOLTS strategy we have created t independent instances of the HyParView overlay (by letting nodes join each instance by different random orders) and then embedded a single tree in each of these instances.

We evaluated both strategies by simulating a system composed of 10.000 nodes, and the target of building 5 independent spanning trees (we will describe the experimental setup employed in detail in Section 5). For NUTS we employed a single HyParView instance with a node degree of 25. For BOLTS we configured each of the HyParView instances to have a node degree of 5. The *fanout* value used by the Plumtree instances was set to 5 which is related with the number of neighbors maintained by HyParView for 10.000 nodes [14]. These configurations ensure that each node has an identical number of overlay links in both approaches. Figure 1 plots the percentage of nodes that are interior in 0, 1, 2, 3, 4, and 5 trees.

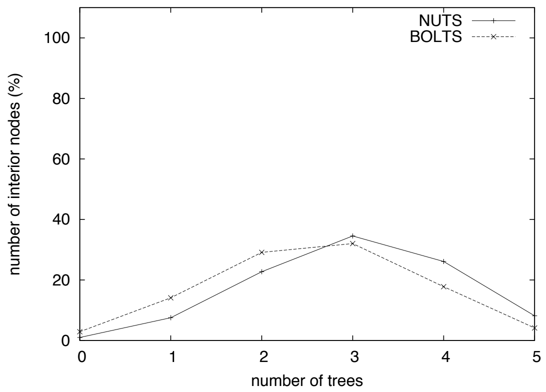


Figure 1. K-interior node distribution.

The figure shows that in both strategies only a small fraction of nodes (between 7% and 17%) are interior in a single tree. The majority of nodes in the system are interior in either 2, 3, or 4 trees (with a small fraction being interior in all trees for both strategies). Notice that, for BOLTS, there are some nodes that do not act as interior nodes in any tree (0). Such nodes do not contribute to the data dissemination process, acting always as free riders.

Algorithm 1: Data Structures & Inititalization

```
data structure Tree
    field activePeers : Set

data structure Load : int[]

upon event Init do
    foreach $t \in trees$ do
        t.activePeers ⟵ $\emptyset$
    backupPeers ⟵ getPeers()
    announcements ⟵ $\emptyset$
    receivedMsgs ⟵ $\emptyset$
    loadEstimate$_p(t)$ ⟵ $\emptyset$
```

This clearly shows that these strategies create (even in steady state) suboptimal configurations, where many nodes are required to forward messages in more than one tree. Additionally, this also indicates that the single failure of a node can disrupt the operation of a significant number, or even all, spanning trees, which clearly compromises the reliability of the data dissemination process.

These results can be explained by the random and uncoordinated nature of tree construction, in which each tree is built in an independent way. In fact, although a large measure of randomness is implicit in the unstructured overlay networks in the BOLTS solution, and the selection of peers is independent in NUTS, there is still a significative probability that nodes can be selected to be interior in several trees.

4 Thicket

4.1 Architecture

Thicket relies on an unstructured overlay network that implements a reactive peer sampling service and exports a symmetric partial view of the system[1]. The peer sampling service is responsible for notifying the Thicket layer whenever there is a change on the partial view of the node using the *NeighborUp*(p) and *NeighborDown*(p) calls.

Thicket operates by employing a gossip-based technique to build T divergent spanning trees (T is a protocol parameter, we discuss the selection of values for T later in the section), where most nodes are interior in a single tree and leaf in all other trees. Furthermore, Thicket uses the remaining overlay links for the following purposes: $i)$ ensure complete coverage of all existing trees *i.e.*, that all nodes in the system are connected to all trees, notice that to ensure this, some nodes may be required to be interior in more than a single tree; $ii)$ detect and recover from tree partitions when nodes fail; $iii)$ ensure that tree heights are kept small, despite the existing dynamics of the system; and finally, $iv)$ that the forwarding load of each participant (for all trees where it operates as an interior node) is limited by a protocol parameter named *maxLoad*.

The *maxLoad* parameter must be low enough in order to limit the forwarding load imposed to each node, avoiding overloading situations. However, if the chosen value is too low, nodes might be unable to coordinate among themselves in order to generate trees with full coverage (*i.e.* that connect all nodes). Following epidemic theory *maxLoad* should be logarithmic with the number of nodes in the system.

Algorithm 1 depicts the data structures maintained by Thicket, as well as its initialization procedure. Each node n in Thicket keeps a set of *backupPeers*$_n$; with the identifiers of the neighbors that are not being used to receive (or forward) messages in any of the T trees. Initially, all neighbors of n are in this set. Additionally, for each

[1]By reactive, we mean that the contents of partial views maintained by nodes are only updated in reaction to external events such as a peer joining or leaving the system. Symmetric means that the resulting overlay denotes an undirected graph

tree t maintained by Thicket, each node n maintains a set $t.\mathit{activePeers}_n$ with the identifiers of the neighbors from which it receives (or forwards to) data messages in t.

Each node n also maintains an *announcements*$_n$ set, in which it stores control information received from peers that belong to the *backupPeers*$_n$ set. This information is used to detect and recover from tree partitions due to node failures or departures. We will later explain in detail how the recovery procedure operates. In order to avoid routing loops, each node also maintains a *receivedMsgs*$_n$ set, with identifiers of messages previously delivered and forwarded by a node.

Finally, in order to balance the load of the nodes, i.e., to ensure that most nodes are only interior in a single tree and to limit the message forwarding load imposed to each participant, each node n keeps an estimate of the forwarding load of its neighbors. For this purpose, every time a node s sends a message to another node, it includes a list of values denoting the number of nodes to which s has to forward messages in each tree[2]. Since this information can be encoded efficiently, it is piggybacked to all data and control messages exchanged between neighbors. This allows every node to keep fresh information about the load of its peers without explicitly exchanging messages just for this purpose. Each node n maintains the most recent information received from its neighbor p for each tree t in the variable *loadEstimate*$(p,t)_n$.

4.2 Tree Construction

Algorithm 2 depicts a simplified version of the pseudo-code for the tree construction procedure. We have omitted some obvious aspects from the pseudo-code (for instance the update of the *loadEstimate*) to improve its readability.

The creation of each tree t is initiated by the source node. To that end, and for each tree t, the source node n selects f nodes at random from the *backupPeers*$_n$ set and moves them to the $t.\mathit{activePeers}_n$ set. After this, the source initiates the dissemination of data messages in each tree t, by sending messages to the nodes in $t.\mathit{activePeers}_n$.

All messages are tagged with a unique identifier, *muid*, composed of the pair (*sqnb*, *t*), where *sqnb* is a sequence number and *t* the tree identifier. The *muid*s of previously delivered (and forwarded) messages are stored in the *receivedMsgs*$_n$ set[3]. Periodically, each node n sends a SUMMARY of this set to all nodes in its *backupPeers*$_n$ set (this messages also include load information used to update *loadEstimate*).

When a node n receives a data message from s in t, it first checks if the tree has been already created locally. The first message that is received in a given tree t triggers the local tree branching procedure for t. The construction step for an interior node is different from the one executed by the source node. First, n removes s from *backupPeers*$_n$ and adds s to $t.\mathit{activePeers}_n$. Furthermore, if $\nexists t' : |t'.\mathit{activePeers}_n| > 1$ (i.e., the node is not interior in some other tree t'), then n moves at most $f-1$ peers from *backupPeers*$_n$ to $t.\mathit{activePeers}_n$. On the other hand, if n is already an interior node in some other tree, it stops the branching process, becoming a leaf node in t.

The data message is then processed. If the message is not found to be a duplicate (by inspecting the *receivedMsgs*$_n$ set), it is forwarded to the nodes in $t.\mathit{activePeers}_n \backslash \{s\}$. On the other hand, if the received message is a duplicate, the node moves s from $t.\mathit{activePeers}_n$ to *backupPeers*$_n$ and sends a PRUNE message back to s. Upon receiving the PRUNE message, s will move n from $t.\mathit{activePeers}_s$ to *backupPeers*$_s$. This procedure results in the elimination of a redundant link from t and removes any cycles created by the the gossip mechanism.

By executing this algorithm, nodes become interior in at most one spanning tree. The algorithm also promotes load balancing (as long as the number of data messages sent through each tree is similar). On the other hand, since our mechanism selects random peers for establishing each tree, there is a non negligible probability that some nodes do not become connected to every tree. Such occurrences are addressed by the a tree repair mechanism described in the following section.

[2]We assume that tree identifiers are sequential numbers starting at zero. This list has a size of T. The number in position t represents the forwarding load of that node in tree t (which is the size of $t.\mathit{activePeers}_n$ minus 1).

[3]For techniques on how to garbage collect obsolete information from this set see for instance [12].

Algorithm 2: Tree Construction

```
upon event Broadcast($m$) do
    tree $\longleftarrow$ nextTree()
    muid $\longleftarrow$ (nextSqnb(), tree)
    if tree.activePeers = $\emptyset$ then
        call SourceTreeBranching(tree)
    call Forward (m, muid, tree, myself)
    trigger Deliver($m$)
    receivedMsgs $\longleftarrow$ receivedMsgs $\cup$ {muid}

upon event Receive (DATA, $m$, muid, load, tree, sender) do
    if muid $\notin$ receivedMsgs then
        trigger Deliver($m$)
        receivedMsgs $\longleftarrow$ receivedMsgs $\cup$ {muid}
        if $\forall$ (id) $\in$ missingFromTree(announcements, tree) : id $=$ muid then
            cancel Timer(mID)
        announcements $\longleftarrow$ removeMuid(muid, announcements)
        if tree.activePeers = $\emptyset$ then
            if sender $\in$ backupPeers then
                tree.activePeers $\longleftarrow$ tree.activePeers $\cup$ {sender}
            backupPeers $\longleftarrow$ backupPeers $\setminus$ {sender}
            call treeBranching(tree)
        call Forward (m, mID, round$+1$, tree, myself)
        call Balance (mID, mask, tree, sender)
    else
        tree.activePeers $\longleftarrow$ tree.activePeers $\setminus$ {sender}
        backupPeers $\longleftarrow$ backupPeers $\cup$ {sender}
        trigger Send(PRUNE, sender, tree, myself)

procedure SourceTreeBranching (tree) do
    peers $\longleftarrow$ getRandomPeers(backupPeers, $f$)
    foreach $p \in peers$ do
        tree.activePeers $\longleftarrow$ tree.activePeers $\cup$ {p}
        backupPeers $\longleftarrow$ backupPeers $\setminus$ {p}

procedure TreeBranching (tree) do
    if $\nexists$ t $\in trees$ : $|t.activePeers| > 1$ then
        peers $\longleftarrow$ getRandomPeers(backupPeers, $f - 1$)
        foreach $p \in peers$ do
            tree.activePeers $\longleftarrow$ tree.activePeers $\cup$ {p}
            backupPeers $\longleftarrow$ backupPeers $\setminus$ {p}

every T seconds do
    if $\sum_t Load < maxLoad$ then
        SUMMARY $\longleftarrow$ GetNewSummnary (receivedMessages)
        foreach $p \in$ backupPeers do
            trigger send(SUMMARY, Load)

procedure Forward (m, muid, tree, sender) do
    foreach $p \in$ tree.activePeers: $p \neq$ sender do
        trigger Send(DATA, $p$, $m$, muid, Load, tree, myself)

upon event Receive (PRUNE, load, tree, sender) do
    tree.ActivePeers $\longleftarrow$ tree.ActivePeers $\setminus$ {sender}
    BackupPeers $\longleftarrow$ BackupPeers $\cup$ {sender}
```

4.3 Tree Repair

The goals of the tree repair mechanism are twofold: i) it is responsible for ensuring that all nodes eventually become connected to all existing spanning trees and, ii) it detects and recovers from tree partitions that might happen due to failure of nodes. This component relies on the SUMMARY messages disseminated periodically by

Algorithm 3: Tree Repair

```
upon event Receive (SUMMARY, load, sender) do
    foreach (muid, p) $\in$ SUMMARY do
        if $\nexists$ Timer(t) : t = muid.t then
            setup Timer(muid.t, timeout)
        announcements $\longleftarrow$ announcements $\cup$ {(muid, sender)}

upon event Timer(tree) do
    (muid, p) $\longleftarrow$ removeBest(announcements, tree)
    tree.activePeers $\longleftarrow$ tree.activePeers $\cup$ {p}
    backupPeers $\longleftarrow$ backupPeers \ {p}
    trigger Send(GRAFT, p, null, loadEstimate$_p$, tree, myself)

upon event Receive (GRAFT, muid, load, tree, sender) do
    if $\sum_t Load < maxLoad$ $\wedge$ sender $\in$ tree.backupPeers $\wedge$
            ($|tree.activePeers| > 1 \vee load =$ Load) then
        tree.activePeers $\longleftarrow$ tree.activePeers $\cup$ {sender}
        backupPeers $\longleftarrow$ backupPeers \ {sender}
    else
        trigger Send(PRUNE, sender, Load, tree, myself)

procedure Balance (muid, load, tree, sender) do
    if $\exists$ (id, p) $\in$ announcements : id.t = tree then
        newLoad $\longleftarrow$ IncTreeLoad(loadEstimate$_p$, tree)
        if $nInterior(newLoad) < nInterior(load)$ then
            trigger Send(GRAFT, n, $null$, loadEstimate$_p$, t, myself)
            trigger Send(PRUNE, sender, Load, tree, myself)
```

each node. We recall that SUMMARY messages contain the identifiers of data messages recently added to the *receivedMsgs* set. More precisely, each SUMMARY message contains the identifiers of all fresh messages received since the last SUMMARY message was sent by the node.

When a node n receives a SUMMARY message from another node s, it verifies if all message identifiers are recorded in its *receivedMsgs*$_n$ set. If no messages have been missed, the SUMMARY is simply discarded. Otherwise, a tuple (*muid*, s) is stored in the *announcements*$_n$ set for each data message that has not been received yet. Furthermore, for each tree t where a message has been detected to be missing, a *repair timer* is initiated: if the missing messages have not been received by the time this timer expires, the node assumes that t has become disconnected from that tree and takes measures to repair it, as follows.

Consider that node n has received from a set of nodes S a SUMMARY message with the *muid* of a data message detected to be lost in tree t. Node n is going to select a single target node $s_t \in S$ to repair the tree t. The selection procedure uses the information that nodes keep about the load of their peers (see variable *loadEstimate*$(p,t)_n$ in Section 4.1). Namely, s_t is selected at random among all peers in S for which the forwarding load is below a threshold (*maxLoad*) and that are estimated to be interior nodes in a smaller number of trees, or that are already interior in t and has not reached a load of *maxLoad*.

After selecting s_t, node n performs the following two steps: s_t is removed from *backupPeers*$_n$ and added to t.*activePeers*$_n$ and a GRAFT message is sent to s_t. The GRAFT message includes the current view of n concerning the load of s_t (note that n's information about s_t may be outdated, as this information is only propagated when it can be piggybacked on data or control messages).

When s_t receives a GRAFT message from n for tree t, it first checks if n based its decision on up-to-date values for the load of s_t (i.e., if the current forwarding load of s_t matches the information owned by n) or if, despite eventual inaccuracies in the estimate, s_t can nevertheless satisfy the request of n without increasing the number of trees where it is interior nor increasing its current forwarding load to values above *maxLoad*. If this is the case, s_t adds n to t.*activePeers*$_{s_t}$. Otherwise, s_t rejects the GRAFT message by sending back a PRUNE message to n

Algorithm 4: Overlay Network Dynamics

```
upon event NeighborDown(node) do
    foreach $tree \in trees$ do
        tree.ActivePeers ⟵ tree.ActivePeers \ {node}
    BackupPeers ⟵ BackupPeers \ {node}
    foreach ($muid$,$s$) ∈ announcements : s = node do
        announcements ⟵ announcements \ {($muid$,$s$)}

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

(since load information is piggybacked to all messages, this will also update n' s information on s_t' s load).

Finally, if n receives a PRUNE message back from s_t, n will move back s_t from t.*activePeers*$_n$ to the *backupPeers*$_n$ and attempt to repair t by picking new targets from the *announcements*$_n$ set.

Algorithm 3 depicts a simplified version of this procedure in pseudo-code.

4.4 Tree Reconfiguration

The tree construction and repair mechanisms described above are able to create spanning trees with complete coverage, where a large portion of nodes is interior in a single spanning tree (this happens due to the repair mechanism, as confirmed by experimental results presented in Section 5). This is true in a stable environment (*i.e.*, when there are no joins or leaves in the system). However, multiple executions of the repair mechanism above may lead to configurations where several nodes are interior in more than one tree, which is clearly undesirable.

To circumvent this problem, we developed a reconfiguration procedure that operates as follows: When node n receives a non-redundant data message m from a node s in a tree t for which it had previously received an announcement from a peer a, it compares the estimated load of s and a.

If $\sum_t loadEstimate(s,t)_n > \sum_t loadEstimate(a,t)_n$ and n can replace the position of s in tree t without becoming interior in more trees, node n attempts to replace the link between s and n by a link between a and n. For this purpose, n sends a PRUNE message to s and a GRAFT message to a.

Note that the reconfiguration is only performed if the announcement from a is received before the data message itself from s. This ensures that a reconfiguration contributes to reducing the latency in the tree while avoiding the construction of cycles. Note that, because nodes which forwarding load reaches the *maxLoad* threshold are unable to help their peers repairing spanning trees, they cancel the periodic transmission of SUMMARY messages in this situation.

4.5 Network Dynamics

As stated previously, the peer sampling service is responsible for detecting changes in the partial view maintained locally and for notifying Thicket when these changes occur, using the *NeighborDown*(p) and *NeighborUp*(p) notifications (see Algorithm 4).

When a node n receives a *NeighborDown*(p) notification it removes p from all t.*activePeers*$_n$ sets and also from the *backupPeers*$_n$ set. Additionally, all records of announcements sent by p are also deleted from the *announcements*$_n$ set. This might result in the node becoming disconnected from some trees (most of the times from a single tree). The tree repair mechanism however is able to detect and recover from this scenario.

On the other hand, when a node n receives a *NeighborUp*(p) notification, it simply adds p to the *backupPeers*$_n$ set. As a result, p will start exchanging SUMMARY messages with n. As explained above, these messages will allow not only joining nodes to become connected to all spanning trees, but also to leverage on new overlay neighbors to balance the load imposed over participants (using the tree reconfiguration mechanism).

4.6 On the Selection of Parameter T

Considering the number of trees created (T) and the protocol *fanout* (f), the maximum value for parameter T is intimately related with the parameter f. In fact, take the case where f is equal to 2. In this scenario each of our trees is a binary tree where half the nodes are interior. Therefore, in such a scenario only 2 trees can be built using the same overlay without having a node acting as interior in more than a single tree. Therefore, the maximum number of trees (T) is limited by the fanout (f) used in branching the trees[4].

The degree of the unstructured overlay network should at least be equal to f (for the tree where each node acts as interior) plus a link for each additional tree ($T - 1$, these links are used to receive the messages disseminated through the remaining tree) however this would render a decentralized mechanism to build such trees infeasible. Therefore, we rely on overlay degrees in the order of $f * T$, which provides each node with access to enough links to find suitable configurations for its role in all trees.

5 Evaluation

In this Section we report experimental results obtained using the PeerSim simulator [10]. To this end we have implemented Thicket for this simulator. In order to extract comparative figures we also tested the performance of the single-tree Plumtree protocol [13] (that serves as baseline to our solution) as well as the “NUTS and BOLTS” alternatives discussed in Section 3. For fairness, all protocols were executed on top of the same unstructured overlay, maintained by the HyParView protocol [14]. HyParView is able to recover from failures as large as 80% of concurrent node failures. Since HyParView uses TCP to maintain connections between overlay neighbors, we do not model message losses in our system (TCP is also used to detect failures).

We have tested all protocols firstly in a stable scenario, where no node failures were induced, and later in faulty scenarios. For faulty scenarios, we have evaluated the reliability of the broadcast process under sequential failures of nodes and the reconfiguration capacity of Thicket in a catastrophic scenario, where 40% fail simultaneously. In the following we describe in more detail the experimental setup and the relevant parameters employed in our experiments.

5.1 Experimental Setup and Configuration

Our simulations progress in cycles (using the cycle-based engine of the simulator). Each simulation cycle corresponds to 20s. In each cycle the source broadcasts T messages simultaneously, one message using each of the existing trees (in the case of Plumtree, which only builds one shared tree, all T messages are routed through the existing tree). As stated before we assume perfect links, however messages are not delivered to nodes instantly, instead we consider the following delays when routing messages between nodes (these delays are implemented by using the event based engine of the simulator[5]):

Sender delay. We assume that each node has a bounded uplink bandwidth. This allows to simulate uplink congestion when nodes are required to send several messages consecutively. In particular we assume that each node can transmit $200K$ bytes/s. Furthermore we assume that the payload of data messages had 1250 bytes, while SUMMARY messages have 100 bytes.

Network delay. We assume that the core of the network introduces additional delays. In detail, in the simulations a message that is transmitted suffers an additional random delay selected uniformly between 100 and 300 ms.

[4]We have determined the value of f used in our evaluation experimentally. This value is related with the *fanout* of gossip protocols that operate over symmetric overlay networks [14]

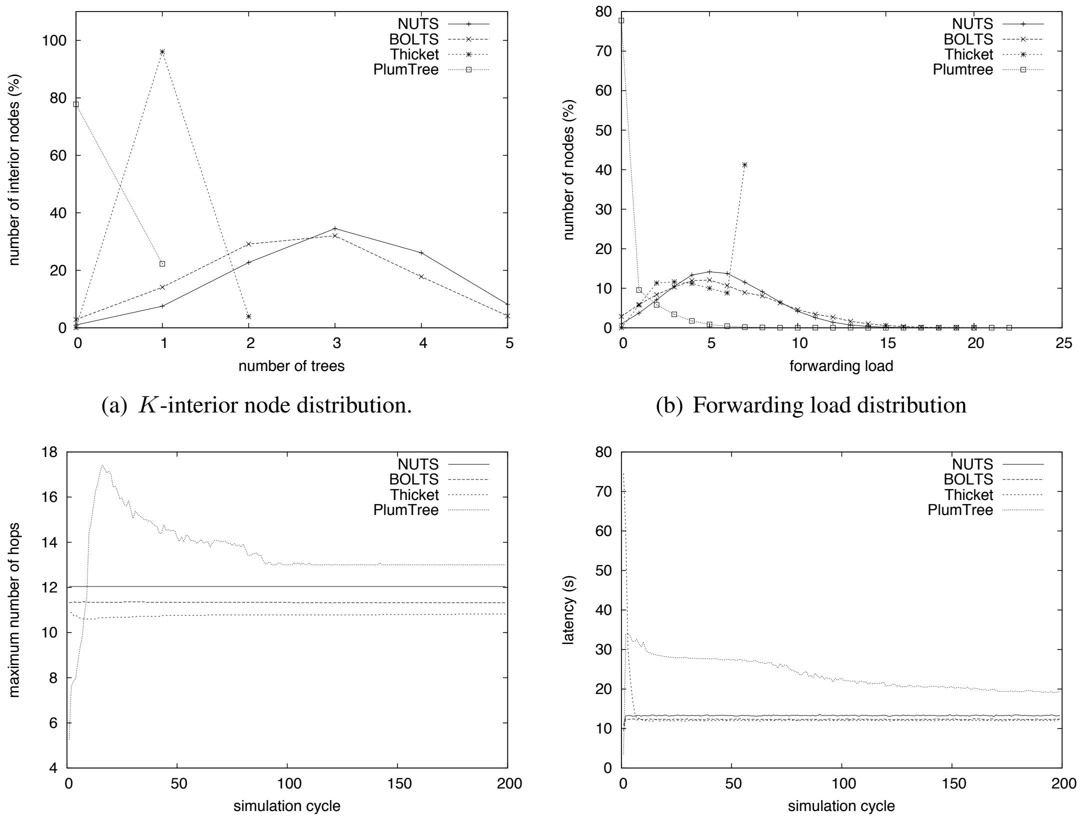
[5]The minimum time unit in our system is 1ms.

These values were selected by taking into consideration round trip time measurements that were performed using the PlanetLab infrastructure[6].

We have conducted all the experiments using a network of 10.000 nodes and all presented results are an average of 10 independent executions of each experiment. All tested protocols, with the exception of Plumtree, were configured to generate $T = 5$ trees. Additionally, Thicket establishes trees using a gossip fanout of $f = 5$ and NUTS initiates the eager push set of each spanning tree with 5 random selected overlay neighbors. Thicket, Plumtree, and NUTS operate on top of an unstructured overlay network with a degree of 25, while each of the 5 overlays used by BOLTS has a degree of 5. Furthermore, we have configured Thicket to have a maximum forwarding load per node (parameter *maxLoad*) of 7. The timeout employed by protocols when receiving an announcement was set to 2s.

All experiments start with a stabilization period of 10 simulation cycles, which are not taken into account when extracting results. During these cycles, all nodes join the overlay network and the overlay topology stabilizes. After this stabilization period, we start the broadcasting process; this triggers the construction of trees.

5.2 Stable Environment



(a) K-interior node distribution.

(b) Forwarding load distribution

(c) Number of maximum hops

(d) Latency

Figure 2. Experimental results in a stable environment.

First, we analyze the relevant performance metrics for Thicket in a stable environment where no node failures are induced. We start by evaluating the distribution of nodes accordingly to the number of spanning trees in which

[6]The measurements can be found in `http://pdos.csail.mit.edu/~strib/pl_app/`

they are interior. A value of 0 trees means that such nodes are not interior in any of the trees, *i.e* they act as leaves in all trees. The results are depicted in Figure 2(a). Plumtree is plotted in the figure to serve as a baseline for a scenario with a single tree. Note that, with a single tree, only 21% of the nodes are interior nodes, and 79% are leaf nodes.

When using both the NUTS and BOLTS strategies, only a small fraction (below 20%) of nodes are interior in a single tree (we repeat here the plot from Section 3 for the convenience of the reader). Also, for both approaches, there is a small number of nodes that are interior in all 5 trees. As noted before, this motivates the need for some sort of coordination during the tree construction.

In sharp contrast, Thicket has almost all nodes in the system acting as interior nodes in a single tree. A very small fraction (around 1%) serve as interior in 2 trees. This is a side effect of our localized tree repair mechanism, that ensures full coverage of all spanning trees. Still, no node (with the exception of the source node) acts as interior for more than 2 trees. This validates the design of Thicket. Notice also that almost no node is a leaf node in all trees; this contributes to the reliability of the broadcast process (see results below) and ensures a uniform load distribution among participants. Furthermore, it allows us to use a much larger fraction of the available resources in the system.

Figure 2(b) depicts the distribution of forwarding load in our system *i.e.*, the distribution of nodes accordingly to the number of messages they must forward across all trees. Because Thicket leverages on its integrated tree construction and maintenance strategy to limit the maximum load imposed to each node, no participant is required to forward more than 7 messages across all trees where it is interior (usually 1 as we explained earlier). Additionally, more than 40% of nodes are forwarding the maximum amount of messages, with more than 55% of nodes forwarding a smaller amount of messages. The other solutions however have much more variable loads, with several nodes forwarding more than 10 messages and some with loads above 15 messages. Notice that Thicket is the only protocol where almost no participant has a forwarding load of 0. This is a clear demonstration of the better resource usage and load distribution that characterizes Thicket.

We also conducted experiments to evaluate the effect of Thicket in the dissemination of payload messages. In particular we have evaluated the maximum number of hops required to deliver a message to all participants, and the maximum latency between the source node and a receiver. Figure 2(c) depicts the number of messages hops required to deliver a data broadcast message to all participants. Plumtree exhibits the highest value. This happens because Plumtree has some difficulties in dealing with variable network latency. This leads to situations where Plumtree triggers message recoveries too early, which increases the number of hops required to deliver a single message to all participants. Plumtree keeps on adjusting the topology during the entire simulation, with the effect of slightly reducing the number of hops, stabilizing at 13 hops.

Thicket presents the best values (11 hops), as the trees created by the protocol are adapted, using the reconfiguration mechanism, to promote trees with lower height, resulting in lower values of last delivery hop (notice that these metrics are related to each other). The BOLTS approach presents a similar result. This happens because the use of several independent overlay networks forces the produced spanning trees (generated with flooding) to use the shortest paths between the source node and all receivers. NUTS has a higher value due to the use of a gossip-based tree construction scheme, that does not guarantee the use of all shortest paths.

Figure 2(d) presents the maximum latency for all protocols. These values are consistent with the last delivery hop values observed. One interesting aspect is that, contrary to all remaining protocols, Thicket presents higher initial values of latency, but these drop quickly in just 5 simulation cycles. This is due to the operation of the tree reconfiguration mechanism.

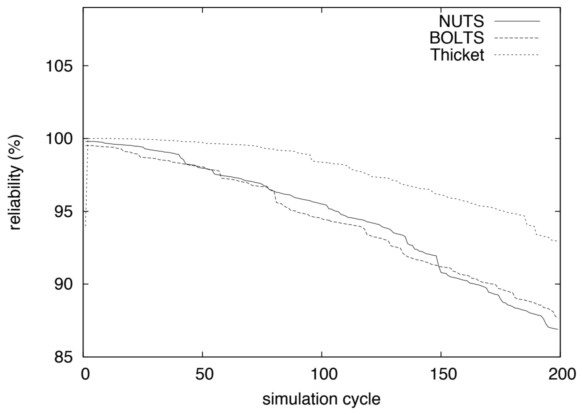
5.3 Fault-Tolerance

In this section we evaluate the performance of thicket in two distinct failure scenarios. In particular we study the impact of sequential node failures in the broadcast reliability when using Thicket, NUTS, and BOLTS. Later, we present results that illustrate the recovery and reconfiguration capacity of Thicket in a catastrophic scenario that is characterized by a large number of simultaneous node failures. In our experiments the source node and the nodes that serve as root for trees in NUTS never fail.

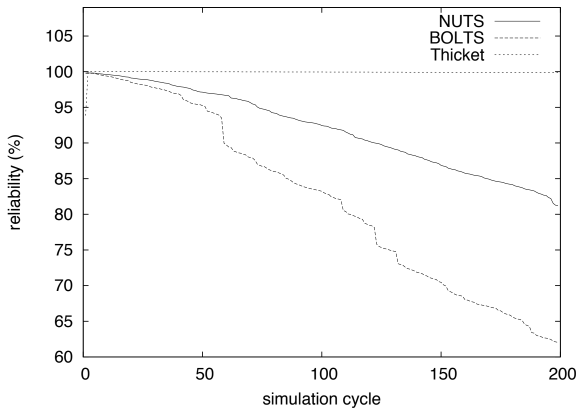
5.3.1 Sequential Node Failures

We now depict the reliability of the broadcast process in face of sequential node failures. Here we consider the reliability assuming that the broadcast process leverages in the co-existing spanning trees to introduce redundancy in the disseminated data (for instance by using network coding techniques). Furthermore we assume that for each segment of data 5 messages are disseminated, one for each spanning tree, such that if a node is able to receive at least 4 of these messages it is able to reconstruct the data segment, otherwise we consider that the node misses the reception of this segment. We define reliability here as the percentage of correct nodes that are able to reconstruct disseminated data segments.

After an additional stabilization period (5 cycles) we configure the source node to disseminate a data segment per cycle. In each cycle we also force a single node to fail. We measure the reliability of the broadcast process at the end of each simulation cycle. Furthermore, we select the node that fails in each cycle using two distinct policies: i) we select the node that fails at random; ii) we select the node that fails at random among the nodes that are interior in more trees. We do not allow nodes to execute the repair mechanism during these simulations, to better capture the resilience of the generated spanning trees. The results for all protocols using the repairing mechanism in this scenario would depict reliability measures close to 100%.



(a) Random node failures.



(b) Targeted node failures.

Figure 3. Experimental results for a catastrophic scenario.

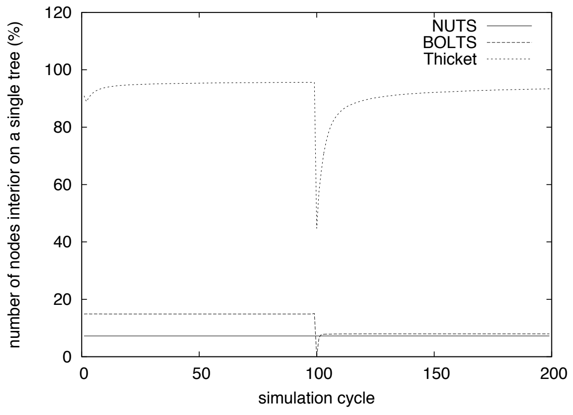
Figure 3 depicts the results for both scenarios. When we select random nodes to fail (Figure 3(a)) the reliability of Thicket drops slowly. This happens because most nodes are interior in a single tree. So each failure, affects only nodes bellow the failed one in a single tree, because nodes can reconstruct the data segment even if they miss messages conveyed by one of the trees, most of them are still able to rebuild data segments as they remain connected to (at least) 4 trees. The reliability drops in a more visible way for both NUTS and BOLTS. This happens because a large majority of nodes are interior in more than a single tree, which results in a single node failure affecting the flow of data in more than a tree.

Note that failing nodes at random may not provide the best metric for reliability. For instance, failing random nodes in a star network only has a noticeable effect in the reliability when the central node fails (this is a single but also the only point of failure). The second experiment is more interesting, as it assesses what happens when "key" nodes crash.

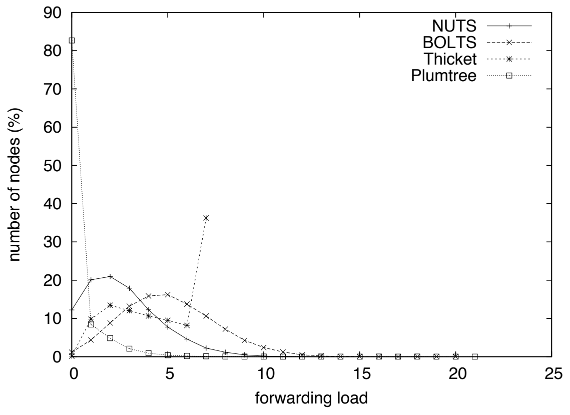
Interestingly, Thicket is extremely robust in face of such a targeted adversary (Figure 3(b)), and its reliability remains constant at 100%. This happens due to the following phenomena: because we limit the forwarding load imposed to each Thicket node, nodes that act as interior in more than a tree are responsible for forwarding messages to a smaller amount of nodes for each tree. Therefore, the effective number of nodes that are affected in each tree is small. Furthermore, because links are never used for more than a tree, these groups of nodes are disjoint, and therefore can still receive messages sent through 4 trees. On the other hand, NUTS and BOLTS are severely affected by this scenario due to the fact that some nodes are interior in all trees, which failure disrupts the flow of data in all trees.

5.3.2 Catastrophic Scenario

We now present results in face of a large number of simultaneous node failures (in particular 40%). Note that, with this number of failures, all trees are affected. Therefore, there are no significant advantages of ensuring that nodes are only interior in a single tree. Thus, we do not expect advantages from a reliability point of view in this scenario. However, it is worth evaluating if Thicket is able to recover from this amount of failures and if, after recovery, the trees preserve their original properties, namely in terms of nodes that are interior in a single tree and in terms of load distribution. Failures are induced after 100 cycles of message dissemination, to ensure that the spanning trees were already stabilized. Figure 4 summarizes our results.



(a) Percentage of Nodes Interior in a single tree.



(b) Forwarding load distribution.

Figure 4. Experimental results for a catastrophic scenario.

Figure 4(a) depicts the variation, for each protocol based on multiple trees, of the percentage of nodes that are interior in a single tree. Before the node failures, all protocols exhibit results consistent with the ones presented earlier, for a stable scenario. After the induction of failures the percentage of interior nodes in a single tree drops in BOLTS as result of its recovery procedure, that increases the percentage of nodes acting as interior nodes in multiple trees. NUTS remains unaffected, as the percentage of nodes in this condition is only 10% in steady state. Thicket drops to values in the order of 40% after the failures. However the protocol is able to reconfigure itself in only a few simulation cycles.

Figure 4(b) depicts the forwarding load distribution for each protocol. The relevant aspect of this graph is that Thicket is able to regain a similar configuration to the one exhibited in a stable environment. The other protocols configuration remains similar, with nodes exhibiting a wide range of forwarding loads. This is a clear indication that Thicket can regain its properties despite a large number of concurrent failures.

6 Conclusions

In this paper we have proposed Thicket, the first decentralized algorithm to efficiently build and maintain multiple and *independent* spanning trees over a single unstructured overlay network. In Thicket most of nodes in the system (almost 100%) act as an interior node in a single spanning tree, and no node is interior in more that 2 trees. This allows us to significantly improve the load balancing of participants in tree-based multicast systems, as long as each tree is used to transmit a similar amount of data.

Additionally, Thicket employs a tree reconfiguration procedure that allows it to build trees with limited height. This allows Thicket to present lower, and more stable latency values when compared with other solutions. Additionally, because Thicket operates on top of an unstructured overlay network that is extremely resilient to failures, it can tolerate catastrophic failure scenarios where a large fraction of the nodes in the system fail simultaneously. We do this by exploiting the overlay links that are not used as tree branches.

For future work, we intend to experiment the Thicket protocol over the PlanetLab infra-structure, evaluating its resilience using real world scenarios which reflect undesirable properties such as link failures, heterogeneous and varying latencies, link congestion, and heterogeneous capacities.

Acknowledgment

This work was partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program fund and by FCT projects “Redico” (PTDC/EIA/71752/2006) and “HPCI” (PTDC/EIA-EIA/102212/2008).

References

[1] M. Allani, J. Leitão, B. Garbinato, and L. Rodrigues. Rasm: A reliable algorithm for scalable multicast. In *Proceedings of Euromicro PDP'2010*, page (to appear), Pisa, Italy, Feb. 2010.

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

[3] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *Proceedings of SOSP'03*, pages 298–313, New York, NY, USA, 2003. ACM.

[4] P. A. Chou and Y. Wu. Network coding for the internet and wireless networks. *IEEE Signal Processing Magazine*, page 7785, 2007.

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

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

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

[8] D. Frey and A. L. Murphy. Failure-tolerant overlay trees for large-scale dynamic networks. In *Proceedings of P2P'08*, pages 351–361, Washington, DC, USA, 2008. IEEE Computer Society.

[9] Y. Huang, T. Z. Fu, D.-M. Chiu, J. C. Lui, and C. Huang. Challenges, design and analysis of a large-scale p2p-vod system. *SIGCOMM Comput. Commun. Rev.*, 38(4):375–388, 2008.

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

[11] D. Johnson, J. Lenstra, and H. Rinnooy. The complexity of the network design problem. *Networks*, 8(4):279–285, 1978.

[12] B. Koldehofe. Buffer management in probabilistic peer-to-peer communication protocols. In *Proc. of SRDS'03*, pages 76–87, Florence,Italy, Oct. 2003.

[13] J. Leitão, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *Proceedings of SRDS'07*, pages 301 – 310, Beijing, China, Oct. 2007.

[14] J. Leitão, J. Pereira, and L. Rodrigues. Hyparview: a membership protocol for reliable gossip-based broadcast. In *Proceedings of DSN'07*, pages 419–429, Edinburgh, UK, June 2007.

[15] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. MON: On-demand overlays for distributed system management. In *Proceedings of WORLDS'05*, 2005.

[16] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. In *Proceedings of NOSSDAV '02*, pages 177–186, New York, NY, USA, 2002. ACM.

[17] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware '01*, pages 329–350, London, UK, 2001. Springer-Verlag.

[18] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In J. Crowcroft and M. Hofmann, editors, *Networked Group Communication*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2001.

[19] V. Venkataraman, K. Yoshida, and P. Francis. Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast. In *Proceedings of ICNP '06*, pages 2–11, Washington, DC, USA, 2006. IEEE Computer Society.

[20] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of NOSSDAV'01*, June 2001.