UNIVERSITY CARLOS III OF MADRID

Department of Telematic Engineering

Master of Science Thesis

Nozzilla: A Novel Peer to Peer Architecture for Video Streaming

Author: **Alexandru Iosif Bikfalvi**
Dipl. Eng. in Telecommunications

Supervisor: **Jaime García-Reinoso, Ph.D.**

Leganés, July 2008

Abstract

Many peer-to-peer video streaming systems use application-level multicast where the peers are responsible of forwarding the video packets between them, usually following the path of one or more tree overlays. Since the performance of such systems depends on the uplink capacity available at each peer (measured as bandwidth, delay and jitter), previous work focuses on sharing the load responsibility between all peers by dividing the video traffic into several stripes and requiring a peer to be an interior node in one multicast tree. However, although these techniques consider the uplink capacity limit, they do not adapt easily for situations where the resources are either very abundant or very scarce.

This thesis proposes Nozzilla, a new approach based on structured peer-to-peer that dynamically takes into account the available capacity resources. Increased efficiency and reliability is achieved in several ways. First, the peers in the overlay are grouped according to their current resources, improving the search for a suitable parent inside the multicast tree. Second, the system uses path diversity by splitting the video traffic in several stripes and distributes each stripe along a different multicast tree. Third, the protocol can be used in a quality-of-service enabled network, which can assign different classes of service for each video stripe. This makes possible the usage of techniques such as multi-description coding and scalable video coding and is more robust to failures since the loss of a single stripe does not cause the loss of video reception. Finally, the algorithm improves the performance of the multicast tree by increasing the load distribution between peers and reducing the resource demand on the root.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet Protocol built-in multicast was designed for purposes of point-to-multipoint applications such as audio and video streaming. Its advantage is that it enables live content delivery from a single source to a group of destination users where the data replication is done entirely by the network. Consequently, the multicast traffic should traverse each network link at most once, reducing the load on the source and using the resources of the network in the most efficient way. However, the reluctance of many providers to support IP multicast in their routers despite the growth in available bandwidth and progress in audio-video compression, hampered the deployment of multicast-based streaming solutions.

As a result, peer-to-peer (P2P) solutions have emerged as a promising alternative, where the task of replicating and forwarding the video traffic is transferred to the end-users interconnected by a P2P overlay network. Because the peers communicate via unicast IP connections, the IP multicast is not used. The major obstacle to the design and deployment of P2P video systems is the lack of any guarantees regarding the performance and the video quality experience by the users. Most peers rely on the service provided to them by other peers, where they can join and leave at any time and where the characteristics of the connections between them are constrained at modest values. Nevertheless, previous research has shown that in the conditions of the current Internet, P2P video streaming is possible on a large scale, even though in some cases the uplink capacity (from the user to the network) is a limiting factor [9].

Most P2P-based video streaming proposals work by using the P2P protocol to create a multicast tree between the peers rooted at the source of the video traffic. The tree is similar to the IP multicast tree, with the difference that the tree nodes are the peers rather than the routers of the network. The major challenge consists in creating a suitable multicast tree in the most efficient way, which from the peer perspective is reduced to locating a suitable parent peer. For this reason, the tree-based approaches consider structured P2P protocols (using distributed hash tables - DHTs), since they guarantee locating any target peer in a limited number of steps, always a function of the total number of peers in the network.

Because the construction of a multicast tree is conditioned by the uplink resources each peer has available, proposals like SplitStream [1] and P2PCast [6] rely on several multicast trees. In this way, peers that are parents (also called *interior nodes*) in one tree can be leaf nodes in the other trees. However, these approaches incur a lower efficiency when they have to redistribute the extra resources a peer might have. In addition, the geometry of the multicast tree is determined by the resources of the peers and the search for parent strategy that can lead to an over-solicitation of the root. Other proposals like Zigzag [10] guarantee a better geometry and usage of resources but require a larger amount of overhead in terms of maintenance bandwidth and computing power.

To address some of the shortcoming of the existing architectures, this thesis describes Nozzilla, a structured P2P-based system for video streaming aimed at improving the usage of available resources in such a distributed environment. The proposed solution targets P2P networks where either the available uplink resources can be determined (by means of measuring the throughput, delay, jitter to other peers) or they are guaranteed by a quality of service mechanism.

Nozzilla features the following characteristics:

- Divides the video traffic into several flows called *stripes*, which are then forwarded using different multicast paths. This method can be used with multi-description coding algorithms making a single stripe enough to watch the video and with an increasing quality as more stripes are received ([1], [6]).
- Uses Scribe [2] and Pastry [7] as foundation for the P2P multicast infrastructure.
- Improves the search for a suitable parent in any multicast tree, by grouping the peers according to their current resources. Nozzilla allows each stripe to have particular QoS requirements, and peers can compute their local resources for each individual tree.
- Shapes the geometry of the multicast tree, by reducing the probability of choosing the root as a parent. Although a regular DHT lookup might select the root of the multicast tree as the first candidate parent (especially when the number of peers is low), our approach allows us to select an alternative peer as parent whenever such alternative exists and without increasing the number of search steps. This method leads to a significant load reduction on the root, which is particularly useful in hybrid implementations where the root peer is the media streaming server. Consequently, the resulting end-to-end delay is higher as the depth of the multicast tree increases, but this is not identified as a major issue for applications like IPTV for which Nozzilla is targeted.

The rest of the work is organized as follows: Chapter 2 surveys some of the existing P2P-based video streaming approaches, in order to identify the general design requirements, obstacles and directions for improvement. Chapter 3 overviews Pastry and Scribe, the two P2P protocols on which Nozzilla is based. Chapter 4 gathers the main contribution of this thesis: the P2P architecture, the protocol specification and multicast operations. Finally, Chapter 5 evaluates experimentally some performance aspects of the protocol and Chapter 6 concludes identifying the topics covered and future work.

Chapter 2

State of the Art

The P2P alternative is an elegant and cheap solution in achieving multicast communication for video streaming applications in situations where the IP multicast is not an option and other solutions (such as MBONE [3] and content distribution networks) may be too difficult or costly to implement. P2P multicast is an application-level multicast because it work as an application layer protocol using IP unicast connections. However, creating an efficient P2P video streaming protocol can be difficult for the following reasons [10]:

- The behavior of the peers is unpredictable. They can join and leave at any time, consequently making the overlay a very unstable network fabric.
- The users may experience a high end-to-end delay, since the video traffic is routed through several intermediate peers. Considering a multicast overlay tree, the largest delay in the network (for the most unfavorable situated peer) will depend on the depth of the tree as the sum of the delay on each individual branch.
- A P2P overlay requires complex protocol exchanges between peers, especially when the overlay changes as peers join and leave. For the P2P multicast, this requires extra processing at the user side and incurs additional overhead for the peer's link.

2.1 Multicast Trees

One of the simple methods of constructing a P2P multicast overlay is to create an overlay tree with the root at the source of the multicast traffic and spanning all users that join the multicast group (i.e. the set of all users that join the service). Compared to the IP multicast, in P2P the routing role is shifted to the edges of the network. Thus, the traffic replication is performed by the peers, in this way bypassing the traditional limitation of an IP multicast network.

From the service quality perspective, the overlay multicast can work efficiently if the tree topology matches the network resources available to the peers. In this sense, two important topology parameters are the *depth* (or *height*) and the *spread* of the tree. If the tree is balanced (all interior nodes have the same number of children and all leaf nodes are at the same level), the spread is proportional to the node *fan-out*: the maximum number of children accepted by an interior node. If we assume the most simple overlay model where the connections between peers are identical, a greater depth determines a greater latency and jitter for the video traffic that reaches the peers located on lower tree levels. The maximum latency is obtained for the maximum depth, when the tree is a chain (the fan-out of each peer - including the root - is one).

The spread influences the number of tree branches that emerge from any given node. If the spread increases, so does the probability of having a throughput bottleneck as the necessary uplink bandwidth is proportional to the number of children. In particular, the maximum spread is obtained when all peers are connected to the root of the tree, yielding a minimum delay but maximum throughput requirements. In general, for a balanced multicast tree with a depth d and where each node has a fan-out f, the number of interior nodes is given by Eq. 2.1 and the number of leaf nodes is given by Eq. 2.2 [6]:

$$N_I = \sum_{i=0}^{d-1} f^i = \frac{f^d - 1}{f - 1} \tag{2.1}$$

$$N_L = f^d \tag{2.2}$$

The interior nodes require an available uplink bandwidth of at least the product between their fan-out and the throughput needed by the video stream or stripe. Hence, the selection of the fan-out (which is conditioned by existing network resources) will in turn determine the depth of the multicast tree. For a low fan-out balanced tree, such as the case of a binary tree, the interior nodes require an uplink available bandwidth of only the double of the video bit rate. Thus, many peers may meet the bandwidth requirements of being an interior node. On the other hand, for such a tree, almost 50% of the peers need to be interior nodes. Such deep trees are not practical because they will increase the transmission latency and are more prone to service interruptions. Given the large percentage of interior nodes, the rate at which interior nodes are leaving the network is very high.

For a tree with a fan-out of 100, a little over 1% of the peers are interior nodes. It is feasible in such cases to promote as interior nodes, the most stable peers. A study of the stability of P2P systems proved that the probability for a peer to remain online increases with the uptime (the most stable peers are the oldest peers from the overlay) [8]. However, the large fan-out requires those peers to have an available bandwidth of a hundred times the video streaming bandwidth. Since in both of the two extreme cases presented before, the video quality experienced by the average user may be unsatisfactory, most architectures attempt to construct an overlay tree having a balanced depth and fan-out (although the fan-out is limited by the available capacity).

2.2 Peer-to-Peer Video Streaming Systems

Most of the P2P video streaming issues presented so far were of general nature, dealing only with the most common aspects. This section studies two P2P video streaming proposals: SplitStream[1] and P2PCast[6], which served as a starting model for Nozzilla. We shall focus on some of the notions presented so far, such as multi-path delivery, multicast trees construction details, routing aspects, optimizations specific to each proposal and particular unique ideas.

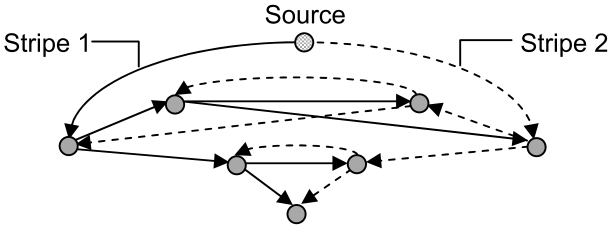

Figure 2.1: The basic idea of SplitStream

2.2.1 SplitStream

SplitStream is a balanced approach that works by dividing the video stream into k stripes. In this way, the forwarding load is distributed across all peers. For example, in Fig. 2.1 the video stream is divided into two stripes and an independent multicast tree is created for each stripe, such that a peer is an interior node in one tree and a leaf node in the other.

For a given bandwidth requirement B of the video stream and depending on the number of stripes, each peer needs a downlink available bandwidth of B. However, since only a fraction of the peers are interior node in any given tree, the required uplink is only $\frac{B \cdot k}{f}$, where k is number of stripes and f is the node fan-out. Thus, each node can precisely control the needed uplink bandwidth by limiting the number of child peer it accepts.

Even though the authors of SplitStream leave open to the implementers the choice of a P2P protocol, their proposal is based on Pastry [7] and Scribe [2]. Like any DHT-based P2P protocol, each Pastry node receive a peer identifier (or peer ID), selected from a numeric space, when they initially join the overlay. The peer's routing table is used to locate the peers closest to a given key, from the same numeric space as the identifiers, and it is an essence a mapping between some node ID and its contact network address and transport port. Because it is unfeasible to have a routing table that stores the contacts of all peers in an overlay within a huge hash space, Pastry works by storing more node contacts for peers that have a node ID with more digits in common, and less contacts for peers that have a node ID with less digits in common. We can say that peers have a good knowledge of their neighbors and a lesser knowledge of the distant peers. The Pastry protocol is further detailed in Chap. 3.

On top of Pastry, SplitStream uses Scribe that assigns to each multicast group a unique hash key. The root of the multicast tree is the overlay peer closest to the key. Group management is handled in a decentralized way. An existing peer from the overlay can join the group by simply routing toward the group's key. The routing process finishes when a node already in the group has been found; in the worst case, the routing will finish when the root of the tree - the source of the multicast traffic - is reached. SplitStream uses a separate Scribe tree for each of the k stripes. Using the Pastry routing, SplitStream tries to create multicast trees that do not share any interior nodes with each other (all interior nodes from a tree are leaf nodes in all others). Since the basics of the routing is that at each hop the next hop is closer to the key - meaning it has a node ID that shares more of its significant digits to the key - trees with disjoint interior nodes can be built by simply selecting group keys different in their most significant digits.

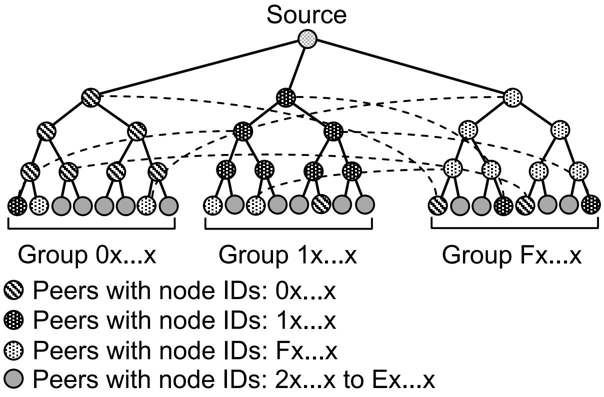
Source
Group 0x...x
Group 1x...x
Group Fx...x
Peers with node IDs: 0x...x
Peers with node IDs: 1x...x
Peers with node IDs: Fx...x
Peers with node IDs: 2x...x to Ex...x

Figure 2.2: Creating the interior nodes-disjoint multicast trees in SplitStream

Fig. 2.2 illustrates how interior nodes-disjoint multicast trees are created in SplitStream. The nodes are assigned as interior nodes for a given multicast tree based on their identifiers. In this example, hash values (group keys and node identifiers) are represented in a hexadecimal format. The number of stripes (k) is 16 with group keys ranging from $0x...x$ to F$x...x$. Points connected through dashed lines represent a single peer that is simultaneously an interior node in one group (tree) and a leaf node in other groups (trees). Since the SplitStream multicast trees are disjoint, the downlink for any given node is given by the number of stripes that peer receives. At most, any peer can be an interior node in one group and a leaf node in all other groups it belong. In addition, the uplink bandwidth is precisely controlled at an interior node level by the number of children the node chooses to accept. Thus, the fan-out of each node is decided by the node itself based on its known uplink capacity.

In other proposals, if the fan-out threshold is reached when a new peer wants to join the group at that particular node, it will be forwarded to the child that has the lowest delay. This method, however, does not work well in SplitStream, since the child node that supposed to accept the new peer might have already reached its fan-out threshold in another tree. Therefore, in SplitStream a node of a tree always accepts the new peer as child, regardless of the its fan-out and then tries to find one of its children that can be rejected. The rejection process takes into account the children that match these criteria in the following order:

- One of the children that belong to another group (this is possible through the spare capacity feature of SplitStream, described below);

- One of the children that belong to the group and have the shortest prefix match between the node ID and the group key.

If each criteria returns more than one eligible node that include the new child, than the new child is always selected for rejection.

Fig. 2.3 outlines the join and rejection mechanism when a new peer wants to join the tree at a given node, and that node reached its fan-out limit. In both cases A and B, the new node wants to join the group 0800 to which the candidate parent belongs. In case A, the candidate parent finds a child that joined for a different group and rejects this child after accepting the

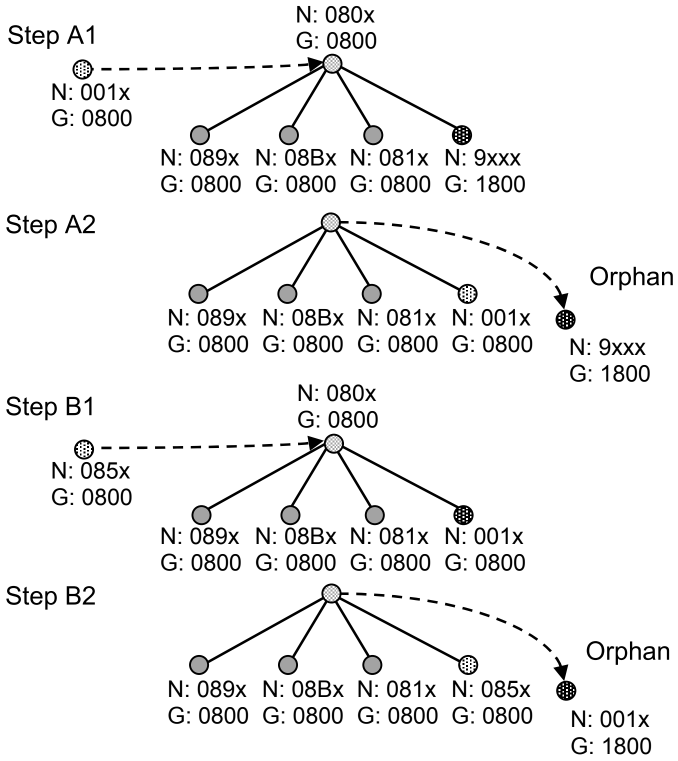

Figure 2.3: Accepting a new child in SplitStream when reaching the fan-out limit

new one. In case B, the candidate parent selects the node with the most distant identifier from the group. For each node, N represents the node ID and G represents the group (or stripe) key.

After a node is rejected, it is informed that has been orphaned for that specific group and it begins searching for a new parent starting with its former siblings that have IDs that match the prefix of the desired group. If no former siblings can accept it, then it tries with their children until either a peer in the desired group accepts it or no parent has been found. Finally, in the second case it tries to attach to any tree using the spare capacity group feature of SplitStream. This special group contains all peers with fewer children than their fan out limit and that can server children for other stripes.

The procedure to find a candidate parent in the spare capacity tree begins by sending a anycast message to the spare capacity tree, message that is delivered to the node closer to the orphan peer (node 1 in Fig. 2.4). From there, a search starts inside the spare capacity tree to locate the peer with available resources for the target group. The search always starts down the tree, until the bottom of the tree is reached. Only if a suitable parent is still not found, the search will continue one level up from the closest node and from there again down the tree. For example, in Fig. 2.4 an orphan node joins the spare capacity tree: the orphan node has ID 0 and wants to join the group 6. The anycast message is routed to the closest node, in this case with ID 1. This starts forwarding the message to its children. First,

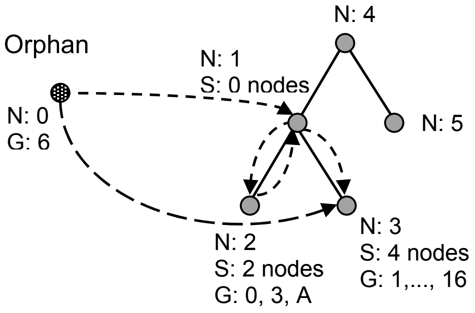

Figure 2.4: The spare capacity tree in SplitStream

node 2 receives the message; however, node 2 is not part of group 6. Since it has no other children, it returns the message to its parent. Next, the message is forwarded to node 3. Since node 3 is part of group 6, has available capacity and node 0 is not an ancestor (this can be verified because each peer keeps a list of upstream nodes up to the root), it accepts the orphan node as child. If none of the children of node 1 could have accepted the orphan, and since node 1 has no available capacity (even if it were a member of group 6), the message would have been forwarded to its parent, node 4. The process would continue either until a suitable parent is located or until all nodes in the spare tree have been checked. Whenever the fan-out threshold of a node from the spare capacity tree is reached, that node removes itself from the spare capacity group.

In the worst-case scenario, anycast to the spare capacity group may also fail. Either because the group is empty or because there is no peer in the spare group that can deliver the requested streaming stripe. In such situations, the orphan node is announced of the failure because all available capacity has been exhausted.

2.2.2 P2PCast

P2PCast uses the same principles as SplitStream. In essence, the multicast video data is split in a number of stripes. The reason behind this division is similar: not to overwhelm the peers uplink available bandwidth, when their fan-out is greater than one. Each node has a *proper tree*: the tree for which that specific node is an interior node. The fan-out limit of interior nodes is considered equal to the number of stripes and it is denoted by f. According to their fan-out at any given time, nodes are classified in:

- Incomplete nodes: with less than f children in their proper tree;
- Only-child nodes: leaf nodes in the tree (and whose parents are incomplete node);
- Complete nodes: have f children in their proper tree (stripe);
- Special node: it is a single peer in the overlay, which is a leaf in all trees and it has the bootstrapping role when the overlay is first created.

The P2PCast does not suggest any particular P2P protocol. Instead, the multicast tree operations are given in a general manner with respect to the tree structure. The explanation

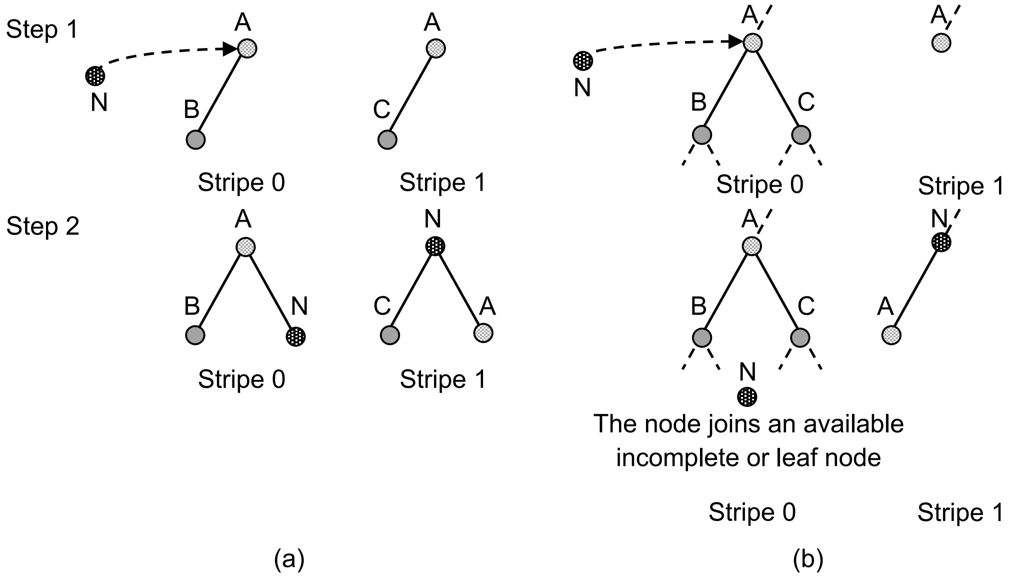

Figure 2.5: Joining P2PCast to an incomplete node (a) and complete node (b)

of the protocol presented here will consider a fan-out limit of two, in which case the multicast trees are binary trees. New users arriving to the overlay are added to the bottom of the multicast tree. The reason for this is that new peers have a greater probability of remaining online short periods, and the damage to the overlay if these peers would leave or fail relatively quickly after joining is therefore mitigated. Always, the new peer will be a child-only or incomplete node (since it does not have any children of its own).

The joining process starts when the node contacts a random bootstrap node from the overlay. If the contacted node is an incomplete node, the incomplete node will select for which stripe it wants to be an internal node and will adopt the new node in that tree. If the incomplete node previously had another child (yet not in its proper tree), the new peer must adopt that child. In addition, the incomplete node is also adopted by the new node in the other tree, for which the new peer becomes an interior node.

When the bootstrap node is an only-child node (a node with an incomplete node as parent), the new peer is redirected to the incomplete node and the adoption is handled like in the previous case. If the only-child node available is the special node, than the new peer joins by replacing the special node in the tree and adopting it as its child.

For bootstrap nodes that are complete nodes, they have all *f* children in their proper stripe, which means that they must be leaf nodes in the other tree (otherwise, the number of children could exceed their fan-out limit). Therefore, the new peer becomes an interior node for that tree adopting the bootstrap node as its child. For the proper stripe of the bootstrap node, the new peer starts parsing down the tree in order to find an incomplete node or a leaf.

Fig. 2.5 illustrates the joining process to an incomplete bootstrap node (a) and a complete bootstrap node (b). In the incomplete case, the new peer, *N*, want to join *A* to stripe 0. *A* has two children, *B* and *C*, in each stripe and its proper tree can be any of them. The node

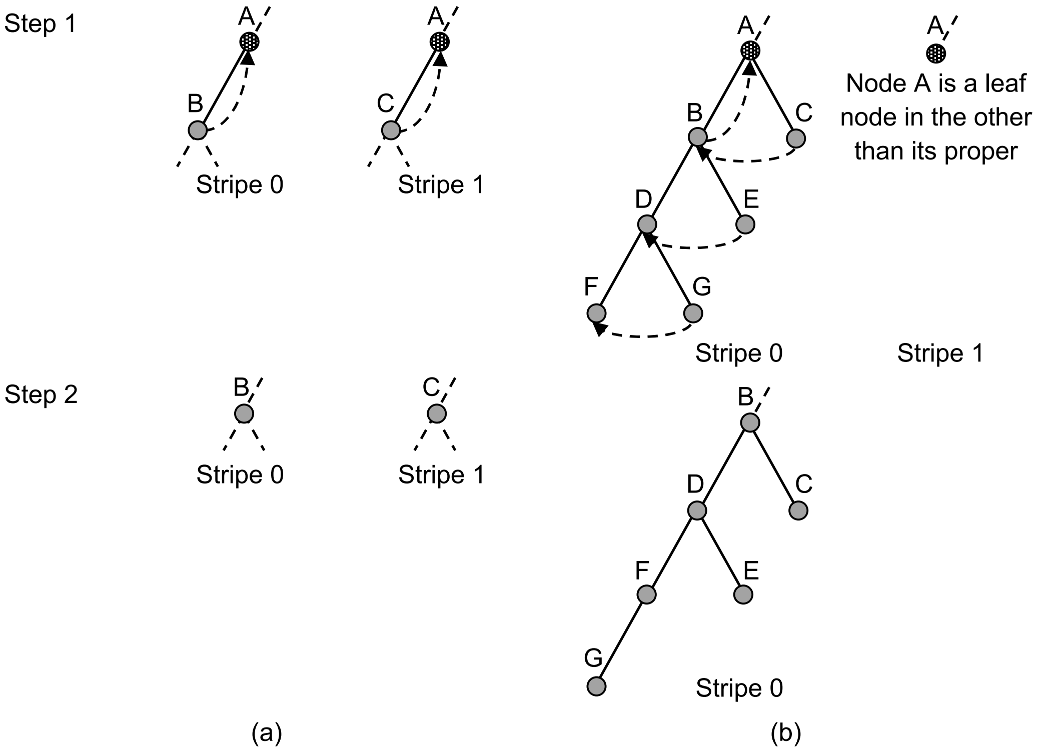

Figure 2.6: Removing from P2PCast an incomplete node (a) and a complete node (b)

N chooses stripe 0 as its proper stripe and adopts *N* as its child. Because its fan-out limit is exceeded (it now has three children: *B*, *C* and *N*), the node *C* from not its proper stripe is dropped. For this reason *N* takes the place of *A* in that tree and adopts both *A* and *C* as its children.

In the joining to a complete node, the new peer joins *A*, which has f children in stripe 0 and is a leaf node in stripe 1 (hence a complete node). Since *A* cannot handle any other children in stripe 1, the new node *N* replaces *A* in this tree and adopts *A* as its child. In the proper tree of *A* (stripe 0), *N* searches down the tree for an incomplete or leaf node and joins in the manner presented before.

The selection of the bootstrapping node by the new joining peer should be done randomly such that the multicast trees are relatively balanced. This is an important aspect because the removal of nodes from the multicast tree depends on the height of the failed node inside the tree. The P2PCast studies only the case when a complete or incomplete tree node fails (the failure of the source and special node means the failure of the entire network, while the failure of leaf nodes has no impact). In both these cases, an algorithm is used to remove the node from the overlay that minimizes any disturbance effects. In the first case, if the peer to be deleted from the tree is an incomplete node, then it will be replaced by its corresponding child from each stripe (see Fig. 2.6, a). However, the second case of complete nodes is more difficult since in its proper tree an empty space - a bubble - is formed, which cannot be easily replaced by either of its children. The authors of P2PCast argue that replacing the deleted node with a leaf node has two disadvantages: first, it is vulnerability, because a former leaf node would be quickly promoted too high in the tree hierarchy and such its failure will affect a large number of nodes. Second, the process of searching for a suitable leaf node might take too long. Instead, they propose that the bubble is filled by a random child. If this child is also a complete node, it needs to discard one of its children in order to adopt the other child of the deleted node. Thus, the discarded node becomes a child of its sibling. If the sibling is also a complete node, one of its children is discarded, which then tries to be adopted by its sibling and so on. The process continues down the tree until all nodes have been readopted. Fig. 2.6 (b) illustrated the removal of a complete node *A* from the multicast tree: in its proper stripe, one of its children (in this case, *B*) replaces it. Its other child, *C*, is adopted by *B*. Because *B* is also a complete node, it rejects one of its children in order to be able to adopt *C*, in this case *E*. The orphan node *E* is adopted by its sibling *D* and since D is a complete node, it must reject one of its children, *G*. *G* can be adopted by its sibling, *F*, without requiring other nodes to be orphaned. Usually the down tree process stops at an incomplete or leaf node.

The distribution of the video stream in P2PCast is made the same as in SplitStream. A node must join a number of stripes in order to decode the video content. In essence, $f - 1$ stripes are used to deliver the media content while one stripe is used for error-correction control data.

The next chapter is a thorough overview of Pastry and Scribe, which, as in the case of SplitStream, are used as foundation in Nozzilla. However, as we shall see, Nozzilla not only builds on the functionality provided by Pastry and Scribe, but it also extends it.

Chapter 3

Pastry/Scribe Overview

The Nozzilla P2P protocol is built using modified versions of Pastry and Scribe: *mPastry* and *mScribe*, although other DHT-based P2P protocol could have been used as the base system. This chapter provides an overview of both protocols.

3.1 Pastry

Pastry is a self-organizing P2P protocol that uses distributed hash tables. Before joining, each peer is assigned a unique number called *node identifier* from a circular numerical space of 128 bits. The objects that will be stored by the peers are also assigned unique values called *keys*, within the same numerical space. The keys are in general calculated using a cryptographic hash function over the stored resources in order to ensure their binding and uniqueness. For this reason, the numerical space from which both the node IDs and the keys are selected is called a *hash space*. The node IDs and the keys are represented using digits in base 2^b, where b is usually 3 or 4.

Regardless of whether object information is stored or retrieved, the goal of Pastry is to locate the peers in the overlay, which are closest to a given key. Each peer maintains three routing data structures: a routing table (*R*), a neighborhood set (*M*) and a leaf set (*L*). The architecture of Pastry guarantees that the closest node to a given key is found in less than $log_{2^b}N$ hops, where N is the number of peers in the overlay (and supposing that the numbers of peers that can fail simultaneously is bounded).

The routing table has $log_{2^b}N$ rows and 2^b columns and maps node identifiers with the associated contact information, such as IP addresses and transport ports. At each row r and column c it can store one contact record for a peer identifier having the first r digits equal to the digits of the local node identifier and the $r + 1$ digit equal to c. If more peers match the same table entry, only the closest one in terms of distance metric is kept.

The neighbor set helps identifying the peers that are closer in the physical network, making peers located at lower distances more desirable. Closeness can be evaluated in various ways related to performance, such as the number of IP hops or geographic distance; in this way peers located at lower distances are more desirable.

Finally, the leaf set is a small list of the closest peers with the numerically smallest (the first half of the set) and largest identifiers (the second half of the set). While routing, the leaf set is always used first to locate another peer closest to the key and only if the key value falls

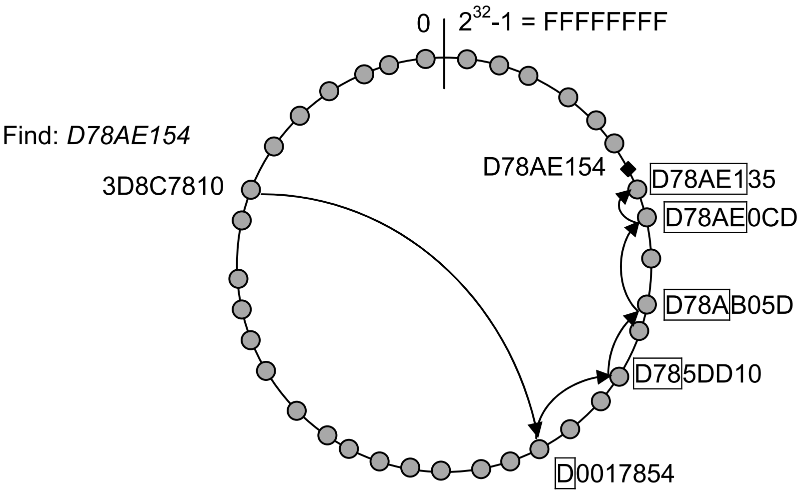

Figure 3.1: The lookup process in Pastry

outside the range covered by the leaf set, the lookup is performed using the routing table. The forwarding and the lookup stops when no other peer closer to the key is identified (see Figure 3.1).

3.2 Scribe

Scribe is a scalable application-level multicast infrastructure built on top of Pastry. Scribe works by allowing each peer to create a *multicast group* to which other peers can join. For each group the protocol builds a fully decentralized multicast tree, meaning that each peer can act as multicast source, root of the tree or being just a group member. Scribe assigns a unique identifier from the hash space to each group and the peer in the group closest to the group ID acts as the root of the tree.

Peers wanting to join a Scribe multicast group use Pastry to perform a hob-by-hop search for the group's ID. In the search, each hop peer will join the multicast tree and will accept the previous hop as child (it supposes that each node has infinite resources). The search stops when either a hop is already a group member or when it reaches the root of the tree. When an existing peer wants to leave a multicast group, it must check whether it has no other children and if true, it can leave after informing its parent.

Chapter 4

Nozzilla

4.1 Overview

The objective of our proposal is to achieve a multi-path delivery mechanism in which the video traffic is divided into several stripes and where each peer can use only its available resources to serve one or more children for each stripe. These available resources can be evaluated on a per stripe basis, making possible of establishing different classes of service, in cases where some stripes carry more sensitive video information than others. For the purposes of better understanding, we shall assume that Nozzilla uses three stripes, a high priority one (HP), a medium priority one (MP) and a low priority one (LP). However, as a general consideration the priority of the stripes, if any, is a secondary aspect, since it only affects how the available resources are computed.

In our case, once the uplink available resources are determined, each peer can decide to become an interior node in any of the high-priority, medium-priority or low-priority multicast trees. If no resources are available, the node will be a leaf in any multicast tree it wants to join. In order to do this, the Pastry hash space is divided into four regions, one for each priority, and one for leaves.

For each video stripe, a separate multicast tree is created. The root of the tree is the video server, which is also the first peer in the overlay. However, unlike SplitStream, the peers are not uniformly distributed as interior nodes to these multicast trees. Instead, each peer decides, based on its available uplink resources, whether it can support any children for a certain stripe and, if so, it will join the network as a candidate interior node for the multicast tree corresponding to that stripe. Using this P2P mechanism, a given node can easily discover potential parents for each multicast tree (stripe). In addition, in Pastry/Scribe the ID of the root is used as the multicast group key during the joining process, leading to many peers joining to the root, simply because they find it first. To reduce this possible overload and because the interior nodes of any multicast tree are identified beforehand, in Nozzilla a peer will select during the join process a random candidate parent.

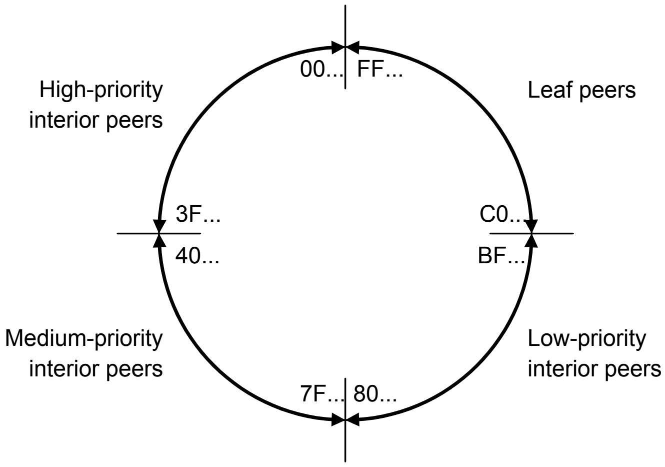

Figure 4.1: The division of the hash space between the peer groups.

Peer Group	Prefix Bits	Node identifier range
High Priority	00	0000...00 - 3FFF...FF
Medium Priority	01	4000...00 - 7FFF...FF
Low Priority	10	8000...00 - BFFF...FF
Leaf	11	C000...00 - FFFF...FF

Table 4.1: The node identifier range for each peer group in Nozzilla

4.2 Peer-to-peer Architecture

Unlike other video streaming protocols using multi path delivery (e.g. SplitStream), in Nozzilla each peer having available resources can become an interior node for more than one multicast tree. To make this possible, it will use a separate node identifier to advertise its presence in the P2P overlay for each multicast group. Eventually, when a peer has no available uplink resources (or the initial resources have been exhausted) the peer should advertise itself using only a corresponding leaf identifier, indicating that, although it is part of the network, it cannot accept any more children. This new concept is called *peer multiple identification* and the decision on which identifiers are advertised is taken entirely by the peer, based on its wish to become an interior node for a certain multicast tree. The collection of all the peers that advertise themselves as interior nodes for a multicast tree is called a *peer group*. There are four peer groups: one for each multicast tree and one for leaf peers. Figure 4.1 illustrates the four regions of the hash space and their mapping to the different types of peers, while Table 4.1 contains the node identifier ranges for each peer group.

By default all peers are members of the leaves peer group, since it is assumed that any peer is able to receive the service. Being a member peer of the leaves group does not necessarily imply to be a leaf of any of the multicast trees, but instead it provides the means for any peer to join a tree and receive the service. If a peer decides to join any other peer group it is mandatory to also join its corresponding multicast tree. Whenever a new child is accepted or an existing child leaves, the peer reevaluates its available resources and the decision of being a member of each of the peer groups. Consequently, the peers can dynamically join and leave the peer groups, while in the same time not modifying their tree membership.

The rationale behind the multi-identification in the same P2P overlay is that the peers may have reserved resources to contribute as interior nodes to several multicast trees. Hence, a peer no longer has a single multicast tree where it should always be an interior node, but rather the membership decision is dynamically taken. The peer checks whenever the resources change and it decides if it should advertise itself as a candidate interior node; thus belonging to one or more peer groups.

Each peer group will receive an equal portion of the hash space, with the first two bits of the hash identifier being reserved to identify the group. The number of identifiers each peer has is equal to the number of the possible peer groups. These identifiers are unique in the overlay and are fixed during the lifetime of the peer. Each of them is composed of a two bit prefix part that is unique for each peer group, and a common 126-bit suffix part that is randomly selected when the peer is created. If a peer decides to become a member to a certain peer group (i.e. a possible interior node for the associated multicast tree), it should advertise (*positive advertisement*) to its neighbors the identifier for that particular group. Similarly, when a peer decides not to be a member of some group (regardless of whether it is already a member of the associated multicast tree or not), it will advertise the removal of the identifier (*negative advertisement*).

This behavior resembles to having four independent overlays, where peers join and leave independently from each other. However, the advantage of using a single overlay divided in four groups is that peers in one group can help to locate peers in other groups. As long as their identifiers are advertised, routing information is exchanged between them and it is possible for a peer that joined a certain group to locate nodes in any other group, thus enabling a fast joining to any multicast tree.

With the exception of the leaf peers, when a peer becomes a member of a peer group it must also join the corresponding multicast tree to be able to fulfill its role as an interior node. If the peer cannot join the corresponding multicast tree in a finite amount of time, it is required to leave the peer group and negatively advertise its identifier for that group to its neighbors. On the other hand, peers from any peer group can join any multicast tree. The only condition is that membership to at least one peer group is required because the peer needs to have at least one identity in the overlay.

The P2P protocol requires several changes to Pastry in order to support the multiple node identifiers and negative identifier advertisements proposals, and in Scribe to support tree joining only with interior nodes, unsuccessful query returns and random parent selection. The modified versions of the existing protocols are called *mPastry* and *mScribe*.

4.3 Modified Pastry

The mPastry protocol supports multiple identifiers for each peer, re-advertisements of these identifiers to the neighboring peers and random parent selection. Thus, it must also feature a slightly changed API provided to the upper layer.

column	0	...	j	...	15
row 0	$0x...x$	...	$jx...x$	...	F$x...x$
row 1	$M_{1,1}0x...x$	...	$M_{1,1}jx...x$	...	$M_{1,1}$F$x...x$
row 2	$M_{2,1}0x...x$	...	$M_{2,1}jx...x$	...	$M_{2,1}$F$x...x$
row 3	$M_{3,1}0x...x$	...	$M_{3,1}jx...x$	...	$M_{3,1}$F$x...x$
row 4	$M_{4,1}0x...x$	...	$M_{4,1}jx...x$	...	$M_{4,1}$F$x...x$
...	...	...	...	...	...
row $4i-3$	$M_{1,1}...M_{1,i}0x...x$	...	$M_{1,1}...M_{1,i}jx...x$	...	$M_{1,1}...M_{1,i}$F$x...x$
row $4i-2$	$M_{2,1}...M_{2,i}0x...x$	...	$M_{2,1}...M_{2,i}jx...x$	...	$M_{2,1}...M_{2,i}$F$x...x$
row $4i-1$	$M_{3,1}...M_{3,i}0x...x$	...	$M_{3,1}...M_{3,i}jx...x$	...	$M_{3,1}...M_{3,i}$F$x...x$
row $4i$	$M_{4,1}...M_{4,i}0x...x$	...	$M_{4,1}...M_{4,i}jx...x$	...	$M_{4,1}...M_{4,i}$F$x...x$
...	...	...	...	...	...

Table 4.2: The routing table of mPastry

4.3.1 Data Structures

Due to the multiple identification, the changed data structures are the routing table and the leaf set. Each peer maintains four leaf sets, one for each of its identifiers. This is because the content of the leaf sets is highly dependent on the identifier considered local for each set. The routing table is also extended in order to include routing information for each identifier. The extended routing table contains a common first line with peer contacts that do not share any common prefix digit in any of the identifiers. The rest of the lines contain four entries in each column corresponding to digits that match in every identifier. Table 4.2 depicts the new organization of the routing table for a hash space with hexadecimal identifiers. The symbol $M_{N,i}$ means a match with the digit i of N^{th} identifier, while x represents any digit.

The use of separate entries for each identifier is needed because the assigned identifiers have a different first digit. However, the routing table is used in a unitary manner meaning that when a target key is received, the mPastry determines the next node based on the local identifier that shares a prefix with the key having the maximum length. For example, a peer that is a member of two peer-groups uses two identifiers in the overlay with the four digit prefixes 1A31 and 5A31 (note that only the two bit prefix of the identifiers differs, the rest of the bits being the same). When a target identifier of 1953 is received, the peer will use the entries in the routing table corresponding to the first identifier. Obviously, when there is no prefix match with any identifiers, the first line in the routing table is used, which is common for all of them. Irrespective of the identifier being used, the rules of accessing and filling either the routing table or the leaf sets remain the same as in Pastry.

4.3.2 Routing

mPastry uses two routing algorithms. The first (Algorithm 1) is used by each peer to determine the next hop where to forward a message targeting a certain key, *K* and in the framework of Nozzilla it is used by mPastry operations and by upper layer applications such as mScribe when forwarding messages. This is a slightly modified version of the original

Pastry algorithm, since it considers the fact that each peer has now four leaf sets and a changed routing table. The following notations were used:

- ID_i - the local node identifier of index i, where $0 \leq i < 4$.
- $R_{i,j}$ - the entry at row i and column j in the peer's routing table, where $0 \leq i < 4 \times \frac{128}{b} - 3$ and $0 \leq j < 2^b$
- L_j^i - the entry on position j in the leaf set i, where $\alpha_i \leq j \leq \beta_i$ and $0 \leq i < 4$ with α_i and β_i corresponding to the indexes of the lowest and highest ID entry in that set.
- K_i - the value of digit i of the key K, where $0 \leq i < \frac{128}{b}$ with i being zero for the most significant digit.
- shl(A, B) - returns the length of the prefix shared in base b digits by two hash identifiers, A and B.
- shb(A, B) - returns the length of the prefix shared in bits by two hash identifiers, A and B.

Algorithm 1 The default routing algorithm in mPastry

```
$id \leftarrow i$ such that shl$(ID_i, K)$ is maximal and $|ID_i - K|$ is minimal with $0 \leq i < 4$
if $L_{\alpha_{id}}^{id} \leq K \leq L_{\beta_{id}}^{id}$ then
    return $L_i^{id}$ such that $\left|K - L_i^{id}\right|$ is minimal
else
    $l \leftarrow$ shl$(K, ID_{id})$
    if $l = 0$ then
        $r \leftarrow 0$
    else
        $r \leftarrow l \times 4 + id - 3$
    end if
    if $R_{r,K_l} \neq empty$ then
        return $R_{r,K_l}$
    else
        return $T \in L \cup R \cup M$ such that shl$(T, K) \geq l$ and $|T - K| < |ID_{id} - K|$
    end if
end if
```

The routing process starts by selecting the leaf set and the rows of the routing table that will be used. Because it is not possible to have two local identifiers with the same two most significant digits, the algorithm selects the local identifier that shares the longest prefix with the key. In the situation when no identifiers share a common prefix, it selects the identifier closest to the key, since in this particular circumstance, the selection of a local identifier will only determine the selection of the leaf set (the routing table has a single row for these keys). Then, the algorithm checks whether the target key falls in the range of the selected leaf set. If true, the next hop is returned as the peer with the smallest distance to the key.

Otherwise, the algorithm uses the routing table to determine the next peer. At this point, the only difference from the original Pastry is the selection of the row in the routing table (the selection of the column is made in the same way). The row index depends on both the length of the shared prefix and the index of the local identifier used by the expression $l \times 4 + id - 3$, with the exception that if the shared prefix length is zero, the first row is used regardless of the identifier index.

The second routing algorithm (Algorithm 2) is a variation of the first one and is intended to provide support for random parent selection at the first hop. This is used by mScribe when searching for a parent in the multicast tree and improves the distribution of parent nodes.

Algorithm 2 The random parent selection routing algorithm in mPastry

```
$P$ such that $P \subset L \cup R \cup M$ and $\forall T \in P$ we have shb$(K, T) \geq 2$
**if** $P = \emptyset$ **then**
    **return** $T$ selected with the default routing algorithm
**else**
    **return** $T \in P$ where $T$ is randomly selected
**end if**
```

The random parent selection routing algorithm creates a subset of the combined routing table, leaf set and neighbor set, with all the nodes that belong to the same peer group as the key (the shared prefix length between any node ID in this subset and K is longer than 1 bit). If this subset is an empty set (i.e. the combined routing table, leaf set and neighbor set do not store any contact in the target peer group), the routing algorithm returns the same contact as the default routing algorithm. Otherwise, a random contact is selected and returned from this subset. Because this contact is in the same peer group as the key, it is a candidate parent in the tree joining process.

4.3.3 API Functions

The mPastry API contains the following functions:

MPASTRYINIT($application$) receives as an argument a handler to the upper layer application using mPastry (in the case of Nozzilla is mScribe) and returns the set of four identifiers assigned to the peer. When each peer is created, it is assigned four identifiers, one for each peer group and calculated using the method described in the previous section. By default, the identifiers are not advertised to the network, meaning that when the peer is initialized, no message exchanges are made with neighboring peers. A local resource management function handles the membership of the peer to each peer group and, respectively, which node identifiers are advertised and which are not. This is done by evaluating the uplink resources, and deciding based on the requirements of the video traffic, if the peer affords to accept new children in each stripe.

MPASTRYJOIN(ID) causes the peer to join the peer-to-peer overlay with the specified identifier, effectively publishing the identifier in the P2P network. In Nozzilla, the join procedure is called by the resource management function of the peer, which decides based on the status of the available resources the membership to each peer group.

MPASTRYLEAVE(ID) informs mPastry that it should leave the peer group associated with ID. When a leave is requested the peer will advertise to all its neighbors the change in the published identifier.

MPASTRYROUTE(msg, K) is a request to mPastry to route the given message (msg) to the neighbor having the identifier closest to the provided key, K. The default routing algorithm is used to determine that neighbor.

MPASTRYPARENTROUTE(msg, K) is similar in functionality to MPASTRYROUTE but it uses the random parent selection routing algorithm to determine the next hop.

MPASTRYSEND(msg, $peer$) sends the given message, msg, to the specified $peer$.

MPASTRYREMOVE(ID, $peer$) is a request to Pastry to send a negative advertisement for the identifier ID to the $peer$.

MPASTRYREJECT(msg, $peer$) is a request to Pastry to send a rejection message to $peer$ informing it that the message msg could not be routed.

The following callback procedures are implemented by the application (mScribe) and are used by mPastry to send notifications of occurring events.

MPASTRYDELIVER(msg, K) is called by Pastry when a message, msg, is received and either the local node is the one closest to the key K (usually during routing) or it was the intended recipient of the message (usually during a direct send).

MPASTRYFORWARD(msg, K, ID_{next}) is called by Pastry before a message, msg, is forwarded to a node closer to the key K during the routing process. ID_{next} indicates the identifier of the next peer where the message will be forwarded. The application can cancel the forward by setting the value of ID_{next} to NULL.

MPASTRYNOTIFYID(IDs_{active}, $IDs_{inactive}$) is called by mPastry when the advertisement of any of the local identifiers change. IDs_{active} indicate the identifiers that are currently advertised while $IDs_{inactive}$ indicate the identifiers that are not advertised.

MPASTRYNOTIFYREJECT(msg) is called by mPastry when a routing rejection is received. msg is the message that was rejected.

4.4 Modified Scribe

mScribe is intended to work with the mPastry and to manage the multicast trees. The multicast tree will still provide best effort delivery and the failure of interior nodes will be managed by the orphan peers by attempting to rejoin the multicast group. mScribe uses the following message types:

- JOIN - to join the multicast tree;
- CREATE - to create a new multicast tree (this message is router to the peer closest to the group ID, which becomes the root of the tree);
- LEAVE - used by parents to inform their children they are about to leave;
- MULTICAST - used to mark the mScribe messages that contain multicast information.

4.4.1 Data Structures

A mScribe peer maintains six data sets:

- A list of the multicast group IDs to which the peer is currently a member.

- A list of its $children$ in the multicast tree, that is initially empty.
- A list of its $parents$ for each multicast tree, that is initially empty.
- Two lists of the advertised (IDs_{active}) and non-advertised identifiers ($IDs_{inactive}$). This information is provided by mPastry through *mPastryNotifyId* callback function.
- In addition, whenever there is a change in the advertised identifiers, mScribe computes a list of the peer group $prefixes$. In the case of four peer groups, the prefixes are the two most significant bits of the advertised identifiers and are calculated using a *gpref* function.

4.4.2 API Functions

In the mScribe API, the *mPastryForward* (Algorithm 3) functions modifies the routing behavior of mPastry when mScribe sends a JOIN message. If left undisturbed, mPastry would route the message to the root (or node closest to the root) of the multicast tree and deliver the message there. However, if the prefixes list of an intermediate peer contains the prefix of the multicast group, it means that the intermediate node is a candidate parent. If it is also a member of the group and if resources are available, the initiator (not the sender) of the message is added to the $children$ list and the mPastry routing is stopped by setting ID_{next} to NULL. If the peer is not an interior node and the destination ID of the message was one of its non-advertised identifiers (meaning that the sender was not aware of an identification advertisement change), it will not prevent forward routing of the message but it will send a remove notification to the sender, informing it that the identifier used is no longer advertised. In the Algorithm 3 we assume that the received message, msg, contains the following fields: $type$, the type of the mScribe message; $group$, the target group ID; ID_{dest}, the peer identifier that the sender used to address the current recipient; $init$, the contact information of the peer that initiated the joining request; $source$, the contact information of the peer that sent the message.

Algorithm 3 mPastryForward procedure

```
procedure MPASTRYFORWARD($msg$,$K$,$ID_{next}$)
    if $msg.type = JOIN$ then
        if gpref($msg.group \in prefixes$) then
            if ($msg.group \in groups$) and ($resources[msg.group] > 0$) then
                $children_{msg.group} \leftarrow children_{msg.group} \cup msg.init$
                $ID_{next} \leftarrow NULL$
            end if
        else if ($msg.ID_{dest} \in IDs_{inactive}$) then
            mPastryRemove($msg.ID_{dest}, msg.source$)
        end if
    end if
end procedure
```

The *mPastryDeliver* (Algorithm 4) function processes the delivered mScribe messages. The JOIN messages are handled in a way similar to the mPastry forwarding function, with the exception that if the current node is advertised as an interior node for the multicast tree but did not join yet or resource reservation for the new child has failed, a rejection message is sent back to the last sender. When a CREATE message is delivered, if the peer did not join the multicast tree yet and if it can be an interior node, it will join the tree and will be able to accept additional children. LEAVE messages are always directed to a particular node and will inform that node that a child has gracefully left the multicast tree. If the number of children reaches zero, and local membership for that multicast tree is no longer needed, the peer can then leave the multicast tree by sending a LEAVE message to its parent. Finally, the MULTICAST message that is usually routed to the root of a multicast tree will be forwarded to all the children in the multicast group.

Algorithm 4 mPastryDeliver procedure

```
procedure MPASTRYDELIVER($msg$,$K$)
    if $msg.type = CREATE$ then
        if gpref($msg.group$) $\in prefixes$ and ($msg.group \notin groups$) then
            $groups = groups \cup msg.group$
        end if
    else if $msg.type = JOIN$ then
        if gpref($msg.group$) $\in prefixes$ then
            if ($msg.group \in groups$) and ($resources[msg.group] > 0$) then
                $children_{msg.group} \leftarrow children_{msg.group} \cup msg.init$
            else
                mPastryReject($msg, msg.source$)
            end if
        else
            mPastryRemove($msg.ID_{dest}, msg.source$)
            mPastryReject($msg, msg.source$)
        end if
    else if $msg.type = MULTICAST$ then
        for all $peer \in group$ do
            mPastrySend($msg, peer$)
        end for
        if current node $\in msg.group$ then
            process($msg$)
        end if
    else if $msg.type = LEAVE$ then
        $children_{msg.group} \leftarrow children_{msg.group} - msg.source$
    end if
end procedure
```

4.5 Multicast Tree Operations

The multicast video infrastructure is created by the video streaming servers of the provider. They use the *mScribeCreate* function (which in turn calls *mPastryInit*) to ini-

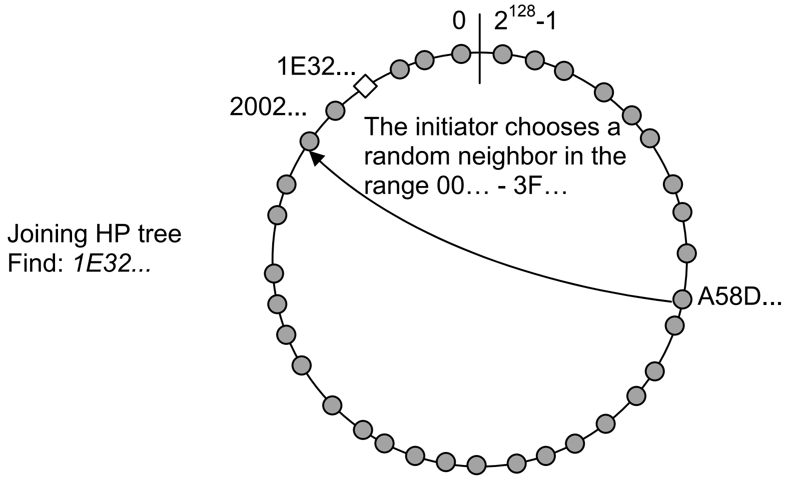
0
2^128-1
1E32...
2002...
The initiator chooses a random neighbor in the range 00… - 3F…
Joining HP tree
Find: 1E32...
A58D...

Figure 4.2: Joining a multicast tree when nodes in the target peer group are known

tialize and set themselves as the root of each multicast tree. The service information (group IDs, server addresses) is published by the provider in an off-band channel. These are usually serviced by one or more bootstrap servers, but their architecture is outside the scope of this work.

After initialization, the peers will connect to the service using *mScribeJoin* resulting in an mPastry search in the P2P network for an interior node in the desired tree. The search stops at the first node that already joined the same multicast tree and it can accept the new peer as a child. Since by requirement these nodes are also members of the peer group associated to that multicast tree, the search is very fast because it only requires locating any peer in that particular peer group. Thus, the traditional Pastry search is reduced to the match of a two bit prefix in the identifier. Because peers advertising their identifiers in that range are candidate interior nodes for the multicast tree, they should in general accept the new peer as child.

A new peer joins a multicast tree by performing a mPastry search in the P2P network for the desired multicast group ID. The search stops at the first node that already joined the same multicast tree and it can accept the new peer as a child. These nodes are also members of the peer group associated to that multicast tree. For example, joining the high-priority multicast tree implies searching for the HP group ID (which is in the range 00... to 3F...), with the search finishing when a peer with an identifier in the same range is located. Eventually, it is possible that the request is sent to destination peers that are not suitable parents for the target multicast tree. This happens when such a destination is the only peer known by the sender that is closest to the target group ID, or when the destination has recently left the peer group but the negative advertisement information did not reach the sender. Under these circumstances, the destination will simply forward the query. Finally, when no path to a peer group member is found the message is returned to the last sender or to the initiator that may start looking for an alternate path.

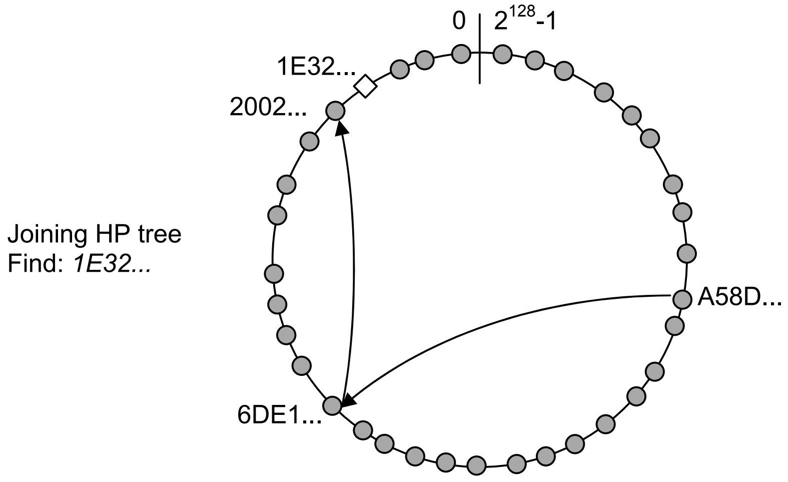

Figure 4.3: Joining a multicast tree through a passive intermediate peer.

Figure 4.2 illustrates one join process scenario: a new peer, member of the LP peer group (and an interior node for the LP multicast tree) with node ID prefix A58D wants to join as leaf to the HP multicast tree. The root of the HP tree has the identifier prefix 1E32. To join, the new peer calls the *mScribeJoin* function which in turn will prepare a JOIN message. When selecting the first hop in join search, mScribe always uses the *random parent selection* algorithm. Thus, the initiator will choose a random neighbor with the first digit of its identifier in the range 0 - 3. This approach reduces the probability of selecting the root as parent especially when the number of peers is low and the root is known by many of them.

In the example from Fig. 4.3 we assume that the same initiator does have any neighbor in the HP peer group. In this situation, it will select any peer that is closest to the group ID. Since this peer is not a member of the HP peer group (its first digit is 6 outside of the range 0 - 3), it will forward the message to its neighbor closest to the group ID. In the example, the second peer is a member of the HP peer group and will accept the initiator as child. In this case, the intermediate peer is a passive node that only forwards the request. After the request message is forwarded, the lookup will stop at the second hop, because this peer is already a member of the HP group which means it is also a member of the HP tree and an interior node for that tree.

In Nozzilla, peers communicate to each other to inform themselves about exceptional situations. For example, when a peer receives a message targeting one of its identifiers that is no longer advertised it will use *mPastryRemove* to inform the sender about the change, which in turn will update its routing table. To prevent a dead end when a request cannot be forwarded, a peer will call *mPastryReject* to send the message back to the sender, which is forced to look for an alternate path.

The last scenario is when a former interior node of a tree receives a join request. This happens when a peer leaves a peer group but still there are peers unaware of the change. A peer leaves a peer group either when it decides to leave a multicast tree altogether or when it

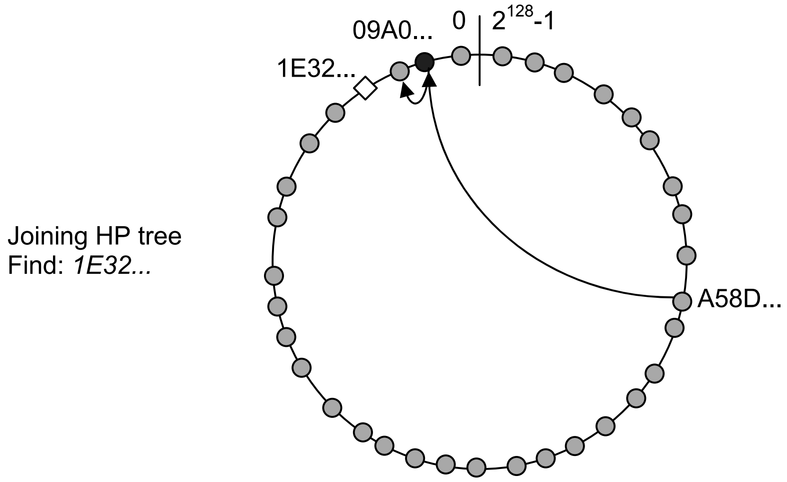

Figure 4.4: Joining through a former interior node that finds next hop

no longer has enough reserved resources to accept new child nodes. In the latter case, even if the peer will continue to be a parent for children that joined previously, it should no longer advertise itself as a candidate interior node.

If such a peer receives a join request, there are two possible courses of action. It may act as a passive intermediate node, forwarding the request to the next peer it knows closest to the target group ID. Or, it can return the request to the peer that sent the query in the first place. In both situations, it informs the peer sending the query that it left that peer group (sends a negative advertisement with *mPastryRemove* for its identifier in that peer group).

Figures 4.4 and 4.5 present a scenario where the initiator node (prefix A58D) wants to join the HP tree, and in the tree join message is forwarded to the peer with prefix 09A0. The peer 09A0 is still online, but it previously left the HP peer group and sent a negative advertisement for its ID to its neighbors. Since the initiator did not learn about the negative advertisement, when the request is received the peer 09A0 behaves as a passive intermediate node and forwards the query to the next peer in knows closest to the tree group ID. In the same time, it will inform the initiator, that it left the HP group, such that the initiator can update its routing table.

This behavior has the advantage of speed. Even though the intermediate peer is no longer a member of the HP group it only has to forward the query to the next peer using one of its other valid identifiers. A potential danger with this approach is the possibility that the intermediate peer does not find another peer in the group closer to the root (either because there is not one, or because the peers it knows also left the group). Therefore, if the intermediate node (prefix 09A0) cannot locate a next hop inside the group will use the second approach (Fig. 4.5, step 2), returning the query with the negative advertisement of its ID to the previous hop (node with prefix A58D in the example at step 3). The previous hop will search for a different path toward the target peer group (step 4).

Leaving a multicast tree follows the same principle as in Scribe. If a peer performs a graceful leaving it will execute the *mScribeLeave* informing its parent. In turn, its parent

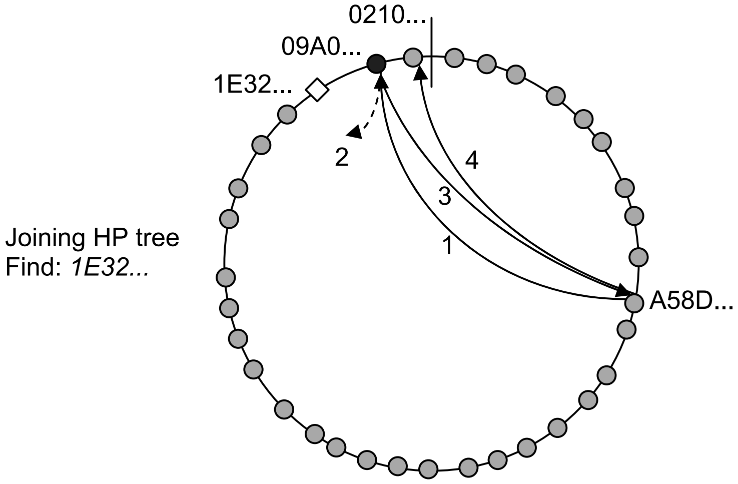

Figure 4.5: Joining through a former interior node that can not find a next hop

will update its children list, and in the case the children list is empty and it no longer needs to be a member of that tree, it will perform a graceful leave as well. When peers are leaving unexpectedly, their parent and children are notified by the lack of heartbeat messages. In such as situation the parents release all resources that were previously committed; clear the children list and update their peer group membership if necessary. Orphan child peers will attempt to rejoin the multicast tree and if the attempt is unsuccessful in a limited duration, they will inform their children of the failure by stop sending them heartbeat messages. This approach is intended to mitigate the service failure for a large portion of the tree, when because an intermediate peer cannot rejoin all its descendants experience a service interruption.

Chapter 5

Performance Evaluation

In this section, we present some experimental results with the goal to assess the performance and the feasibility of the multicast protocol. The results were obtained with a prototype implementation of Nozzilla. The protocol was implemented in Java and the PeerfactSim [5] simulation engine was used to emulate the network. The goal was only to examine the feasibility of the peer-to-peer protocol (i.e. whether peers can join in reasonable amount of time, the root of the multicast tree has an acceptable fan-out, the depth of the tree is small enough, etc), without going into the evaluation of the video path.

The initial evaluation focused on the feasibility of constructing the priority-based multicast tree. Because it is possible for a new peer joining the service to be rejected because no path to the target peer group is found, we considered two recovery mechanisms: first, the rejected messages returned to the sender can be forwarded again up to five times in the attempt to find a good path (under the assumption that the next hop is the second best one). If after the retry attempts limit no path is found and the message still gets rejected, the message is returned to the initiator that will also retry up to five times to restart the forwarding operation. Preliminary experimental data showed that taking a proactive approach (rather than waiting for the operation to time out), improves significantly the performance and results in a join success ratio above 99.9 percent.

The evaluation scenarios considered the practical usage aspects of Nozzilla and the limitations of the simulator. Thus, we focused on a small-to-medium overlay network having from 100 to 10000peers joining with a negative binomial distribution at an average rate of one peer every 2 seconds and with several initial resources configurations. The mPastry layer used $b = 4$ and $|L| = 16$, and we disabled the Pastry neighborhood set. We divided the peer in eight classes: first class has equal res resources for all priorities, second only for HP and MP, third for HP and LP and so on, with the eighth class having zero resources. Because each peer was uniformly selected from these classes, the average resources a peer has for a particular priority is $res/2$. All simulations were run in four different scenarios, corresponding to four different values for the number of resources: $res \in \{1, 3, 5, 7\}$. The implementation does not include a maintenance operation; therefore the routing state converges solely from negative advertisements.

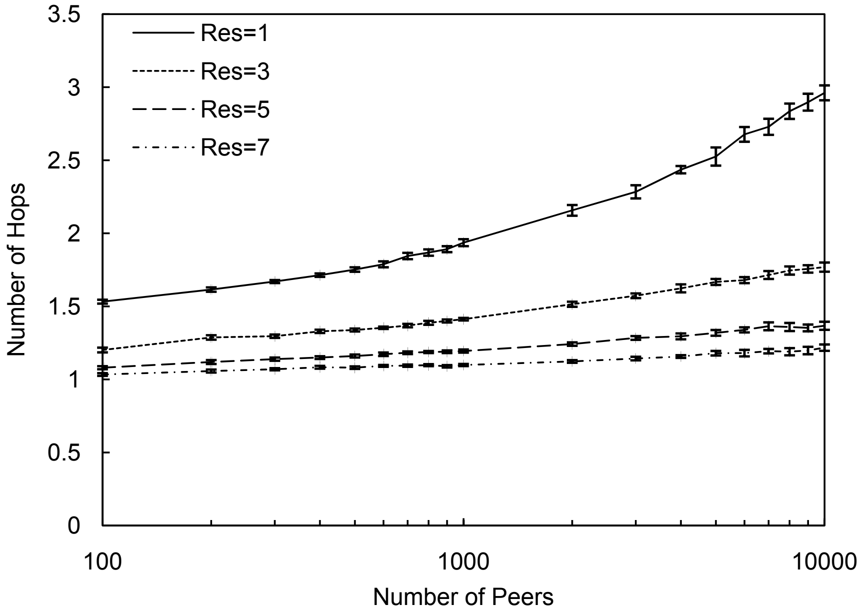

Figure 5.1: The routing performance in Nozzilla

5.1 Routing Performance

One of the objectives of this experiment was to evaluate the performance of joining the multicast tree considering the dynamic nature of the overlay. Peers send negative advertisement whenever local resources for a multicast tree are exhausted which in turn may increase the routing distance and in the most unfavorable situations may cause peers to be rejected. Figure 5.1 illustrates the average number of hops versus the overlay network size for the four selected resource values.

These results validate the routing algorithm and indicate that a new node can join a multicast tree in a reasonable number of hops: negative advertisements decrease the number of hops necessary to join a multicast tree, because they reduce the number of advertised identifiers. The routing performance is improved when increasing the value of res. We obtained a larger improvement when the average number of resources per peer is smaller but the improvement becomes insignificant for larger values of res. Anyway, when the value of res is equal or greater than three, the average number of hops is below two that it seems to be a reasonable value in a real scenario.

5.2 Peer Distribution in the Multicast Tree

To assess the scalability of the proposal we analyzed the distribution of peers inside the multicast tree. This is an important factor because it helps to determine whether the P2P overlay plays an important role distributing the video traffic. We analyzed the peer distribution from the following perspectives: the percentage of peers joining to the root of the multicast tree, the multicast tree depth and the distribution of peers versus the tree level.

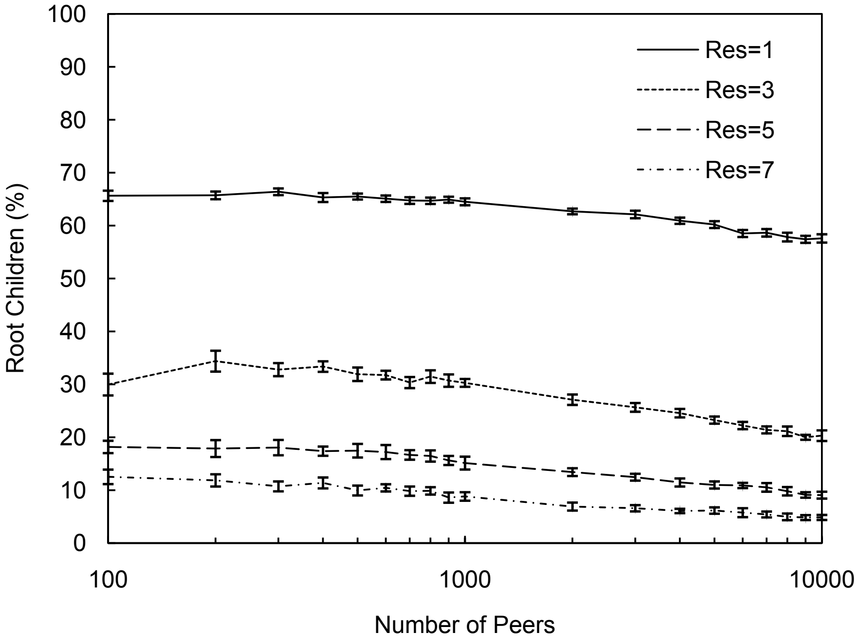

Figure 5.2: Root children ratio

The percentage of peers joining to the root of the multicast tree shown in Fig. 5.2 identifies the load on the root. The target is to minimize the number of children of the root node by avoiding selecting the root as the first hop whenever possible. When $res = 1$ the percentage of root children is large since a given peer can have either one uplink resource (i.e. can accept one child) or none with equal probability. Thus, immediately as such a peer accepts a child it can no longer be an interior node and has to leave the peer group. Because the lookup algorithm is build to minimize the joining delay rather than to search for a suitable parent other than the root, most newcomers will join by choosing the most stable peer in the targeted group, which is the root. This tendency is also observed in Fig. 5.3 and Fig. 5.4 which illustrate the distribution of the peers with the tree level for an overlay of 100 and 10000 peers respectively.

In such a low-resource scenario, besides the root, the bulk of the peers able to accept new children are the last peers that joined (therefore many older peers do not know them yet) and the peers that are isolated (have an ID with a large distance to the root and the probability to encounter them during a regular join lookup is low). Not only these peers are hard to find in most lookups for the mentioned reasons, but the situation is worsened by the older and more stable peers that are forced to leave the peer group when they accept new children.

The behavior is significantly improved when the average number of resources is increased, due to the selection process of the first hop that is able to reach a parent other than the root. However, the drawback of a scalable parent selection is the delay of the video traffic experienced by the average peer. When the average number of resources per peers is higher, most parents are regular peers leading to a greater average tree level and a higher tree depth. This aspect can be observed in Fig. 5.3 and Fig. 5.4, where for res equal to 5 or 7 it is easy

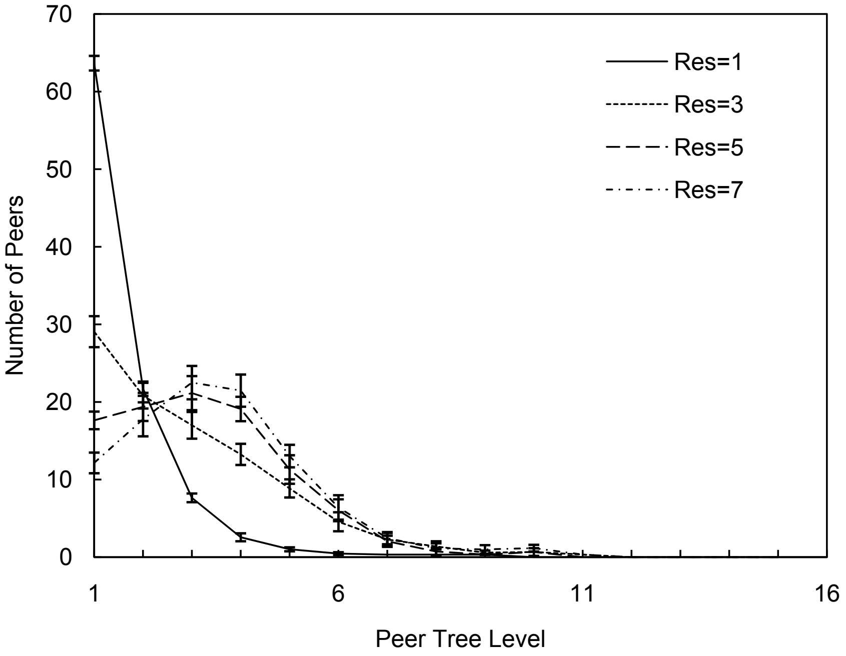

Figure 5.3: The distribution of the peers with the tree level in an overlay with 100 peers

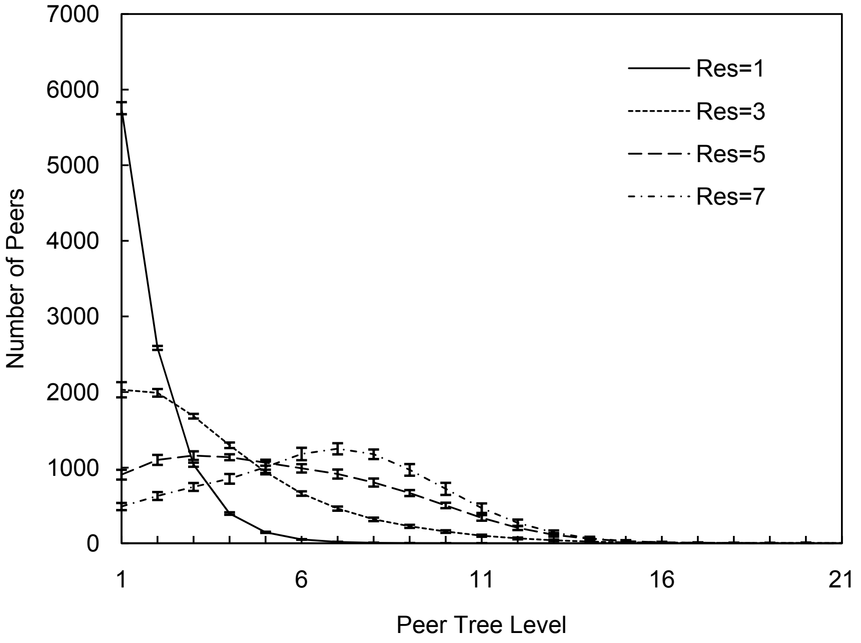

Figure 5.4: The distribution of the peers with the tree level in an overlay with 10000 peers

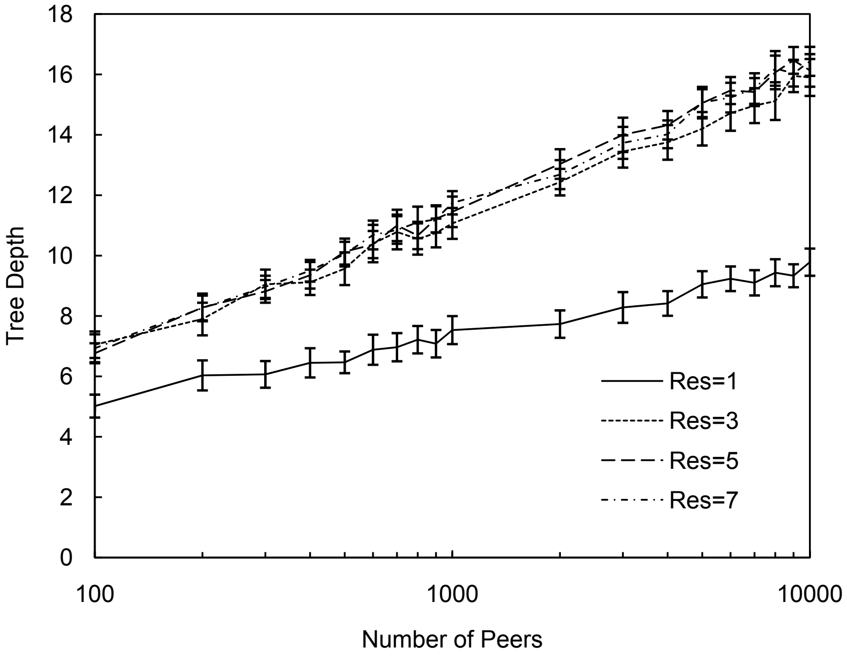

Figure 5.5: The multicast tree depth

to observe that most of peers have a tree level greater than one, thus not being children of the root. Similarly, in Fig. 5.5 the average depth of the multicast tree increases with the increase of the number of resources (although the difference when res equals 3, 5 and 7 is smaller than the measurement error).

5.3 Comparative Analysis with Scribe

Finally we compare the behavior of Nozzilla to other proposals based on the traditional Scribe, in order to determine whether we achieved the design targets of improving the tree join and the peer distribution. Because Scribe does not support the notion of resources, and because we wanted to obtain comparative results taken under the same conditions, in these experiments the resource check was disabled, effectively meaning that each peer has infinite resources. Thus, the results presented here illustrate the best-case scenarios of Nozzilla versus Scribe. Although a worst-case analysis would have been interesting as well, because Scribe does not specify how to implement some of the performance improving features part of Nozzilla, in such scenarios the results would be obvious biased in Nozzilla's favor.

Under the current assumptions, both protocols behaved excellent with a 100 % success ratio and a routing performance of approximately 1 hop. However, the peer distribution that determines the root children ratio and the tree depth was dramatically different, because Scribe does not include the random parent selection algorithm. Figures 5.6, 5.7, 5.8 and 5.9 illustrate this behavior. The most striking observation is that the percentage of root children in Scribe is significantly higher than in Nozzilla especially for a small number of peers. This

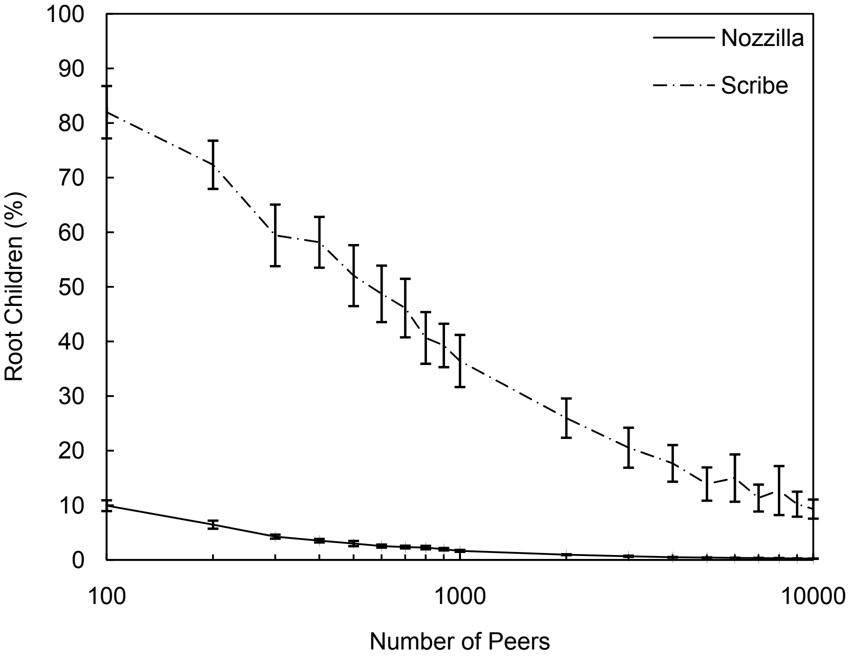

Figure 5.6: Nozzilla and Scribe root children ratio

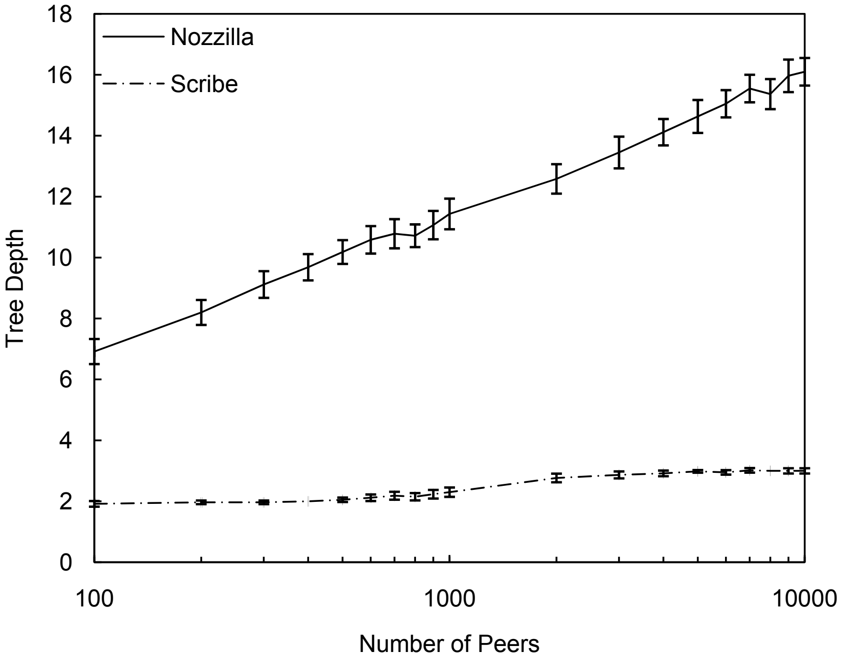

Figure 5.7: Nozzilla and Scribe tree depth

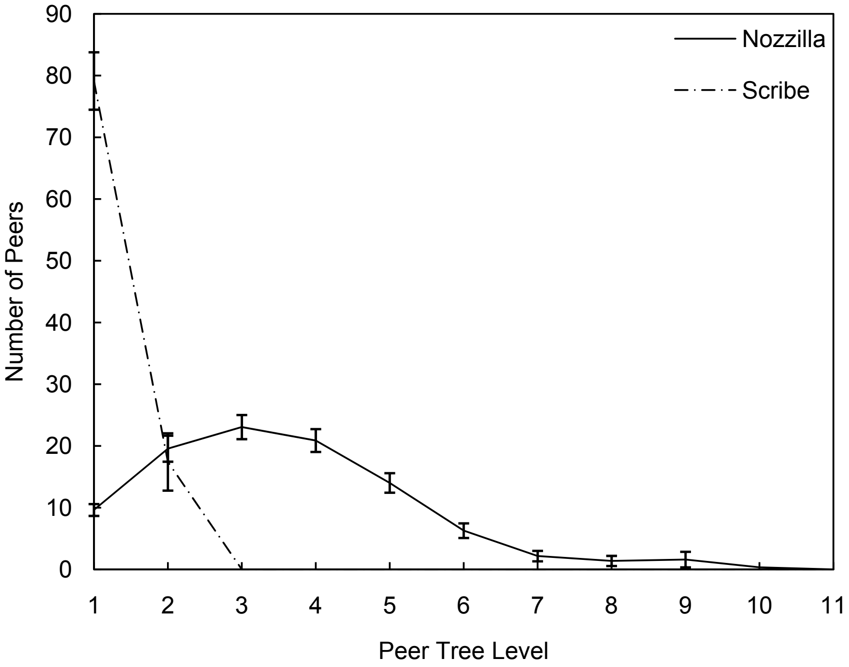

Figure 5.8: Nozzilla and Scribe peer distribution with peer level in an overlay with 100 peers

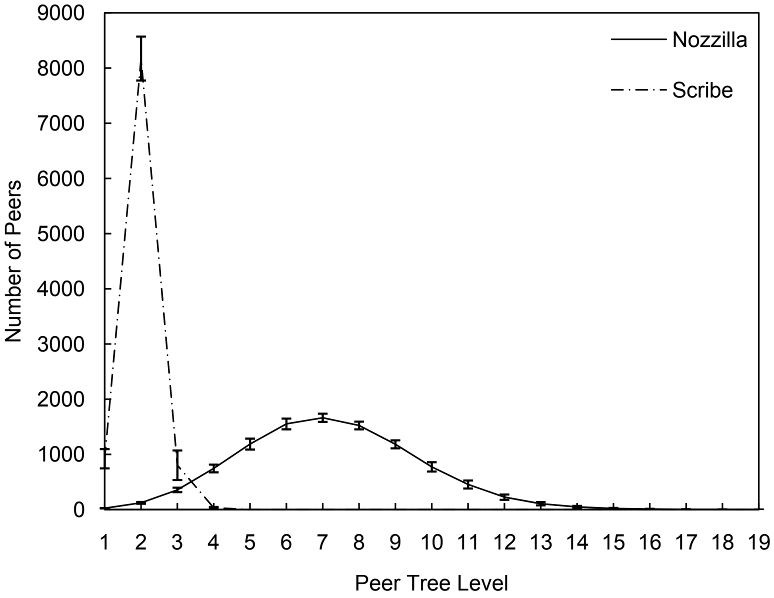

Figure 5.9: Nozzilla and Scribe peer distribution with peer level in an overlay with 10000 peers

is because routing is always done according to the leaf set, and when the number of peers is small there is a higher chance to find the root directly in the leaf set. On the other hand, in Nozzilla the selection of the first hop is done by choosing randomly any neighbor that is found the target peer group. The selection is done from the reunion of the leaf set, neighbor set and routing table and the distance to the root is not involved in this decision.

Chapter 6

Summary

In this thesis we presented Nozzilla, a novel P2P architecture to distribute video using multiple multicast trees. This mechanism is especially useful when the video is coded using multi-description coding or scalable video coding techniques, which generate several stripes: the more stripes received, the better the quality obtained. The main objectives are to proactively take into account the local uplink resources of each peer in order to increase the efficiency of multicast operations such as joining to the service. In addition, we strive to influence the geometry of the multicast tree such that no peer (as the root of the tree) is subjected to a higher load, which is proved to be a disadvantage of other proposals that have to rely on additional overhead to solve it.

Nozzilla is based on well known P2P algorithms such as Pastry and Scribe, modifying some of their functions in order to split the hash space in several groups. When a node advertises in the P2P network a given identifier it means that this node can be an interior node for the corresponding multicast group.

The experimental evaluation shows that our proposal has a very good success rate and a lower percentage of root children. These correspond to both a higher average of the peer tree level and to a higher tree depth when increasing the number of peers. This is expected behavior of Nozzilla and it is intended to reduce the joining effort (thus making failures quickly recoverable) and the load on the root of the multicast tree, which is the video streaming server in many applications. However, this is done at the expense of a lengthier video path (i.e. higher delay that is not always important in such applications) that increases with both the available resources and number of peers.

Our future work will be directed to improve the response of the protocol when peers leave or fail, to extend the random peer selection algorithm for intermediate hops, and to provide a path diversity by providing a forwarding soft state in both initiator peers (in case of rejections, this soft state will help select an true alternate path).

References

[1] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 298–313, 2003.

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

[3] H. Erikson. MBONE: The Multicast Backbone. *Communication of the ACM*, 37(8):54–60, 1994.

[4] J. Garcia, F. Valera, I. Vidal, and A. Azcorra. A broadcasting enabled Residential Gateway for Next Generation Networks. *Broadband Convergence Networks, 2007. BcN'07. 2nd IEEE/IFIP International Workshop on*, pages 1–12, 2007.

[5] A. Kovacevic, S. Kaune, H. Heckel, A. Mink, K. Graffi, O. Heckmann, and R. Steinmetz. PeerfactSim. KOM-A Simulator for Large-Scale Peer-to-Peer Networks. *Technische Universitat Darmstadt, Germany, Tech. Rep. Tr-2006-06*, pages 2006–06, 2006.

[6] A. Nicolosi and S. Annapureddy. P2PCAST: A peer-to-peer multicast scheme for streaming data. *1st IRIS Student Workshop (ISW'03). Available at: http://www.cs.nyu.edu/nicolosi/P2PCast.ps*, 2003.

[7] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes In Computer Science*, 2218:329–350, 2001.

[8] S. Saroiu, P. Gummadi, S. Gribble, et al. A measurement study of peer-to-peer file sharing systems. *Proceedings of Multimedia Computing and Networking*, 2002, 2002.

[9] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 107–120, 2004.

[10] D. Tran, K. Hua, and T. Do. A peer-to-peer architecture for media streaming. *Selected Areas in Communications, IEEE Journal on*, 22(1):121–133, 2004.