Πολλαπλή αποστολή δεδομένων σε DHT δίκτυα

Καπρίτσος Εμμανουήλ
AM 457

Τμήμα Μηχανικών H/Υ & Πληροφορικής
Πανεπιστήμιο Πατρών

Επιζήτης καθηγητής
Παναγιώτης (Peter) Τριανταφύλλου

Διπλωματική εργασία που εκπονήθηκε στα πλαίσια του
Μεταπτυχιακού Διπλώματος Ειδίκευσης
‘Επιστήμη και Τεχνολογία των Υπολογιστών’ στο
Πανεπιστήμιο Πατρών

· 2007 ·
Abstract

The rapid evolution of the Internet and network technologies has led to an equally rapid increase in data dissemination applications. At the same time, the needs for quick data transfer increase daily. One of the most demanding categories of information disseminating applications are file sharing applications. In these applications, one transmitter wants to send data to a group of receivers, that are geographically distributed.

It is obvious that the transmitter can not send the data to all receivers simultaneously, because his bandwidth is limited, while the receivers may be hundreds or even thousands. Therefore, the most common method is that of the creation of a dissemination tree, where the initial node forwards the information to some recipients and they forward it to their children etc. The dissemination tree is usually constructed over a peer-to-peer network and specifically over a Distributed Hash Table (DHT) network. This method, although solves the problem of multicasting, faces some problems. First, the dissemination tree is static, which means that the connections between the nodes can not be rearranged, if the condition of the underlying network changes. Moreover, there is no control over the efficiency of the nodes in the highest levels of the tree. This leads to a significant drop in the tree efficiency.

In this thesis, we study the creation of a dynamic dissemination tree, which can adapt to the network conditions, thereby increasing the tree performance. Specifically, the basic evaluation metric is the bandwidth that end users perceive. We present evidence that shows the improvement that our algorithm imposes.

We also study the creation of a dissemination tree that uses the existing DHT structure to efficiently and reliably (using erasure coding techniques) disseminate information and guarantees a logarithmic number of hops for message delivery. At the same time, the system tries to balance the load among all network nodes.

Both systems were implemented and evaluated using the Pastry system and its implementation in Java (FreePastry).
Περίληψη

Η ραγδαία ανάπτυξη του Διαδικτύου και των τεχνολογιών που το υποστηρίζουν έχει οδηγήσει στην ραγδαία αύξηση των εφαρμογών διαμοίρασης δεδομένων. Ταυτόχρονα, οι ανάγκες για ταχέα μεταφορά δεδομένων γίνονται πιο συνεχείς και μεγαλύτερες. Μια από τις πιο απαιτητικές κατηγορίες εφαρμογών που διανέμουν πληροφορία είναι οι εφαρμογές πολλαπλής αποστολής δεδομένων. Σε αυτές τις εφαρμογές, ένας αποστολέας θέλει να στείλει δεδομένα σε μια ομάδα παραλήπτων, οι οποίοι στη γενική περίπτωση είναι γεωγραφικά κατανεμημένοι. Είναι προφανές ότι ο αποστολέας δεν μπορεί να στείλει τα δεδομένα σε όλους τους παραλήπτες ταυτόχρονα, γιατί το εύρος ζώνης που διαθέτει είναι περιορισμένο, ενώ οι παραλήπτες μπορεί να είναι χιλιάδες. Ετσί, υιοθετείται συνήθως το στρατηγικό τεχνικό δημιουργίας ενός δέντρου διανομής, όπου ο αρχικός κόμβος στέλνει σε μερικούς μόνο παραλήπτες, οι οποίοι προωθούν το μήνυμα στα παιδιά τους κ.ο.κ. Το δέντρο διανομής συνήθως κατασκευάζεται πάνω από ένα δομημένο δικτύο ομοτίμων (P2P network) και πιο συγκεκριμένα πάνω από ένα δικτύο βασισμένο σε Κατανεμημένους Πίνακες Κατακερματισμού (Distributed Hash Tables - DHT). Αυτή η τεχνική, καθώς και άλλες, προφανώς λύει το πρόβλημα πολλαπλής αποστολής, αντιμετωπίζει όμως κάποια προβλήματα. Πιο συγκεκριμένα, το δέντρο διανομής είναι στατικό, δηλαδή δεν μπορεί να μεταβληθούν οι συνδέσεις μεταξύ των κόμβων αν αλλάξουν οι συνθήκες του υφιστάμενου δικτύου. Ακόμη, μεταξύ των κόμβων που βρίσκονται στα υψηλότερα επίπεδα του δέντρου, δεν υπάρχει κάποια ελεγχόμενη διάδοση δεδομένων. Αυτό έχει ως αποτέλεσμα το δέντρο να χάνει ένα μεγάλο μέρος από την αποδοτικότητά του. Στα πλαίσια της εργασίας αυτής, μελετάμε την δημιουργία ενός διανομικού δέντρου διανομής, χρησιμοποιώντας το σύστημα Pastry. Και το σύστημα Pastry υλοποιείται σε Java (FreePastry).

Παρουσιάζουμε διάφορες στατιστικές κοινωνίες που δείχνουν τη δραστική βελτίωση που επιτύχθηκε με τη χρήση του αλγόριθμου erasure coding, να εγγυάται ένα λογαριασμό δέντρου διανομής ενώ οι παραλήπτες μπορούν να επικοινωνήσουν κατά τη διάρκεια της αποστολής. Και το σύστημα Pastry υλοποιείται σε Java (FreePastry) και επιδεικνύεται ότι επιτύχθηκε μεγάλη βελτίωση που επιτύχθηκε με τη χρήση του αλγόριθμου erasure coding.
Contents

Acknowledgments 6

1 Introduction 7
  1.1 The Internet today ............................................ 7
    1.1.1 Scalability ............................................... 7
    1.1.2 Structure ................................................ 7
    1.1.3 Performance ............................................. 8
  1.2 Peer-to-peer systems .......................................... 8
  1.3 Multicast ..................................................... 9

2 Related work 10
  2.1 DHT systems ................................................ 10
    2.1.1 Chord .................................................... 10
    2.1.2 Pastry ................................................... 11
  2.2 Multicast over DHTs ......................................... 12
    2.2.1 End System Multicast - Narada ......................... 12
    2.2.2 Scribe ................................................... 12
    2.2.3 SplitStream ............................................. 12
    2.2.4 Bullet ................................................... 13
  2.3 Problems and motivation .................................... 13

3 BAD: Bandwidth Adaptive Dissemination 14
  3.1 Motivation and Contributions ................................ 14
    3.1.1 Contributions ........................................... 15
3.2 Bandwidth Adaptive Dissemination

3.2.1 Basic idea
3.2.2 Node state
3.2.3 Nodes joining/leaving BAD
3.2.4 Bandwidth calculation
3.2.5 Probing
3.2.6 Changing father algorithm
3.2.7 Algorithm for bandwidth distribution

3.3 BAD optimizations

3.3.1 “Bush” prevention algorithm for the store & forward model
3.3.2 Message reduction algorithm
3.3.3 High bandwidth root
3.3.4 Reliability
3.3.5 Discussion

3.4 Experimental evaluation

3.4.1 Experimental setup
3.4.2 Congestion modeling
3.4.3 Experimental results

4 NOVA: Logarithmic-Latency, Reliable, Scalable Multicast

4.1 Introduction
4.2 Problem Statement and Contributions
4.3 Algorithm basic idea
4.3.1 Basic principles
4.3.2 NOVA over DHTs
4.3.3 Load Balancing and Optimization During Multicast

4.4 Robustness
4.4.1 Segment loss
4.4.2 Recovery

4.5 Formal Arguments
4.5.1 Latency
4.5.2 Correctness
4.5.3 Load Balancing Optimization ........................................... 38
4.5.4 Messages and Bandwidth ............................................. 38

4.6 Implementation and Performance Evaluation ......................... 39
  4.6.1 Performance evaluation ............................................ 39
  4.6.2 Experimental Setup ............................................... 40
  4.6.3 Experimental Results ............................................. 41
  4.6.4 Discussion ..................................................... 43

5 Conclusions ........................................................................ 45

Bibliography ........................................................................... 48
Acknowledgments

First of all, I would like to thank my supervisor, Professor Peter Triantafillou, for his valuable insight and guidance. It was an honor being part of the “selected” group of people that he has worked with until past midnight, in order to finish editing a paper. I would also like to thank Nikos Ntarmos and Yannis Aekaterinidis for their continuous support. Specifically, I would like to thank Yannis for helping me “peek” into Peter’s dark office, in order to view the sketch I had drawn on the board (see image below). I would like to thank Nikos for being a constant source of knowledge and inspiration.
Chapter 1

Introduction

1.1 The Internet today

Since the creation of the Internet, its size has grown rapidly. Nowadays, there are billions of web pages and millions of users, resulting in the most complex and vast collection of data ever. The evolution of network technology has made it possible for a large number of services to emerge. In the most recent years, optical technology has multiplied the bandwidth available to end users. Inevitably, bandwidth-demanding applications like file sharing and streaming video-conferencing have emerged. As the Internet tries to cope with increasing user demands, it is of high importance that the applications are designed to take advantage of the underlying network capabilities.

1.1.1 Scalability

Obviously, this rapid evolution creates many problems. The Internet grows bigger every day, and yet the demand for performance increases. As the number of users participating in web-based applications continues to increase, scalability is the primal concern. Every system and application must be designed to scale to thousands or even millions of users. Scalability is a general notion that affects most aspects of a system. The amount of state that each node maintains, the number of messages sent, the bandwidth used and many other parameters must be sublinear with respect to the number of nodes in the system. Imagine what would happen in a multi-million user system, if a joining node would have to send a message to each other node in the system. This would obviously create a huge overhead, which would be enhanced by the very high frequency by which users log in and out of web-based applications.

1.1.2 Structure

In order to create efficient systems, we need them to have some sort of structure. Unstructured systems, although easier to implement, are far less efficient than structured ones. In structured systems, nodes (i.e. computers that participate in the network) maintain information about a relatively small subset of all nodes. This way, nodes can communicate with each other, thereby creating some sort of geometrical structure. Most structured systems are far more scalable than unstructured ones, because the state that needs to be kept, as well as the messages being sent, are significantly reduced.
1.1.3 Performance

Real life systems are usually evaluated according to several parameters. The most common among them are:

- the delay for delivering messages
- the bandwidth offered to end users
- the network overhead imposed by the system
- various Quality of Service parameters, including reliability issues

Apart from the aforementioned parameters, the performance of a system can be evaluated based on some more intuitive notions, like self-organization and adaptation. It is desirable for a system to be able to adapt to changing circumstances and reach an efficient state, without needing manual configuration.

1.2 Peer-to-peer systems

In the past decade, the most common web-based application has been file sharing. Millions of users have used application like Kazaa, Napster etc. to download audio and video files, applications or games. The large number of users that these applications attracted, made the use of traditional client-server systems almost impossible, due to the high load. Inevitably, decentralized architectures became increasingly more popular. Fully distributed systems, with no nodes acting as servers, or even special-purposes computers, are named peer-to-peer. Peer-to-peer (P2P) systems are distributed systems that connect geographically distributed nodes via largely ad-hoc connections. Unlike the server-client model, all nodes in a P2P network are considered to be equal. Although that is seldom the case (computers all around the world vary in computing power, bandwidth and storage capabilities), the functionality of P2P networks is based on this assumption. The main idea behind such a network is that all information is distributed throughout the network, instead of being kept in special purpose computers (servers). This approach has many advantages, but there are also some drawbacks. Since the information is distributed throughout the network, there is no single point of failure and the load is more evenly divided among all nodes in the network, avoiding the creation of bottleneck points. On the other hand, since nodes can join/leave the network at any time (and sometimes unexpectedly), the information needs to be replicated in several nodes in order to make sure that it is always accesible.

P2P networks have been widely used over the past decade. The most popular P2P applications are file sharing systems like Limewire, Kazaa, etc. In general, P2P application can be divided in two main categories:

Unstructured P2P networks This is the first form of P2P networks. Unstructured P2P networks use only ad-hoc connections between nodes.

Structured P2P networks They are the evolution of unstructured networks. Each node maintains information about other nodes in the network, creating a neighborhood.

Structured networks are superior to unstructured from many aspects. For example, searching for a specific file in an unstructured network would require to flood the request, in order to find a node that has it. On the other hand, a structured network uses specific mechanisms to greatly reduce the number of messages that are sent for file lookups.

DHT(Distributed Hash Table) networks are a special case of structured P2P networks that provide scalable resource lookup and routing. Some popular DHTs are Chord [30],
1.3 Multicast

Pastry [28], Tapestry [32], CAN [25] etc. DHT systems use a circular identifier space and a secure hash function [1] to uniquely map nodes and objects to that space. Objects are assigned to nodes using a DHT-specific function (e.g. the object is assigned to the node with the ID closest to the ID of the node). We shall present the design of Chord and Pastry, which are among the most popular DHT systems.

1.3 Multicast

Many applications need to send data to a large number of receivers at the same time. For example, the distribution of a new software package needs to be delivered to multiple clients all over the world. Moreover, video-conferencing is an example of multicasting streaming data from one transmitter to all receivers.

In the generic form of a multicast system, there is a source that wants to send some information to a number of receivers (which form a multicast group). The naive solution would be to send the information directly to each receiver. However, this solution is far from efficient, since the source has limited bandwidth. The most popular approach is to create a multicast “tree”. The source will send the information only to a limited number of nodes (e.g. 4) and each node will forward the information to some other nodes etc. This way, each node sends the information to few nodes (and can therefore achieve a high bandwidth with each of them), accelerating the process.

Since the first proposal of a network-level multicast protocol [10], back in the 90s, it has been obvious that such an idea could not be implemented easily, mainly because of the difficulty in wide deployment. Inevitably, application-level multicast has been gaining in popularity ever since. The main reason is that no additional infrastructure is needed for a multicast application to be deployed.
Chapter 2

Related work

2.1 DHT systems

In this section, we present Chord and Pastry, two popular DHT systems. Both systems offer a scalable lookup mechanism. Chord is more naive and simple, while Pastry is more sophisticated and more efficient from many aspects.

2.1.1 Chord

Chord [30] uses a secure hash function to map objects and nodes to a circular 160-bit ID-space. A predecessor of a node on the ID circle is the node with the ID that is closest to the node, going counterclockwise. In the same way, the successor of a node is the node with the ID that is closest to the node, going clockwise. Nodes are responsible for all objects with IDs between the node’s ID and the node’s predecessor’s ID. That means that when an object is inserted into the DHT, it is assigned an ID using a secure hash function. The object is stored in the successor of that ID (i.e. the first node that is equal to or greater than the ID).

Let \( m \) be the number of bits in the key/node identifiers. Each node \( n \) maintains a routing table with (at most) \( m \) entries, called the finger table. The \( i^{th} \) entry in the table at node \( n \) contains the identity of the first node, \( s \), that succeeds \( n \) by at least \( 2^{i-1} \) on the identifier circle, i.e., \( s = \text{successor}(n + 2^{i-1}) \), where \( 1 \leq i \leq m \) (and all arithmetic is modulo \( 2^m \)). We call node \( s \) the \( i^{th} \) finger of node \( n \), and denote it by \( n.\text{finger}[i] \). A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Note that the first finger of \( n \) is the immediate successor of \( n \) on the circle; for convenience we often refer to the first finger as the successor.

When a node wants to locate a certain key on the ring, it uses its finger table. It searches the finger table for an entry that is as far as possible but does not exceed the key, and forwards the request to that node. The node that receives the request repeats this process, until a node can not find an appropriate next step. Then, the successor of that node is the node that stores the desired key. As each node maintains information about nodes that have exponential distances from it, in each step at least half the distance between the node and the key is covered. Eventually, the lookup process takes at most \( O(\log N) \) steps, where \( N \) is the number of nodes in the network. This supports the argument that Chord offers a scalable lookup mechanism, which is a highly desired property.

Nevertheless, Chord suffers from some problems that its successors (like Pastry) try to solve. First of all, there is no guarantee that nodes that are close in the ring are actually close in the underlying network. Some proximity between each node and its neighbors is desired. This issue is addressed by Pastry.
2.1 DHT systems

2.1.2 Pastry

Pastry [28], as well as Tapestry [31] and Bamboo [27], are all based on location and routing mechanisms introduced in [24]. Plaxton et. al. present in [24] a distributed data structure (a.k.a. Plaxton Mesh) optimised for routing and locating objects in a very large network with constant size routing tables.

Assuming a static network, routing tables consist of multiple levels, where in each level \( i \) there are pointers to nodes whose identifiers (or node ids) have the same \( i \)-digit long suffix with the current node’s id. The routing of messages is achieved by resolving one digit of the destination id in each step \( i \) and looking at the \( i + 1 \) level of the local routing table for the next node. This mechanism ensures that a node will be reached in at most \( m = \log_{\beta}(N) \) logical hops, where \( N \) is the namespace size, \( \beta \) is the base of ids, and \( m \) the number of digits in an id. The size of the routing table is constant and equal to \( \beta \times \log_{\beta}(N) \).

Pastry [28] offers a robust, scalable, and self-organising extension to Plaxton’s Mesh under a dynamic environment. The routing scheme in Pastry, is similar to the one proposed by Plaxton et. al. with routing tables of size \( \beta \times \log_{\beta}(N) \) (with \( \log_{\beta}(N) \) levels/rows and \( \beta \) columns per level), resulting in \( \log_{\beta}(N) \) logical hops to locate a node. However, prefix (instead of suffix) matching is performed in each routing step towards the destination node, while routing table entries point to the closest node with the appropriate id prefix in terms of a proximity metric (such as round-trip time, RTT). Moreover, in order to achieve reliable routing, there is the notion of a leaf set for each node consisting of \( L \) pointers to nodes with id numerically close to the current node’s id. In Pastry there is also the notion of neighbouring nodes, which is a set of \( M \) pointers to nearby nodes according to a proximity metric and used for maintaining locality properties. Tapestry [31] and Bamboo [27] are DHTs with similar routing functionality.
Chapter 2: Related work

2.2 Multicast over DHTs

Many systems [3, 8, 9, 15, 26, 33] have used a DHT infrastructure [25, 28, 30, 32] in order to disseminate a multicast message to all recipients. The main idea behind such an approach is to use the DHT infrastructure to route messages to a rendez-vous point (the root of the multicast tree) using a logarithmic hop count. We shall overview four systems that each represent a different design philosophy.

2.2.1 End System Multicast - Narada

Narada is a protocol that implements End System Multicast [9], an early approach to an application-level multicast system. End System Multicast tries to overcome problems inherent to the design of IP Multicast. It is a quite simple solution of creating an overlay mesh, where all nodes know about all others. Moreover, nodes probe each other at random, in order to acquire information about the link between them. The link is added or dropped from the mesh, depending on its performance.

While it manages to build an effective overlay mesh, there are key points in its design that make it inappropriate for large-scale applications. In particular, all nodes maintain information about every member within a multicast group. Moreover, this information is updated every time a node joins or leaves the group. Such a mechanism prevents the application from scaling to a large number of group members. The authors of [9] mention this restriction explicitly.

2.2.2 Scribe

Scribe is an application-level multicast system that is built on top of Pastry. Scribe’s main goal is to build a multicast tree, using a rendez-vous point. Each multicast group has a unique identifier on the ID space (it can be the hash of the creator’s IP address). When a node wants to join the multicast group, it must route a JOIN message towards that identifier. If there are no other members for this group, the message will be delivered to the node that is arithmetically closest to the group identifier. This node is named the root of the tree. All nodes along the path of the JOIN message to the root become forwarders for that group and add the node that forwarded them the message as a child for that group. If a node receives a JOIN message, but is already a forwarder for that group, it adds the previous forwarder as a child for that group, but does not forward the message any further. This way, node joins are handled in a distributed way, as only few joins will actually involve the root itself.

When a node wants to leave the group, it sends a LEAVE message to its parent (the node towards which it forwarded its JOIN message). The parent deletes that node from his list of children and checks whether he should continue being a forwarder for that group. If he is not a member himself and has no other children for that group, then he leaves the group by sending a LEAVE message to its own father. This procedure continues until a node is found that either is a member for the group or has children for that group. Scribe uses “are-you-alive” messages to handle failures. This way, if a child realizes that its parent has failed, it can route a JOIN message, in order to be reconnected to the multicast tree.

2.2.3 SplitStream

SplitStream [7] is an application-level multicast system, like Scribe. It’s main goal is to balance the load among all nodes in the network, improving the way Scribe works. Scribe builds a multicast tree, where most nodes are leaves. Even in a binary tree, 50%
of the nodes are forwarders and 50% are leaves. As the fanout of the nodes increases, the percentage of leaves increases, too. As a result, almost all of the forwarding load during a multicast procedure is carried by about 10% of the nodes in a 16-ary tree.

SplitStream aims to address this inequality by splitting the data into stripes and forwarding each stripe along a different multicast tree. Given the way that prefix-based routing works (SplitStream uses Scribe trees over Pastry), all trees will have different internal nodes, as long as their identifiers have different first digits. This causes the load to spread over more nodes in the network, alleviating the inequality.

2.2.4 Bullet

A more recent and complete work on overlay meshes is Bullet [19]. Bullet is an overlay mesh that uses Erasure codes [20–22]. Bullet’s main idea is to construct an overlay tree that is enhanced by non-tree links, forming an overlay mesh. The data is encoded using Erasure codes and nodes use a distributed algorithm to locate disjoint data from other peers on the tree. Although Bullet is enhancing the functionality of a simple tree, it provides no guarantee about the depth of the tree and requires a significant overhead for maintaining the mesh structure and probing for overlay peers.

2.3 Problems and motivation

Most application-level multicast infrastructures depend on the underlying DHT in order to build an efficient multicast tree (considering one or more metrics). Moreover, they operate statically, in that the multicast tree can not change once it has been created. In doing so, these algorithms adopt a static nature, failing to comply with the rapidly changing nature of the underlying network. Moreover, most physical networks, like the Internet, perform network-level congestion control. However, the network traffic varies significantly over time, creating bottlenecks. There is much data [5, 29] that shows how quickly the state of the network can change. This leads to the conclusion that the available bandwidth between servers (and the users that are connected to them) is not stable; it may change drastically and frequently over time, due to changing network traffic characteristics. It is obvious that under such circumstances, a multicast tree that can not be rearranged is practically less useful, since its performance can drop significantly.

Moreover, many of the algorithms in related work can be deployed only over a specific DHT, which is not a desired property. We would prefer a multicast system that can be deployed over any DHT, which makes it more flexible to implement, while retaining the advantages of a DHT network (structure, locality properties etc.).

One of the main aspects of the Internet that remains to be dealt with by modern multicast systems is heterogeneity. The bandwidth capacity available to users around the world varies significantly. Therefore, a multicast system that does not take this into consideration is less desirable. For example, nodes with low bandwidth capacity can not have many children in the multicast tree, because their ability to forward large amounts of data is limited.

Finally, none of the approaches in related work can give any guarantee concerning the delay for delivering the message. All multicast systems in related work create a multicast tree or mesh, but have no way of limiting its depth. This way, the delay for delivering the message to nodes that are very deep in the tree could become significantly large. This could become an issue for real time data, that are extremely sensitive to the message delivery delay.
Chapter 3

BAD: Bandwidth Adaptive Dissemination

3.1 Motivation and Contributions

Why adaptive
As mentioned before, BAD’s [17] main goal is to improve the bandwidth perceived by users during multicast. Conventional systems, like Scribe (one of the most influential works), offer a static approach. When a node joins a multicast group, it can no longer change its position within the multicast tree. We will show that such a static approach has a big effect on the tree’s performance under both a static and a dynamic environment, as explained below.

Static environment
We will first consider a static environment, modeling an underlying network where no changes are made to the congestions of the links, or with perfect congestion control algorithms. Conventional multicast systems would operate just fine on such an environment, if all nodes had the same bandwidth capacity. Unfortunately, that is not the case in the Internet, where the bandwidth capacity of the users ranges from 400Kbps (simple connection) to 80Mbps (broadband connection) and more. Obviously, the approach of conventional systems that ignores heterogeneity can not lead to an efficient multicast tree. Building a multicast tree based on proximity or other similar metrics only, while not allowing nodes to reposition themselves on the tree, may prove catastrophic. For example, when a node with very low bandwidth must forward a multicast message to \(k\) other nodes, then each of those nodes will receive a very low data rate and the performance of the multicast tree will deteriorate dramatically.

Dynamic environment
We will now go further to a more realistic environment, where the congestion (and thus the “effective bandwidth”) of links changes over time. This change can be quite big, affecting greatly the performance of conventional multicast systems. These systems claim that a node chooses as his parent a node that is relatively close to it. This proximity however is not enough, if one is to claim that the system should maintain its functionality and performance through time. If, for example, the link connecting a node with its father becomes congested, then the node has no alternative than to suffer a very low data rate. The worst part is that the delay imposed by this congestion would affect the whole subtree rooted at this node.
It is therefore the rapidly and widely changing nature of the Internet’s traffic characteristics that commands the use of adaptive multicast algorithms. BAD overcomes such problems
by rearranging the multicast tree in order to improve the data rate that end users perceive during a multicast operation.

To sum up, the main problems that a multicast system should address are:

- Frequent group joins/leaves. They should be handled in a distributed way, to prevent a bottleneck at the root.
- Multicast trees should be properly formed, so that desired properties are met (e.g. optimization of average bandwidth perceived by end users).
- Heterogeneity. The bandwidth capacity available to end users around the world varies significantly.
- The underlying network traffic characteristics change very often and sometimes abruptly. The performance of the system should be as independent as possible of such changes.
- The overhead imposed (messages sent and state kept), in order to reach high performance, should be small.

It is obvious that the last problem is actually a trade-off between high performance and small overhead. It depends on the needs of the application to define how small the overhead must be, in order to achieve the desired performance.

BAD tries to solve all of the above mentioned problems.

### 3.1.1 Contributions

The contributions of BAD are:

- A novel suit of distributed algorithms for creating and maintaining high-performance adaptive multicast trees
- An algorithm for bandwidth distribution among heterogeneous nodes
- An algorithm for determining the amount of adaptivity desired by the system. This amount has a direct impact on the overhead imposed. If no adaptivity is desired, the system is reduced to a static approach.
- An algorithm for reducing the overhead imposed by the system, in terms of reducing the state being kept and the messages needed to be sent between nodes.
- A novel algorithm for efficiently choosing a high-bandwidth root for the multicast tree.

### 3.2 Bandwidth Adaptive Dissemination

BAD’s main goal is to alleviate some fundamental problems that appear in conventional application-level multicast systems, as mentioned above. BAD adopts an adaptive nature, in order to comply with the frequently and abruptly changing nature of the Internet (which in most cases is the physical network, upon which the application is built).

As with all multicasting systems, the group concept is essential. A group consists of a set of nodes that wish to receive information relative to the group’s topic. Each group is identified by a unique group ID and corresponds to a unique multicast tree. Nodes that belong to the tree are divided into four categories:
Producers Nodes that provide information relevant to the group’s topic. This information is sent to the root of the multicast tree, so that the multicast operation can begin.

The root of the tree The node to which the join messages are routed. This node is determined by DHT-specific mechanisms based on the value returned by hashing the group ID (e.g. in Chord [30] it would be the node numerically closest to that value). In the following analysis we use the terms “root of the tree” and “root of the group” interchangeably.

Forwarders Nodes that forward the multicast messages to their descendants on the tree.

Consumers Nodes that joined the tree to receive information relevant to the group’s topic.

We should note that a node can be a consumer and a forwarder at the same time.

BAD was designed to improve the performance of existing application-level multicasting systems. The main performance metric is the bandwidth that the end users (consumers) experience. That is the downlink bandwidth between the root of the tree and the consumer. Since this is a multicast tree, data is disseminated along the paths of the tree. Therefore, the bandwidth perceived by the consumer depends on the bandwidths of each overlay link along the path from the root. We will discuss later on how this bandwidth is calculated, with respect to the forwarding method used by the system.

3.2.1 Basic idea

BAD’s basic function is to rearrange the multicast tree, so as to make the dissemination more efficient. In order to do so, each node maintains a set of “relatives” that it can probe. At predefined time intervals (which can be dynamically set) each node probes his relatives to check if some of them are more appropriate fathers than his current father. If such nodes are found, the most appropriate among them is chosen to be the new father of the node and the transition procedure begins.

In order to deal with heterogeneity, BAD divides nodes into a number of categories with respect to their bandwidth capacity. As a result of that, not all nodes are treated the same when a decision must be made.

3.2.2 Node state

All nodes in a group need to maintain information about the following nodes (relative to that specific group):

1. Tree connectivity state
   - father: its own father
   - children: a list of its direct descendants on the tree

2. Algorithm-specific state
   - brothers: its father’s children
   - uncles: its father’s brothers
   - grandfather: its father’s father

   For each of the above mentioned nodes, the information that must be kept includes the following:
3.2 Bandwidth Adaptive Dissemination

**ID:** the node’s unique ID on the DHT.

**IP address:** The node’s IP address can be used to send direct messages to that node (e.g. probing).

**Bandwidth class:** which of the bandwidth capacity categories it belongs to.

For each of the above mentioned nodes except the children nodes, we keep additional information that is acquired through probing and is used to decide on the most appropriate father.

**Bandwidth from root:** an estimation of that node’s bandwidth from the root of the tree (downlink from the root).

**Congestion:** an estimation of the congestion on the lines between the remote and the local node. Later we will discuss further how this estimation is carried out.

For each of the children, we keep the following additional information:

**Bandwidth percentage:** the percentage of the local node’s total bandwidth that is reserved for the child.

### 3.2.3 Nodes joining/leaving BAD

BAD can be deployed over any DHT. Nodes that are already part of the DHT can join or leave BAD at any time. When a node joins BAD, it asks the underlying DHT to route a JOIN\_GROUP message to the root of that group. In most DHTs the root is the node which is numerically closest to the group ID, but this may not be the case in all DHTs. The function of this node is irrelevant to the definition of the root, giving BAD its independence from the underlying system. As the message traverses the DHT, all nodes that receive it become forwarders for that group. If they do not have an entry for that group already, they add one and add the sender of the message as a child for that group. When the message reaches the root or a node that already is a forwarder for that group, it is not forwarded any more. This way joins can be dealt with in a “local” way, without having to involve the root (unless the node becomes a child of the root). When a node state changes, all the changes affecting its relatives are propagated to them so that they can change their state, too. For example, when a node joins a group, its new father sends a message to all of its children to notify them of their new brother. Those nodes then send a message to their children to notify them of the arrival of their new uncle. Note that when a node joins a group (or changes father) only one level of this node’s subtree (its children) and only two levels of its new and old fathers’ trees (brothers and nephews) are affected. We shall show later that such a change is more than worth the overhead imposed by it. When a node wishes to leave a group, it sends a LEAVE\_GROUP message to its father. Then the father deletes this child. If there are no other children for this group and the node is not a member (consumer) of this group, then it sends a LEAVE\_GROUP message to its father. This procedure continues until a node is found with a child left for this group.

### 3.2.4 Bandwidth calculation

As you have noticed, we refer often to the term “bandwidth from the root”. The definition of bandwidth has a lot to do with the way the information is disseminated. Our approach uses the store and forward model, where a forwarder must first receive the entire message, before starting forwarding it. In this case, the bandwidth perceived by the end user is the inverse sum of the latencies on each link on the path from the root to the end user, as
proved below:
\[
BW(\text{root}, k) = \frac{1}{\text{latency}(\text{root}, k)} = \frac{1}{\sum_{i=0}^{k-1} \text{latency}(i, i+1)} = \frac{1}{\sum_{i=0}^{k-1} BW(i, i+1)}
\]

where \(BW(\text{root}, k)\) denotes the bandwidth (data rate) perceived by a node in depth \(k\) on the tree during a multicast operation. \(\text{latency}(i, i+1)\) denotes the latency between a father \((i)\) and its child \((i+1)\), while \(BW(i, i+1)\) denotes the bandwidth between these two nodes.

In order to calculate its bandwidth from the root, a node does not have to know the bandwidth between each of the nodes along its path to the root. Instead, the node needs only know the bandwidth between it and its father, as well as its father’s bandwidth from the root. Specifically, we prove the following to be true:

\[
BW(\text{root}, k) = \frac{1}{BW(\text{root}, k-1) + BW(k-1, k)}
\]

**Proof** The bandwidth from the root is the inverse latency between the node and the root, which is further analyzed as follows:

\[
BW(\text{root}, k) = \frac{1}{\text{latency}(\text{root}, k)} = \frac{1}{\text{latency}(\text{root}, k-1) + \text{latency}(k-1, k)} = \frac{1}{BW(\text{root}, k-1) + BW(k-1, k)}
\]

As mentioned above, each node maintains information about its father’s bandwidth from the root. It can also estimate the bandwidth between it and its father, as described in section 3.2.5. Using this information it can estimate its current bandwidth from the root.

We should note that the links that we mention are not physical links, but overlay links. Therefore a link’s bandwidth is the data rate with which two nodes on the overlay network can exchange information.

### 3.2.5 Probing

In order to maintain recent information, nodes need to send probe messages to each other at predefined intervals (alternatively, these intervals can be set dynamically). Each node sends (periodically) an ASK_BANDWIDTH message to all his relatives (brothers, uncles and grandfather). Each relative answers to this message by sending a large enough file back to the node along with some information. The node A can now estimate the congestion between it and its relative B based on the response time, the size of the file and the bandwidth capacity of its relative. Since the bandwidth capacity of B is known, and the average data rate at which the file was sent can be calculated, node A has an estimation of the congestion on the lines between A and B. Using this estimation and the information sent along with the file (mainly the percentage of the B’s bandwidth that would be reserved for A if A were a child of B) node A can now estimate the bandwidth between it and node B. This information will be used to check for a more suitable father than the current one, as will be discussed in the next paragraph.

In an effort to reduce the overhead of sending a large enough file, we could use a packet dispersion method, like packet pairs or packet trains [14, 16]. The bandwidth estimation method is orthogonal to the algorithm, as long as it provides a fairly good estimate of the available bandwidth.

### 3.2.6 Changing father algorithm

The main idea of BAD is the ability to rearrange the tree to improve its performance. BAD addresses that goal in a distributed way. Each node tries to find the best father among a set of relatives in order to improve its current bandwidth from the root. If this father can provide the local node with a better bandwidth from the root, then it is chosen as the new father. This way a node’s bandwidth from the root can only ascend (under stable network conditions).

Using the probing scheme that we described above, a node can probe all of its relatives,
estimate the bandwidth that it would experience if it were their child and determine if there are some nodes that provide it with better bandwidth (from the root) than its current father. In practice, in order to prevent many changes from happening, a parameter stabilize_c is defined. A candidate node’s offered bandwidth must be stabilize_c times greater than the bandwidth through the current father; otherwise the candidate node is rejected. If there are a lot of nodes that surpass this threshold, then the greatest offered bandwidth among them is chosen. At this point, we use a second parameter, group_c to alleviate small differences in bandwidths due to network effects. If a candidate node’s offered bandwidth is no more than group_c times smaller than the greatest offered bandwidth, then these two bandwidth offers are considered to be equal. In such a case, when two or more offers are considered to be equal, the best candidate is determined by other criteria (e.g. number of hops).

When a node A selects its new father NF, it sends a DIRECT_JOIN_GROUP message to it, without having to route the message through the DHT. It also sends a LEAVE_GROUP message to its current father CF, so that CF erases A from its children table.

Choosing father algorithm

```plaintext
answers ← probe Relatives()
for each x in answers do
    offered_bandwidth ← bandwidths[x.bandwidth_class] * x.bandwidth_percentage * x.congestion
    if offered_bandwidth > congestion_c * current_bandwidth then
        good_offers.add(x)
    end if
end for
if good_offers.isEmpty() then
    print '"No better father found"
else
    best_offers ← getEliteGroup(good_offers, grouping = group_c)
    new_father ← chooseFather(best_offers, metric = #hops)
    sendDirectJoin(new_father)
end if
```

3.2.7 Algorithm for bandwidth distribution

As mentioned before, one of BAD’s main goals is to deal with heterogeneity. A basic feature of BAD is that it takes into account the bandwidth capacity of children nodes in relation to the local node’s bandwidth capacity, in order to make sure that all nodes will receive a high data rate during a multicast operation. We want to avoid situations where a node has more children than it can handle, dividing his available bandwidth to them. To achieve that, each node keeps some state about the bandwidth class of its children. We need an algorithm that corresponds a percentage of the available bandwidth to each child. The algorithm is as follows:

We consider the node’s available bandwidth, capacity. We also consider the number of children, #children, and the number of children belonging to each capacity category, #level_X, where X = 1 denotes the lowest bandwidth level. First we check if the available capacity is large enough to provide all children with at least level_1 bandwidth. If not, there is nothing better that we can do, than assign bandwidth equal to capacity/#children to each child. If there is enough bandwidth, we first assign a level_1 bandwidth to each level_1 child and keep the remaining bandwidth for the rest of the nodes. Note that the remaining bandwidth is enough to supply the rest of the nodes with at least level_1 bandwidth. The next step of the algorithm is basically a repetition of the first, only now the remaining bandwidth is #children-#level_1 and the lowest level is considered to be level_2. Similarly to the first
step, we check whether we can supply the remaining children with level_2 bandwidth. If not, we divide the remaining bandwidth equally to all remaining children. If we can, we assign level_2 bandwidth to all level_2 children and repeat the procedure for the remaining children/bandwidth in step 3.

**Bandwidth distribution algorithm**

\[
\text{children}_{\text{left}} \leftarrow \text{children} \\
\text{capacity}_{\text{left}} \leftarrow \text{capacity} \\
\text{for level} = \text{lowest level} \text{ to } \text{highest level} \text{ do} \\
\hspace{1em} \text{if } \text{children}_{\text{left}} \times \text{level} > \text{capacity} \text{ then} \\
\hspace{2em} \text{assign } \text{capacity}_{\text{left}} \text{ to everybody} \\
\hspace{1em} \text{else} \\
\hspace{2em} \text{assign level bandwidth to nodes of that level} \\
\hspace{2em} \text{children}_{\text{left}} \leftarrow \text{children}_{\text{left}} - \#\text{level} \\
\hspace{2em} \text{capacity}_{\text{left}} \leftarrow \text{capacity}_{\text{left}} - (\#\text{level} \times \text{level}) \\
\hspace{1em} \text{end if} \\
\text{end for}
\]

3.3 BAD optimizations

3.3.1 “Bush” prevention algorithm for the store & forward model

The probing process involves all of the node’s relatives, including its grandfather. When a node “considers” becoming a child of its current grandfather, the offer that it receives is on an average case better than that of its uncles or brothers. That happens because the grandfather is one level higher than all of the above mentioned nodes. If the store & forward model is used, the latency from the root to the grandfather is smaller (depending on the depth of the node in the tree) than that from the root to the node’s father or uncles. Therefore, its bandwidth from the root is higher, resulting in a better offer (on an average case). This effect is more noticeable for higher levels (near the root). If no action is taken, all nodes would prefer their grandfather to their current father. This, in turn, would result in the creation of an extremely “bushy” tree, where all nodes would be children of the root. It is obvious that such a tree is undesired, since it does not possess any of the fundamental attributes of a multicast tree (the multicast process is reduced to a multiple unicast process).

To prevent that from happening, we try to alleviate the “bonus” gained when a node ascends a level (i.e. becomes the child of its grandfather). That can be achieved if a node multiplies the bandwidth offered by its grandfather with a certain factor, which we call *ascension factor*:

\[
\text{asc}(k) = \frac{k-1}{k}
\]

where \(\text{asc}(k)\) denotes the ascension factor at level \(k\).

3.3.2 Message reduction algorithm

**Logarithmic state**

We have tried to keep the number of messages used by the system as low as possible. We managed to do that, while reducing the state that nodes have to keep to a logarithmic limit. Each node does not need to “know” all of its relatives in order for the algorithm to perform well. Experimental evaluation shows that BAD performs very well even if nodes only know \(\log M\) of their brothers and \(\log M\) of their uncles, where \(M\) is the number of nodes in the group (\(M\) can be estimated using a number of techniques [11,13]). In order to do so, we must alter the algorithm that propagates information to the children. Whenever a father wants to inform its children that their brothers (or uncles) have changed, it chooses
Figure 3.1: Routing of a JOIN message through the SuperRing to get the root’s ID on the DHT ring

$logM$ brothers (or uncles) at random and propagates information only about these nodes to each of its children. For each child, there is a different random selection performed. Otherwise, all children would keep information about the same nodes and would all prefer to move to the “strongest” among them. That could lead to that node’s capacity being overweahlmed, while other nodes with high bandwidth would receive no new children and their bandwidth would remain unexploited. This way, whenever a node wants to probe its relatives, it will send at most $2\log M + 1$ messages.

**Happiness threshold**

In order to further reduce the number of messages that the system uses, we divide nodes into 2 categories. Nodes that have a current bandwidth from the root that is greater than a certain percentage of their bandwidth capacity, are considered to be *happy*. Those nodes do not probe their relatives in search for a better father. Of course, this can change for many reasons (e.g. the link between a *happy* node and its father has become congested). All other nodes are considered to be *unhappy* and continue to search for a better father. In order to separate *happy* from *unhappy* nodes, we set a threshold $0 \leq th \leq 1$. If a node’s current bandwidth from the root is less than $th$ times its bandwidth capacity, the node is considered to be *unhappy*. This way, if $th = 0$, the system is reduced to a static approach, like Scribe. If $th = 1$, all nodes are *unhappy*, and therefore are always searching for a better father.

We should note that when a node with high bandwidth (e.g. 0.9 times its bandwidth capacity) is searching for a better father, it burdens the system in two ways. First of all, the messages needed to communicate with its relatives are an unnecessary overhead. Second, though it may not be obvious, this node prevents the system from increasing its performance. For example, if a high bandwidth node $L$ finds a node $K$ with high bandwidth and decides to become its child, a part of $K$’s bandwidth is reserved for $L$. That way, it partially prevents other nodes (which probably need it more than node $L$) from repositioning themselves under node $K$. 
3.3.3 High bandwidth root

BAD’s main goal is to overcome the problems of previous multicast systems, in order to create high performance multicast trees. However, there is little that can be done about a multicast tree’s performance if the root’s available bandwidth is low. Other systems ignore completely the limitations that are imposed by the bandwidth capacity of nodes. That way, a root with very low bandwidth capacity can be selected. We have developed an algorithm to prevent that from happening. In particular, the root is selected among nodes of the highest bandwidth class. In order to select a root that always has a high bandwidth capacity, we organize high bandwidth nodes (nodes that belong to the highest bandwidth class) into a separate DHT network, which we call SuperRing. We assume that when a node wants to join BAD, it “knows” of a member of SuperRing (this assumption is not much different than the assumption made by all p2p systems that a node must know a member of the system in order to join). A high bandwidth node can join this DHT simply by contacting a node that already is in it. This node now has two IDs, one for the generic DHT and one for the SuperRing. When a message needs to be routed by any node to the root of the tree (e.g. a JOIN_GROUP message), it is directly sent to a member of SuperRing (we have assumed that all nodes know such a member). That member routes the message in SuperRing. The message destination is the hash of the group ID. The node that corresponds to this ID is considered to be the root of the tree. When a JOIN_GROUP message is delivered to the root, it informs the sender of the message of its ID on the generic DHT. Then the sender routes the message on the generic DHT, following the procedure that we have described earlier. The method is illustrated in figure 3.1.

This method provides us with a high bandwidth root, which is essential for a high performance multicast tree to be built. We should note however, that the usage of this method causes all group joins to involve the root \(^1\). That is of course an overhead, but we believe that the benefit acquired by a high bandwidth root are much more important. Besides, the root is guaranteed to be of the highest bandwidth class, so it can handle a significant amount of messages without being bottlenecked. Moreover, since we discuss a multicast tree, the join rate is much lower than that of a DHT. We therefore believe that the overall improvement achieved with this method is more than worth the overhead imposed by it. The experimentation section provides more detailed information on the benefits acquired and the overhead imposed by the use of BAD.

3.3.4 Reliability

BAD’s adaptive nature makes it resilient to node failures. Failure handling is inherent to the adapting mechanism of BAD. When a non-root node fails, the node’s children will realize (through the time-out of “are you alive” messages) that their father is offline and will simply choose one of their relatives (the one offering the best bandwidth) as their new father. Therefore, the impact of any node failure is limited to the node’s children and has no further effect on the tree. If the root of the tree fails, then its children will route a JOIN_GROUP message on the SuperRing. The node to which the message will be delivered is the new root of the tree and will act accordingly, taking up the “orphans” of the failed root. Again, the failure only affects the children of the failed node.

\(^1\)An alternative would be to cache the root’s ID in SuperRing nodes that have already joined the group. However, that approach can not handle root failures, because in such a case the ID of the correct root would be totally different than that of the cached one. That will cause joining nodes to route the JOIN_GROUP message to the wrong node (making all nodes along the path forwarders for the group).
3.3.5 Discussion

In order to maintain the tree, there are a lot of messages that need to be sent. This overhead could become quite big, having an impact on the performance of the underlying network. That is, of course, undesired. In order to avoid that, we have taken advantage of the “are you alive” messages that are sent between a child and its father, so that possible node failures can be detected. These “are you alive” messages are part of every multicast system, since they are vital for maintaining the tree connectivity. If we take a closer look, we will see that most maintenance messages are sent from a father to its child. Those messages can be piggybacked on the “are you alive” messages that are sent between a child and its father. That way most of BAD’s functionality is available without any additional messages. Within a typical 1500-byte packet, we can easily send information about $2\log M + 1$ nodes ($\log M$ brothers, $\log M$ uncles and a grandfather), for values of $M$ up to 20 or more. Of course there are some messages that cannot be piggybacked on the “are you alive” messages. Those are the messages that a node sends when probing its relatives, in its attempt to locate a better father. In the worst case, each node sends $2\log M + 1$ such messages every time it tries to locate a better father. Using the algorithms described above for reducing the messages sent, the average number of messages deteriorates even further, depending on the threshold chosen. These messages are considered to be the overhead that BAD imposes on the network load. The experimental evaluation shows that these messages are quite few, compared to the significant improvement of the minimum and average bandwidth of the multicast group.

3.4 Experimental evaluation

In this section, we present the experimental evaluation of BAD. These experiments compare the performance of BAD against that of Scribe, which is a very popular tree-based multicast system. We demonstrate the most important parameters and show how they affect the performance of the system. In the experiments we first construct a DHT with $N$ nodes. Next we perform $K$ random group joins, thereby creating a tree, as it would be created by Scribe. Nevertheless, we believe the performance of the tree created this way to be similar to the ones created by most of the conventional multicast systems that are built on top of a DHT network. After creating the tree, we run BAD on each of the nodes that joined the tree. As the experiments are run based on the simulated version of BAD, the algorithm does not run periodically. Instead it runs once on each node (starting from the root and heading to the bottom of the tree). When the algorithm has run in all nodes once, we consider that to be the first run of the experiment. Each node calculates its current bandwidth from the root.

We have evaluated the performance of BAD under two basic metrics. First we consider the minimum among the bandwidths (from the root of the tree) that all nodes experience. This metric is indicative of the improvement that is achieved for low bandwidth nodes, offering them a much better data rate than that offered by conventional multicast systems. The second metric is the average among the bandwidths perceived by all nodes. This metric indicates the overall improvement that BAD has achieved, including both low and high bandwidth nodes.

Additionally, we evaluate the overhead imposed by the use of BAD. In particular, the only overhead is the number of messages sent between nodes. Remember that we only count the probing messages, as the rest of the messages are sent from the father to its child and can be piggybacked on “are you alive” messages.
3.4.1 Experimental setup

We implemented BAD on top of FreePastry [12]. FreePastry is a Java implementation of Pastry, from Rice University. BAD was integrated in FreePastry as a p2p network, using FreePastry’s common API. BAD is available for anyone to test and use. The source code of BAD, as well as detailed instructions on how to incorporate it into FreePastry can be found here [2]. BAD was implemented using the direct simulator but can also run in distributed mode (actual implementation over the Internet). We have created Pastry networks of size 5000 - 50000 nodes. Various experiments have been run for each case, where the percentage of Pastry nodes that join BAD is ranging from 10% to 100%. We have also experimented with different values of threshold. In the experiments demonstrated here, threshold varies from 0.1 to 0.7.

We have divided nodes into 4 categories with respect to their bandwidth capacity: i) Low bandwidth DSL, ii) High bandwidth DSL, iii) T1 and iv) Broadband. The capacities are assigned to nodes using a uniform distribution.

3.4.2 Congestion modeling

As mentioned before, BAD uses an estimation of the congestion between 2 nodes, in order to calculate the effective bandwidth between them. Since the evaluation was conducted in a simulated environment, the congestion between any 2 nodes, as well as any changes affecting it, must be properly modelled. We model congestion as the percentage of available bandwidth, relative to the maximum available bandwidth (which is equal to the minimum of the bandwidth capacity of the 2 nodes). Congestion values range from 0 (fully congested link) to 1 (no congestion). In order to model the unpredictable nature of the Internet, congestion values are assigned from a normal distribution with mean value 0 and standard deviation 1. In the evaluation, we consider two kinds of congestion types. The first is a mild congestion, that is applied to the whole network and models a usual situation where most links are moderately congested. The second is a more intense congestion, which is applied only to a percentage of links, so that we can evaluate the performance of the algorithm under abrupt changes.

**Mild congestion** Mild congestion is calculated as follows:
\[\text{congestion} = 1 - \min(1, \frac{|n|}{3})\], where \( n \) is the random variable following the normal distribution \( N(0,1) \). Given the fact that 99.2% of the samples of a normal distribution are within 3 standard deviations from its mean value, almost no value of \( n \) is more than 3, and therefore very few congestion values are 0. Also, 68% of the samples are within one standard deviation from its mean value. That way, almost \( \frac{2}{3} \) of the congestion values are greater than \( \frac{2}{3} \). We can therefore claim that the congestion assigned to each link is mild.

**Intense congestion** Intense congestion is calculated as follows:
\[\text{congestion} = 1 - \min(1, |n|)\], where \( n \) is the random variable following the normal distribution \( N(0,1) \). Given the characteristics of the normal distribution that we mentioned above, almost \( \frac{1}{3} \) of the congestion values will be 0 (since \(|n|\) will be greater than 1). The rest \( \frac{2}{3} \) are distributed in the range \([-1, 1]\), so congestion values are in \([0, 1]\) (instead of \([\frac{2}{3}, 1]\) in mild congestion scenarios).

3.4.3 Experimental results

In this section, we demonstrate the results of the performance evaluation, based on various parameters. In most experiments, the metric evaluated is the minimum and average bandwidth that the end users perceive (from the root of the tree). In the rest, we evaluate the number of messages needed by the algorithm, in order to achieve a certain level of performance.
3.4 Experimental evaluation

Performance The first set of experiments is about the improvement that BAD can achieve for different number of nodes joining a certain group. In the first experiment, the number of nodes in the DHT is constant, while the number of nodes joining the group varies. The results of the experiment are demonstrated in figures 3.2 and 3.3 for the minimum and the average bandwidth respectively. We observe that the improvement of the minimum bandwidth increases as the percentage of joins increases. That is expected, because a high join percentage leads to a more "bushy" tree (i.e. each node has more children in average). That way nodes have more options when they try to reposition themselves under a better father. However, even in the worst case, where the join percentage is low (10%) and “skinny” trees are created, the minimum bandwidth increases about 39% after the first run of the algorithm. This means that the minimum bandwidth after the algorithm was run is 1.39 times greater than before. Furthermore, the average bandwidth (which is even more important, because it reflects the improvement enjoyed by all of the nodes) increases by 59% after the first run. As the join percentage increases, the improvement of the minimum bandwidth reaches up to 1700% and the improvement of the average bandwidth up to 356% after the first run. We have omitted results after the second run, because in most cases there is no further significant change in the minimum or average bandwidth. That does not mean that the algorithm has failed to improve the bandwidth any further. On the contrary, it means that the algorithm has managed to maintain the performance of the multicast tree, despite the fact that link congestions between all nodes have changed (as described in section 4.2).

To sum up, figures 3.2 and 3.3 show the improvement that can be achieved using BAD instead of Scribe. This improvement ranges between 39% and 1700% for the minimum bandwidth and between 59% and 356% for the average bandwidth. The performance of the system increases as more nodes join the multicast tree (for a constant number of nodes in the DHT).

We also observe that the improvement of minimum bandwidth varies a lot more than that of the average bandwidth. That happens because changes in the minimum bandwidth occur even if only one node (the one with the worst bandwidth) manages to find a better father. Things get even better if this father finds a better father for itself etc.

When it comes to average bandwidth, things are not so easy. In order to achieve a 50% improvement, we must have all nodes improve their bandwidth 50%. Since that is not possible (some nodes already have a very good father), some nodes will have to compensate for...
some others. Therefore, many nodes must achieve a 100% or even greater improvement of their bandwidth. It is clear that average bandwidth (and its improvement) is much more stable than the minimum bandwidth, since it is the average of thousands of nodes. Given the results of the previous experiments, we try to figure out how the join percentage affects the performance of the BAD tree. We have seen that an increasing join percentage leads to an increase of the performance of the tree. We further analyze the case where the join percentage is kept constant and observe the performance of the tree as the number of nodes increase. The results of this experiment are shown in figures 3.4 and 3.5, with join percentages 20% and 50%, respectively. Note that we no longer present the actual bandwidth achieved by BAD and Scribe, due to space constraints. Instead, we present the performance ratio for minimum and average bandwidth on the same chart. For example, a value of 2 for the minimum bandwidth means that BAD achieves 2 times higher minimum bandwidth than Scribe. Both these figures show that the improvement of minimum bandwidth declines slightly as the number of nodes increases. The improvement of the average bandwidth, on the other hand, seems to increase slightly. However none of these changes is large enough to imply a dependency on the number of nodes (when the join percentage is kept constant). We therefore conclude that the performance of the BAD tree is mostly a function of the join percentage, ranging from 60% to 350% for the improvement of average bandwidth and from 40% to 1700% for the improvement of minimum bandwidth.

**Happiness threshold** As mentioned before, we use a threshold algorithm to reduce the number of messages used by BAD. We have experimented on how different values of this threshold affect the performance of the system. We have tested the performance of the algorithm for various values of the threshold \( th \). We will present results for values of \( th \) ranging from 0.1 to 0.7. In figure 3.6 we present the performance of the BAD tree as a function of the threshold \( th \). As we have mentioned before, the choice of \( th \) is a tradeoff between a static and a dynamic tree. Each side has its advantages and disadvantages. That is confirmed by the results of the experiments, where 0.5 seems to be the best value in terms of bandwidth improvement. Combined with the results of figure 3.7, where we can see the messages used by the algorithm for each value of \( th \), we conclude that \( th = 0.5 \) provides us with the best combination of high performance, with a relatively low number
3.4 Experimental evaluation

Figure 3.6: Effect of threshold on tree performance of messages used. Further increase of the threshold will lead to a decrease in performance coupled with an increase in messages. On the other hand, further decrease of the threshold will lead in a decrease in performance that is followed by a respective decrease in messages. Therefore, if the network load becomes an issue, this option is open. The performance will not be as good as it can be, but it is still a lot better than that achieved by conventional systems.

**Intense congestion** The last set of experiments that we have conducted concerns the congestion between nodes. We have previously discussed two types of congestion, the mild and the intense congestion. In all previous experiments we have used mild congestion on all links between nodes. We present a final set of figures that demonstrate the effect that extreme changes in network congestion have on the performance of BAD. We have created a typical network of 20000 nodes and evaluated the performance of BAD as the number of joins increase.

In the first experiment the intense congestion affected 10% of the links. This percentage is quite high, considering the intensity of the congestion. It practically models a situation where the network is under an unusual amount of load. Even in such situations, BAD performs very well. The minimum bandwidth still manages to improve. This improvement ranges from 40% to 360% while the improvement of average bandwidth ranges from 75% to 450%. We observe that the improvement of minimum bandwidth is not as stable as it was in previous examples, while improvement of average bandwidth is very stable (with an increasing tension). That is explained by the fact that minimum bandwidth is a much more “fragile” metric than average bandwidth, as discussed before. It is therefore natural that its deviation increases as the intensity of the congestion increases, since there can be nodes that get congested (i.e. the link between them and their father gets congested) and can not find a better father, because all links between them and high-bandwidth relatives are congested. Results of this are shown in figure 3.8.

The second experiment models a situation that resembles a network split. The percentage of links that suffer from the intense congestion is increased to 50%. That is a very extreme case but we find it interesting to observe how the system reacts to such pathological cases. Results of this experiment are shown in figure 3.9. Observation of the figure shows that the improvement of minimum bandwidth is not stable (as expected), but in an average case it ranges between 20% and 350%. Furthermore, the improvement of average bandwidth
Figure 3.8: Effect of intense congestion (10% of links) on tree performance manages to remain in quite high levels, ranging between 60% and 200%, which is very good given the extreme nature of the situation.
Chapter 4

NOVA: Logarithmic-Latency, Reliable, Scalable Multicast

4.1 Introduction

Many of the existing multicast systems [8,9,33] try to build a single source multicast tree, using the routing protocol of the underlying DHT. This approach has many drawbacks, like bottleneck points, single points of failure (root) etc. More recent work has shown that a different approach can be adopted, using the Digital Fountain [4] model. In a Digital Fountain, the information is encoded into a number of segments (with a certain degree of redundancy, which is called stretch factor). In recent years, many systems have used the Digital Fountain approach [9,19]. In these systems, the idea of a multicast tree was advanced to that of an overlay mesh. In a mesh, a node receives data from many nodes, not only its own “father” (a father is the node directly above the current node in the multicast tree). This approach has many significant advantages. First, the reliability offered by such a system exceeds that of previous systems, because the data can be successfully delivered, even if some segments are lost. Second, the ability to handle each segment individually makes the system more flexible. This way, the dissemination of the information can be distributed among many nodes across the system, taking better advantage of the available network resources, in contrast to the single tree approach, which tends to “focus” the network load on few nodes at a time. These two advantages are the main reason why the standard Digital Fountain approach is utilized in the design of NOVA [18].

4.2 Problem Statement and Contributions

Existing multicast algorithms so far can provide no guarantees regarding the delivery delay for multicast messages. For example, multicast tree depths can become arbitrarily large. While most modern application-level systems operate upon DHTs, which guarantee logarithmic costs (in number of hops) for message delivery, unlike related work, we aim to utilize the DHT infrastructure in order to provide the same guarantees. Further, all existing solutions require extra routing state for multicast message dissemination, on top of that used by the DHT substrate. This typically introduces extra significant maintenance overhead which, in turn, hurts overall performance and scalability. In addition complex novel algorithms for distributed state maintenance need to be developed, implemented and tested.

In emerging applications (e.g. distributed social network sharing), semantic groups (of
nodes/users) may wish to form their own separate DHT, in which traditional one-to-one communication (e.g. DHT lookup) as well as multicast communication is desirable. Would it not be great if, with the same state used by the DHT protocol, multicast communication could be supported as well?

With NOVA we attempt to address the above performance issues. In addition, we wish to provide a robust multicast infrastructure. For this we will utilize the standard Digital Fountain scheme, augmented with novel recovery algorithms that can reduce the risk of having group members not receiving a multicast message.

Finally, we wish to design a multicast system that can be applied on top of several DHT architectures, which we believe is the key to achieving wide-scale applicability and usefulness.

The contributions of NOVA are:

- A novel suit of algorithms for creating and maintaining a multicast group, using the state and algorithms already present in a DHT system.
- Design and implementation of our algorithms for different popular DHTs, namely the Chord system and all networks utilizing Plaxton et al routing (e.g. Pastry, Tapestry, Bamboo, etc.).
- A novel algorithm for balancing the load during multicasting, without needing additional hops.
- An algorithm for recovery of any lost segments, that makes the delivery of the original data reliable.
- Implementation of the system over the Internet, as well as in a simulator, and detailed evaluation of its performance.

4.3 Algorithm basic idea

4.3.1 Basic principles

In this section we present the basic idea of the algorithm. We shall present the key elements of the algorithm first, without going into details about implementation on specific DHTs. The fundamental conceptual difference between this system and its predecessors is that NOVA is not built on top of an existing DHT system. Instead, it is built as a DHT system. The group concept is essential to our design, only now a multicast group is formed as a DHT. Nodes that want to join the multicast group need only to know a bootstrap node, just as in any DHT, and follow the exact same procedure. The nodes do not keep any additional state, except the routing state of the DHT.

As we have previously mentioned, the Digital Fountain approach has many advantages, including reliability and flexibility in global load balancing. Another key feature of NOVA is to split the original data in $k$ segments (using Erasure codes). In broad strokes, NOVA delivery works like this: the multicast source uses a hash function [1] to correspond each segment to a unique ID on the DHT\(^1\). The nodes responsible for these IDs are called starting nodes. The source uses the routing procedure of the DHT to locate the nodes\(^2\) and sends the corresponding segment to each one, along with a list of all starting nodes (we will use this list for the recovery algorithm later). Each starting node is then responsible to disseminate its segment to the whole network. The dissemination is done using the node’s routing table, using DHT-specific mechanisms. Fundamentally, NOVA operates as follows:

\(^1\)We can produce $k$ different IDs by hashing the group ID concatenated with the numbers 1 to $k$.
\(^2\)The IP addresses of the nodes can be cached to be reused in later multicasts.
it exploits the way a DHT’s per node routing table is defined and utilized during routing, so to define a unique territory of the name space for which each node is responsible, with the union of all territories being the complete namespace. Then, the NOVA algorithms basically instruct each node to ensure that all nodes that belong in its territory receive the multicast message. In different DHTs node territories are defined differently, depending on how routing is done and how routing tables are constructed. Thus, NOVA differs for different DHTs.

### 4.3.2 NOVA over DHTs

We now present the design of the system, using DHT-specific algorithms. First, we will present NOVA on Chord, which is a very popular DHT system. Although the performance is quite good with Chord (with high probability Chord guarantees $O(\log_2 N)$ lookup hop count), there are still some drawbacks inherent to the Chord system, like the lack of locality properties. Hence, to address this issue and showcase the wide applicability of NOVA, we will present the design of NOVA using the Pastry DHT structure. Pastry has good locality and routing properties. Specifically, in a network with $N$ nodes, messages are routed in less than $\lceil \log_2 N \rceil$ hops under normal operation ($b$ is a configuration parameter with typical value 4).

We should note that there are other systems, except Pastry, that use Plaxton et al routing [24]. We will only mention Pastry from now on, but the same arguments and algorithms can be applied to all DHTs based on Plaxton et al routing (Tapestry, Bamboo etc.). In the rest of the chapter, we shall refer to the set of nodes for which the current node is responsible for, as the node’s territory.

**NOVA on Chord**

Chord uses a circular ID space which typically ranges from 0 to $2^m - 1$, $m \geq 128$. Nodes and data are hashed using a secure hash function (e.g. SHA-1 [1]) and the output is their ID. Data is matched to a specific node according to its ID and is stored in that node (or its successor in the ring, if there is no node with that ID). The state kept by each Chord node consists of $O(\log_2 N)$ fingers that point to selected nodes in the network. The $i^{th}$ finger of a node $n$ contains information (usually the IP address) about the node that succeeds $n$ by at least $2^{i-1}$ on the ID circle. This way, look-ups can be performed in an efficient way. For example, the look-up for a key $k$ would be performed the following way: node $n$ would follow the finger that points as far as possible away from $n$, but before node $k$. This procedure continues until a node is found that has no such finger. Node $k$ is the successor of this node. Any look-up can locate a certain node within at most $O(\log_2 N)$ hops with high probability, as proved in [30]. Intuitively, at each routing step the remaining distance to be travelled is cut in half.

NOVA utilizes the routing table of Chord for multicast message dissemination, without having to employ any extra state or any extra maintenance algorithms. As mentioned above, the multicast procedure begins by encoding the original data into $k$ segments using Erasure Codes. Each of these segments is sent to a different (starting) node, which is responsible for disseminating it to the rest of the network. The selection of nodes is done as mentioned in section 4.3.1. Each of the starting nodes sends the segment to each of its fingers, along with the ID of the next finger. This way, each fingered node knows the subset of nodes that it is responsible for (its territory), which include all nodes between itself and the next fingered node. This procedure is repeated until a node has an empty territory (i.e. its immediate successor has already received the message). In figure 4.1 we illustrate the basic idea of the algorithm.

Note that each node on the DHT receives and forwards segments during the multicast
Figure 4.1: NOVA multicasting on Chord. The message is recursively forwarded to the fingers of each node, never exceeding each node’s territory.

process. This way, the network load is better distributed among more nodes of the network. Instead of having few nodes carrying a heavy load, we have many nodes carrying a lighter load. Moreover, the way that NOVA utilizes the routing properties of Chord, as proved in section 4.5, ensures that the maximum path for the delivery of any segment is $O(\log_2 N)$ hops with high probability, thereby setting an upper bound on the perceived latency. Furthermore, in practice a node does not necessarily have to wait for all the segments to arrive in order to reconstruct the original data. Instead, the use of Erasure Codes suggests that it only needs to receive a fraction of them (depending on the selected stretch factor). That is expected to reduce the average latency, as the data can be reconstructed regardless of the latency of the most delayed packets.

**NOVA on Pastry**

Pastry, like Chord, is based on a circular ID space, with typical range from 0 to $2^m - 1$, $m \geq 128$. The hashing scheme is similar to that of Chord, but the routing scheme is different. Think of Pastry node IDs as a sequence of digits with base $2^b$. The basic idea of the routing algorithm is to route messages increasingly closer to the key of the message. When a node receives a message with a certain key, the node forwards the message to a node that shares with the key a prefix that is at least one digit ($b$ bits) longer than the prefix that the current node shares with the key. In this way, with each hop the message gets one digit closer to its final destination, which is the node with the ID that is numerically closest to the key. In order to do that, a node needs to maintain routing state. The state consists of the routing table, the neighborhood set and a leaf set. The routing table is organized into $\lceil \log_2 N \rceil$ rows with $2^b - 1$ entries each. The $2^b - 1$ entries of row $n$ of the routing table each refer to a node whose ID shares the present node’s ID in the first $n$ digits, but whose $n + 1^{th}$ digit has one of the $2^b - 1$ possible values, other than the $n + 1^{th}$ digit in the present node’s ID. The neighborhood set is used to maintain locality properties and the leaf set is used to make routing consistent, in case an entry in
4.3 Algorithm basic idea

Figure 4.2: Forwarding of a segment. Digits are in base 16 and x represents an arbitrary suffix. Node 65a1x receives the segment in the second forwarding step. It forwards it to all rows, except the first two. The grayed areas represent the territory of each node. Node 6567x (highlighted in 65a1x’s table) receives the segment and forwards it to its own (smaller) territory.

the routing table is missing.

NOVA uses the routing table of Pastry to efficiently disseminate the information to the network. The original data as before is encoded into \( k \) segments and each one is sent to the \( k \) starting nodes, just as with Chord. A starting node sends the segment to all nodes in its routing table. Along with the segment, it also sends the number of row that the receiving node has in the starting node’s routing table. For example, a node that is in the 4th row in the routing table of the starting node will receive the segment along with the number 4. This node will then forward the segment to all nodes in its routing table (informing each node of its “row number”), except for the first 4 rows. Intuitively, each node sends the message to its territory. The “row number” informs the receiver of its own territory. An example is shown in figure 4.2 that illustrates the algorithm.

In section 4.5 we prove that the average path length for any segment is \( \lceil \log_{2^b} N \rceil \) hops with high probability. Moreover, since the forwarding mechanism uses the routing table of Pastry, we take advantage of Pastry’s locality properties. It can be shown [6] that the average delay of Pastry messages is less than twice the IP delay between the source and destination. These locality properties are directly inherited by NOVA.

4.3.3 Load Balancing and Optimization During Multicast

In section 4.5 we prove the correctness of the NOVA algorithm, when the routing tables are consistent. Furthermore, we prove that NOVA has very good latency characteristics, achieving a logarithmic average latency. However, its performance may suffer, because it does not distribute load evenly among nodes. In its present form, the algorithm overloads the starting nodes, as they have to forward the information (even a single segment) to all nodes in their routing table. In the same way, the nodes that receive it in the second
Figure 4.3: NOVA routing table for a node with ID 65a1x. Digits are in base 16 and x represents an arbitrary suffix. Gray and white areas denote the “curly columns” used in the optimization algorithm.

The step of the algorithm must forward it to a slightly lower, but still very large, number of nodes (all nodes in the routing table, except those in the first row). In each step, the number of nodes the segment must be forwarded to, becomes smaller, resulting in an uneven distribution of load among nodes of the network. We have modified the algorithm so that it will balance this load evenly among nodes, without disturbing the good latency properties of the initial algorithm. We introduce the concept of a “curly” column. The \( i^{th} \) “curly” column consists of the \( i^{th} \) non-empty entry in each row. Figure 4.3 illustrates the concept of the “curly” column. Every other one column is highlighted, in order to distinguish adjacent columns (gray and white).

The intuition behind this optimization is to assign a subset of a node’s territory to another node, without disturbing the latency properties. The first step of the optimization is that the starting node \( N_0 \) forwards the segment to all \( (2^b - 1) \) nodes in the first row of its routing table. In addition, the \( i^{th} \) node, say \( N_1 \), in the row (not counting the empty entry of the row) receives, along with the segment, a list of the IPs of the nodes in the \( i^{th} \) “curly” column of \( N_0 \), from the second to the last row. In this way, each node in the first row is responsible for all nodes in its routing table, but also for the nodes in the “curly” column that were assigned to it. However, it does not forward the segment to all of these nodes, but only to the first node of the list (and passes the rest of the list to it). In the second forwarding step, node \( N_1 \) will:

1. send the segment to all nodes in the second row of its routing table

2. send to the first node of the “curly” column assigned to it by \( N_0 \) the segment and the rest of the “curly” column

3. For each column \( j \) in \( N_1 \)’s routing table, let \( N_2 \) be the node in position \( (2^j) \). \( N_1 \) will send to \( N_2 \), in addition to the segment, the IPs of all nodes \( (k, j) \) in \( N_1 \)’s routing table.
4.4 Robustness

As mentioned above, robustness is inherent to the design of NOVA, because of the use of Erasure Codes. The stretch factor can be chosen appropriately, in order to provide the desired reliability degree. Thus, a node has to receive only \( n \) out of \( k \) segments in order to reconstruct the original message, where \( \frac{k}{n} \) is the stretch factor.

In the case that more than \( k - n \) segments are not delivered to a node, the node will initiate a recovery procedure in order to be able to locate as many of the missing segments as possible. The recovery algorithm guarantees that a node will eventually be able to reconstruct the original data.

4.4.1 Segment loss

There are several possible causes for a segment not to be delivered to all nodes. Among them are:

- Packet loss in the network layer
- Node failure
- Missing entries in the routing table of forwarding nodes

The reasons for the first two are obvious, so we will discuss the third one. Under high churn, the network can not reach a stable condition. In other words, the routing tables may not have all the appropriate values. It is therefore possible that a node \( A \) is responsible for node \( B \), but node \( A \) does not have an appropriate entry in its routing table\(^3\). In that case, node \( B \) will not receive the segment. However, this probability is very low and, as demonstrated in the Performance Evaluation section, the segment probability loss due to this cause is extremely small, even under churn.

\(^3\)An appropriate entry would be either node \( B \) or a node that is responsible for node \( B \).
4.4.2 Recovery

The recovery algorithm is initiated if the node \( A \) has received less than \( k - n \) segments after a timeout period has passed since the arrival of the first segment. The recovery algorithms are different for different DHTs, but the main idea is the same: the node tries to contact a node that is likely to have received the missing segment without problems. The obvious choice for such a node would be the starting node of the specific segment. Although the node does not know the starting node, it can ask one of the starting nodes that it knows (one that corresponds to a segment that the node has already received). As mentioned earlier, starting nodes keep a list of all starting nodes specifically for this purpose.

Although this approach works well, we would not like to focus the load on the starting node. If possible, the nodes should find another node that already has the segment, without having to contact the starting node. The problem is to find a node that has a high probability of having received the segment without problems. In order to do that, the node must avoid contacting nodes that belong to the same territory as the current node, because it is very likely that they have not received the segment, as a result of the failure of the node that is responsible for that territory. Below, we present the versions of this algorithm for Chord and Pastry.

Recovery with NOVA on Chord

The main goal of the recovery algorithm is to find a node that has a high probability of having the missing segment. The best option for a Chord node \( A \) is to ask its last finger. The reason becomes clear if we consider why node \( A \) has not received the segment. Consider node \( C \) that is responsible for node \( A \). There are three possibilities:

- node \( C \) has not received the message
- node \( C \) has failed
- the message from \( C \) to \( A \) has been lost

Since we can not know whether the last case has occurred, we will suppose the worst: either one of the first two cases has occurred. In such a case, all nodes in node \( C \)'s territory will not have received the message. Therefore, the best option for node \( A \) is to avoid looking in this territory. Note that node \( C \) is preceding node \( A \) by at least 1 position in the ring and by at most half a circle. Thus, following the last finger of node \( A \) will lead to a node that is outside of \( C \)'s territory, with high probability. If that node does not have the missing segment, node \( A \) will ask the starting node following the procedure mentioned above. In the extreme case that the starting node has failed, node \( A \) will ask the multicast source. If the source has also failed, \( A \) can route the query for the missing segment to a random node in the DHT, until it finds a node that has the missing segment.

Recovery with NOVA on Pastry

In Pastry, there are more options, as the routing table has more entries. In order to locate a node that has a high probability of having the missing segment, node \( A \) (defined as above) must again avoid the territory of node \( C \). In the Pastry case, this can be done by asking a node in the first row of \( A \)'s routing table. Since that node has a different first digit than \( A \), it can not be inside \( C \)'s territory, unless \( C \) is the starting node (which means that node \( C \) has failed). Since \( A \) can not know whether \( C \) is the starting node, the best option for \( A \) is to ask a node in its first row. If that node does not have the segment, then the following procedure is followed (same as with Chord):
1. node A asks another starting node for the IP of the segment’s starting node
2. node A contacts the segment’s starting node
3. if it has failed, the source is contacted
4. if the source has failed too, a random node in the DHT is contacted

4.5 Formal Arguments

In this section we present formal arguments to prove some key qualitative and quantitative properties of NOVA. These arguments are proved for Pastry. Similar arguments hold for the Chord system.

4.5.1 Latency

Claim 1

The average path length (number of hops) needed for the delivery of any segment from a starting node to any recipient node is $\lceil \log_2 N \rceil$, where $N$ is the number of nodes in the multicast group.

Proof NOVA multicast proceeds in steps. In forwarding step $i$, the segment is forwarded to the nodes in the $j^{th}$ row in the routing table of the current node, $\forall j \geq i$, and stops when all these rows are empty. The routing tables in Pastry have on average $\lceil \log_2 N \rceil$ rows. Thus, there are at most $\lceil \log_2 N \rceil$ forwarding steps on average. □

We should also note that the maximum number of hops needed on average during a multicast procedure does not have to be as high as this upper bound. For example, even if a node $A$ has an entry for node $B$ in the 10th row, that does not mean that node $B$ will necessarily receive the segment through node $A$.

4.5.2 Correctness

Claim 2

NOVA disseminates a segment to all nodes in the group, as long as the routing tables are consistent (i.e. if there is a node suitable for a table entry, that entry can not be empty).

Proof Given a segment, we consider the starting node $F_0$ and the random node $N$. We prove that $N$ will receive the segment.

Let $d_0$ be the prefix length that $N$ shares with $F_0$. Since routing tables are consistent, $F_0$ has an entry in the $d_0^{th}$ row of its routing table that has a common prefix with $N$, of length at least $d_0 + 1$. This entry could be $N$ itself, but for the sake of generality we will suppose that it is another node that has the first $d_0 + 1$ digits common with $N$. We will name this node $F_1$. Let $d_1$ be the number of digits that $N$ shares with $F_1$, where $d_1 \geq d_0 + 1$. Then, in the same way as before, $F_1$ has an entry in the $d_1^{th}$ row that has at least a common prefix with $N$, of length at least $d_1 + 1$.

In the same way, the $F_i$ forwarder has an entry in the $d_i^{th}$ row of its routing table that shares a prefix of length at least $d_i + 1$ with $N$, where $d_i \geq d_{i-1} + 1$. This procedure ends when, for a forwarder $F_j$, $N$ is the only node that shares a prefix of length $d_j$ with $F_j$.

It is straightforward to see, that in each forwarding step the prefix length shared between $N$ and $F_i$ is strictly ascending as $i$ increases. In other words, in each step we come at least 1 digit closer to $N$. In particular, in the $i^{th}$ step we come $d_i + 1$ steps closer to $N$. In the worst case (when all $d_i$s are 0), $N$ will receive the segment after $D$ steps, where $D$ is the number of digits of the IDs. □
4.5.3 Load Balancing Optimization

Claim 3

Claim 1 holds even when the optimization algorithm described in section 4.3.3 is employed.

Proof As proved in Claim 1, the average path length between the starting node and any recipient is \([\log_2 N]\). We prove that the use of the optimization algorithm does not affect this. According to the non-optimized algorithm, in the \(j\)th forwarding step a segment is sent to all nodes in the \(j\)th row of the routing table, \(\forall j \geq i\). In this way, the nodes in the last row (the \([\log_2 N]\)th on average) will receive the segment during the \([\log_2 N]\)th forwarding step. In other words, all we need to prove is that the optimization algorithm preserves this property (i.e. a node in the \(i\)th row of the routing table will receive the segment in the \(j\)th step, where \(j \leq i\)). The tricky point is that, with the optimization algorithm, the extra nodes assigned to each node seem to receive the segment later than they should. We prove that this does not happen.

The main difference of the optimization algorithm is that in the \(i\)th forwarding step the segment is sent to all nodes in the \(i\)th row of the routing table, along with a list of nodes, one in each row \(j\), \(\forall j > i\). Let \(A\) be the node that forwards the segment in step \(i\), \(B\) a node in the \(i\)th row of \(A\) and \(E\) the list of nodes that is appointed to \(B\), where \(E = [E_1, E_2, ...]\).

In the next forwarding step, all \(B\)'s will forward the segment to all nodes in the \(i + 1\)th row of their routing table, but also forward it to the first node in the list (i.e. \(E_1\)), along with the rest of the list. In this way, \(E_1\) (which is in the \(i + 1\)th row of node \(A\)) will receive the segment during the \(i + 1\)th step (as will all the nodes in the \(i + 1\)th row of node \(B\)). In the same way, node \(E_k\), which is in the \(i + k\)th row of \(A\), will receive the segment during the \(i + k\)th forwarding step. We have therefore proven that the optimized algorithm preserves the above mentioned property. It is straightforward to see that, as the forwarding procedure continues in this way, the segment will reach the \([\log_2 N]\)th row of node \(A\) in the \([\log_2 N]\)th step. 

\(\square\)

4.5.4 Messages and Bandwidth

The number of network messages used for a multicast procedure is analogous to the number of segments used and to the average path length between a starting node and any recipient node. We can reduce the number of messages either by reducing the number of segments (i.e. by using a lower stretch factor) or by reducing the average path length. The choice of the stretch factor is a tradeoff between the number of messages (i.e. the network load imposed) and the reliability of the system. As for the average path length, according to Claim 1, it is \([\log_2 N]\). We further discuss the choice of the stretch factor in section 4.6.4.

We should note that the number of messages used by the algorithm is the lowest possible. This becomes apparent if we consider the concept of a territory. A node is responsible for all nodes in its territory. In our algorithm, each node sends exactly one message to each node in its territory, in order to forward the segment. Since the territories of all nodes that receive the segment are non-overlapping, a node receives exactly one message. This is obviously a lower bound for the messages that can be used by any algorithm. If a node \(A\) in a territory \(T\) is not notified (implicitly or explicitly) by the node that is responsible for \(T\) (say node \(C\)), then \(A\) will not receive the message. The proof is straightforward, since nodes do not forward information to nodes outside their territory (unless \(C\) has notified \(A\) implicitly, through another node).

We will present an example that illustrates how some nodes will not receive the segment, if an algorithm tries to use less messages than NOVA. Consider the following simple example of a trivial DHT with \(2^k = 2, 2^m = 8\) and all 8 nodes are present in the DHT. Let \(x\) be the starting node with ID 010. Note that all nodes have \(2^k - 1 = 1\) node in each row. Suppose
4.6 Implementation and Performance Evaluation

We have implemented NOVA using the Pastry DHT infrastructure. We used the FreePastry [12] implementation of Pastry. We have implemented NOVA, using the Direct protocol (simulated environment) as well as the Socket protocol (actual deployment over the Internet). In the performance evaluation section we present results from the simulated version of NOVA. The main reason for this is the ability of the Direct protocol to simulate many thousands nodes. In this way, we can demonstrate the scalability of our results. The FreePastry version that we used is 2.0 and the parameter $2^b$ was set to 16. The source code of NOVA, along with detailed instructions on how to incorporate it into FreePastry, can be found in [23].

4.6.1 Performance evaluation

In this section we present the results of the experimental evaluation of NOVA. The evaluation uses the following performance metrics:

**Latency:** expressed as the average path length (number of hops) required for a segment to be delivered from a starting node to any recipient. We are mainly concerned about the maximum and average path length per segment, but also present detailed statistical properties.

**Reliability:** expressed as the percentage of nodes that receive enough segments so to be able to reconstruct the original message. We analyze the number of partial deliveries and discuss their relation to the encoding stretch factor, which has a direct effect on the number of messages.

As far as the latency metric is concerned, we compare the performance of NOVA to that of SplitStream, which is acknowledged by the community as one of the most efficient multicast systems over DHTs. We present detailed results that show that NOVA can achieve a far lower average path length, which is actually logarithmic, in terms of network
4.6.2 Experimental Setup

We have ran various experiments to evaluate the performance of NOVA. In these experiments, we create a multicast group by having many nodes join the NOVA DHT with a certain join rate. The number of nodes ranges from 1000 to 20000. The join rate is set to 1 node/sec, in order to evaluate the operation of NOVA under high churn circumstances. Immediately after the last node has joined the DHT (but way before the DHT manages to stabilize), we initiate the dissemination of $k$ segments. The stretch factor is typically set to 2, but we further discuss applying lower values, when churn rates are not very high, or reliability is not very important.

After the dissemination of the segments, we wait for a predefined period and then check how many segments has each node received. We present various statistics about the average percentage of segments received by the nodes, the number of partial deliveries (the number of nodes that did not receive all segments, but received enough to reconstruct the original message) and the number of failed deliveries (the number of nodes that did not receive enough segments to reconstruct the original message).

We further collect information about the per segment path length from starting node to recipients. Specifically, we present the mean value and the standard deviation of the path size. In the rest of this section we will use the terms “path length” and “hop count” interchangeably.

Figure 4.6: Comparison of the hop-count distribution of NOVA and SplitStream
4.6 Implementation and Performance Evaluation

Figure 4.7: Average hop-count of NOVA and SplitStream compared to $\log_{16} N$

length, as well as a detailed graph demonstrating the hop count distribution among the nodes. In the same graph, we present the corresponding distribution for the SplitStream system, so that the performance improvement becomes obvious.

4.6.3 Experimental Results

In this section we present the results of the experimental evaluation. Figure 4.6 presents the hop-count distribution for NOVA and SplitStream. The results where taken from a 18000-nodes multicast group. It is obvious that NOVA achieves a far better distribution. For example, over 93% of NOVA nodes receive a segment after only 4 forwarding steps, while in SplitStream 70% of the nodes require between 5 and 7 forwarding steps. Moreover, as shown also in following experiments the average hop count of NOVA is 1.1-1.3 hops lower that that of SplitStream, for group sizes ranging between 2000-20000 nodes. We also observe that NOVA has a lower maximum hop count than SplitStream (5 hops instead of 8). As the network size increases, this performance comparison becomes increasingly better for NOVA. The reason for this is that NOVA exploits Pastry’s routing tables to achieve a $[\log_{2} N]$ average path length, while SplitStream fails to provide this guarantee. Figure 4.7 presents the results about the average hop count that NOVA and SplitStream achieved under varying group sizes. The “logN” curve corresponds to the $log_{16} N$, where
Figure 4.8: Average delivery percentage per node (left Y-axis) and standard deviation (right Y-axis).

$N$ is the number of nodes in the group. The performance of NOVA is very close to the logarithmic curve, thereby confirming what we have already proven in section 4.5. On the other hand, SplitStream uses more hops than $\log_{16} N$ on average. Therefore, as the network size increases, NOVA’s performance becomes increasingly better than that of SplitStream. This performance difference is explained by the different structures that NOVA and SplitStream use. NOVA uses the routing table of Pastry, whereas SplitStream uses a number of Scribe trees to disseminate the information. While the SplitStream stripes have proven to have good performance, they can not achieve $\log_{2} N$ hop-count, because the tree they are based on is not perfectly balanced and does not have the same fan-out (16) in all nodes\(^4\).

The next experiment is concerned with reliability. We analyze the behavior of NOVA with respect to the number of segments that are delivered to each node. Figure 4.8 presents the average and the standard deviation of the distribution of values. The left Y-axis corresponds to the average percentage of segment deliveries per node, while the right Y-axis corresponds to the standard deviation of per-node segment deliveries. For example, with a group size of 8000, nodes receive on average over 99% of all segments with a standard deviation of 2. We observe that some nodes do not receive all segments. The main reason for this is the instability of the DHT. We have already explained that we have conducted the experiments under churn, in order to simulate unfavorable situations.

\(^4\)A perfectly balanced tree with fan-out 16 would achieve an average hop-count slightly lower than $\log_{16} N$ hops
4.6 Implementation and Performance Evaluation

Since the DHT can not be perfectly stabilized under churn, the routing tables have some outdated entries. As a result, some nodes may not receive a segment. However, the average percentage of segments that are delivered is over 99.5%. This, combined with the low standard deviation of the values (below 3 in all cases), guarantees that all nodes will receive the original data with high probability.

In the next experiment, we show that, even under churn, there are very few nodes that do not receive enough segments and will therefore use the recovery algorithm.

Figure 4.9 shows, for a varying group size, how many nodes received the message partially (on the left axis) and how many nodes failed to receive the message. Partial message reception by a node means that the node did not receive all segments, but received enough to reconstruct the original message. Failed reception means that the node received less than the minimum number of segments required to reconstruct the message (and will therefore have to use the recovery algorithm). For example, with 10,000 nodes about 350 nodes did not receive all segments, but only 1 of them was not able to reconstruct the message. Further, for group sizes up to 8000 nodes, no failed deliveries occur.

4.6.4 Discussion

Based on the experimental results mentioned above, we further discuss the choice of stretch factor. In our experiments, the stretch factor was set to 2. However, much lower values are possible. The stretch factor must take into consideration the desired reliability of the system, combined with the expected join/leave rate (churn). We have already shown
that all nodes will ultimately receive a necessary number of segments, so that they can reconstruct the original data (even the extremely small percentage of nodes experiencing failed deliveries will run the recovery algorithm to obtain missing segments). However, the value of the stretch factor will determine the number of nodes that will need the recovery algorithm. Our experimental evaluation has shown that, even under churn, the average percentage of segments that is delivered to each node is over 99.5%, with a standard deviation of about 2.5. It is therefore possible to decrease the value of the stretch factor dramatically and still enjoy high reliability. In particular, a system that does not expect high churn would have very consistent routing tables in the DHT nodes. In this way, the possibility that a node does not receive a segment would practically be 0, and even a stretch factor of 1 would perform very well. It is also possible that the stretch factor can change dynamically, if the join/leave rate can be estimated.
Chapter 5

Conclusions

We have presented the design, implementation and performance evaluation of BAD and NOVA, two application-level multicast infrastructures. They both aim to improve the performance of application-level multicast, but each from a different angle.

BAD’s main goal is to improve the bandwidth perceived by end users. It can be built over any DHT and it uses a novel distributed algorithm for rearranging a multicast tree that is built over a DHT, in order to improve its performance. Moreover, nodes distribute their available bandwidth to their children, using a novel algorithm that takes into consideration the bandwidth capabilities of the children. Additional novel algorithms are presented to reduce the number of messages used by BAD, so that the overhead imposed on the underlying network is minimized.

We have evaluated the performance of BAD under two basic metrics, the minimum and the average bandwidth among members of the group. We have observed that the improvement of minimum bandwidth is more dependent on the parameters of the system and therefore deviates more from its mean value than the improvement of average bandwidth. Experimentation has proved that BAD performs well, even under the most unfavorable circumstances. In particular, BAD’s performance remains at high levels, even if the underlying network is heavily congested.

Although BAD’s performance is very good, there are some issues left open. We are currently working on a forecasting algorithm that takes into consideration the number (and bandwidth class) of the brothers that a node would have, if it were to select a certain father. That way, depending on the candidate father’s bandwidth, a node can estimate the probability that the bandwidth that is offered to it will not change drastically. Based on this estimation, the node can choose to move to another father, that provides it with somewhat less bandwidth, but which is less prone to change. A threshold can be used to determine the amount of stability that nodes demand and how much this stability affects their choice. Moreover, we are in the process of evaluating the relevant performance of BAD, compared to that of non-tree approaches, hoping to develop a hybrid system that inherits the advantages of both approaches.

NOVA, on the other hand, seeks to improve latency characteristics by sending all multicast messages using a logarithmic number of hops, while achieving high reliability and load balancing. NOVA enjoys several important advantages compared to related research efforts. First, it provides guarantees on the latency observed for message segments to be delivered to any recipient node, in terms of the, on average, path length traversed by a segment en route to a destination node. Second, in order to achieve this it requires no extra state other than that required for the underlying DHT consisting of the nodes in
the multicast group. Thus, when multicast groups form overlay networks, which require one-to-one and multicast communication, the above latency benefits are achieved at no additional state (implying avoidance of state-maintenance overheads). In any case, off-the-shelf state-maintenance algorithms are used by NOVA, as provided by the underlying DHT. An interesting unique feature is that, since NOVA utilizes the DHT routing tables for message delivery, it can import all extensive research (and related optimizations) conducted to improve routing-performance. For instance, routing tables that are constructed taking into account locality properties immediately translate into benefits for the performance of multicast message delivery in NOVA.

We have proved the basic properties of NOVA concerning performance and correctness and we have implemented NOVA over the Pastry DHT and performed experimentation testing the performance of NOVA in terms of segment delivery path length and its reliability. Our results confirm NOVA’s superior performance. Finally, we have designed NOVA over Chord and DHTs based on Plaxton et al routing (as exemplified by Pastry). This shows that NOVA’s fundamental principles are applicable to a variety of DHT architectures.
Bibliography


