INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Live Streaming in Overlay Networks

Mário Rui Vazão Vasco Ferreira

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Prof. Doutor Alberto Manuel Ramos da Cunha
Orientador:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Vogal:	Prof. Doutor Sérgio Marco Duarte

October 2010

Acknowledgements

I would like to start thanking my advisor, Prof. Luís Rodrigues, for accepting to guide me during the elaboration of this thesis and for all his providential advices.

I want to thank João Leitão as well, for all his support and helpful comments during the development of this work.

I also want to thank all the members of the Distributed Systems Group (GSD) for their comments on my work and for the football matches during the time I spent there.

Likewise, I want to thank my home mates and best friends namely Pedro Vala, André Louçã and Carlos Rosa for their help, support and fruitful or non-fruitful comments.

Finally, I would like thank my parents, my family and my girlfriend, Soraia Silva, for all their support and encouragement.

This work was performed at INESC-ID and was partially funded by FCT under the project "Redico" (PTDC/EIA/71752/2006) and INESC-ID multi-annual funding through the PIDDAC Program funds.

Lisboa, October 2010

Mário Rui Vazão Vasco Ferreira

For my parents and girlfriend,

Carlos, Cidália and Soraia

Resumo

Os sistemas entre-pares surgiram como uma tecnologia promissora para suportar a disseminação de informação na Internet, incluindo dados multimédia com requisitos de qualidade de serviço. Este trabalho apresenta o Thicket, um protocolo para construir e manter múltiplas árvores de disseminação numa rede sobreposta, oferecendo um esquema de disseminação que promove a distribuição da carga por todos os participantes e a tolerância a falhas. O desenho proposto foi avaliado recorrendo a simulações e a experiências usando um protótipo concretizado em Java. Uma rede composta por 10.000 nós foi simulada usando o Peersim e o protótipo foi testado sobre a infra-estrutura do PlanetLab.

Abstract

P2P systems have emerged as a promising technology to support the dissemination of information on the Internet, including multimedia streams with quality of service requirements. This work presents Thicket, a protocol for building and maintaining multiple spanning trees over an overlay network, providing a dissemination scheme that promotes load distribution across all participants and fault-tolerance. The proposed design was evaluated using simulations and a prototype implemented in Java. The Peersim was used to simulate a network composed of 10.000 nodes and the prototype was tested on the PlanetLab infrastructure.

Palavras Chave
Keywords

Palavras Chave

Transmissão ao vivo

Sistemas entre-Pares

Árvores de Disseminação

Tolerância a Falhas

Keywords

Live Streaming

P2P Systems

Spanning Trees

Fault-tolerance

Index

List of Figures

List of Tables

1 Introduction

Live streaming applications are emerging in the Internet. Due to the lack of widespread of IP Multicast support, the Peer-to-Peer (P2P) paradigm became an attractive alternative to build these applications. The resulting systems support a large number of participants connected by self-organizing overlays. In P2P systems, the available resources such as upload bandwidth, CPU cycles, or data storage capacity, scale with the demand; since clients also participate in the system, when the number of system participants increases the quantity of available resources also increases.

The key strategy to perform live streaming to a large number of participants is to use a multicast service. Several approaches have been proposed to support multicast in P2P systems. Some approaches focus on building specialized structures to support data dissemination while others employ different techniques, such as flooding or distance vector routing, to perform the multicast operation. While the former adds complexity in overlay management to maintain the structure, the latter offers potential to better cope with the underlying topology allowing the overlay to explore more efficient routing paths.

This work addresses the problem of implementing large-scale P2P live streaming systems, with empahsis on decentralized mechanisms that allow to distribute the forwarding load evenly among the participants.

1.1 Motivation

Live streaming applications need to deliver content to a large set of receivers. Therefore, in order to be scalable, they need to rely on some form of multicast mechanism. One way to efficiently disseminate messages to a large number of receivers is using a spanning tree structure, where each node forwards every message to its children nodes, until all participants receive the message(Zhuang, Zhao, Joseph, Katz, & Kubiatowicz 2001; Rowstron, Kermarrec, Castro, &

Druschel 2001; Liang, Ko, Gupta, & Nahrstedt 2005; Leitão, Pereira, & Rodrigues 2007a). Structured approaches, such as the ones using spanning-trees, are often more efficient than completely unstructured approaches, such as flooding. However, when using a single spanning tree, interior nodes, which are a just a subset of all nodes, support the entire system load, while leaf nodes just receive data and do not help in the dissemination.

Designs that rely on the use of multiple trees offer opportunities for load balancing(Padmanabhan, Wang, Chou, & Sripanidkulchai 2002; Castro, Druschel, Kermarrec, Nandi, Rowstron, & Singh 2003; Venkataraman, Yoshida, & Francis 2006). The idea is to build several spanning trees, each connecting all nodes. In this way, each node is interior in some trees and a leaf in the remaining. Load balancing is achieved by sending an equal fraction of all messages through each tree, making all nodes contribute in the trees which they are interior.

Ideally, each node should only be interior in at least one tree. This constraint is more easily achieved using a centralized approach to tree construction(Padmanabhan, Wang, Chou, & Sripanidkulchai 2002), or by relying on structured overlays(Castro, Druschel, Kermarrec, Nandi, Rowstron, & Singh 2003). But these designs also have disadvantages. Relying on a centralized component to coordinate the joins/exits of participants can limit scalability, specially in unstable environments such as the Internet, where systems are constantly subject to churn. On the other hand, using a structured overlay adds overhead during membership changes, since measures need to be taken in order to maintain a specific overlay structure.

Therefore, this work addresses the design of multi-tree dissemination schemes, that are decentralized and can be executed over unstructured overlays.

1.2 Contributions

This work addresses the problem of designing fully distributed algorithms capable of efficiently disseminate data to a large number of participants. More precisely, the thesis analyzes, implements, and evaluates techniques to disseminate content over a P2P unstructured overlay while balancing the load through all participants. As a result, the thesis makes the following contribution:

It proposes a novel algorithm to build and manage an efficient multi-tree data dis-

semination structure, called Thicket. The algorithm is fully decentralized, and builds the trees such that nodes are interior in just one or in a small subset of all trees. In this way, it promotes load balancing of the system load across all participants.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- Two prototype implementations of Thicket, one for the PeerSim Simulator and a real implementation that has been deployed in the PlanetLab infrastructure.
- An experimental evaluation of the algorithm, both using simulations and experiments using the PlanetLab deployment.

1.4 Research History

This work was performed in the context of the Redico[1] project (Dynamic Reconfiguration of Communication Protocols), where some work on multicast for large-scale peer-to-peer systems had already been performed. During my work, I benefited from the fruitful collaboration with the remaining members of the GSD team working on Redico, namely João Leitão, the main designer of the Plumtree protocol, one of the sources of inspiration for this work.

Parts of the work reported in this dissertation have been published in (Ferreira, Leitão, & Rodrigues 2010b) and (Ferreira, Leitão, & Rodrigues 2010a).

1.5 Structure of the Document

The rest of this document is organized as follows. Chapter 2 provides survey of the background work that has inspired Thicket design. Chapter 3 introduces Thicket, a Protocol for Building and Maintaining Multiple Trees in a P2P Overlay and Chapter 4 presents the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by discussing its main contributions and providing hints for future work.

[1] http://redico.gsd.inesc-id.pt

2 Related Work

2.1 Introduction

This section presents an overview of the existing related work on supporting the dissemination of information with quality-of-service constraints in large-scale peer-to-peer systems. Although the goal of this work is to support live-streaming, for completeness, some solutions that are focused on the video-on-demand and multicast support with soft real-time constraints are also described.

The remaining of this section is organized as follows. Section 2.2 presents three representative classes of bandwidth-consuming applications. Section 2.3 compares the two main approaches to perform multicast. Section 2.4 identifies the main challenges and performance metrics and Section 2.5 presents the main design choices concerning live streaming. Finally, Section 2.6 presents a number of selected systems.

2.2 Types of Bandwidth-Consuming Applications

Some applications depend on remote data from the network to fulfill its operation. This applications have bandwidth requirements and are called bandwidth-consuming applications. They can be divided into three representative classes (Benbadis, Mathieu, Hegde, & Perino 2008), namely: *fixed rate*, *required rate* and *elastic Rate*. These classes of applications differ on how content is generated and consumed, and on the quality of service metrics used to assess their performance.

- In *fixed Rate* applications, data content is generated at a fixed rate r and distributed on the fly. There are strict time requirements for content distribution, because peers should receive the streamed data at the same rate r. A quality of service metric for this

applications is the download continuity, which means that the download rate should always be equal to r. An example of such application is live-streaming (Padmanabhan, Wang, Chou, & Sripanidkulchai 2002; Huang, Ravindran, & Kumar 2009).

- In *required Rate* applications, the content is available at a subset of nodes and it should be downloaded by the other nodes at a minimum rate r. Similarly to *fixed Rate* applications, this restriction is imposed to ensure continuity, but unlike in the previous class, content is already completely available a priori, and can be cached or stored until it is consumed. Therefore, the quality of service metric can be expressed as: a node must download content at a rate greater or equal than r. A typical example of this class of applications is Video-on-Demand (VoD) (Annapureddy, Guha, Gkantsidis, Gunawardena, & Rodriguez 2007; Huang, Ravindran, & Kumar 2009).

- In *elastic Rate* applications the content is also initially available at a subset of nodes, but there are no constraints on the minimum download time or rate. The quality of service metric is the download time, which should be minimized. This kind of applications includes file-sharing applications (Bittorrent ; Emule ; Kazaa).

2.3 IP Multicast vs Application-Level Multicast

There are two main approaches to multicast data to a potentially large set of recipients in the Internet, which main difference is the layer at which they are implemented. IP Multicast (Deering & Cheriton 1990) operates at the IP-layer while Application-Level Multicast (ALM) (Chu, Rao, Seshan, & Zhang 2002; Rowstron, Kermarrec, Castro, & Druschel 2001) operates at the application-layer.

IP Multicast Since 1990, multicast mechanisms at the IP-layer have been proposed (Deering & Cheriton 1990). By supporting multicast at IP level it is possible to prevent the same data from crossing any given link twice. It it also possible to ensure that packets follow almost IP-optimal paths. Therefore, IP multicast has the potential to achieve good performance and efficient network consumption. However, the deployment of IP multicast has been limited due to a number of drawbacks of the technology. First, it requires routers to keep per group state which, in addition to violating the "stateless" principle of the original IP

design, leads to scalability limitations. Moreover, IP Multicast, similarly to IP unicast, is a best effort service. Unfortunately, providing high level guarantees such as reliability, congestion control, flow control, or security in multicast systems, is much more challenging than providing such guarantees for unicast communication. Additionally, IP Multicast imposes infrastructural changes, slowing down the deployment phase. Recently, some effort has been made to overcome this difficulties; in (Ratnasamy, Ermolinskiy, & Shenker 2006), the authors attempt to provide a new approach for IP Multicast deployment which uses routing protocols that already exist but, with a loss of efficiency because the proposed solution increases bandwidth requirements.

Application-Level Multicast Because of the aforementioned deployment issues of IP Multicast, P2P systems have emerged as an alternative to support multicast service using decentralized and scalable solutions. These solutions typically create an *overlay network* among the participants in the system, namely among the content providers and all the recipients, which is later used for media/data dissemination. Given that the overlay supports the communication among application level entities directly, it facilitates the development of high level mechanisms to enforce reliability, congestion control, etc, that are difficult to introduce at the IP Multicast level.

2.4 Challenges and Performance Metrics

Building P2P streaming solutions and applications is not an easy task. Internet is an heterogeneous environment, where most nodes and links have different characteristics. To build efficient and scalable P2P streaming solutions these differences need to be considered in order to meet applications requirements. Now, the main challenges and performance metrics that need to be considered when designing these solutions are presented.

Latency The latency of the multicast is a function of the sum of the latencies introduced by each link in the path from the source to the recipient. Ideally, the multicast service should forward messages using the links with the lowest latency, if possible. In turn, the latency of each overlay link depends on the properties of the underlying connection (including the maximum throughput, the physical distance among the endpoints, and the network

congestion). Latency can be estimated in runtime using multiple Round-Trip Time (RTT) measurements (Comer & Lin 1994).

Bandwidth Bandwidth is typically measured in bits per second, and captures the capacity of a system to transmit data from the source to the destination. In most P2P systems, where many nodes are connected to the Internet via ADSL or cable connections, the participants bandwidth is typically limited and asymmetric: nodes have much larger download bandwidth than upload bandwidth. Limited upload bandwidth needs to be managed to efficiently disseminate data while preserving application-level requirements.

Link loss Streaming systems using UDP as the transport protocol need to deal with packet losses. TCP can mask omissions at the network level but at the cost of increasing the communication overhead which limits the resource management at the streaming protocol level. Furthermore, in live-streaming applications, it is often preferable to deal with the loss of information than to introduce a lag in the delivery flow to recover a packet (Padmanabhan, Wang, Chou, & Sripanidkulchai 2002; Hefeeda, Habib, Xu, Bhargava, & Botev 2005). Link losses may also add complexity to the maintenance of the overlay network. For instance, failure detectors (Chandra & Toueg 1996; Renesse, Minsky, & Hayden 1998) may report a high number of false positives if a large number of control messages (such as heartbeat messages) are dropped.

Churn Any distributed application needs to deal with the failure or abrupt departure of nodes. This is also the case in P2P streaming applications. For instance, when a node fails, several other nodes will be affected as the failing node could be in charge of forwarding data to them. In large-scale P2P systems, failures may be common, not only due to the size of the system but also because participants may leave shortly after joining the system. A long sequence of join and leave / failures at very high pace in P2P system is a phenomena that is known as *churn* (Godfrey, Shenker, & Stoica 2006).

Free-Riding P2P systems are designed to exhibit the best performance when all participants cooperate in the execution of the distributed algorithm. However, it is possible that a selfish user deviates itself from the algorithm, such that it is able to receive content without contributing to its dissemination. Nodes that avoid to contribute to the data dissemination are known as free-riders. Therefore, a P2P streaming system must incorporate appropriate

incentive and punishment mechanisms to mitigate or eliminate this behavior (Haridasan & van Renesse 2008; Li, Clement, Marchetti, Kapritsos, Robison, Alvisi, & Dahlin 2008).

It is worth noting that is often impossible to optimize the performance of the system with regard to all of the challenges and performance metrics listed above. For instance, there is usually a trade off between the latency and bandwidth consumption.

2.5 Main Design Choices

This section discusses the main design choices in the implementation of live-streaming P2P systems. It identifies relevant design choices analyzing the advantages and disadvantages of existing design options.

2.5.1 Single vs Multiple View

Many algorithms are designed to support the dissemination of a single multimedia stream. These systems are called *single view* (Padmanabhan, Wang, Chou, & Sripanidkulchai 2002; da Silva, Leonardi, Mellia, & Meo 2008; Hefeeda, Habib, Xu, Bhargava, & Botev 2005; Picconi & Massoulié 2008; Liang, Guo, & Liu 2008; Huang, Ravindran, & Kumar 2009) systems. Naturally, it is possible to use such systems to disseminate multiple streams, by running multiple *independent* instances of the algorithm, one for each multimedia stream. This approach, however, may suffer from scalability issues.

On the other hand, it is possible to design a system in such a way that it is prepared to disseminate multiple multimedia streams. These systems are called *multi-view* systems (Wang, Xu, & Ramamurthy 2009; Wang, Xu, & Ramamurthy 2008). The advantage of having an integrated multi-view system is that it has the potential to achieve a better resource usage, for instance, by implementing schemes that promote resource sharing and load balancing. Multi-view systems are of particular importance to support IP-TV, because a user usually watches multiple TV channels (Oh, Kim, Yoon, Kim, Park, Lee, Lee, Heo, Lee, Park, Na, Hyun, Kim, Byun, Kim, & Ho 2007).

2.5.2 Centralized vs Decentralized Approaches

A centralized streaming system is a system where a unique node is responsible for transmitting the stream directly to all recipients. These solutions have the appeal of simplicity and may work in small-scale scenarios while suffering from obvious scalability problems.

In fully decentralized approaches (Leitão, Pereira, & Rodrigues 2007b; Chu, Rao, Seshan, & Zhang 2002; Rowstron, Kermarrec, Castro, & Druschel 2001), the source is not required to interact directly with all recipients, as the recipients coordinate to distribute the load of disseminating the data among themselves. Such solutions have the potential to overcome the scalability issues of centralized solutions.

For scalability, solutions only requiring local knowledge are favored, i.e., each individual participant should not be required to have global knowledge of the system. For instance, it may not know the total number of participants, their location, the bandwidth of all links, etc. Instead, it only has a partial view of the system, in particular, of its neighbors in the overlay (Leitão, Pereira, & Rodrigues 2007b; Ganesh, Kermarrec, & Massoulié 2003; Voulgaris, Gavidia, & van Steen 2005).

Naturally, there are optimizations that cannot be achieved only with local information. Due to this fact, some systems (Wang, Xu, & Ramamurthy 2009; Padmanabhan, Wang, Chou, & Sripanidkulchai 2002) adopt an intermediate form that combines both approaches, using a decentralized structure for data dissemination, but relying on a centralized component to facilitate or perform control operations, such as coordinating the join and leave of participants.

2.5.3 Structured vs Unstructured Approaches

There are two main approaches to build an overlay to support message dissemination: *structured* and *unstructured*.

- Structured overlays assume a regular topology, imposed by construction. They were initially designed to support distributed lookup services, but have also been used to disseminate data in an efficient manner. Example topologies include rings (Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001), hypercubes (Ratnasamy, Francis, Handley, Karp, & Shenker 2001) and Plaxton Meshes (Rowstron & Druschel 2001; Zhao, Kubiatowicz, &

Joseph 2001). The disadvantage of these overlays is the overhead required to maintain the structure during join and leave operations. This can lead to a poor performance when experiencing high levels of churn.

- On the other hand, unstructured overlays, typically named meshes, impose little or no specific topology to the system. Therefore, there are not as many constraints as in a structured overlay, enabling these overlays to better cope with network dynamics being more resilient to churn. Unfortunately, it is harder to design efficient dissemination schemes over unstructured overlays. While some systems (Chu, Rao, Seshan, & Zhang 2002) use unstructured overlays to run a distance-vector routing protocol (Deering & Cheriton 1990), the simplest approach consists in flooding the overlay, which in theory allows to reach all participants with a high cost due to the high communication redundancy imposed by this approach. In practice, the redundancy of the flood procedure may congest the network causing additional packet losses that may impair the reliability of the streaming protocol.

As the name implies, tree-based models forward the multicast messages along a spanning tree. This tree defines a single parent/father and a set of children to every node. To disseminate a message, a node receiving a message from its father simply has to forward it to all its children.

Tree-based approaches are not easy to include in a single classification mentioned above. In fact, they can be classified as both structured or unstructured approaches depending on how the tree is constructed.

In Padmanabhan, Wang, Chou & Sripanidkulchai (2002), tree management is held by the stream source which also manages all filiation operations (nodes joining and leaving), imposing a specific structure to the overlay topology. Thus, the approach is structured because it forces the overlay to assume a regular structure by construction.

Other approaches (Leitão, Pereira, & Rodrigues 2007a; Ripeanu, Foster, Iamnitchi, & Rogers 2005) store in each node routing information to eliminate unnecessary redundant messages and, at the same time, choose the more efficient path to route data streams. These approaches do not impose a specific organization by construction; instead they implicitly allow a loose structure to emerge in the overlay (Carvalho, Pereira, Oliveira, & Rodrigues 2007), which can adapt easily to changes in the membership and, at the same time, provide efficient resource usage.

Tree-based approaches perform better when there is a unique stream source, for which the tree is optimized. Multiple sources can be addressed by using a shared tree, which will result in sub-optimal efficiency. Another alternative is to create multiple trees, having each tree optimized for each different source. However, there are scalability limits to the number of sources that can be supported simultaneously as the tree maintenance cost will grow linearly with the number of sources. Multiple trees can also be used to overcome network packet losses, sending the stream or a sub-stream through each tree. This increases the probability of the stream being received at all destinations, even in the presence of packet losses or node failures. When experiencing node leaves or failures, the tree-based models must recover the tree ensuring that none of the receivers becomes disconnected.

2.5.4 Single vs Multiple Overlay

Another choice in the design of P2P streaming systems is to use an unique or multiple overlays to disseminate data. Although the use of multiple overlays is more intuitive in the case of multi-view systems (Wang, Xu, & Ramamurthy 2009; Wang, Xu, & Ramamurthy 2008), it is possible to use a single overlay to disseminate multiple streams and also to use multiple overlays to disseminate a single stream. The latter approach may be useful, for instance, if the the stream is decomposed in multiple sub-streams, so that the quality of the stream delivered to the application is dictated by the number of received sub-streams. In this case, each sub-stream may be disseminated using a different overlay, and participants with limited download bandwidth could join only a few overlays and still receive the required data.

2.5.5 Push vs Pull-Based Forwarding

There are two main strategies to control the data flow in an overlay topology. If the control is held by the node sending the data, i.e., the node that decides when and where it forwards the data to, the strategy is called *push-based*. On the other hand, if the node that receives the data has the control, i.e., it decides from when and where it requests the data from, the strategy is called *pull-based*.

Eager push-based schemes forward messages to their neighbors as soon as they are received. Thus, data is propagated as fast as the network allows. The drawback of this approach is that,

without the proper coordination, two or more nodes may forward the same message concurrently to the same node, which creates unnecessary (and, most of the time, undesirable) redundancy.

There is a variant of the push-based strategy that circumvents this effect named *lazy-push*. In lazy-push, the control is still held by the node that has the data. However, instead of sending the message payload immediately, it sends a control message to notify the potential recipient, which, in turn, indicates if the message is new or redundant. Doing so, a node will only request messages it has not received yet, preventing duplicates. Push-based schemes are suitable for systems where upload bandwidth is limited, because the sender has a better control of the number of messages it forwards.

In a pull-based strategy, the recipient has to contact potential senders, asking for new content. Therefore, it is possible that a node requests a message from one of its neighbors and fails to obtain the required message (because this neighbor has also not received it yet). This model can be effective when the download bandwidth is limited because receivers can control which packets to download and avoid message collisions.

2.5.6 Bandwidth-aware vs Bandwidth-oblivious

Many P2P streaming systems assume that all nodes have similar capacity and, therefore, try to distribute the load evenly among all participants, i.e., bandwidth-oblivious systems. However, in more realistic scenarios, is more common that different participants may have very distinct capacities, including available CPU and bandwidth. A system that is able to take the capacity of individual nodes into account when distributing the load is named *bandwidth-aware* (Ripeanu, Foster, Iamnitchi, & Rogers 2005; da Silva, Leonardi, Mellia, & Meo 2008). For instance, nodes with faster network connections can contribute more for message dissemination than low bandwidth nodes.

Bandwidth-aware systems must also address the problem of managing the competition for the available bandwidth. In a scenario where downstream peers need to be supplied from multiple upstream peers, bandwidth competition, which consists in managing the available upload bandwidth to supply all the receivers, needs to be solved in a fair way for all participants. One interesting approach to solve this problem is through the use of game theory (Wu & Li 2007; Wu, Li, & Li 2008), by modeling the bandwidth competition problem as an auction game.

Finally, a bandwidth-aware system should strive to avoid congesting links in the overlay, as a congested link presents higher delays and packet drop rates which can lead to scenarios with decreased streaming quality.

2.5.7 FIFO vs Topology-Aware Chunk Scheduling

Chunk scheduling is related to the policy that decides which packets are exchanged in each communication step. In a system where there is a single path from the source to the recipient, the most used chuck scheduling policy is FIFO; i.e., packets are forwarded by the order they have been received (which is related to their production order in the case of live streaming). However, when there are multiple paths between a node and the source, more sophisticated scheduling schemes may be used to improve load balancing among links while reducing message redundancy. For instance, a node with two paths to the source could receive half of the packets from one path and the other half from the other path by employing a suitable scheduling policy.

2.5.8 UDP vs TCP

Another important decision when building a P2P streaming system is the choice of the transport protocol, in particular between TCP or UDP.

TCP is an attractive option because of the reliability mechanisms it offers. Besides, TCP can also be used to detect neighbor failures and lower the time the system takes to reconfigure itself to recover from these failures. Finally, TCP embeds congestion control mechanisms and is favored over UDP by Internet Service Providers and carriers. On the other hand, TCP is more expensive, in terms of bandwidth, than UDP and, additionally, the occurrence of omissions results in additional delays in data delivery by the receiver, due to TCP recovery mechanisms design, which may introduce undesirable glitches in the stream.

An interesting approach explored in (Hefeeda, Habib, Xu, Bhargava, & Botev 2005) is to use both protocols. This system uses UDP for fast data dissemination, and TCP for control message exchange and failure detection.

2.5.9 Stream Splitting vs Network Coding

When performing data streaming, it may be desirable to split a data stream in several sub-streams and send each through a different path. This approach is useful to improve the resiliency to packet losses, adding data redundancy in the sub-streams, and also to perform load balancing by spreading the forwarding effort through different nodes.

The most simple approach to obtain such sub-streams is to divide the main data stream by distributing the packets that form the stream using a specific policy. A simple policy to create two sub-streams could simply be to assign half the packets to one sub-stream and half to another.

Another way to obtain the sub-streams is by using network coding techniques to encode the stream data, adding redundancy and allowing the receiver to decode a stream with only a subset of the encoded packets. This is mostly used to improve the resilience of the system to failures, by allowing nodes to, partially or completely, rebuild the data stream using only a fraction of the disseminated data. Multiple Description Coding (Padmanabhan, Wang, Chou, & Sripanidkulchai 2002) is a form of network coding that encodes the source stream into layers. A peer receiving a single layer is able to decode the original stream but the quality increases with the number of layers used in the decoding. Such mechanisms are useful to provide robustness in a P2P streaming scenario with unreliable network links and nodes.

2.6 Selected Systems

In this section, a selection of relevant systems is described, from simple application-level multicast approaches to multi-view, multi-overlay scalable streaming systems.

2.6.1 Narada

Narada (Chu, Rao, Seshan, & Zhang 2002) is an unstructured approach for application-level multicast. In this protocol, a mesh-overlay is built among participating end systems in a self-organizing and fully-distributed manner.

In Narada, every node needs to maintain knowledge about the whole group membership. To ensure this, each node uses the mesh to periodically propagate refresh messages to its neighbors

with increasing sequence number. To reduce the communication overhead of probing the whole membership, nodes only exchange refresh messages with their neighbors. Each refresh message from node i contains the address and the last sequence number that i has received in a refresh message from any other node.

When a new participant wishes to join the overlay, it contacts a list of existent nodes and connects to them. During a leave event, a node informs all its neighbors with a message that is propagated through all the group members. Failures are detected locally and propagated to the rest of the members. Failure detection consists of a node watching if a certain neighbor refresh message was not issued during a certain time. If a timeout is exceeded, the node assumes that its neighbor has failed.

The mesh is periodically optimized, dropping some links to add some others according to a utility function and the expected cost of dropping a link. The utility function measures the performance gain of adding a new link, while the cost, which depends on the number of paths that are routed through the respective link, evaluates the utility of the link to be dropped. For the sake of stability, an exchange is only performed if the associated gain is above a given threshold.

Data delivery is performed with the support of multicast trees. These trees are built over the mesh overlay using a reverse shortest path mechanism between each recipient and the source. A member M that receives a packet from source S through a neighbor N forwards the packet only if N is in the shortest path from M to S. This is verified based on the information exchanged in the refresh messages.

The multicast approach of Narada provides efficient data dissemination thanks to the optimized trees which are constructed over the mesh-based overlay. In the presence of multiple sources, several trees must be constructed whereas each tree is optimized to a single source which offers an adequate substrate for multi-source multicast. One limitation of Narada is the need to maintain the information of the whole group membership in each node, limiting its scalability.

2.6.2 Scribe

Scribe (Rowstron, Kermarrec, Castro, & Druschel 2001) uses a structured approach to Application-Level Multicast. Scribe is built on top of Pastry (Rowstron & Druschel 2001), a

generic P2P object location and routing substrate overlay for the Internet. Pastry imposes an overlay organization among the participants, according to an unique identifier generated for each node.

Pastry is fully decentralized and can locate a node in the structure with $log(N)$ complexity, where N is the total number of participants. Each node also needs to maintain routing information about $log(N)$ nodes only. Scribe uses Pastry to manage group creation and to build a per-group multicast tree used to disseminate multicast messages in the group. All decisions are based on local information, and each node has identical responsibilities. As a consequence, both Scribe and Pastry are scalable.

Scribe allows distinct multicast groups to co-exist. This groups are associated to different interest topics that user can subscribe. Each group, similarly to the Pastry participants, has an unique *groupId* and the node with *nodeId* closest to the *groupId* acts as the rendez-vous point, which means it is the root of the group multicast tree. Alternatively, Scribe allows the creator of the group to become the rendez-vous point by setting the *groupId* according to the respective *nodeId*. This alternative can be useful to optimize performance in situations where the creator will send often to the group. Both alternatives use only group and node names to generate the identifiers and do not require any name service.

The multicast tree is set during group join operations using a scheme similar to reverse path forwarding. A node sends a $JOIN$ message to the rendez-vous point, and all the nodes which forward the messages update the set of children for the respective group with the message source identifier. The $JOIN$ message is forwarded until it reaches a node which is already a forwarder for that group or to the rendez-vous point of the group. When a node performs a multicast, it needs to locate the rendez-vous point which will perform the data dissemination process. However, the rendez-vous address is cached to enhance the performance of subsequent multicasts.

Scribe provides a structured overlay to efficiently disseminate multicast data. It divides the load of the nodes through all participants using the Pastry substrate. However, it assumes all nodes have the same characteristics. On the other hand, Scribe forces nodes which are not interested in a given group to be forwarders if they are part of the routing path between any receiver and the rendez-vous associated with the group.

2.6.3 PlumTree

In Leitão, Pereira & Rodrigues (2007a), the authors propose PlumTree, an integrated approach combining epidemic and deterministic tree-based broadcast strategies. PlumTree is a tree-based approach but does not impose a specific structure to its participants. Instead, the protocol embeds a minimum delay broadcast tree in an unstructured gossip-based overlay, maintaining it with a low cost algorithm. The tree emerges from the combined use of eager and lazy push gossip strategies.

The protocol selects eager push links so that their closure effectively builds a broadcast tree embedded in the overlay. Lazy push links are used to ensure reliability, when nodes fail, and to quickly heal the broadcast tree. The membership protocol used is symmetric, simplifying the task of creating bi-directional trees. Initially, all links use eager push, resulting in a flood-like broadcast. When a node receives a duplicated message, it marks the source link as lazy push and informs the correspondent node to perform the same operation. In the presence of failures, lazy push links are used both to recover missing messages and to heal the tree. If a neighbor fails, a fact detected by the membership protocol using TCP connections, the node will change one of the lazy push links to eager push. Upon the reception of a message, there is an optimization step that verifies if there is a lazy link that is performing better than the eager push link through which the message was sent. In this case, the roles of both links are switched.

Combining eager and lazy push strategies allows the system to recover quickly from failures, and prevents messages from flooding the network when the system is stabilized. Compared to Scribe (Rowstron, Kermarrec, Castro, & Druschel 2001), PlumTree does not balance the load between nodes. In the absence of failures, the forwarders of the multicast messages are always the same. Also, opposed to Narada (Chu, Rao, Seshan, & Zhang 2002), with a single multicast tree data dissemination is optimized for a single source. If consecutive multi-source multicasts were performed, a single tree would be optimized to none of them. The protocol allows the usage of multiple trees, each optimized for one source, but the complexity of managing more than one tree limits the number of possible multicast sources in the system.

2.6.4 Adaptive Chunk Selection

The authors of Zhao, Lui & Chiu (2009) propose a distributed adaptive algorithm to enable a server to notify all peers about an optimal chunk policy. In opposition to the previous presented approaches, the policies referred in Zhao, Lui & Chiu (2009) use a pull-based model, i.e., the receiver makes the decision of which chunk will be exchanged. The policies manage a limited size buffer in each peer. In more detail, a chunk policy determines which chunk will be downloaded to the buffer by the next pull request of each peer. The protocol also assumes that a peer can download chunks directly from the main server or from another peer.

To evaluate the best policy, a priority-based chunk selection model is used to represent a large policy family. This model assumes a priority value associated to each buffer position. The highest priority can be assigned, for instance, to *chunks with lower sequence number*, downloading the chunks immediately needed to play the video (*greedy strategy*), or to *chunks with higher sequence number*, downloading the rarest chunks (the last reaching the peer buffer). The use of similar priorities for all chunks is equivalent to the use of a random policy. This priority model allows the definition of $(n-2)!$ different policies, where n is the size of the buffer (see Fig. 2.1), because the positions 1 and n are reserved for the next chunk downloaded from the main server and the next chunk to be consumed by the video player, respectively.

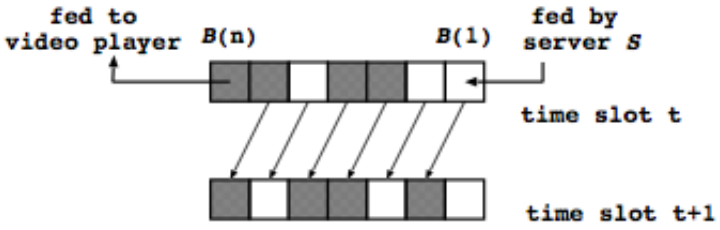


Figure 2.1: Buffer structure B with n = 7, server S fills B(1) at time slot t.

Evaluation results show that the optimal chunk policies are $\vee$-shaped, this means that, considering $B(k)$ the buffer cell with lowest priority value, priority increases as the position moves away from k. On the other hand, worst chunk policies present the inverse behavior named $\wedge$-shaped, meaning that, considering $B(k)$ the buffer cell with highest priority value, priority decreases as the position moves away from k. Another important observation is the fact that the optimal policies vary according to the fraction of downloads a peer accomplishes using the main server, f(opposed to the fraction of downloads accomplished through other peers,

$1 - f$). f depends on the main server upload capacity (C) and the number of receivers (M). The experiments described in Zhao, Lui & Chiu (2009) point to the conclusion that the optimal policies are greedier when the value of f increases. The rationale that explains this result is that with higher values of f the server has additional available bandwidth capacity to serve the peers, resulting in low chunk scarcity. Thus, peers should download those chunks which have earliest playback deadlines.

The distributed adaptive algorithm proposed in Zhao, Lui & Chiu (2009), decides which policy to be used based on f, which can be calculated by the quotient between the main server known capacity C and the number of receivers M. To estimate M, all join and leave operations are performed through a *Tracker* according to a DHT structure (similar to Chord (Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001)). The *Tracker* is responsible for maintaining M updated, for generating the IDs, and determining the neighborhood for new nodes. Peers exchange *keep-alive* messages with their neighbors to detect failures.

When M changes, the new policy is determined accordingly and peers are informed using a gossip protocol.

Under high churn rates, this approach may result in many policy changes, leading to a high message overhead due to the gossip mechanism used to update the peers with the new policy. Despite this, the authors present an evaluation of the existent chunk selection policies for pull-based communication model.

2.6.5 Most Deprived Peer, Latest Useful Chunk

Tree-based designs are able to achieve close-to-optimal rates in real-life conditions. The authors of Picconi & Massoulié (2008) propose a mesh-based communication strategy called DP/LU, which intends to prove its competitive quality against tree-based solutions. This approach differs from the one employed in Narada (Chu, Rao, Seshan, & Zhang 2002) because it does not build a dissemination tree over the mesh. Instead, the data is pushed to the node's neighbors according to a chunk scheduling strategy.

DP/LU intends to overcome the deficiencies presented by another chunk policy called DP/RU. DP/RU stands for *Most deprived peer, random useful chunk* and selects the neighbor which has received less chunks from the current node and sends it a random *useful* chunk.

A *useful* chunk is a chunk which has not been sent to that peer yet. Despite being analytically shown that this policy is rate-optimal, experiments using PlanetLab (Liang, Guo, & Liu 2008) proved that it could not achieve high rates in real world deployment.

DP/LU, which stands for *Most deprived peer, latest useful chunk* uses the same mechanism as DP/RU to choose the neighbor but sends the useful chunk with higher sequence number. Simulations show that this strategy provides better results than the previous approach and supports the usage of mesh-based overlays to perform live streaming.

2.6.6 Bandwidth-aware Clustering

In Huang, Ravindran & Kumar (2009), an approach to minimize the maximum end-to-end streaming delay in *mesh-based overlay networks* is described. Previous presented systems and algorithms assume an uniform node degree and a single-commodity flow for each receiver. These assumptions are not appropriate for P2P streaming, where peers have heterogeneous bandwidth capacities and need to aggregate multiple flows for a smooth playback.

The approximation algorithm proposed by the authors of Huang, Ravindran & Kumar (2009) clusters nodes according to a parameter λ, based on the optimal delay value. Nodes at range λ from the source form the first cluster. Then, a *representative node* is selected from these nodes. The selected node is the one with most residual bandwidth capacity. It is used to form another cluster with some of the unclustered nodes, following the same process. This process is repeated until all nodes are clustered.

The inter-cluster links resulting from the process described above are virtual. They represent the aggregated inter-cluster stream instead of the actual flows to the representative nodes. The real links are decided depending on the bandwidth capacities of the nodes in the cluster and can be divided across several participants.

The distributed version of this algorithm relies on a rendezvous point to maintain the list of the *representative nodes* to manage join operations and to support stream coordination.

This system presents an interesting approach to define clusters in a streaming overlay using bandwidth-aware mechanisms.

2.6.7 Unstructured Multi-source Multicast

In Ripeanu, Foster, Iamnitchi & Rogers (2005), the authors propose UMM, which stands for Unstructured Multi-source Multicast. This system is an attempt to use unstructured overlays to provide efficient multi-source multicast. As in Huang, Ravindran & Kumar (2009), the system is bandwidth and heterogeneity aware.

Similarly to Narada (Chu, Rao, Seshan, & Zhang 2002), this approach builds a base dense overlay where efficient distribution trees are embedded using a flood mechanism. During this process, a node which receives a duplicated message from a certain source informs the respective forwarder not to send more messages from that source through that link. This link filtering mechanism has the ability to eliminate the least efficient links. To detect network performance changes, this information is treated as soft-state and, after a certain timeout, links need to be filtered again. This link filtering mechanism has also similarities with the approach taken in PlumTree (Leitão, Pereira, & Rodrigues 2007a). The main difference is that, in PlumTree, the optimization step uses a reactive strategy to accommodate link performance changes, responding quicker than the cyclic strategy employed by this system. On the other hand, the filtering mechanism used in UMM filters messages per-source, allowing the system to optimize data dissemination for different sources, simultaneously.

Each node in UMM has half of the links labelled as short and half labelled as long. The definition of short and long depends on the latency of the link and a threshold value. During overlay optimization rounds, a node selects a small subset of peers which it knows and to whom it is not directly connected, and evaluates the potential link it may establish with these nodes. Short links are optimized for latency while long links are optimized for bandwidth. Additionally, the number of neighbors a node supports, and thus the node degree, which affects the load imposed to each participant, is proportional to its upload bandwidth capacity.

To avoid node disconnection from the overlay, heartbeat messages are broadcasted from multiple sources for failure detection. A node determines that it is disconnected from the overlay if it receives less than half of the heartbeat messages from all sources.

This system proposes a simple unstructured approach for application-level multicast and a mechanism for supporting source optimization for multi-source multicast. The link filtering mechanism information expires periodically, forcing nodes to return to a flooding strategy and

stabilize again.

2.6.8 SplitStream

SplitStream (Castro, Druschel, Kermarrec, Nandi, Rowstron, & Singh 2003) leverages on a variant of Scribe to build multiple disjoint spanning trees over the Pastry DHT (Rowstron & Druschel 2001). 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. However, the system relies 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 higher that of 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.

2.6.9 ChunkySpread

ChunkySpread (Venkataraman, Yoshida, & Francis 2006) is a tree-based approach for streaming multicast. It is an approach to distribute load over a tree-based multicast approach using multiple distribution trees. It divides the stream into M stripes and sends each one through a different multicast tree. Each node has a target and a maximum load, both expressed as a number of stripes equal or greater to zero.

This system sets the node degree proportional to its target load. The source of a multicast has M neighbors (M being the number of slices of a stream) and sends a sub-stream to each neighbor. Each of these neighbors will be the root of a multicast tree.

The propagation graph is constructed using weighted random walks with support for statistical control over nodes degree. This means that each node can control the probability with

which it is chosen by the random walks.

In this system, nodes try to replace overloaded parents with underloaded nodes. Once a node is satisfied with its neighbors concerning their load, it switches its behavior to optimize latency. Nodes identify the targets for a possible switch to improve latency using the relative time it takes for them to receive the slices of the stream.

ChunkySpread focus, primarily, in optimizing its trees concerning peers load leaving link latency improvement in second plane. Therefore, this approach may lead to sub-optimal paths.

2.6.10 CoopNet

The authors of Padmanabhan, Wang, Chou & Sripanidkulchai (2002) address the problem of distributed live streaming media content to a large number of hosts. To address this problem, they designed CoopNet, a system which attempts to build multiple short trees, providing a node with as many neighbors as its bandwidth can support. The multiple tree approach allows the system to balance the load of forwarding stream packets through all participants.

As in ChunkySpread (Venkataraman, Yoshida, & Francis 2006), data dissemination is performed using multiple trees. The difference is that tree management is centralized at the root node. Trees can be built using one of two methods: *Randomized Tree Construction*, which picks one leaf node with spare bandwidth to support a child, randomly; and *Deterministic Tree Construction*, which builds diverse trees where a node is only an interior node in one tree. In the *Deterministic Tree Construction*, new nodes are inserted in the trees with less interior nodes.

One novelty of CoopNet is the use of *Multiple Description Coding (MDC)* to provide data redundancy needed for robust P2P media streaming. MDC separates a stream in sub-streams such that the quality of the received data depends on the number of the received sub-streams. Each sub-stream is sent using a different tree. A participant receiving a single sub-stream can display the media content with less quality.

One drawback of this approach is the assumption that only the root can send data and that it holds all the tree management information. However, the tree diversity concept can help the system to distribute the multicast load among all nodes more fairly.

2.6.11 Bandwidth-Aware

In da Silva, Leonardi, Mellia & Meo (2008) a Bandwidth-Aware (BA) system is proposed. The fundamental idea behind the design of this system is to deliver chunks of a stream first to peers that can contribute more to the chunk dissemination, based on their upload capacity.

The authors of Picconi & Massoulié (2008) proposed a similar strategy that operates only at the source level. In BA peers with high upload capacity have higher probability of being chosen to deliver a chunk, not only by the source but also by any node that forwards a chunk. To make this decision, peers need to know their neighbors capacities. To support this, a signaling mechanism is employed.

BA also adjusts the neighborhood cardinality according to each node upload capacity. Notice that there is an inherent trade-off between the number of neighbors and the signaling overhead.

The algorithms proposed in da Silva, Leonardi, Mellia & Meo (2008) describe an interesting approach to allocate the heterogeneous bandwidth capacity of a P2P system. The use of probabilistic approach ensures that peers with greater upload capacity will often receive data first but does not undervalue completely nodes with lower capacities. However, the use of a push-based probabilistic model allows low capacity nodes to become starved. Combining the two push and pull-based as in PlumTree (Leitão, Pereira, & Rodrigues 2007a) is a possible strategy to mitigate this issue.

2.6.12 Divide-and-Conquer

Multi-view P2P live streaming systems have recently emerged. This type of systems enables users to simultaneously watch different channels. In Wang, Xu & Ramamurthy (2009), the authors propose a protocol for multi-view P2P streaming that can efficiently solve the inter-channel bandwidth competition problem. The proposed solution is flexible enough to incorporate any streaming protocol. The main design goals of this system were: flexibility, efficiency and scalability.

The Divide-and-Conquer (DAC) system ensures the scalability goal by dividing overlapped overlays in different independent logical overlays, one for each channel. Assigning each channel a different overlay, allows the system to solve the inter-channel competition at the channel-level

opposed to approaches which solve the problem at the peer level. DAC does not have intra-channel requirements allowing any streaming protocol to be used which allows to achieve the flexibility goal.

To provide efficient bandwidth allocation, a mathematical model, based on an utility function, is employed to determine each channel's bandwidth allocation over each overlay link. Executing this model requires the measurement of system information, which is supported by a few sampling servers that are also part of the system and are responsible to statistically capture relevant information, and periodically report it.

The system implements a solution to handle multi-channel streaming. It claims to meet the scalability requirement using a different overlay for each subscribed channel. This means that a peer will have to manage as many overlays as the number of channels it is receiving. It could become difficult to handle this configuration with many channels.

2.6.13 Channel-Aware Peer Selection

In Wang, Xu & Ramamurthy (2008), the authors propose a Channel-Aware Peer Selection (CAPS) algorithm for P2P MultiMedia Streaming (P2PMMS) applications. This algorithm goal is to optimize peer resource allocation in multi-view streaming systems.

In single-view systems there are two main bottlenecks. First, the bandwidth bottleneck, which occurs when neighbors have limited upload bandwidth to supply that peer. This issue can be solved by constructing an efficient overlay. Secondly, the content bottleneck, which occurs when neighbors have sufficient upload bandwidth but they do not have the requested content. The content bottleneck can be mitigated with adequate and efficient chunk scheduling policies.

A multi-view peer is a peer that belongs to more than one overlay and, consequently, receives more that one channel stream, increasing the flexibility of its upload bandwidth allocation. So, joining peers would benefit from setting all their connections with multi-view nodes, but this would result in two new problems. The first one is unfairness, due to the fact that early joining peers would consume most of the available multi-view peers. The second problem concerns the fact that nodes would cluster around multi-view peers building an overlay that could be quite fragile to multi-view peers leaves or failures. To solve these problems, the authors propose a new channel-aware peer selection algorithm for multi-view systems.

CAPS distinguishes peers with different types, in relation to the number of watched channels, in different groups. A joining peer chooses its neighbors based on a matrix which sets a higher probability of choosing neighbors from groups with more spare bandwidth. To do this it is required to estimate the peers total upload bandwidth capacity.

The probability mechanism used to select peers ensures the neighbor diversity, providing a good mechanism to obtain a balanced overlay.

2.6.14 Strategies of Conflict in Coexisting Streaming Overlays

In Wu & Li (2007), the authors propose game theory based algorithms to solve bandwidth competition in coexisting overlays. Coexisting overlays are used in IP TV systems with multiple channels where, usually, each channel is transmitted through a distinct overlay. The authors of Wu & Li (2007) propose decentralized strategies to resolve conflicts among coexisting streaming overlays.

The proposed approach uses game theory concepts, and models the upload bandwidth sharing problem of a node, called the seller, between its peers, called the players, as an auction game. In each round, players place their bids to the sellers they know about. A bid consists of a 2-tuple with the values of the requested bandwidth and the price which the player is willing to pay for it. The sellers job is to analyze the bids and choose the ones that carry out the most profit. To do this, in an auction game, the seller chooses the bid with highest price and allocates the required bandwidth for that player limited by the available bandwidth it still owns. If it still has available bandwidth after this allocation it repeats the process for the remaining bids.

A player is allowed to bid multiple sellers. Its objective is to allocate the required bandwidth it needs. When a player receives a stream from a seller, there are two kinds of costs associated. First, the streaming cost, which is expressed as a function of the requested bandwidth and represents the streaming latency actually experienced by the player. Second, the bidding cost, which is expressed by the bid made to seller and represents the degree of competition and demand for bandwidth in the auction games. A player should place its bids in order to obtain the required bandwidth, minimizing the cost function which is expressed by the sum of both the referred costs.

A player also adjusts its bidding price according to the previous bidding round. It starts

with a bidding pricing of 0. When the upstream peer allocates its bandwidth, it sends the results to the players and, if the player has not been fully supplied it increases the bid prices, otherwise decreases it.

In order to achieve a prioritization mechanism, the authors set a limited budget and a player bidding cost cannot exceed this budget. With this mechanism, overlays with higher budgets will perform better achieving higher streaming rates. This approach is useful on IPTV systems to enhance, for instance, the quality of premium channels.

This approach offers a set of strategies to share an upstream peer bandwidth between a set of downstream peers in a fully decentralized way. The proposed algorithms are targeted for multiple overlay bandwidth competition scenarios, and they provide an interesting solution for multi-view systems which face such competition issues.

The proposed strategies tackle the problem of bandwidth sharing in multi-overlay scenarios but do not account for lossy or congested links. If these parameters were incorporated within these strategies, auctions could make a better use of available upload bandwidth.

2.6.15 CollectCast

In Hefeeda, Habib, Xu, Bhargava & Botev (2005), the authors propose CollectCast, a P2P streaming system that assumes a multiple-to-one streaming model. In CollectCast each receiver of a stream gets data from multiple sources in order to achieve the required download rate needed for continuous video playing.

CollectCast service is layered on top of a P2P lookup substrate, and it is independent of the chosen substrate. The lookup service is required to return a candidate set of sender peers for a certain media file. The authors present tomography-based techniques (Coates, Hero III, & Nowak 2002) that use inferred network topology information and select peers from the candidate set accordingly. This approach can detect situations where end-to-end measures are not accurate. For instance, if paths from two senders to the receiver overlap in the same physical link, it is not possible to receive at a rate equal to the sum of both paths due to the common physical link which is a bottleneck. This approach has considerable overhead. The peer selection method chooses the most appropriate peers for the streaming session and the remaining candidate set is simply left in stand by mode.

In CollectCast a receiver uses lightweight UDP channels to receive stream data but it also owns reliable TCP channels for exchanging control messages. The control channels are used to detect node failures and stream flow degradations. When both situations occur, the respective peer is replaced with the best candidate in standby, according to the peer selection method.

CollectCast makes an effort to maintain a high quality level of the received stream, by setting multiple sources to feed a single receiver. The approach taken is not applicable for a system where every peer is both a receiver and a sender at the same time, as in P2P live streaming applications. However, some of the choices made by the authors could enhance the quality of the received streams in those scenarios as well. The multiple-to-one communication models used by CollectCast also fits nicely on mesh-based overlays.

2.6.16 Dynamic Video Rate Control

In Papadimitriou, Tsaoussidis & Mamatas (2008), the authors propose Dynamic Video Rate Control (DVRC), a receiver-centric protocol resilient to congestion and adaptive to network packets losses. The protocol attempts to maintain the video streaming desired smoothness, avoiding at the same time the significant damage usually caused due to network congestion.

DVRC is an extension to the UDP protocol. It employs acknowledgments not to perform retransmissions of lost packet but to estimate the RTT and loss rate values. There are two different strategies to adjust a stream rate: additive and multiplicative. Additive adaptation has the advantage of performing smoother changes while multiplicative adaptation responds faster to overloading situations.

The authors discuss two approaches to adjust the received stream rate according to experienced load and both employ an additive increase to perform a smooth increase on the received rate when the experienced load is under the target value. The difference remains on the decrease strategy:

Additive Increase Additive Decrease (AIAD): When the receiver is overloaded AIAD reacts slowly, using an additive strategy, resulting in a poor adaptation to critical situations.

Additive Increase Multiplicative Decrease (AIMD): AIMD approach employs a multiplicative decrease, reacting faster, but it can cause some unnecessary instability when random losses occur, because of its aggressive adaptation strategy.

To overcome these problems exhibited by both these approaches and to and get the best of both worlds, DVRC employs an hybrid strategy called Additive Increase Additive Multiplicative Decrease (AIAMD). This approach uses the receiving rate history to decide when to use an additive or multiplicative rate adaptation. To maintain the receiving rate history, the protocol updates a moving average value at each received rate measure. The authors divide the elapsed time in epochs. An epoch is the elapsed time between two lost packet events. At the end of each epoch the experienced received rate is measured and the rate moving average is updated. If the difference between both values exceeds a certain threshold, multiplicative adaptation is performed to quickly recover from congestion. Otherwise, the additive strategy is used in order to provide a smooth adaptation.

The proposed protocol is aimed at client-server systems. However, the approach could be extended to perform congestion control in P2P streaming applications. Assuming multiple sources, the main issue would be to define a strategy for choosing which peers to increase or decrease the exchanged rates.

The protocol's goal is to maintain the receiver load under a target value, but at the cost of decreasing the received rate, resulting in a lower stream quality. Again, in P2P systems, this issue could be solved by taking advantage of the flexibility in changing the stream sources.

Summary

In this Section, we presented some of the most significative approaches to support P2P multicast operation discussing the impact of their design choices. An interesting observation that steams from this overview of related work is that unstructured approaches such as PlumTree, which deduces dissemination trees from the data flows, are able to combine the natural resiliency of such approaches with some of the efficiency of structured approaches.

Systems which deliver content in a bandwidth-aware fashion typically organize the participants according to their bandwidth capacities which, usually, results in situations where data is delivered through non-optimized paths. Tree-based approaches are able to disseminate data using optimized and efficient paths, however they allow situations where some nodes become overloaded in the dissemination tree. To address this problem many systems adopted the multiple tree design. This design requires a mechanism to split the original streams in sub-streams

which are disseminated using different trees.

Some streaming systems intended for single-receiver scenarios were also presented, which do not address live streaming issues, but identify some details which may enhance data communication of those systems. These systems address issues of the used communication protocols and adaptation of the stream download rates during overloading situations that may be useful in live streaming scenarios.

Next chapter introduces Thicket, a protocol to build and maintain multiple spanning trees over an unstructured P2P overlay network. This protocol operates in a fully distributed fashion and creates the multiple trees in a way that each node is only interior in one or few of all trees. This property guarantees load distribution across all participants, also avoiding overloading scenarios. Besides, each node keeps its forwarding load under a certain threshold.

3 Thicket

3.1 Introduction

This chapter describes Thicket, a protocol for building and maintaining multiple trees in an overlay network. The protocol operates on top of a random overlay network and ensures that each of the participant nodes is only interior in 1 or few of all trees. This configuration forces all nodes to contribute to support the system load, also limiting the load of each node to a certain threshold.

The chapter is organized as follows. Section 3.3 describes the target environment in which Thicket operates. Section 3.4 enumerates the protocol desirable properties. An architectural description is presented in Section 3.6. Section 3.7 focuses on the protocol operation, describing its main functionality. A discussion of the protocol parameter values is done in Section 3.7.5.

3.2 Design Space

Table 3.1 illustrates several relevant combinations in the design space for the addressed problem, providing some notable examples of solutions for each region. Partially centralized category includes all systems that rely on centralized components to perform a required functionality. Bayeux (Zhuang, Zhao, Joseph, Katz, & Kubiatowicz 2001), for instance, uses centralized mechanisms to manage the membership while CoopNet (Padmanabhan, Wang, Chou,

	Partially Centralized	Decentralized	
		Structured overlay	Unstructured overlay
Single tree	Bayeux	Scribe	Mon, Plumtree
Multiple tree	CoopNet	Splitstream	Chunkyspread, *THICKET*

Table 3.1: Thicket in the design space

& Sripanidkulchai 2002) centers the multiple tree management on the root node.

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.

3.3 System Model

The Thicket protocol is intended for P2P overlay networks. More precisely, a random overlay network where each node has a partial view of the whole system, keeping knowledge of just a small subset of all nodes. If a node has some other node in its partial view, there is a link connecting both nodes. In this model symmetric links are considered, therefore if node n_1 contains n_2 in its partial view then n_2 also contains n_1 in its partial view. Thus, the resulting overlay denotes an undirected graph.

Nodes have limited bandwidth capacity, if they send or receive too much data during a time interval they become congested. Also, failures can occur forcing nodes to leave the overlay without notice.

3.4 Protocol Desirable Properties

The goal of this work is designing a protocol with several properties:

Load Distribution The Thicket protocol was designed to support streaming applications. Therefore, to ensure a constant data flow, the forwarding load should be divided across all nodes. So, the protocol must ensure that the trees are created in such a way that load is distributed across all participants. This property avoids overloading situations where nodes may become congested due to the high number of messages they are required to forward, during a time interval.

Resiliency The protocol is intended for unstable environments as the Internet, where nodes are constantly joining and leaving the overlay network. Therefore, Thicket must be resilient to constant changes in the system membership. Also, because nodes can leave the overlay

due to the occurrence of failures, the protocol must also be fault-tolerant, recovering from failures of one or more participants and guaranteeing the data reception to all the correct nodes.

Scalability Finally, Thicket is designed to support a large number of participants. Thus, it must scale as the number of nodes present in the overlay increases. This issue is addressed by designing a protocol which is fully distributed making each participant managing the same amount of data and the same load independently of the number of nodes present in the whole system.

3.5 Some Naive Approaches

The goal of this work 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. This approach is named *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. This approach is named *Basic multiple OverLay-TreeS*, or simply, BOLTS.

These two basic "nuts and bolts" strategies have been implemented to assess how good they

perform in practice. Their resulting performance was analyzed to extract some guidelines for the design of Thicket.

For these experiments HyParView (Leitão, Pereira, & Rodrigues 2007b) was used 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 the goals of this work, because it makes load balancing easier. Plumtree protocol (Leitão, Pereira, & Rodrigues 2007a) was used for building the trees. Plumtree embeds a tree in topologies such as the ones created by HyParView.

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

Both strategies were evaluated by simulating a system composed of 10.000 nodes, and the target of building 5 independent spanning trees (the experimental setup employed will be described in detail in Chapter 4). For NUTS a single HyParView instance was employed with a node degree of 25. For BOLTS each of the HyParView instances was configured 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 (Leitão, Pereira, & Rodrigues 2007b). These configurations ensure that each node has an identical number of overlay links in both approaches. Figure 3.1 plots the percentage of nodes that are interior in 0, 1, 2, 3, 4, and 5 trees.

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. 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,

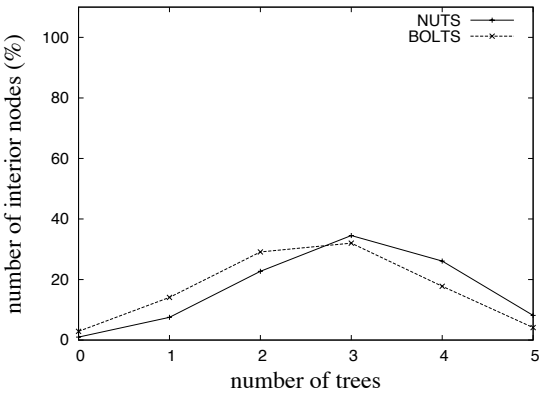


Figure 3.1: K-interior node distribution.

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.

3.6 System Architecture

Each node of a system using the Thicket protocol uses the components depicted in figure 3.2. Each component

Transport Layer The bottommost layer which interacts directly with the network. It handles the received messages, delivering them to the respective layers above, and scheduling the dispatch of sent messages by the current node.

Peer Sampling Service Defines the links for the random overlay network. This layer offers an interface to provide the above layers data concerning the partial views of the current nodes. The proposed design assumes a reactive strategy for this component, meaning that the contents of partial views maintained by nodes are only updated in reaction to external events such as peer joining or leaving the system. Therefore, when this changes occur suffers the Peer Sampling Service is required to notify the above layers using $NeighborUp(p)$ and $NeighborDown(p)$ calls.

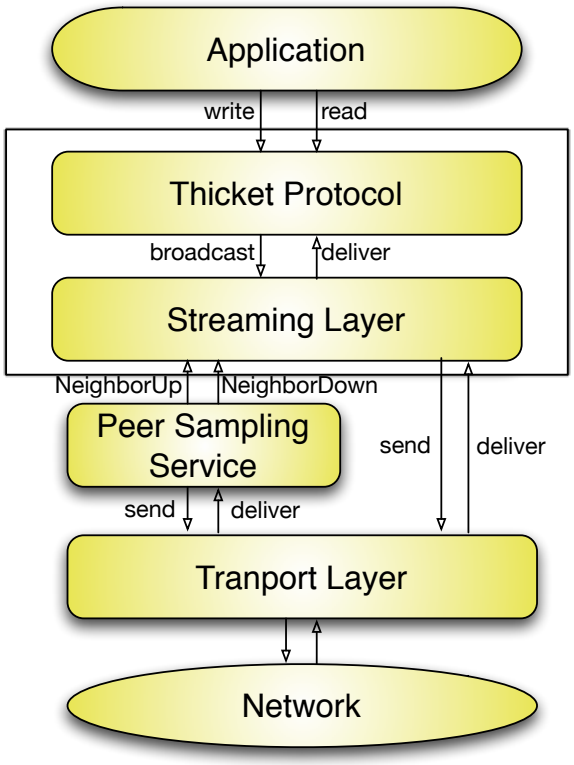


Figure 3.2: Components of a node using Thicket Protocol

This component specific messages are exchanged using the Transport Layer.

Thicket Protocol Is immediately above the Peer Sampling Service and is responsible for building and managing the multiple spanning trees used during the message dissemination. This layer uses the partial view information provided by the Peer Sampling Service to set the possible tree branches. The Thicket layer is also responsible to perform repairing processes during node failure situations.

It also uses the Transport Layer directly to exchange the protocol messages.

Streaming Layer Interacts with the Application from which receives the data stream to be broadcasted and splits it in several chunks. These chunks are then disseminated using the Thicket protocol. On the receiver side, an incoming set of packets delivered by Ticket is rearranged to obtain the original stream which is then read by the Application.

This layer can introduce data redundancy for fault-tolerance purposes.

3.7 Protocol Operation

Thicket operates by employing a gossip-based technique to build T divergent spanning trees (T is a protocol parameter, the selection of values for T is discussed later in the chapter), 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*.

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

Algorithm 1: Data Structures & Initialization

```
1  data structure Tree
2      field activePeers : Set

3  data structure Load : int[]

4  upon event Init do
5      foreach t ∈ trees do
6          t.activePeers ⟵ ∅
7      backupPeers ⟵ getPeers()
8      announcements ⟵ ∅
9      receivedMsgs ⟵ ∅
10     loadEstimate_p(t) ⟵ ∅
11     currentFlow_p(t) ⟵ 0
```

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. Later it will be explained 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.

A variable $currentFlow_p(t)$ is maintained by each node containing the identifier of the current data flow state for each node in each tree. This information is used to verify if received messages are part of the current data flow sent to a node by one of its peers or if they belong to a outdated flow. By verifying this, the protocol is able to reduce the number of control messages exchanged when breaking redundant data flows.

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[1]. 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 $loadEstimate(p,t)_n$.

[1]It is assumed 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 $t.activePeers_n$ minus 1).

Therefore, the goal of Thicket is to build, and maintain, the *activePeers* sets associated to each spanning tree, ensuring the constraints discussed earlier.

3.7.1 Tree Construction

Algorithms 2 and 3 depicts a simplified version of the pseudo-code for the tree construction procedure. Some obvious aspects from the pseudo-code have been omitted (for instance the update of the *loadEstimate*) to improve its readability.

Algorithm 2: Tree Construction (1/2)

```
1  upon event Broadcast(m) do
2      tree ⟵ nextTree()
3      muid ⟵ (nextSqnb(), tree)
4      trigger Deliver(m)
5      if tree.activePeers = ∅ then
6          call SourceTreeBranching(tree)
7      call Forward (m, muid, tree, myself)
8      receivedMsgs ⟵ receivedMsgs ∪ {muid}

27 procedure SourceTreeBranching (tree) do
28     peers ⟵ getRandomPeers(backupPeers, f)
29     foreach p ∈ peers do
30         tree.activePeers ⟵ tree.activePeers ∪ {p}
31         backupPeers ⟵ backupPeers \ {p}

38 every T seconds do
39     if ∑_t Load < maxLoad then
40         SUMMARY ⟵ GetNewSummnary (receivedMessages)
41         foreach p ∈ backupPeers do
42             trigger send(SUMMARY, Load)

43 procedure Forward (m, muid, tree, sender) do
44     foreach p ∈ tree.activePeers: p ≠ sender do
45         trigger Send(DATA, p, m, muid, Load, currentFlow_sender(tree), tree, myself)
```

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 $t.activePeers_n$ set. After this, the source initiates the dissemination of data messages in each tree t, by sending messages to the nodes in $t.activePeers_n$.

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[2]. Periodically, each node n sends

[2]For techniques on how to garbage collect obsolete information from this set see for instance (Kaldehofe 2003).

Algorithm 3: Tree Construction (2/2)

```
9  upon event Receive (DATA, m, muid, load, flowID, tree, sender) do
10     if muid ∉ receivedMsgs then
11         trigger Deliver(m)
12         receivedMsgs ⟵ receivedMsgs ∪ {muid}
16         if tree.activePeers = ∅ then
17             if sender ∈ backupPeers then
18                 tree.activePeers ⟵ tree.activePeers ∪ {sender}
19             backupPeers ⟵ backupPeers \ {sender}
20             call treeBranching(tree)
21         call Forward (m, mID, round+1, tree, myself)
22         call Balance (mID, mask, tree, sender)
13         if ∀ (id) ∈ missingFromTree(announcements, tree) : id = muid then
14             cancel Timer(mID)
15         announcements ⟵ removeMuid(muid, announcements)
23     else
24         if currentFlow_sender(tree) >= flowID then
25             flowID_sender(tree) ⟵ flowID
24             tree.activePeers ⟵ tree.activePeers \ {sender}
25             backupPeers ⟵ backupPeers ∪ {sender}
26             trigger Send(PRUNE, sender, tree, flowID_sender(tree), myself)

32 procedure TreeBranching (tree) do
33     if ∄ t ∈ trees : |t.activePeers| > 1 then
34         peers ⟵ getRandomPeers(backupPeers, f − 1)
35         foreach p ∈ peers do
36             tree.activePeers ⟵ tree.activePeers ∪ {p}
37             backupPeers ⟵ backupPeers \ {p}

46 upon event Receive (PRUNE, load, tree, flowID, sender) do
25     flowID_sender(tree) ⟵ flowID_sender(tree) + 1
47     tree.ActivePeers ⟵ tree.ActivePeers \ {sender}
48     BackupPeers ⟵ BackupPeers ∪ {sender}
```

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 $t.activePeers_n$. Furthermore, if $\nexists t' : |t'.activePeers_n| > 1$ (i.e., the node is not interior in some other tree t'), then n moves at most $f - 1$ peers from $backupPeers_n$ to $t.activePeers_n$. 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.

If the message is not found to be a duplicate (by inspecting the $receivedMsgs_n$ set), it is forwarded to the nodes in $t.activePeers_n \backslash \{s\}$. On the other hand, if the received message is a duplicate, the node moves s from $t.activePeers_n$ to $backupPeers_n$ and sends a PRUNE message back to s. Upon receiving the PRUNE message, s will move n from $t.activePeers_s$ 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.

Notice that, when receiving a data flow, the described operation to remove redundant links will not break a data flow instantaneously so, n may continue to receive redundant messages. In order to prevent that this messages trigger the sending of unnecessary PRUNE message, $currentFlow_n(t)$ is compared to the flow identifier received from s and only takes the measures to break the tree branch if the flow is not outdated, which means the value received should be greater or equal than $currentFlow_n(t)$.

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 this 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.

3.7.2 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 each node. As stated before, 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 $loadEstimate(p,t)_n$ at the beginning of this section). 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.

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). This is the only case where the load data added in a message does not correspond to the sender but to the receiver of the message.

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 informa-

tion 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 (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 4 depicts a simplified version of this procedure in pseudo-code.

Algorithm 4: Tree Repair

```
1  upon event Receive (SUMMARY, load, sender) do
2      foreach (muid, p) ∈ SUMMARY do
3          if ∄ Timer(t) : t = muid.t then
4              setup Timer(muid.t, timeout)
5          announcements ⟵ announcements ∪ {(muid, sender)}

6  upon event Timer(tree) do
7      (muid, p) ⟵ removeBest(announcements, tree)
8      tree.activePeers ⟵ tree.activePeers ∪ {p}
9      backupPeers ⟵ backupPeers \ {p}
10     trigger Send(GRAFT, p, muid, loadEstimate_p, tree, myself)

11 upon event Receive (GRAFT, muid, load, tree, sender) do
12     if ∑_t Load < maxLoad ∧ sender ∈ tree.backupPeers ∧
13             (|tree.activePeers| > 1 ∨ load = Load) then
13         flowID_sender(tree) ⟵ flowID_sender(tree) + 1
14         tree.activePeers ⟵ tree.activePeers ∪ {sender}
15         backupPeers ⟵ backupPeers \ {sender}
16         trigger Send(DATA, sender, m, muid, Load, currentFlow_sender(tree), tree, myself)
16     else
17         trigger Send(PRUNE, sender, Load, tree, myself)

18 procedure Balance (muid, load, tree, sender) do
19     if ∃ (id, p) ∈ announcements : id.t = tree then
20         newLoad ⟵ IncTreeLoad(loadEstimate_p, tree)
21         if nInterior(newLoad) < nInterior(load) then
22             trigger Send(GRAFT, n, null, loadEstimate_p, t, myself)
23             trigger Send(PRUNE, sender, Load, tree, myself)
```

3.7.3 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 (the coverage is assured by the repair mechanism, as confirmed by experimental results presented in Chapter 4). 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, a reconfiguration procedure was developed 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 with the estimated load of a assuming there is a tree branch between a and n.

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.

Comparing the estimates of s and a assuming, in both cases, that there is a branch to n prevents oscillating situations where a node is constantly changing its parent in a certain tree.

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 reduce 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.

3.7.4 Network Dynamics

Algorithm 5: Overlay Network Dynamics

```
1  upon event NeighborDown(node) do
2      foreach tree ∈ trees do
3          tree.ActivePeers ⟵ tree.ActivePeers \ {node}
4      BackupPeers ⟵ BackupPeers \ {node}
5      foreach (muid,s) ∈ announcements : s = node do
6          announcements ⟵ announcements \ {(muid,s)}

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

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 5).

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).

3.7.5 Parameters

This section identifies and discusses the parameters used by the Thicket protocol during its operation. First, the number of trees T is debated. Next, the degree of the unstructured overlay is discussed and related to the T parameter. Finally, a debate over the *maxLoad* parameter is presented.

3.7.5.1 Number of Trees 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 tree 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[3].

3.7.5.2 Unstructured Overlay Degree

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

[3]The value of f used in the evaluation was determined experimentally. This value is related with the *fanout* of gossip protocols that operate over symmetric overlay networks (Leitão, Pereira, & Rodrigues 2007b)

to receive the messages disseminated through the remaining tree) however this would render a decentralized mechanism to build such trees infeasible. Therefore, overlay degrees were set 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.

3.7.5.3 *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).

3.8 Implementation

The Thicket protocol was implemented using two different platforms. First, the PeerSim Simulator was used to validate the dissemination scheme and the properties of the created trees. Next, the Java prototype permitted the evaluation of the protocol performance in a real-world scenario.

This section describes each platform, how the protocol was implemented on it and the problems that appeared during the process.

3.8.1 PeerSim Implementation

3.8.1.1 Overview

PeerSim is a peer-to-peer simulator which allows to validate distributed protocols in a fast and easy way, simulating large overlays with thousands of nodes. The simulator has two operating modes: cycle-based and event-based. The cycle-based engine, to allow for scalability, uses some simplifying assumptions, such as ignoring the details of the transport layer in the communication protocol stack. The event based engine is less efficient but more realistic. Among other things, it supports transport layer simulation as well. In addition, cycle-based protocols can be run by the event-based engine too.

3.8.1.2 Cycle-driven approach

Thicket was first implemented using the cycle-driven engine of PeerSim. To that end, a Java class was developed implementing the protocol state variables and handlers for all protocol messages. Each cycle of execution triggered a broadcast step which sent a broadcast message through each tree.

All sent messages were stored in a queue to be processed using FIFO strategy. This process was made by two phases. First, all messages resulting from the gossip dissemination mechanism were processed. Then, a repairing phase was executed, triggering the recovery of not received messages and healing the built trees.

This implementation allowed the observation of the protocol operation but it did not simulate the network delays and the timeouts of the repairing mechanism due to changes in network traffic delivery. This happens because cycle-driven engine permits the definition of protocol behavior which will be executed in every execution cycle. However, this engine only permits the definition of periodic behavior which is not suitable to simulate message delivery.

3.8.1.3 Event-driven approach

Since cycle-driven engine is not able to model the network behavior, Thicket was also implemented using the event-driven engine. This implementation allowed to set values for message delays and to better simulate the triggering of repairing timeouts.

The event-driven engine permits to set different delays to various events. This way message latencies and timeouts could be simulated as event delays. The code developed for the cycle-driven approach was adapted to use the event-driven engine. A transport layer was also developed which increases the message delays when the amount of data sent per broadcast cycle was too large, simulating overloading situations.

With this approach, it was possible to better simulate the network behavior and also node congestion.

3.8.1.4 Details

During the development of the PeerSim prototype, some simplifications of the Thicket protocol were performed:

- None of the messages contained the actual data to be sent because it would increase the memory consumption with no benefit for the extracted results.
- The *receivedMsgs* set contains the identifiers of all received messages at any given moment. This means that no process was employed to garbage collect useless data.

3.8.2 Java Prototype

3.8.2.1 Overview

The Java prototype permitted the evaluation of Thicket in real-world scenarios. This section describes the components included in the prototype implementation and discusses the some issues encountered during the development of the prototype.

3.8.2.2 Components

To deploy a system using the Thicket protocol, some components were required opposed to the PeerSim prototype:

Transport Layer A non-blocking transport layer was used to handle the message exchange. This layer supplied an interface to send messages without blocking the application and also a set of callbacks to notify the above protocols of the received messages they are interested in.

Streaming Layer The Streaming Layer was implemented to act as both an input and output stream. In the source node, the stream to be broadcasted is written to the Streaming layer which stores data in a buffer until the size is enough to fill a packet. At this time, the packet(s) are sent using the Thicket protocol. When acting as an input stream, the Streaming Layer waits until it receives the chunks and inserts them in an internal buffer,

in correct order, waiting for a read operation. If a read is performed and the next chunk is not available the operation will block until the missing data is received.

Application Two different types of applications were developed. First, a *FileSharing* application which uses the Thicket Protocol to stream file contents to all recipients. Next, a *MP3Player* application that permits the broadcasting of a music file to be received and played at all the participants in the overlay network.

3.8.2.3 Forward Error Correction

In order to make the developed prototype more resilient to failure scenarios, the streaming layer encodes data segments using a Forward Error Correction (FEC) Library. With FEC, a segment composed of N chunks can be encoded in M chunks in a way that the reception of N of the encoded chunks permits the decoding of the original data segment. This is accomplished by introducing redundancy on the encoded chunks.

This feature is useful to tolerate temporary disconnections of nodes to a subset of trees. Using FEC, the messages received from remaining trees can still be used to obtain the original data.

3.8.2.4 Issues

During the development of this prototype, several issues were encountered in the protocol operation that required special attention:

Tree Repair Considering an incoming data stream, when a node n becomes disconnected from a tree t, several messages will not be received and, consequently, the *muids* of those messages will be present in the SUMMARY messages received from the neighbors. At this point, n needs to recover the messages starting from the one with the lowest *sqnb*. Therefore, each node stores the next *sqnb* to be received in each tree in a variable $nextSqnb_t$. This information is used to choose the neighbor during the repairing process, a neighbor p is chosen to repair tree t if there is an *announcement* entry with *muid* (*sqnb*, t) for it, such that $sqnb = nextSqnb_t$. When n repairs the branch to p for a tree t, n will send all data messages with sequence number between the requested value in the respective GRAFT message and the $nextSqnb_t$ of n.

Garbage Collection of Received Messages To determine when a received message will no longer be useful and discard it, an interval of time Δt was set as the minimal time that a message should remain in the $receivedMsgs$ set. To accomplish this feature, the *muids* of the messages in the $receivedMsgs$ set are inserted in a queue. Then, every Δt the number of messages in the queue is recorded and, in the next execution of the periodic handler those messages are removed, leaving only the messages added in between the current and last handler executions. This process guarantees that the messages are maintained at least for Δt.

Algorithm 6: Message Garbage Collection

```
1  n = 0

2  every Δt seconds do
3      receivedMsgs ⟵ removeFirst(receivedMsgs, n)
4      n ⟵ |receivedMsgs|
```

Storing Announcements Every data message received by a peer results in an entry in the next SUMMARY message sent to each of the $backupPeers_n$ set. Considering the rates of streaming applications this would lead to a large amount of data to be managed in the receivers of SUMMARY messages. To better store this data, announcements are stored in intervals composed of only two integers. An interval is stored for each neighbor and each tree. This way the memory requirements of storing the announcements set are reduced.

Store Outdated Data Flows as Announcements Due to the highly unstable delays observed in the Internet, a node n may receive messages through a tree t from two different peers interleaved in a way that at least one message from each neighbor results in a duplicate. This phenomena triggers the sending of a PRUNE message for both peers leaving n disconnected from t. To avoid this scenario, Thicket uses the $currentFlow_n(t)$ variable to detect if an incoming data flow is outdated (which happens if a PRUNE message was already issued to the flow sender). If a fresh message is then received from a peer to which a branch was previously dropped, the message is simply treated as an announcement and will only be delivered if the corresponding timer expires.

Summary

This chapter presents the Thicket protocol motivating its design, describing its target environment, the architecture of a node using the protocol and the way it operates, creating multiple spanning trees with very few interior nodes in common. A discussion on the most important parameters is also provided. The chapter also described how Thicket protocol was implemented describing both the platforms used to evaluate it: PeerSim and Java Prototype. Some particular details of each implementation were discussed.

Next chapter presents the evaluation of the two implementations described in this chapter.

4 Evaluation

4.1 Introduction

This Chapter reports experimental results obtained when evaluating the Thicket protocol on both implementations described on Section 3.8. The PeerSim evaluation validates the proposed tree design and the Java prototype deployed in the PlanetLab infrastructure evaluates its performance in a real network.

4.2 PeerSim

4.2.1 Overview

This section presents the results obtained using the PeerSim simulator (Jelasity, Montresor, Jesi, & Voulgaris). All results presented in this section were obtained using the event-based engine of the simulator. In order to extract comparative figures the performance of the single-tree Plumtree protocol (Leitão, Pereira, & Rodrigues 2007a) (that serves as baseline to the proposed solution) as well as the "NUTS and BOLTS" alternatives discussed in Section 3.5 was also tested. For fairness, all protocols were executed on top of the same unstructured overlay, maintained by the HyParView protocol (Leitão, Pereira, & Rodrigues 2007b). 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, message losses were not modeled in the system (TCP is also used to detect failures).

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

4.2.2 Experimental Settings

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 it is assumed the usage of perfect links, however messages are not delivered to nodes instantly, instead the following delays are considered when routing messages between nodes (these delays are implemented by using the event based engine of the simulator[1]):

Sender delay It is assumed 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 each node can transmit $200K$ bytes/s. Furthermore it is assumed that the payload of data messages had 1250 bytes, while SUMMARY messages have 100 bytes.

Network delay It is assumed 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. These values were selected by taking into consideration round trip time measurements that were performed using the PlanetLab infrastructure[2].

All the experiments were conducted 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, Thicket was configured 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

[1]The minimum time unit in the system is 1ms.
[2]The measurements can be found in `http://pdos.csail.mit.edu/~strib/pl_app/`

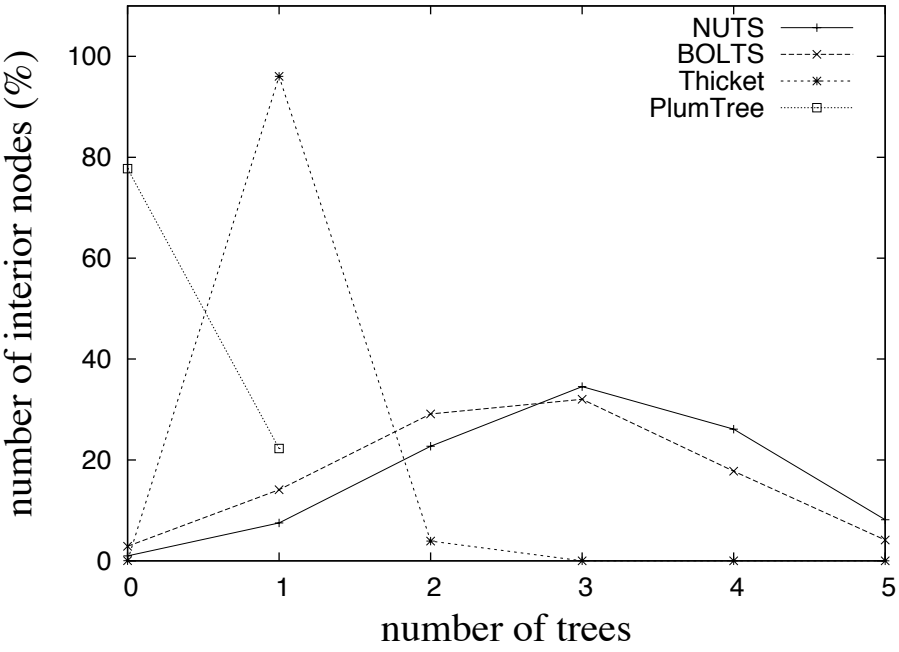


Figure 4.1: K-interior node distribution in a stable environment.

the overlay topology stabilizes. After this stabilization period, the broadcasting process starts; this triggers the construction of trees.

4.2.3 Stable Environment

The relevant performance metrics were analyzed for Thicket in a stable environment where no node failures are induced. First, it was evaluated the distribution of nodes accordingly to the number of spanning trees in which 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 leafs in all trees. The results are depicted in Figure 4.1. 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 (the plot from Section 3.5 is repeated here 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 the 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 4.2 depicts the distribution of forwarding load in the 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 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.

Experiments to evaluate the effect of Thicket in the dissemination of payload messages were also conducted. In particular it was 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 4.3 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

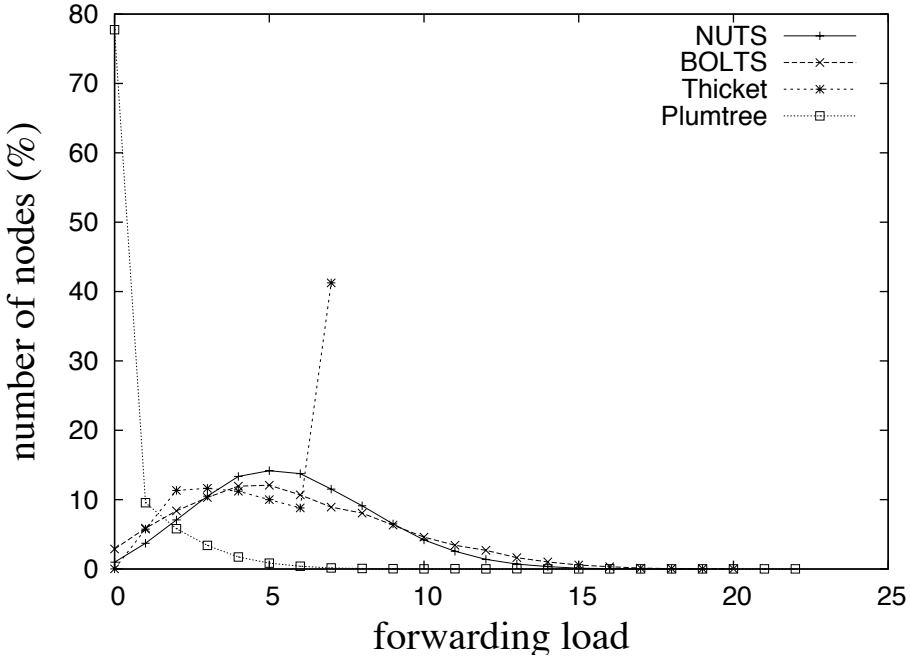


Figure 4.2: Forwarding load distribution in a stable environment.

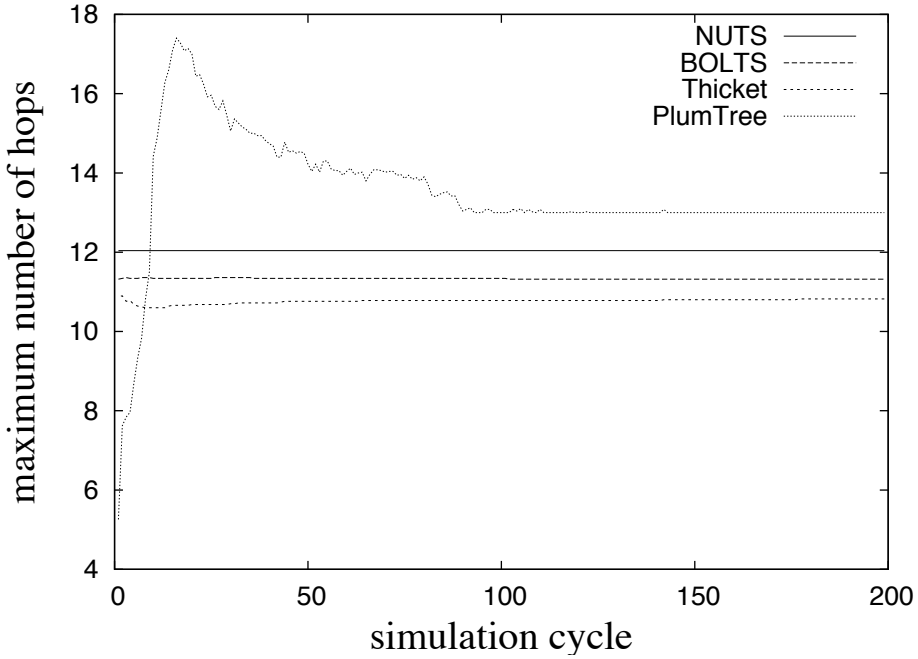


Figure 4.3: Number of maximum hops in a stable environment.

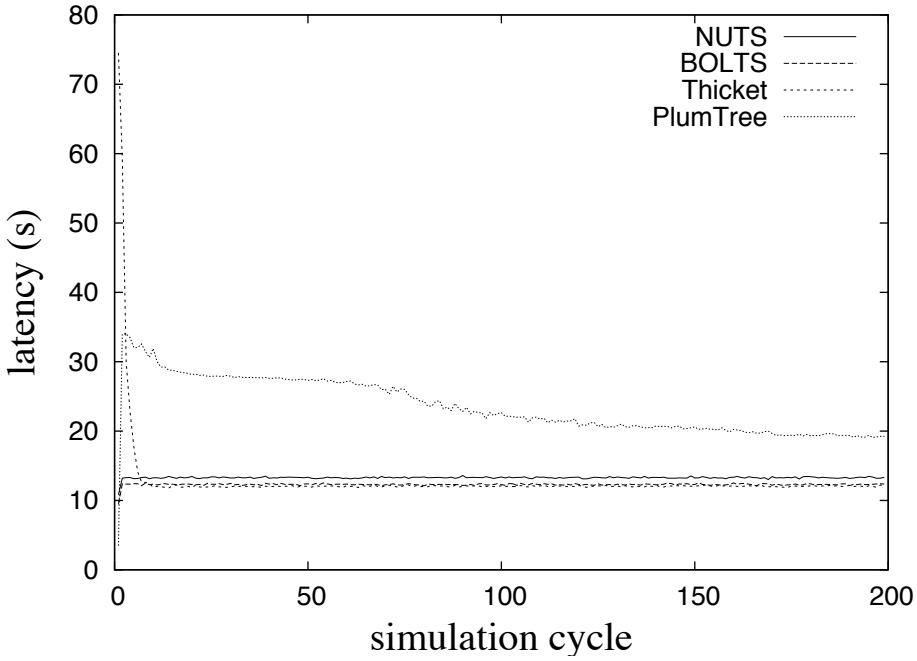


Figure 4.4: Latency in a stable environment.

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 4.4 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.

4.2.4 Fault-Tolerance

In this section the performance of Thicket was evaluated in two distinct failure scenarios. In particular the study the impact of sequential node failures in the broadcast reliability when using Thicket, NUTS, and BOLTS. Later, 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 the experiments the source node and the nodes that serve as root for trees in NUTS never fail.

4.2.4.1 Sequential Node Failures

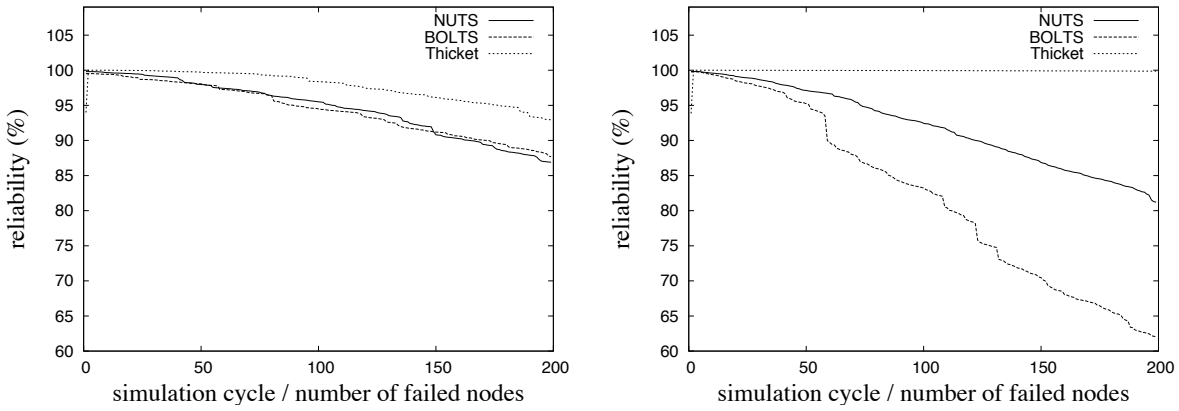
Now the reliability of the broadcast process in face of sequential node failures is depicted. Here the reliability is considered assuming that the broadcast process leverages in the co-existing spanning trees to introduce redundancy in the disseminated data (for instance by using network coding techniques). Furthermore it is assumed 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 it is considered that the node misses the reception of this segment. Reliability was defined here as the percentage of correct nodes that are able to reconstruct disseminated data segments.

After an additional stabilization period (5 cycles) the source node was configured to disseminate a data segment per cycle. In each cycle a single node is also forced to fail. The reliability of the broadcast process was measured at the end of each simulation cycle. Furthermore, the node that fails in each cycle was selected using two distinct policies: *i*) the node that fails is selected at random; *ii*) the node that fails is selected at random among the nodes that are interior in more trees. Nodes are not allowed 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%.

Figure 4.5 depicts the results for both scenarios. When nodes to fail are selected at random (Figure 4.5(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 4.5(b)),



(a) Random node failures.

(b) Targeted node failures.

Figure 4.5: Experimental results for sequential node failures.

and its reliability remains constant at 100%. This happens due to the following phenomena: because the forwarding load imposed to each Thicket node is limited, 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.

4.2.4.2 Catastrophic Scenario

Now results in face of a large number of simultaneous node failures (in particular 40%) are presented. 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, advantages from a reliability point of view are not expected 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 4.6 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

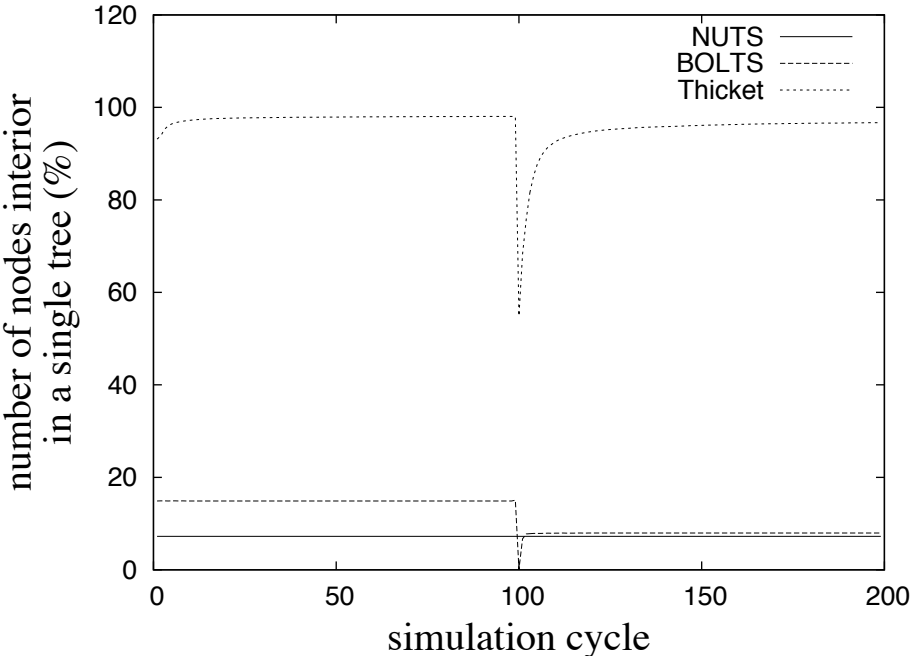


Figure 4.6: Percentage of Nodes Interior in a single tree for a catastrophic scenario.

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 4.7 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.

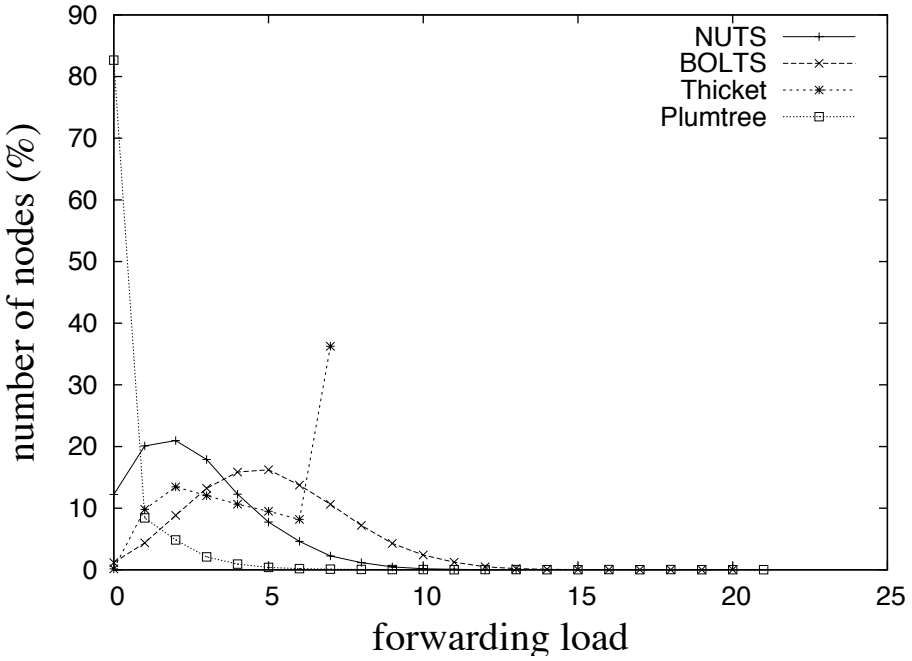


Figure 4.7: Forwarding load distribution for a catastrophic scenario.

4.3 Java Prototype

4.3.1 Overview

In order to evaluate the Java Prototype we have used the PlanetLab infrastructure. PlanetLab consists in a global research network that supports the development and evaluation of new network services. It can be used to deploy distributed applications in a large set of node sites in the Internet.

4.3.2 Experimental Settings

In this experiments the parameter values of T, f and *maxLoad* were the same as in the PeerSim simulations. The results were extracted using an overlay network composed of 400 nodes.

In all the experiments, a mp3 file of size $7311327bytes$ is broadcast at a bit rate of $256Kbit/s$. The size of the data chunks distributed in the network was defined to $10Kbytes$. The FEC Library was used to send data segments composed of 4 chunks in 5 encoded chunks. This prevents a single tree disconnection from affecting the reception of the original data segment at

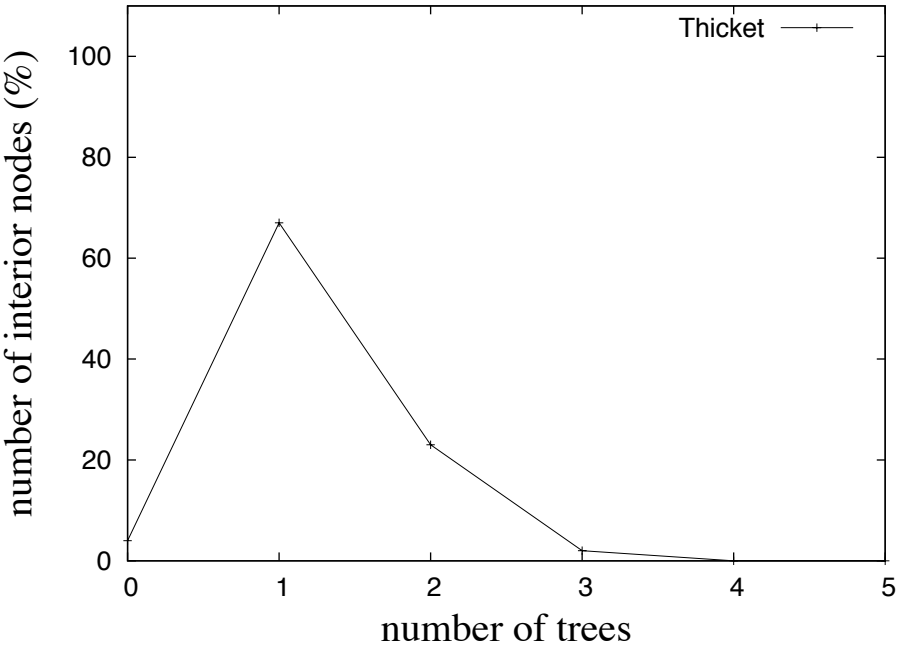


Figure 4.8: K-interior node distribution in a stable environment.

any node. This results in a redundancy factor of $5/4 = 1.25$.

The bitrate of the original data $256Kbit/s$ corresponds to $32Kbytes/s$. The redundancy factor introduce by FEC encoding increases the send rate $32000 * 1.25 = 40Kbytes/s$. This is equivalent to $40000/10000 = 4chunks/s$, which is within the rates suggested by Hedge, Mathieu and Perino (2010).

After being received, a message is buffered by Thicket at least during $30s$.

All presented results are an average of 5 independent executions of each experiment.

4.3.3 Stable Environment

First, the Java prototype was evaluated in a stable environment with no node failures. Figure 4.8 depicts the distribution of Thicket nodes according to the number of trees in which they are interior. As in the simulation experiments, it is expected that most nodes become interior in a single tree (limiting the forwarding load of all nodes promoting the usage of all the available resources).

It can be observed that the PlanetLab deployment achieves the target desirable properties in the tree construction process: most nodes (68 %) remain interior in a single tree while a

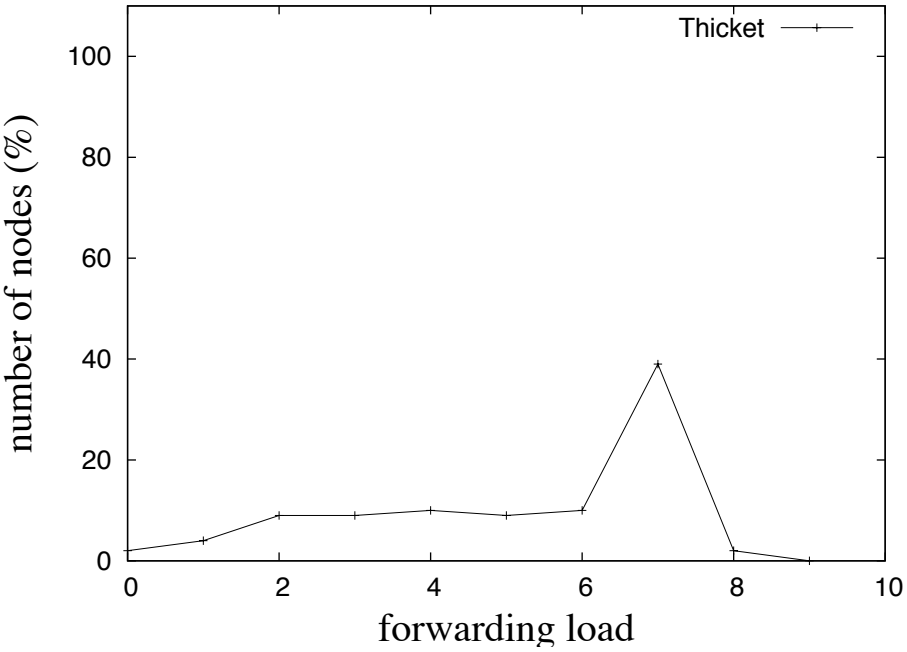


Figure 4.9: Forwarding load in a stable environment.

small fraction, about 24 %, are interior in two trees ensuring the coverage of the trees to all participants. Furthermore, only 4 % of nodes are leafs in all trees, meaning that most nodes contribute to the message dissemination process. Due to the constant changes in the network link throughput during the experiments, there is a small fraction (close to 2%) of the nodes that are interior in more than two trees. This occurs because the repairing mechanism is triggered more often than in the controlled environment of the Peersim simulations. Still, most nodes exhibit the desirable properties target by the protocol.

Next, it is measured the forwarding load distribution of all participants. It was aimed that the value this metric would never exceed *maxLoad*; additionally, the percentage of nodes with a forwarding load of 0 was expected to be low, i.e., that most participants could contribute to the message dissemination process. Figure 4.9 presents the results obtained in the experiments.

As expected, the results show that most participants (about 40%) exhibit a forwarding load equal to the *maxLoad* parameter and, consequently, most nodes contribute equally to the dissemination process.Furthermore, only a small fraction (about 2.5%) of all participants present a forwarding load of 0, not disseminating messages to any tree. There are, however, some nodes that exceed the forwarding load limit. This happens because nodes add tree branches during the repairing mechanism without checking the load limit (otherwise, one could compromise the

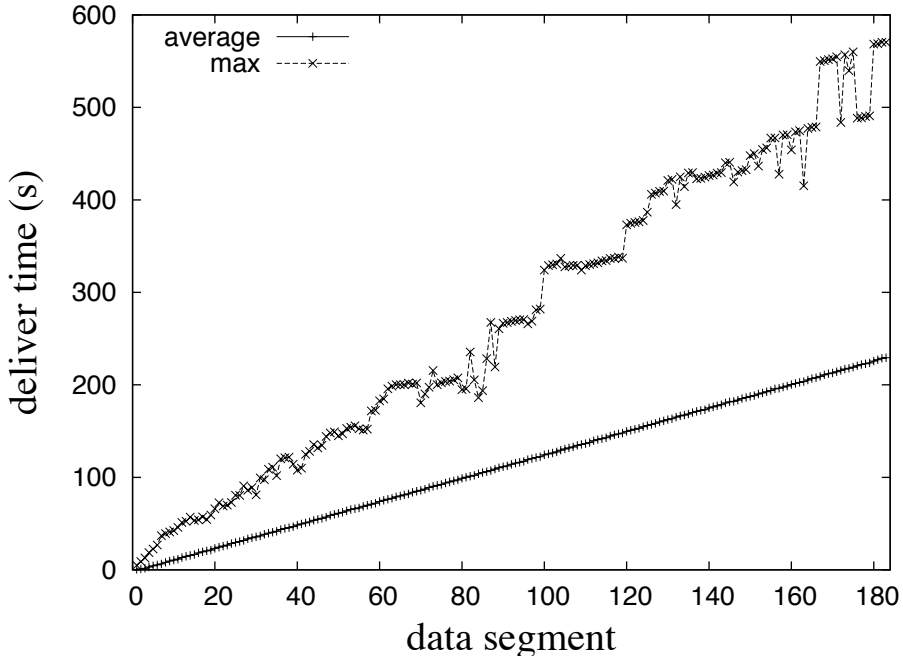


Figure 4.10: Delivery time of data segments in a stable environment.

reception of data segments).

A simple way to avoid a peer from violating the load limit constraint would be to allow to node to disconnect itself from a peer when overloaded. Unfortunately, allowing nodes do break the tree branches may also compromise the data reception at the disconnected peer.

Finally, Figure 4.10 presents the average, and maximum delays between the reception of each data segments and the first data segment received. This metric captures the ability of the protocol to deliver messages with a constant delay, ensuring that data segments are delivered in time to be consumed by the application.

The results show that average delivery time of data segments increases linearly with the data segment number (note that the inverse of the derivative of the line indicates the rate at which data segments are delivered). However, the maximum values observed can be as high as tree times the average. This means that the nodes that observe these delays may experience playback breaks. The most probable cause for this behavior is that some PlanetLab nodes may temporarily lack the resources required to receive the stream without delays.

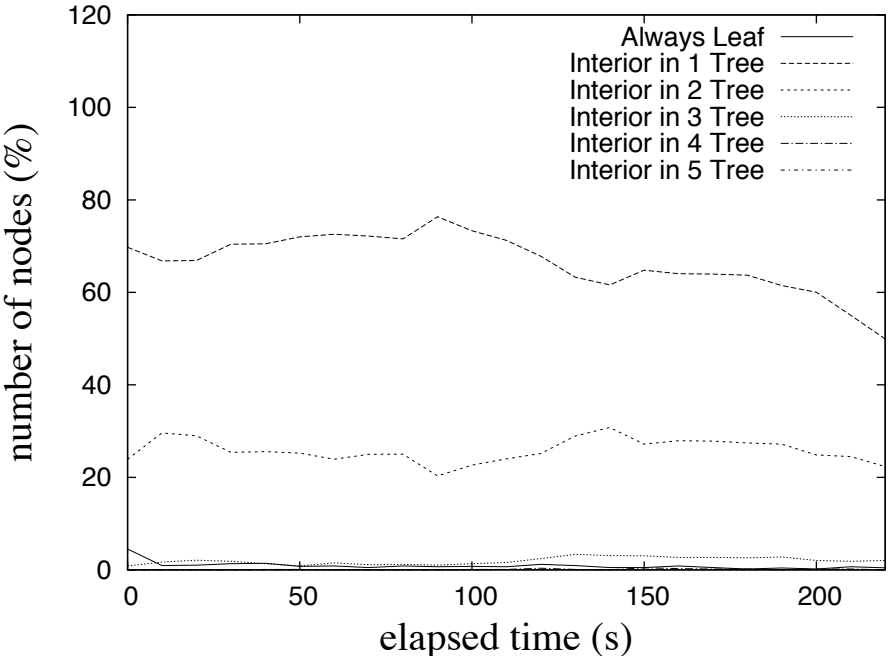


Figure 4.11: K-interior node distribution in a catastrophic scenario.

4.3.4 Fault-Tolerance

Now, we evaluate the performance of the Java prototype in a scenario where a large percentage of participants fail. During this tests, 40% of all nodes fail after receiving 100 data segments. This corresponds approximately to half the execution time. The presented results are for the 60% of the participants which remain correct after the failure event.

Figure 4.11 presents the percentage of nodes that are interior in 1 2, 3, 4 and all trees as time elapses.

When a large number of participants has failed, the percentage of nodes interior in a single tree decreases due to the operation of the repair mechanism but the protocol is able to maintain a configuration in which most nodes are interior in a single tree and minimizing the number of participants that are interior in more than two trees. The decrease is lower than during the simulations experiments due to the fact that, in this experiment, nodes do not fail exactly at the same time (the failure is triggered by the reception of the hundredth data segment), allowing Thicket to take repairing measures between participant failures.

The forwarding load distribution of all participants was also measured in a faulty scenario. Figure 4.12 presents the results. From the figure, is clear that even a large percentage of node

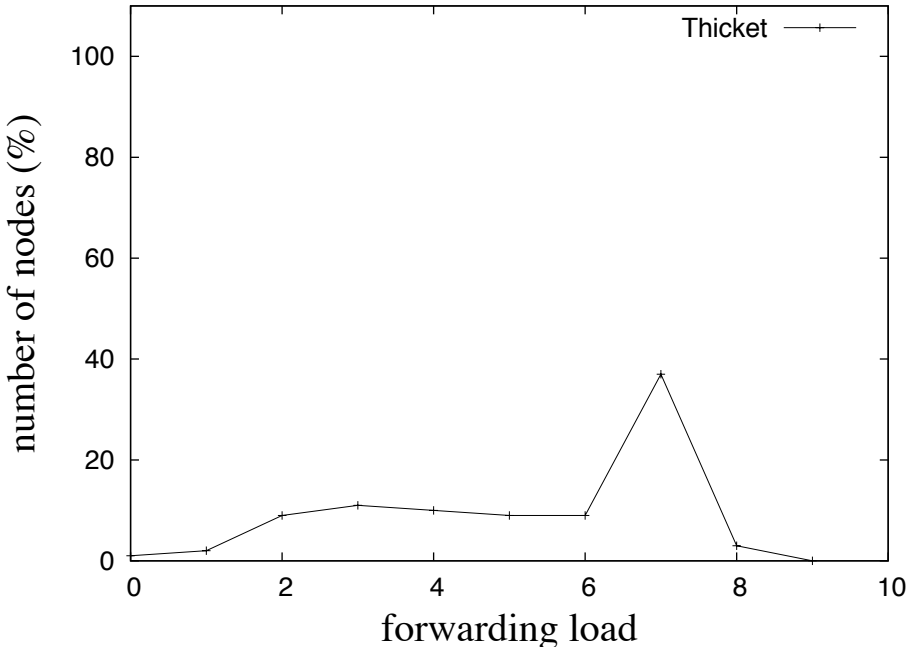


Figure 4.12: Forwarding load in a catastrophic scenario.

failures does not affect the forwarding load of the participants, which often exhibit values equal and below the *maxLoad*, as in the stable scenario.

Figure 4.13 presents the impact of the failures of a large percentage of nodes on the delivery time of data segments. The results show that the protocol is able maintain almost constant delays in the average of all nodes as in the stable scenario. The worst case analysis indicates that failures do not affect negatively the delays on the correct nodes since the delays obtained in this scenario are lower than during the experiments on the stable environment.

Finally, we have performed an evaluation on the reliability of the protocol under the catastrophic scenario. Figure 4.14 presents the impact of a large percentage of failures in the reception of data segments. As the results show, reliability decreases on the segments close to segment 100 (the time at which this segment is sent corresponds approximately to the time where the failure event occurs).

Summary

This chapter presented the results of the evaluation of the two Thicket implementations. The PeerSim simulations validated the employed algorithms, confirming their ability to build

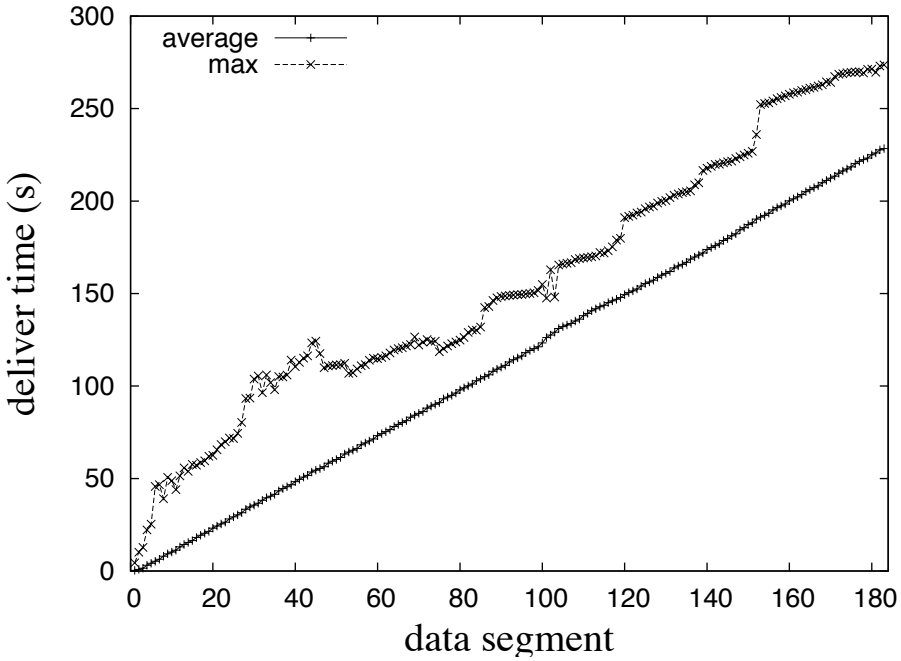


Figure 4.13: Delivery time of data segments in a catastrophic scenario.

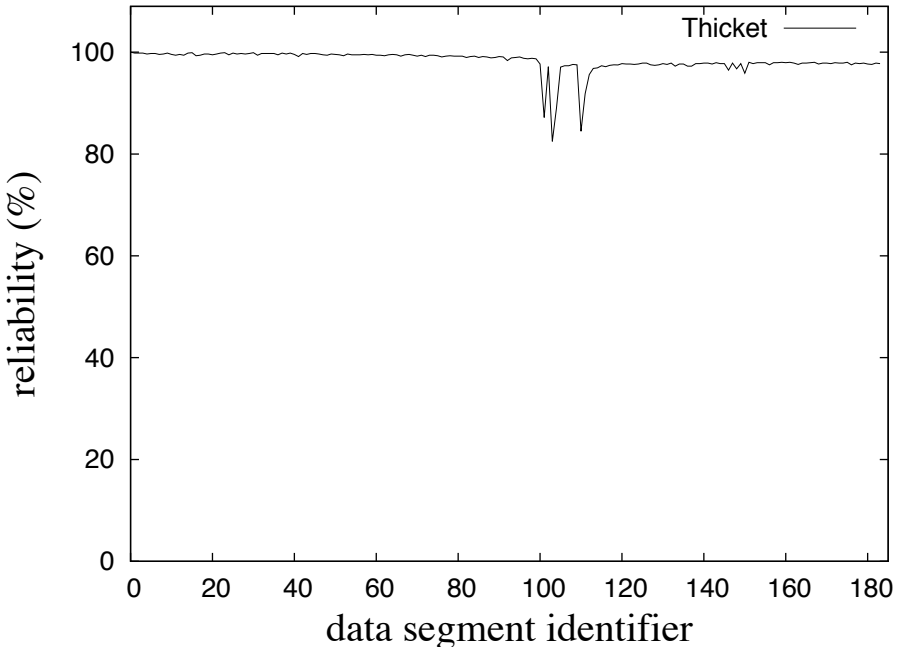


Figure 4.14: Reliability in a catastrophic scenario.

trees with very few interior nodes in common. Thicket is also able to limit the forwarding load to a defined threshold while maintaining the desirable tree properties. The reliability of the created trees is also evaluated in this chapter, showing that the spanning trees created by Thicket are more resilient than the "nuts and bolts" approaches. Finally, the results of the Java prototype deployed in the PlanetLab infrastructure provides evidence that the protocol is able to operate in practical real-world environments.

5 Conclusions and Future Work

P2P systems offer a scalable approach to develop efficient dissemination schemes. A challenge of such approach is to distribute load across all participants avoiding overloading situations and using efficiently all the resources of the system. However, this systems need to deal with node failures ensuring that all correct nodes continue to receive the disseminated data. This thesis addressed this problem by analyzing, implementing and evaluating techniques to perform live streaming to a large set of participants.

To this end, this thesis presents Thicket, a novel dissemination scheme which builds and maintains multiple spanning trees with very few interior nodes in common. The protocol also limits the forwarding load of all participants avoiding overloading situations. The proposed design also created trees more resilient to node failures since the failure of one node will often cause the rupture of only one tree, permitting the affected nodes to receive data from the remaining spanning trees.

Experimental results shows that Thicket creates trees with less interior nodes in common that some presented "nuts and bolts" approaches. Most nodes of the Thicket protocol are interior in only one of all trees. Opposed to the other approaches, Thicket also guarantees that all nodes have a forwarding load under a defined threshold. Results also show that the trees created by Thicket protocol are more resilient than the other approaches to node failures.

The deployment of the protocol over the PlanetLab infrastructure allowed to validate the proposed design under real-world scenarios. The results show that Thicket presents the desirable tree properties and distributes the forwarding load across all nodes.

As future work the protocol could be extended to account for the heterogeneity of the resources of each participant. To this end the load limit, defined by the parameter *maxLoad*, becomes different for each node and this information should be piggybacked in the exchanged messages.

Furthermore, a video decoder can be used at the Streaming Layer to allow the dissemination of video content using the Thicket protocol.

In conclusion, Thicket provides a dissemination scheme which distributes load across all participants and is resilient to node failures. This properties make the protocol suitable for live streaming applications.

References

Annapureddy, S., S. Guha, C. Gkantsidis, D. Gunawardena, & P. Rodriguez (2007, May). Exploring VoD in P2P Swarming Systems. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM '07)*, Anchorage, AK, USA, pp. 2571–2575. IEEE.

Benbadis, F., F. Mathieu, N. Hegde, & D. Perino (2008, September). Playing with the Bandwidth Conservation Law. In *Proceedings of 8th International Conference on Peer-to-Peer Computing (P2P '08)*, Aachen, Germany, pp. 140–149. IEEE.

Bittorrent. http://bittorrent.org.

Carvalho, N., J. Pereira, R. Oliveira, & L. Rodrigues (2007, June). Emergent Structure in Unstructured Epidemic Multicast. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, Washington, DC, USA, pp. 481–490. IEEE.

Castro, M., P. Druschel, A.-M. Kermarrec, A. Nandi, A. I. T. Rowstron, & A. Singh (2003). SplitStream: high-bandwidth multicast in cooperative environments. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, New York, NY, USA, pp. 298–313. ACM.

Chandra, T. D. & S. Toueg (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM) 43*(2), 225 – 267.

Chu, Y.-h., S. Rao, S. Seshan, & H. Zhang (2002, October). A case for end system multicast. *IEEE Journal on Selected Areas in Communications 20*(8), 1456–1471.

Coates, A., A. Hero III, & R. Nowak (2002, May). Internet tomography. *IEEE Signal Processing Magazine 19*(3), 47–65.

Comer, D. E. & J. C. Lin (1994). Probing TCP implementations. *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference 1*, 17 – 17.

da Silva, A. P. C., E. Leonardi, M. Mellia, & M. Meo (2008, September). A Bandwidth-Aware Scheduling Strategy for P2P-TV Systems. In *Proceedings of 8th International Conference on Peer-to-Peer Computing (P2P '08)*, Aachen, Germany, pp. 279 – 288. IEEE.

Deering, S. E. & D. R. Cheriton (1990). Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems (TOCS '90) 8*(2), 85 – 110.

Emule. http://www.emule.com.

Ferreira, M., J. Leitão, & L. Rodrigues (2010a). Thicket: A Protocol for Building and Maintaining Multiple Trees in a P2P Overlay. In *Proceedings of the 29th IEEE International Symposium on Reliable Distributed Systems*, New Delhi, India, pp. (to appear).

Ferreira, M., J. Leitão, & L. Rodrigues (2010b). Thicket: Construção e Manutenção de Múltiplas Árvores numa Rede entre Pares. In *Actas do segundo Simpósio de Informática*, Braga, Portugal, pp. 279 – 290.

Ganesh, A. J., A.-M. Kermarrec, & L. Massoulié (2003). Peer-to-Peer Membership Management for Gossip-Based Protocols. *IEEE Transactions on Computers 52*(2), 139 – 149.

Godfrey, P. B., S. Shenker, & I. Stoica (2006). Minimizing churn in distributed systems. *ACM Computer Communication Review (SIGCOMM '06) 36*(4), 147 – 158.

Haridasan, M. & R. van Renesse (2008). SecureStream: An intrusion-tolerant protocol for live-streaming dissemination. *Computer Communications 31*(3), 563–575.

Hefeeda, M., A. Habib, D. Xu, B. Bhargava, & B. Botev (2005, October). CollectCast: A peer-to-peer service for media streaming. *Multimedia Systems 11*(1), 68 – 81.

Hegde, N., F. Mathieu, & D. Perino (2010, May). On Optimizing for Epidemic Live Streaming. In *Proceedings of the IEEE International Conference on Communications (ICC '10)*, Cape Town, South Africa, pp. 1 – 5. IEEE.

Huang, F., B. Ravindran, & V. A. Kumar (2009, September). An approximation algorithm for minimum-delay peer-to-peer streaming. In *Proceedigns of the IEEE 9th International Conference on Peer-to-Peer Computing (P2P '09)*, pp. 71 – 80. IEEE.

Jelasity, M., A. Montresor, G. P. Jesi, & S. Voulgaris. The Peersim Simulator.

Kaldehofe, B. (2003). Buffer management in probabilistic peer-to-peer communication protocols. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS '03)*, Florence, Italy, pp. 76 – 85. SRDS.

Kazaa. http://www.kazaa.com.

Leitão, J., J. Pereira, & L. Rodrigues (2007a). Epidemic Broadcast Trees. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS '07)*, Beijing, China, pp. 301–310.

Leitão, J., J. Pereira, & L. Rodrigues (2007b). HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, Edinburgh, UK, pp. 419 – 429.

Li, H., A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, & M. Dahlin (2008). FlightPath: Obedience vs. Choice in Cooperative Services. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, USA, pp. 355 – 368.

Liang, C., Y. Guo, & Y. Liu (2008, June). Is Random Scheduling Sufficient in P2P Video Streaming? In *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS '08)*, Beijing, China, pp. 53 – 60. IEEE.

Liang, J., S. Y. Ko, I. Gupta, & K. Nahrstedt (2005). MON: on-demand overlays for distributed system management. In *Proceedings of the 2nd Workshop on Real, Large Distributed Systems (WORLDS '05)*, San Francisco, California, USA, pp. 13 – 18.

Oh, K.-J., M. Kim, J. S. Yoon, J. Kim, I. Park, S. Lee, C. Lee, J. Heo, S.-B. Lee, P.-K. Park, S.-T. Na, M.-H. Hyun, J. Kim, H. Byun, H. K. Kim, & Y.-S. Ho (2007, May). Multi-View Video and Multi-Channel Audio Broadcasting System. In *Proceedings of the 3DTV Conference*, Kos Island, Greece, pp. 1 – 4. IEEE.

Padmanabhan, V. N., H. J. Wang, P. A. Chou, & K. Sripanidkulchai (2002). Distributing streaming media content using cooperative networking. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '02)*, Miami, Florida, USA, pp. 177 – 186. ACM.

Papadimitriou, P., V. Tsaoussidis, & L. Mamatas (2008). A receiver-centric rate control scheme for layered video streams in the Internet. *Journal of Systems and Software 81*(12), 2396 – 2412.

Picconi, F. & L. Massoulié (2008, September). Is There a Future for Mesh-Based live Video

Streaming? In *Proceedings of 8th International Conference on Peer-to-Peer Computing (P2P '08)*, Aachen, Germany, pp. 289 – 298. IEEE.

Ratnasamy, S., A. Ermolinskiy, & S. Shenker (2006). Revisiting IP multicast. *Applications, Technologies, Architectures, and Protocols for Computer Communication 36*(4), 15 – 26.

Ratnasamy, S., P. Francis, M. Handley, R. Karp, & S. Shenker (2001). A scalable content-addressable network. *Applications, Technologies, Architectures, and Protocols for Computer Communication 31*(4), 161 – 172.

Renesse, R. V., Y. Minsky, & M. Hayden (1998). A Gossip-Style Failure Detection Service. Technical report, Dept. of Computer Science, Cornell University.

Ripeanu, M., I. Foster, A. Iamnitchi, & A. Rogers (2005). UMM: A dynamically adaptive, unstructured multicast overlay. In *Service Management and Self-Organization in IP-based Networks*.

Rowstron, A. I. T. & P. Druschel (2001). Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '01)*, Heidelberg, Germany, pp. 329–350. Springer-Verlag.

Rowstron, A. I. T., A.-M. Kermarrec, M. Castro, & P. Druschel (2001). SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. *Networked Group Communication 2233*, 30 – 43.

Stoica, I., R. Morris, D. R. Karger, M. F. Kaashoek, & H. Balakrishnan (2001). Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON '03) 11*(1), 17 – 32.

Venkataraman, V., K. Yoshida, & P. Francis (2006). Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast. In *Proceedings of the 14th IEEE International Conference on Network Protocols (ICNP '06)*, Washington, DC, USA, pp. 2 – 11. IEEE Computer Society.

Voulgaris, S., D. Gavidia, & M. van Steen (2005, June). CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management 13*(2), 197 – 217.

Wang, M., L. Xu, & B. Ramamurthy (2008, August). Channel-Aware Peer Selection in Multi-

View Peer-to-Peer Multimedia Streaming. In *Proceedings of 17th International Conference on Computer Communications and Networks (ICCCN '08)*, St. Thomas U.S. Virgin Islands, pp. 1 – 6. IEEE.

Wang, M., L. Xu, & B. Ramamurthy (2009, September). A flexible Divide-and-Conquer protocol for multi-view peer-to-peer live streaming. In *Proceedigns of the IEEE 9th International Conference on Peer-to-Peer Computing (P2P '09)*, Seattle, Washington, USA, pp. 291 – 300. IEEE.

Wu, C. & B. Li (2007, May). Strategies of Conflict in Coexisting Streaming Overlays. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM '07)*, Anchorage, AK, USA, pp. 481 – 489. IEEE.

Wu, C., B. Li, & Z. Li (2008, June). Dynamic Bandwidth Auctions in Multioverlay P2P Streaming with Network Coding. *IEEE Transactions on Parallel and Distributed Systems 19*(6), 806 – 820.

Zhao, B. Q., J. C. S. Lui, & D.-M. Chiu (2009, September). Exploring the optimal chunk selection policy for data-driven P2P streaming systems. In *Proceedigns of the IEEE 9th International Conference on Peer-to-Peer Computing (P2P '09)*, Seattle, Washington, USA, pp. 271 – 280. IEEE.

Zhao, B. Y., J. D. Kubiatowicz, & A. D. Joseph (2001). Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, UC Berkeley.

Zhuang, S. Q., B. Y. Zhao, A. D. Joseph, R. H. Katz, & J. D. Kubiatowicz (2001). Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '01)*, Port Jefferson, New York, United States, pp. 11 – 20.