Group Communication Techniques in Overlay Networks

by

Knut-Helge Vik

Doctoral Dissertation submitted to
the Faculty of Mathematics and Natural Sciences
at the University of Oslo
in partial fulfillment of the requirements for
the degree of Philosophiae Doctor

December 2008

Abstract

One type of Internet services that have recently gained much attention are services that enable people around the world to communicate in real-time. Such services of real-time interaction are offered by applications most commonly referred to as distributed interactive applications. Concrete examples of distributed interactive applications are multiplayer online games, audio/video conferencing, and many virtual-reality applications linked to education, entertainment, military, etc. A time-dependent requirement generally applies to all distributed interactive applications that aim to support real-time interaction, and is usually in terms of a few hundred milliseconds. The latency requirements are manifested in terms of event-distribution, group membership management, group dynamics, etc., far exceeding the requirements of many other applications.

One general focal point in this thesis is to enable scalable group communication for managing dynamic groups of clients that interact in real-time. By doing this, we want to enable people around the world to dynamically join networks of participants and interact with them in real-time. The main contributions of the thesis are a number of investigations of a wide variety of group communication techniques. The results from the investigations form a foundation to identify the techniques that are particularly suitable for distributed interactive applications.

We investigated membership management techniques, and evaluated both centralized and distributed approaches through empirical and experimental studies on PlanetLab. We proposed 3 membership management techniques and found that a centralized membership management approach is particularly fast and consistent when there are multiple dynamic groups.

We also aimed to identify well-placed nodes in the application network that yield low pairwise latencies to groups of clients. These may, for example, be used for membership managing tasks. We evaluated 5 core-node selection algorithms through group communication simulations and experiments on PlanetLab. From these evaluations we found that there exist core-node selection algorithms that are able to find sufficiently well-placed nodes.

We considered overlay network multicast as the better option to distribute time-dependent events in groups, and found that centralized graph algorithms are suited to meet the latency requirements put on the overlay constructions and reconfigurations. We evaluated a variety of centralized overlay construction algorithms that aim to build low-latency overlay networks. Through rather comprehensive analyses we identified suitable algorithms in the investigations.

Finally, we investigated whether it is possible to obtain accurate all-to-all path latencies to

be used by the centralized graph algorithms. For this, we evaluated 2 latency estimation techniques and measured their accuracy by comparing the estimates to all-to-all ping measurements. For the evaluation, we implemented a real-world system and performed group communication experiments on PlanetLab. We found that when latency estimates are used by core-node selection algorithms and overlay construction algorithms, they are sufficiently accurate such that the graph algorithms still find solutions that are close to the real-world.

Acknowledgements

I would like to thank my supervisors Dr. Pål Halvorsen and Dr. Carsten Griwodz for hiring me as a PhD Stipendiat at the Department of Informatics, University of Oslo, September 2004. Their continuous tireless interest has contributed to making the road towards a PhD a better one. In addition, I would like to thank my previous supervisor Dr. Sirisha Medidi for all her kind words that somehow convinced Pål and Carsten to call me in for an interview.

During my Stipendiat period at the University of Oslo and Simula Research Laboratory I have met many exceptional persons that have often unwittingly motivated me to continue my work towards a PhD. It was especially the two and a half years at Simula Research Laboratory that brought the surroundings that really enabled me to work hard and remain motivated. Silly moments with childish humor and irony is important for me, and this was something that my office-mate and fellow PhD Stipendiat Andreas Petlund and myself became quite good at. At least in our own eyes.

The ND-department at Simula has many outstanding researchers that are not only bright and hardworking professionals but also people that are impossible to dislike. These researchers are without a doubt major contributors that made it so much easier for me to work hard and efficiently. They gladly participated in discussions about random things that really made my day many a hard working period.

I met many friends and fellow students when studying at Høgskolen i Ålesund and Washington State University. It has become clear to me now how important they were for me. Without friends like Alexander, Finn-Tore, Mikke, Thomas, Odd-B, Thor-Egil, Espen, Ketil, Bjørn, Amund, Gunnnar, Jan, and many many more, I may have chosen a different direction.

I would like to thank my family for their unconditional support and love during the last four years. There were periods where I was not really a pleasure to talk to but undoubtedly that is just what I needed.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The first decade of this millennium has been a time of globalization, where people take for granted the fact that we can communicate throughout the world using the Internet. But, it hasn't always been like this. In fact, as late as in the early 1990s the Internet was a rather slow network that was not widely accessible. Now that the first decade of the new millenium has almost come to an end, the Internet is generally available and is becoming a natural part of peoples lives.

The current Internet has a huge amount of different services, many that are important in regular people's everyday affairs, but also many that go on unnoticed until they disappear. A few of the current popular Internet services include online searches using Google, instant text messaging on MSN Chat, IP phone technology on Skype, social networking tools like Facebook, video streaming from various TV-channels, multiplayer online games like the recent Age of Conan, and the list goes on. To an untrained eye, it certainly looks like there are enough services to cover any need that people may have. But, it is clear that many Internet services are yet to be deployed and even discovered, and that these services are going to cover everyday needs that most people today don't realize they have. As a humorous example, figure 1.1 gives a prediction to which services Google may possibly include in the year 2084.

One type of Internet services that have recently gained much attention are services that enable people around the world to communicate in real-time.

Such services of real-time interaction are offered by applications most commonly referred to as distributed interactive applications. These applications typically support group communication functionalities, in which clients can communicate and interact with each other over large distances. However, distributed interactive applications currently have multiple unsolved issues that prevent them from easy and cheap deployment.

In the course of the thesis, we investigate a wide range of group communication techniques that are intended to enable a broader range of distributed interactive applications. And, as such, make it easier for developers to create distributed interactive applications.

Figure 1.1: What types of Internet services will be available in 2084?

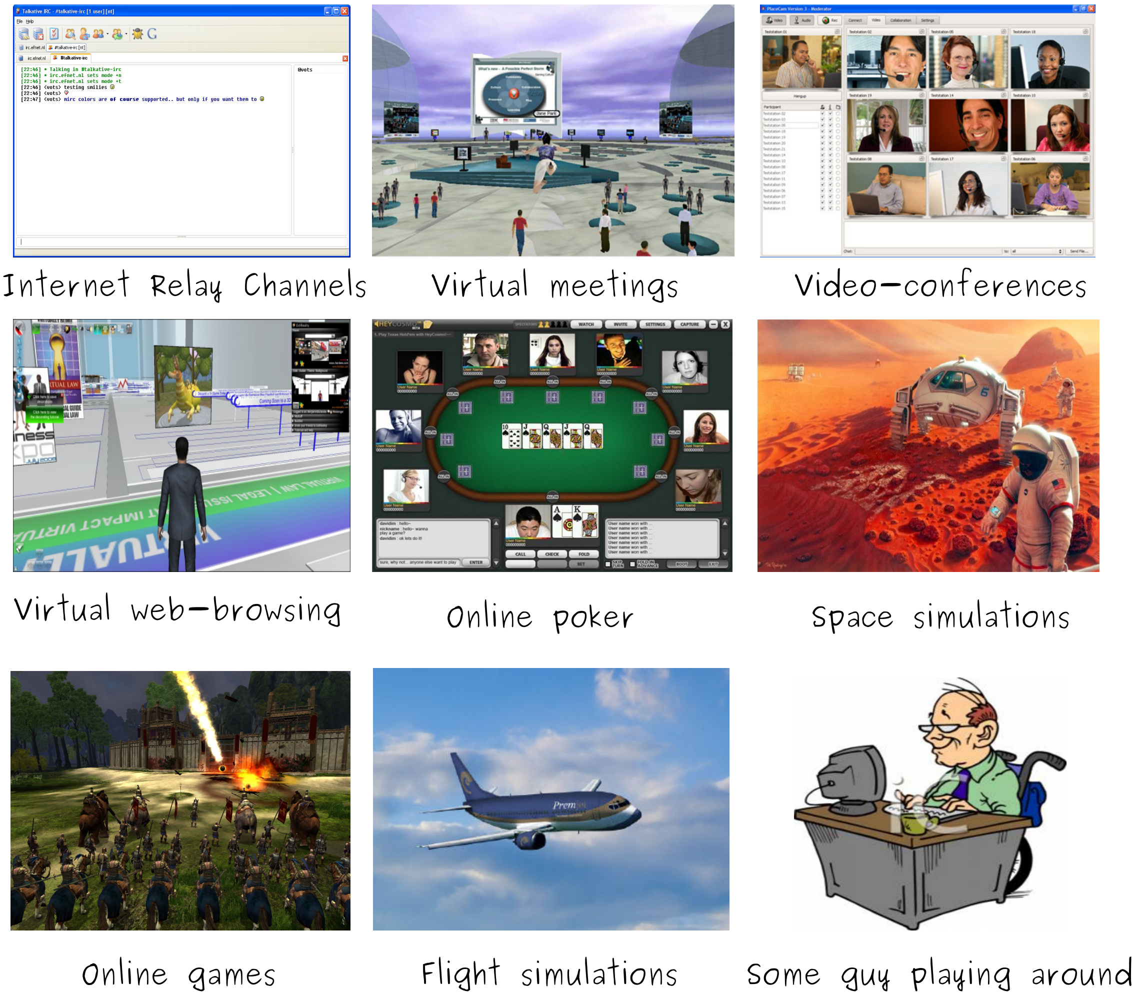
In the following sections, we introduce distributed interactive applications, and highlight their challenges, which are especially linked to achieving sufficiently low latencies between interacting parties to enable real-time interaction. We also discuss the latency and bandwidth limitations that are present in the Internet, which make it hard to support real-time interaction. These Internet limitations must be handled by advanced group communication techniques that are able to identify the opportunities that the Internet does provide. From these observations, we define 4 specific goals for the thesis, and then give a brief summary of our contributions.

1.1 Distributed interactive applications

The target applications for the work done in the thesis are distributed interactive applications.

Distributed interactivce applications aim to offer real-time interaction between multiple participants over the Internet.

Concrete examples of distributed interactive applications are multiplayer online games, audio/video conferencing, and many virtual-reality applications linked to education, entertainment, military, etc. Figure 1.2 includes screen-shots from a few distributed interactive appli-



```
Talkative IRC - #talkative-irc [1 user] [nt]
[22:46] * Talking in #talkative-irc
[22:46] * irc.efnet.nl sets mode +n
[22:46] * irc.efnet.nl sets mode +t
[22:46] <vots> testing smilies
[22:46] <vots>
[22:47] <vots> mirc colors are of course supported.. but only if you want then to
```

Figure 1.2: Some examples of interactive applications.

cations. The differences between such real-time interactive applications and other less time-dependent Internet communication, can be highlighted by comparing video-conferencing to video-streaming.

Applications that support audio/video streaming of live events typically buffer the stream at the client for a number of seconds before it is played out. This is done because it is more important that the audio/video stream runs smoothly during playback, than that the audio/video is "exactly" live and appears jittery. For video-conferencing applications it is not an option to buffer video for multiple seconds, because this makes it very hard to achieve smooth interaction when there are many active participants discussing. Rather, the audio/video streams are strictly time-dependent and must be distributed very fast among the participants and then played out live as quick as possible on each of them.

A time-dependent requirement generally applies to all distributed interactive applications that aim to support real-time interaction, and is usually in terms of a few hundred milliseconds.

1.2 Emerging real-time interaction services in the Internet

Real-time interaction yields strict latency requirements in any setting, and supporting it across the Internet is currently a challenge due to the rather high end-to-end path latencies. This is also one of the reasons why there are rather few distributed interactive applications, and such services of real-time interaction. However, we envision that real-time interaction services on the Internet are emerging services that are expected to become easier to develop and more accessible to the general public within a few years (see figure 1.3). In this context -

We characterize real-time interaction services in the Internet to be services that enable people around the world to dynamically join networks of participants, and then instantly interact with them in real-time.

One example of a real-time interaction service may be a social network of people that share common interests. Such a service exists in the format of text messaging where participants may join an Internet relay channel (IRC) to text message about some topic of interest, be it politics, sports, relationships, etc. Although IRC applications are not within the scope of the thesis, their basic functionality is desirable for distributed interactive applications. However, it is much harder to achieve this for audio, video and virtual reality.

In the Internet today, there are few distributed interactive applications that let participants freely join dynamic social networks using audio, video and virtual reality.

Existing applications that do support audio and video, for example Skype, only let participants talk in rather small non-dynamic networks of people you have to search for explicitly. Applications that include interactive virtual reality, such as online games, are centrally managed in which the participants are kept in closed groups that require subscription to access. Generally, these applications 1) have static user groups, 2) are costly to deploy, and 3) suffer in their scalability. Solving these issues is important to enable cheap deployment of high-quality distributed interactive applications that support time-dependent interactive multimedia (rich media) in the Internet.

Time-dependent rich-media data streams require sufficiently low-latency paths such that they can be delivered on time to the clients that interact.

Time-dependent rich-media streams include audio/video, 3D streams, instant text messages (chat), position updates in online games, etc. The challenge of distributing such time-dependent rich-media is linked to both latency requirements and the bandwidth they require. For example, audio and video streams require more bandwidth than streams of position updates that determine the movement of characters in online games. From these observations, it is clear that group communication techniques should be available to address both the latency and the bandwidth requirements of rich-media streams.

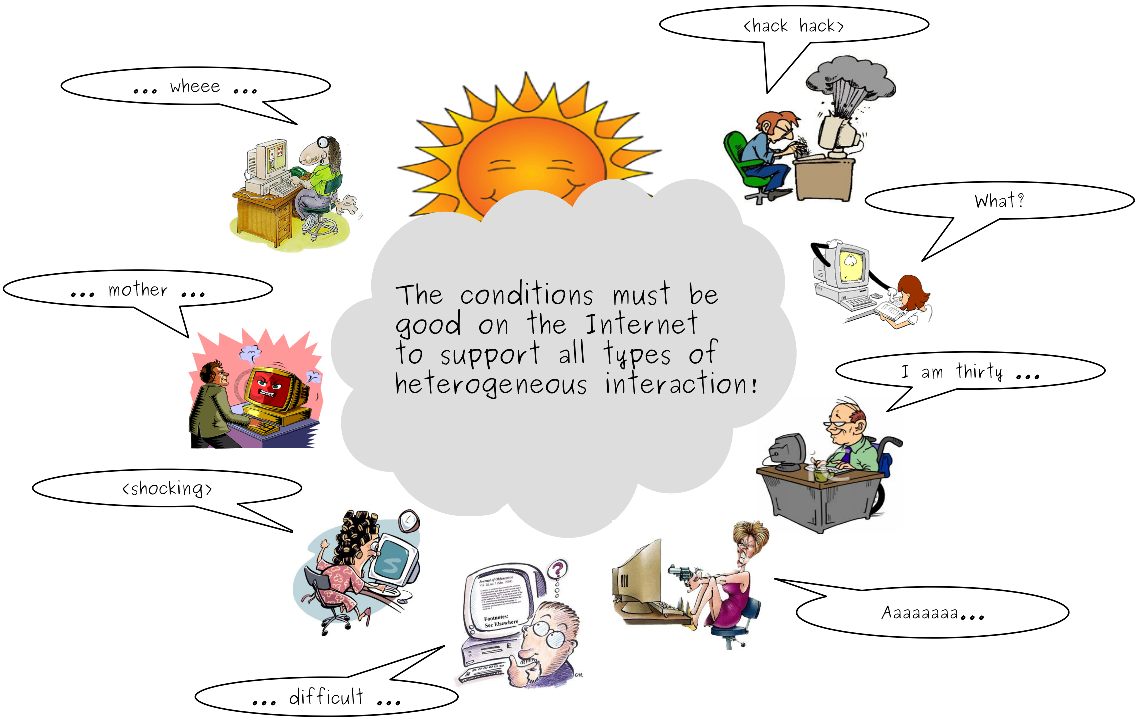


Figure 1.3: It is expected that the demand for interaction services in the Internet increases. Therefore, the demand for an Internet with low end-to-end latencies will also increase.

1.3 Achieving group communication in the Internet

The status of the current group communication techniques available for enabling distributed interactive applications for real-time interaction services is poor. One reason is that group communication across the Internet was for a long time hard to achieve because of the rather low throughput, which again was very much due to the low bandwidths that marred average clients on the Internet. Therefore, the group communication research has been delayed and is still in its infancy.

The challenges of achieving group communication in the Internet are especially related to asymmetry, heterogeneity, resource availability and latency issues in the Internet. Distributed interactive applications have strict latency requirements in order to achieve real-time interaction. These latency requirements are manifested in terms of event-distribution, group membership management, group dynamics, etc, far exceeding the requirements of many other applications. One general observation regarding real-time (live) interaction and latency is:

To achieve real-time interaction across the Internet requires sufficiently low latencies between the interacting participants.

Hence, the path latencies between the interacting parties must be bounded sufficiently (see figure 1.4). This is difficult to achieve for all participants, because they may be located throughout the world, for example, in areas where the bandwidth capacities of the Internet connections are

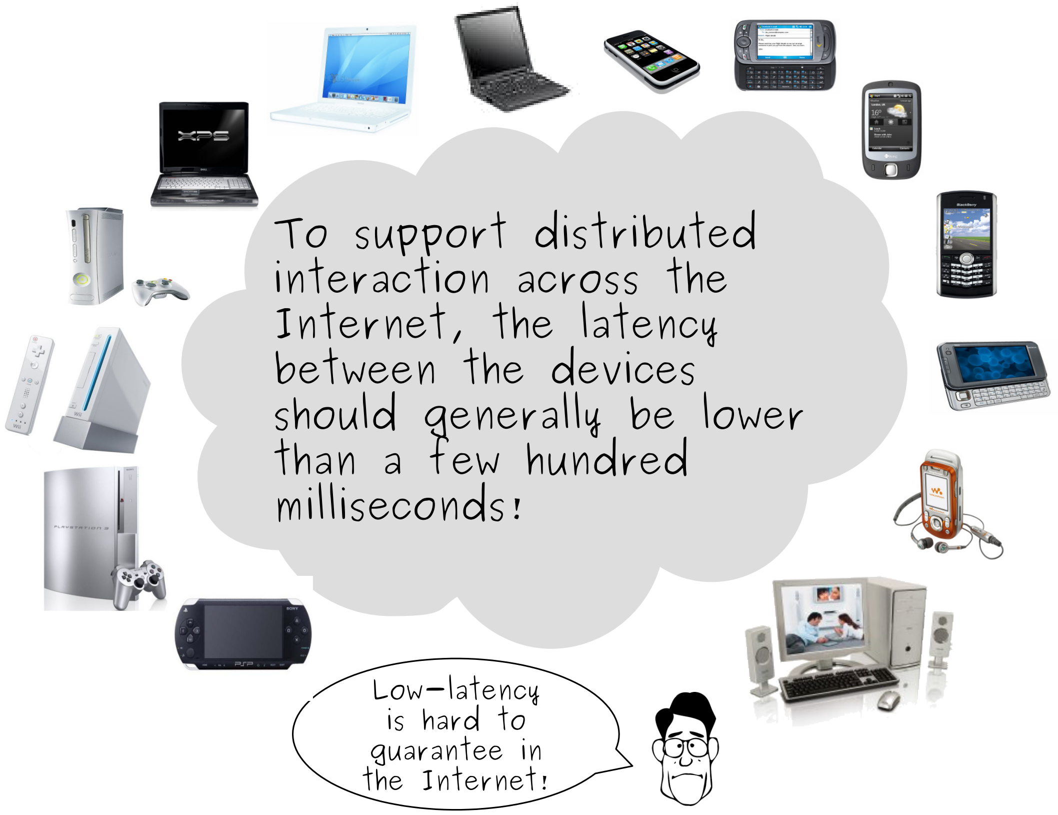

To support distributed interaction across the Internet, the latency between the devices should generally be lower than a few hundred milliseconds!
Low-latency is hard to guarantee in the Internet!

Figure 1.4: Distributed interaction yields strict latency demands that the Internet must handle.

very limited. Therefore, the most important reason for why it is difficult to achieve support for real-time interaction across the Internet is that:

Rich-media are time-dependent and often bandwidth-intensive data streams that have hard-to-meet requirements, which are difficult to support by the current Internet.

It is quite clear that we have to accept that the current Internet has limitations, which often result in too large end-to-end path latencies. However, in the course of the thesis, we shall investigate group communication techniques that try to handle these latency limitations in the best possible manner, and take advantage of the possibilities that the Internet does provide. To this date, the Internet is the only option if we want to communicate in groups throughout the world in an interactive timely manner.

1.4 The main goals of the thesis

The goals of the thesis are very much tied to investigating multiple group communication techniques and then finding the ones that are suitable for distributed interactive applications. In that respect, it is clear from the application scenarios we have described that the clients in these applications should have the possibility of joining and leaving an ongoing session of real-time interaction. Therefore, one general focal point in the thesis is to:

Enable scalable group communication for managing dynamic groups of clients that interact in real-time.

By doing this, we want to enable people around the world to dynamically join networks of participants and interact with them in real-time.

1.4.1 Observations that define the goals

An application that is to support instantaneous interaction must use a distributed system that handles the interactivity and dynamicity of clients. Generally, such systems must have basic mechanisms that support interactivity and dynamicity, such that:

When clients join and leave an ongoing session of real-time interaction, mechanisms should ensure that the service to the remaining clients is not disrupted, and the new clients should be included such that they, in a timely fashion, can interact with the clients online.

Parts of this functionality should be enabled by a membership management that handles incoming join and leave requests. Furthermore, when clients have joined and started the real-time interaction, these client-interactions must be enabled by distributing the application events across the Internet. In other words, the application events that occur on each client, and are vital for the interaction, must be distributed to all the clients that need them. Many of these application events are time-dependent, therefore they must also be distributed such that the real-time interaction is continuous (without glitches).

1.4.2 The four main goals

The previous observations are summarized to four specific goals that we aim to address in the thesis. For each of the goals, we give some motivation to why we believe the goal is important and what the benefits are of reaching the goal.

1) Identify techniques that enable an efficient and timely membership management of multiple dynamic subgroups of clients.

Generally, clients in distributed interactive applications generate events that need to be shared with others. However, in large-scale applications with hundreds or even thousands of clients, it is unlikely that all the events that occur are important for everybody. The reason is that many of the clients in such large-scale applications are not interacting. Rather, only sub-sets of the clients are interacting and need to share their events with each-other. Therefore, distributed interactive applications should support multiple dynamic subgroups because it is likely that it will make them more scalable, compared to just having a single flat group of clients, in which all clients receive every event.

Based on these observations, we want to identify a membership management that enables dynamic sub-groups to form based on incoming join and leave requests. By doing this, the fundament is laid for letting clients join and leave onging sessions of real-time interaction.

2) Identify techniques that enable a resource management to identify nodes in the (application) network that yield low pair-wise latencies to groups of clients.

Well-placed nodes that yield low latencies to groups of clients are a resource for distributed interactive applications because they can be used to execute time-dependent management tasks. When multiple dynamic sub-groups are allowed to form, these groups must be updated sufficiently fast to enable the real-time interaction to continue.

Based on these observations, we want to identify nodes in the application network that yield low latencies to the clients that are interacting to enable them to execute a membership management for sub-groups of clients, or centralized graph algorithms that create and update overlay networks for event-distribution.

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

We mentioned previously that clients in distributed interactive applications generate time-dependent events that have strict latency requirements often in terms of a few hundred milliseconds. Therefore, it is important that clients who are interacting are connected through Internet paths that yield sufficiently low latencies. Multicast is a way of achieving scalable group communication in the Internet and can be achieved both on the network layer and the application layer. Network layer IP Multicast is currently not available throughout the Internet, and has unsolved scalability problems. Therefore, we consider application layer overlay multicast as an appropriate way of distributing time-dependent events.

Based on these observations, we want to identify techniques that construct low latency overlay networks to multicast application events. The overlay construction techniques must consider the time-dependent requirements both in terms of construction time, and the pair-wise latencies in the overlay network that is configured. In a scenario with dynamically changing sub-groups, the overlay construction techniques must also be able to update overlay networks sufficiently fast based on incoming join and leave requests.

4) Identify techniques that are able to obtain accurate all-to-all Internet path latencies.

In the thesis, we consider latency as the most important metric to evaluate in order to achieve real-time interaction. By retrieveing all-to-all path latencies we aim to enable graph algorithmic techniques to find both well-placed nodes (goal number 2), and also construct overlay networks for event-distribution (goal number 3). In addition, the all-to-all path latencies must be available sufficiently such that they enable graph algorithmic techniques to quickly configure dynamically changing sub-groups of clients.

Based on these observations, we want to obtain accurate all-to-all Internet path latencies by using latency estimation techniques. The latency estimation techniques must be both accurate and make the estimates quickly available.

1.5 Thesis contributions

The four goals from section 1.4 are all addressed in the course of the thesis, and for each of them, we give rather comprehensive studies. The general focus was to compare multiple group communication techniques through simulations and experiments, and then come to a conclusion to which techniques would be better to enable group communication applications that support real-time interaction. Many of the evaluations are published in top peer reviewed conferences and journals [131, 130, 125, 127, 128, 129, 60, 17, 126]. We re-state the goals from section 1.4.2 for the sake of convenience.

1) Identify techniques that enable an efficient and timely membership management of multiple dynamic subgroups of clients.

To reach this goal, we evaluated both centralized and distributed approaches through empirical and experimental studies on PlanetLab. We proposed 3 membership management techniques that are all evaluated towards the membership change execution latency and the consistency they yield.

Among the 3 evaluated membership management approaches, we found that a centralized architecture is a fitting approach that yields the consistency desired for such managment. A centralized architecture is also able to execute a membership change request sufficiently fast, but the location of the node (central entity) that executes the membership management influences the latency in a membership change request. Therefore, we found that in a centralized architecture, the central entity should always yield low latencies to the groups it is managing.

2) Identify techniques that enable a resource management to identify nodes in the (application) network that yield low pair-wise latencies to groups of clients.

We addressed the goal by evaluating 5 core-node selection algorithms through group communication simulations and experiments on PlanetLab. Core-node selection algorithms are algorithms that aim to select nodes in a network that yield a desired property, which in our case is low pair-wise latencies to groups of clients.

We evaluated the 5 core-node selection algorithms towards how how well the algorithms could identify well-placed core-nodes in the network that yield low latencies to certain groups of clients. In addition, we tried to estimate how many such core-nodes are needed to be available for management tasks to reduce the management latencies sufficiently for all groups. We found that when core-node selection algorithms are applied, only a limited set of core-nodes is needed to sufficiently bound the management latencies [17].

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

We considered overlay network multicast as the way to distribute time-dependent events in groups, and found that centralized graph algorithms are most suited to meet the latency requirements that is put on the overlay construction and reconfigurations. Therefore, we evaluated a wide range of centralized overlay construction algorithms that aim for low-latency overlay networks. Many of the algorithms are developed by the author of the thesis [131,130,125,127,128, 129]. The algorithms we evaluated are 12 spanning-tree algorithms, 25 Steiner-tree algorithms and 12 Subgraph algorithms. In addition, we evaluated dynamic algorithms that insert and remove single nodes from overlay networks. In the dynamic tree algorithm studies, we evaluated 8 insert and 11 remove strategies, where one insert and one remove strategy forms a dynamic tree algorithm. Similarly, we evaluated 3 insert and 11 remove strategies for dynamic subgraph algorithms. Finally, we proposed 2 reconfiguration algorithms, one for trees and one for subgraphs, that take as input a dynamic overlay algorithm and an overlay construction algorithm.

All of the algorithms are evaluated in a group communication simulator we implemented and in a real-world system that we test on PlanetLab. Both the simulator and the real-world system mimic group communication in which clients join and leave groups throughout the experiment.

The general findings were that when there are dynamic groups of clients, the dynamic algorithms should be used [125]. This is because they are designed to insert and remove single nodes from an existing group's overlay network. However, these dynamic algorithms are often simplistic and may result in group overlays that have too large maximum latencies between the nodes. Therefore, we found solutions where a total reconfiguration is initiated based on an upper bound on the maximum latency using spanning or Steiner overlay algorithms [127,131,130,129]. This ensured that the overlay networks were close-to-optimal.

4) Identify techniques that are able to obtain accurate all-to-all Internet path latencies.

The goal of retrieving all-to-all path latencies is motivated by the fact that centralized graph algorithms are suitable for achieving the latency bounds of real-time interaction. The centralized graph algorithms that we investigate use path latencies to find well-placed nodes or construct overlay networks.

To reach the goal, we evaluated 2 latency estimation techniques and measured their accuracy by comparing the estimates to all-to-all ping measurements. For the evaluation, we implemented a real-world system and performed group communication experiments on PlanetLab.

We found that when latency estimates are used by core-node selection algorithms and overlay construction algorithms, they are sufficiently accurate such that the graph algorithms still find solutions that are close to the real-world [126].

1.6 Thesis outline

The rest of the thesis introduces the background and related-work and then the work that was performed in order to achieve the goals listed in section 1.4. The organization of the chapters is as follows:

Chapter 2, Distributed interactive applications: Background and Motivation, presents the background for the research and introduces the specific research goals that we address in the thesis. A general goal is to identify network-related group communication techniques that enable real-time interaction between multiple participants across the Internet.

Chapter 3, Group communication: State-of-the-art and related work, presents a summmary of the related work we found in related research areas, and we identify that our research goals have not been properly addressed in existing literature. Some of the research areas are: Overlay multicast, core-node selection algorithms, overlay construction algorithms and latency estimation techniques.

Chapter 4, Overlay network design: Problems in graph theory, introduces graph theoretical performance metrics and network design problems found in graph theory. In addition, we provide some background to the algorithmic foundations that are used when graph theoretical problems are addressed.

Chapter 5, Distributed interactive system: Group management techniques, presents how we attempt to address the research goals that are identified in chapter 2. We identify what techniques should be researched in the three main research areas: resource management, membership management, overlay network management and network information managment.

Chapter 6, Characteristics of overlay networks: Latency estimation techniques, introduces and evaluates two latency estimation techniques, Netvigator and Vivaldi, in terms of their ability of retrieving accurate all-to-all path latencies. The latency estimates are later on evaluated in terms of their usability for centralized graph algorithms (chapter 7 and 15).

Chapter 7, Managers in overlay networks: Core-node selection algorithms, introduces and evaluates 5 of core-node selection algorithms, in terms of their ability of finding well-placed core-nodes that yield low pair-wise latencies to groups of clients. In addition, we evaluate how the latency estimates from chapter 6 can be applied to the evaluated (centralized) core-node selection algorithms.

Chapter 8, Group specific enhancements: Graph manipulation algorithms, introduces and evaluates a number of graph manipulation algorithms whose goal is to manipulate a group's complete graph such that it enables an overlay construction algorithm to execute fast and build desirable overlay networks.

Chapter 9, Overlay construction techniques: Spanning-tree algorithms, introduces and evaluates 13 spanning-tree algorithms with the goal of identifying those that yield low latency spanning-trees. More specifically, the goal is to find algorithms that construct spanning-trees with a low diameter, within a resonable time that do not add unreasonable stress to nodes in the tree.

Chapter 10, Overlay construction techniques: Steiner-tree algorithms, introduces and evaluates 25 Steiner-tree algorithms and compares their performance. The focal point is to identify those algorithms that within a reasonable time construct Steiner-trees of a low diameter that do not add unreasonable stress to nodes in the tree.

Chapter 11, Overlay construction techniques: Connected subgraph algorithms, introduces and evalutes 13 subgraph construction algorithms that construct connected subgraphs. We evaluate both spanning-subgraph and Steiner-subgraph algorithms, and compare their performance in terms of their ability to construct subgraphs of low pair-wise latencies within a reasonable time, that have controlled stress-levels on the nodes.

Chapter 12, Overlay construction techniques: Dynamic tree algorithms, introduces a range of dynamic tree algorithms that are able to insert and remove nodes from existing trees. Most of the dynamic tree algorithms are able to include Steiner-points to the trees, which yield lower diameter trees. The evaluation focuses on identifying fast dynamic tree algorithms that are able to maintain trees of a consistently low diameter.

Chapter 13, Overlay construction techniques: Dynamic subgraph algorithms, introduces a range of dynamic subgraph algorithms that also insert and remove nodes, but from connected subgraphs rather than trees. The evaluations are similar to dynamic tree algorithms, and aim to identify fast dynamic subgraph algorithms that are able to maintain subgraphs of a consistently low diameter, but also lower pair-wise latencies.

Chapter 14, Overlay construction techniques: Combining overlay construction algorithms, introduces 2 reconfiguration algorithms, one for tree algorithms and one for subgraph algorithms. The focus is to achieve the close-to-optimal overlays from Steiner-tree or Steiner-subgraph algorithms, but use the quickness of dynamic tree and subgraph algorithms as often as possible.

Chapter 15, Group communication experiments: Overlay construction algorithms, finalizes the evaluation of the latency estimates from chapter 6 when they are applied to various overlay construction algorithms, including spanning-tree and dynamic-tree algorithms. The evaluation measures the penalty of applying latency estimates that may be in-accurate, to centralized overlay construction algorithms.

Chapter 16, Distributed interactive application scenarios: Applying the research, gives discussions to how the research conducted in the thesis can be applied to different application scenarios. Specifically, we give examples to how developers of distributed interactive applications can approach basic design issues by using the techniques we identified.

Chapter 17, Conclusions and future work, concludes the thesis with a summary of the main points and a review of the contributions of the thesis. We also provide a critical assessment of the research where we discuss certain limitations in the experiments and how they may be addressed. Finally, we identify sources for future work and give some final remarks to conclude the thesis.

Chapter 2

Distributed interactive applications: Background and motivation

The background and motivation chapter introduces the target applications for the work done in the thesis: distributed interactive applications. Their functionality can be summarized such:

Distributed interactive applications aim to offer real-time interaction between multiple participants over the Internet.

Concrete examples of distributed interactive applications are multiplayer online games, audio/video conferences, and many virtual-reality applications linked to education, entertainment, etc. Recently, they have become very popular and increased in size and complexity, in particular, multiplayer on-line games. However, many research challenges remain unsolved.

In the thesis, we discuss network-related group communication challenges, especially linked to handling the rather large end-to-end latencies in the current Internet. One general observation regarding real-time (live) interaction and latency is:

To achieve real-time interaction across the Internet requires sufficiently low latencies between the interacting participants.

Therefore, the path latencies between the interacting parties must be bounded sufficiently. Moreover, we discuss how distributed interactive applications can be made more scalable than today. The scalability problems are especially linked to the requirements of low-latency and bandwidth demanding interactive multimedia (rich-media) streams. Due to these scalability problems, real-time interaction between more than a dozen participants across the Internet often requires specialized equipment to work.

The rest of the chapter is organized in the following manner. Section 2.1 introduces a few of the current types of distributed interactive applications, how they are applied, and some of their benefits. Section 2.2 introduces the application massively multiplayer online games, with a wide range of their specific research challenges and open issues. Section 2.3 generalizes the requirements of distributed interactive applications, and has initial discussions on how they can be achieved. Section 2.4 discusses research approaches to study techniques and algorithms for use on the Internet. In particular, how to identify techniques that are suitable for distributed interactive applications. Section 2.5 discusses specific system design issues and states the goals that we want to achieve by the work done in the thesis. Section 2.6 presents the methods we used to evaluate and identify group communication techniques. Finally, section 2.7 concludes the background chapter by summarizing the main observations and the goals we address.

2.1 Examples of distributed interactive applications

As mentioned, a few of the more well-known examples of distributed interactive applications are virtual-reality applications linked to education, entertainment, etc, multiplayer online games, and finally audio/video conferences.

In *virtual-reality applications*, a participant controls an avatar that interacts with a virtual world (hence the name, virtual-reality). Virtual-reality applications and multiplayer online games are very similar, but virtual-reality applications are recoginzed by the fact that they aim to simulate a real-world situation through a virtual-reality. Examples are combat-, flight- and boat-simulators, virtual museums, shopping malls etc. It is the game companies and the military that are the driving forces for improving virtual-reality applications. For example, it is estimated that the US army spend millions of dollars to develop realistic virtual-reality simulators [141].

Multiplayer online games have received the most attention recently, where the most common game-categories are first-person shooter games, role playing games and real-time strategy games. These game categories are introduced in more details in section 2.2.1. Large-scale multiplayer online games are often referred to as massively multiplayer online games (MMOGs), because they allow thousands of users to interact concurrently in a persistent virtual environment. Today, MMOGs are the largest and most complex distributed interactive applications. They often take several years to develop, and game-companies spend millions of dollars during the development. One recent example is Age of Conan, which was created by Funcom [56] and released in May 2008. Funcom started the development in 2004, and it is estimated that they spent more than 40 million dollars to develop it. Section 2.2 discusses MMOGs further.

Audio/video conference applications differ from the virtual-reality and games applications, in that they are rather used to set up real-world meetings between participants that are geographically separated. For example, many multinational companies use video conference systems to avoid travel expenses and increase the availability of people. Currently, these audio/video conference systems are not readily available for the general public, but rather need additional equipment to work. Tandberg is a multinational company that specializes in providing videoconferencing systems and services [121].

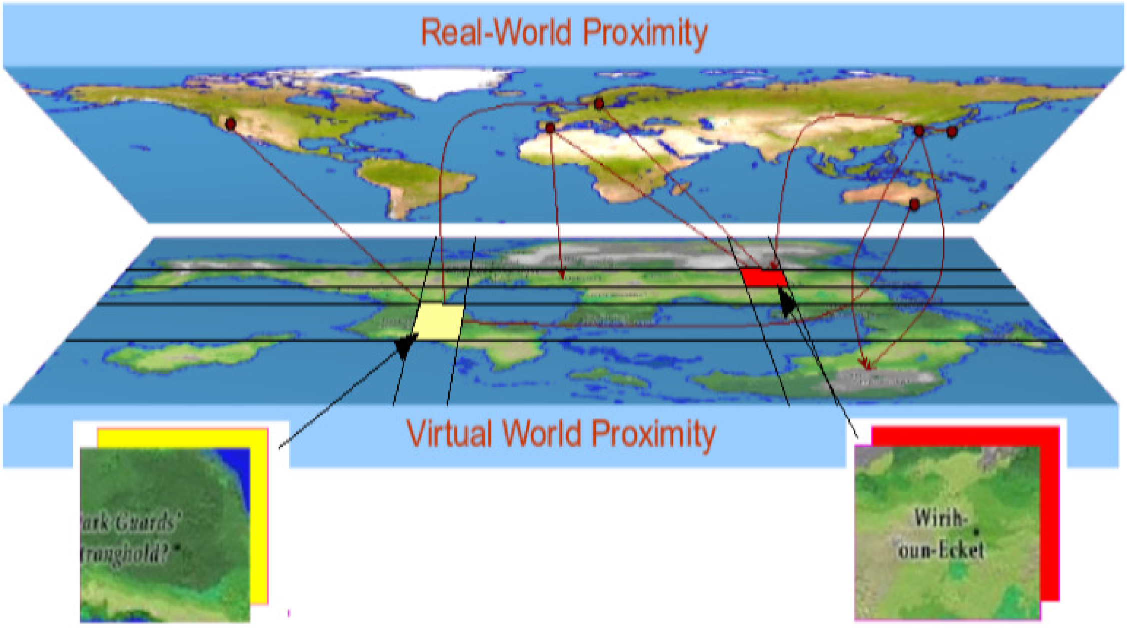


Figure 2.1: The real-world proximity of the clients is different from the virtual-world proximity of the avatars they are controlling.

From these example applications, we delve into some of the challenges and properties of MMOGs. We present further details regarding MMOG game types, their communication architectures, and their issues.

2.2 Scenario: Massively multiplayer online games

MMOG is a game genre that is typically played in a persistent game world. This game-world is almost exclusively a 3-dimensional virtual-world where players control avatars, where an avatar is a fictional in-game character.

Due to the size and complexity of MMOGs, they yield strict requirements, in terms of low-latency, consistency, etc. Therefore, they are important case studies and the current standard to what is achievable by distributed interactive applications. Figure 2.1 illustrates how the real-world proximity of the clients differ in comparison to the virtual-world proximity of the avatars they are controlling. It is this physical separation and virtual closeness that yield great challenges for MMOGs that support such distributed interaction.

2.2.1 Game types and their properties

As mentioned previously, the most popular multiplayer online game types are role-playing games, first-person shooter games and real-time strategy games. Figure 2.2 has some screen-

Figure 2.2: Screenshots from FPS, RTS and MMOGs (RPGs) games.

shots from these game types.

- ***Role-playing games (RPG)*** are often open-ended and based on each player acquiring the "role" of one in-game character (or team), and in the process gain experience, power and possessions through trade and combat. Poupular RPG games are World of Warcraft and Age of Conan [54]. Among RPG games we find the most successful MMOG games in the history of multiplayer online games. It is these RGP games that today are referred to as MMOGs, because they allow thousands of users to interact concurrently.

- ***Real-time strategy games (RTS)*** have a gameplay that often follow a general pattern to i) build up your base and forces, ii) acquire more resources, iii) attack the enemy, and attempt to deprive him of resources and destroy his infrastructure. The gameplay in RTS games progresses in "real-time", that is, it is continuous. Popular RTS games include Command and Conquer, Starcraft and Warcraft [53].

- ***First-person shooter games (FPS)*** involve high-speed combat situations with a significant amount of simultaneous application events. In many FPS games, players either operate by themselves or in teams that attempt to wipe each-other out. Popular examples of FPS games are Doom, Quake and Unreal [51].

The network requirements of FPS, RTS and RPG games vary because of their characteristics. FPS games require fast distribution of events, such that the perceived quality does not

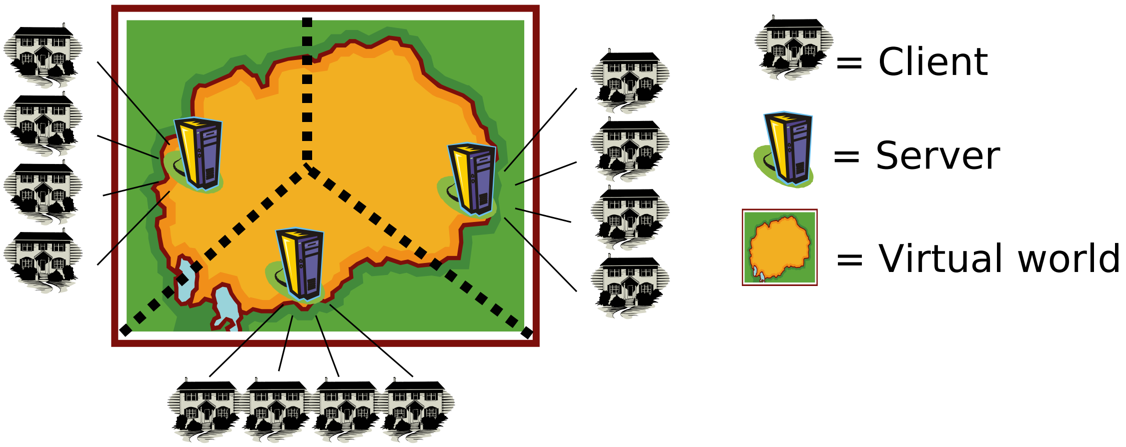


Figure 2.3: Most games today employ a rather simple client/server architecture.

suffer. This is due to many high-speed combat situations in the gameplay. RTS games do not have such stringent latency requirements, but rather need stronger synchronization techniques to handle inconsistencies in the distributed game state. RPGs often have a gameplay that is a mixture between combat situations and slower strategy sequences. Therefore, RPGs yield network requirements that vary depending on the current gameplay.

2.2.2 Basic requirements of user perception and the area-of-interest

MMOGs have many special properties and requirements due to their size and complexity. For example, players in the game should be able to move around, observe other players and interact with them, seemingly as if they were physically next to each other. Since there are so many players interacting in a large environment, it is especially important to distribute the game events efficiently. One important reason is to ensure that the perceived game quality is good for all players regardless of the underlying system capacity, geographical distance, etc.

Most current MMOGs have few, if any, mechanisms to optimize event distribution. Centralized architectures are typical, where the entire game state is stored on a central server (see figure 2.3). To improve these gaming scenarios, it is possible to use application layer multicast and group communication algorithms to enable efficient event distribution. Furthermore, it is also possible to distribute partial game state using proxy technology to achieve better scalability and reduce the latency.

Giving all the players a perfectly consistent view of the virtual world, at all times, is close to impossible [99], due to relatively large end-to-end latencies in the Internet. Rather, the perceived game quality should be the focus. Improving the perceived game quality in MMOGs is tightly linked to reducing the event distribution latency, in addition to making better use of available resources in general.

For example, many players in MMOGs are unaware of each other, because they are virtually far apart or their views are blocked by obstacles. Thus, players do not need to receive all transient events, for example, position updates of other players that are currently out of sight. Restrictions in player awareness are handled by area-of-interest management [5], where one typical area-of-interest is the field of vision of a player. Here, the distance between the players is an important factor. Players far apart, but still within the field of vision, may not need such a consistent view of one another, and the level of detail may be reduced. In addition, there might be other events outside the field of vision, such as sounds from objects (enemies), that are important for a player's reactions.

Area of interest management enables virtual area-of-interest regions within the MMOG virtual game world. Such virtual regions provide the opportunity to organize players into groups. Group communication should be applied to distribute the events within the group efficiently, that is, using the physical and virtual location of the players. Furthermore, having divided the game into virtual area-of-interest regions it is possible to achieve partial game state distribution. A distributed architecture should be used to exploit these observations.

2.2.3 Basic architectures for managing the application state

As mentioned, most MMOGs use a client/server architecture where every packet flows via a centralized server (cluster). Such a client/server model makes it easy to manage the global game state, and to prevent cheaters. In addition, a centralized architecture makes it easy for the game-companies to charge fees for playing their MMOG, enabling a business model. The drawbacks are that the server is a potential bottleneck, both in terms of computing and bandwidth capacity, and that the latency heavily depends on the physical distance from each individual client to the server. Proxy technology and peer-to-peer technology are distributed options.

Proxy technology has an infrastructure consisting of a centralized server and a set of distributed proxy servers. The proxies are usually physically closer to the clients, and may, for example, hold partial game state copies of the virtual game regions needed by the clients connected to it [17]. The proxies can be organized hierarchically, each responsible for a fixed set of clients based on location, or, in a proxy pool fashion, where clients connect to the proxies that are best suited in a given situation.

A flat peer-to-peer architecture distributes the game state among peers. It has no central server, which makes it very hard to administrate the game state such that it is consistent. Currently, there are no MMOGs that solely use such a peer-to-peer architecture because it is harder to base a business model on it. It is also possible to apply a peer-to-peer architecture where the game-state is stored centrally at a few selected peers, while applying both peer-to-peer and client/server communication. Such a mix of client/server and peer-to-peer communication styles can be applicable to MMOGs, because it is likely that it enables a business model.

We observe that the advantage of distributed architectures is that they distribute the load among multiple network nodes. This distributed architecture may therefore be used to increase the scalability of distributed interactive applications, especially when bandwidth intensive multimedia audio/video streams are applied.

2.2.4 Traffic types and their properties

MMOGs may have multiple traffic types, such as, several kinds of events, video streams and instant text messages (chat), all having different characteristics. However, the majority of the traffic flows consist of small packets carrying event information like a position update, where latency is the main issue. In addition, some events are more critical for correct gameplay than others. A level of consistency should be associated with an event to help distinguish and distribute them employing different distribution models.

The distinct properties of the traffic types should be considered when selecting the architecture and communication model. For example, position updates triggered by a player moving, are typical events that require a varying degree of consistency, and will benefit from a distributed architecture like proxy technology. Such events typically require limited distribution within player groups in virtual regions. Events that do not affect the game state and have no particular flow requirements, such as chat messages, could use peer-to-peer communication because it achieves the shortest average latency when events are small. On the other hand, an event type with high consistency requirements affecting a vital part of the game state, that is, changing a global condition, should typically use client to server-side communication. This enables a server/proxy to do necessary checks, for example, to prevent cheats and inconsistencies, before the event is put into action.

2.2.5 Game state distribution using proxy technology

In MMOGs, there are potentially hundreds of simultaneous events, and sending every event to a central server is not very efficient. Proxy technology and area-of-interest management are options to optimize the event distribution in MMOGs. Area-of-interest management enables the formation of dynamic client groups, which again provides an opportunity to distribute partial game state. Proxy technology may be an excellent platform for distributing partial game state closer to the physical location of the players. It also provides load distribution and reduces the average latency. Proxies can also save network resources by aggregating flows within the back-bone of the proxies. In addition, distribution of partial game state using proxies can be used to increase the MMOG scalability, but it requires consistency mechanisms and more advanced cheat prevention. In a proxy technology approach, partial game state may be copied to the proxies. However, the approach differs depending on the proxy model: proxy pool or hierarchical [128].

A proxy pool partitions the load by uniquely assigning parts of the *virtual world* (virtual game regions) to single proxies. A proxy has complete control over the virtual game regions it is assigned. A proxy pool increases the importance of the physical location of the proxy, because each client is connected to a single proxy, and clients can be located throughout the world. The physical location of the proxy and clients within a game region must be taken into account before assigning a copy of the partial game state to a proxy.

In a hierarchical proxy approach the load is partitioned by uniquely assigning parts of the *physical world* to single proxies. For example, clients in Europe are assigned to one proxy, clients in North-America to another, etc. A proxy acts as a distributed server for all of its connected clients, and also holds a copy of the virtual regions that its clients are currently playing in. How many virtual regions a proxy controls depend on the virtual locations of the clients connected to it. Generally, a hierarchical proxy approach requires extra consistency mechanisms, compared to a proxy pool, because there may be multiple copies of the same region among the proxies.

In the following sections, we further investigate the requirements that distributed interactive applications yield, and refine the scenario of MMOGs to apply generally for all distributed interactive applications.

2.3 Problem area of distributed interactive applications

Distributed interactive applications belong to a relatively new application area that have multiple system design issues that require discussion, and open research questions that need answers. In the following we highlight some application characteristics and the challenges they pose. In addition, we discuss some research areas that are important for the further enhancement of distributed interactive applications.

It is apparent from the interactive application scenario described in the previous sections that the clients in these applications should have the possibility of joining and leaving an ongoing session of real-time interaction. Therefore, one focal point is to:

Enable people around the world to dynamically join networks of participants and interact with them in real-time.

From this general and basic focal point we now identify more specific requirements and then give some background and motivation for these.

2.3.1 Instant interaction in groups

It is a major challenge to enable people around the world to dynamically join networks of participants, and then instantly interact with them in real-time. An application that is to support such instantaneous interaction must use a distributed system that handles the interactivity and dynamicity of clients. Currently, such systems do not work out of the box, but rather are costly to deploy and suffer in their scalability. As mentioned, pure client/server architectures are typical where the server needs to be scaled very much for the application to work, which is the case for every current massively multiplayer online game. Generally, such systems must have basic mechanisms that support interactivity and dynamicity, such that:

When clients join and leave an ongoing session of real-time interaction, mechanisms should ensure that the service to the remaining clients is not disrupted, and the new clients should be included such that they, in a timely fashion, can interact with the clients online.

These straight-forward requirements of interactivity and dynamicity together form great challenges that must be enabled by basic mechanisms and then adapted to a specific system's design. The interactivity poses requirements to the latency, and the dynamics pose requirements to the configuration of the event-distribution paths. Together, the interactivity and dynamics should be handled by a system that supports configuration of low-latency networks for event-distribution. In other words, the clients should be within a latency bound to each other in the member network. The system support for interactivity and dynamics must be enabled by basic mechanisms for network configurations and management. These mechanisms can be summarized into five basic requirements:

- ***Join and leave groups of clients:*** Joining and removing clients from distributed interactive applications require timely mechanisms that execute the requests.

- ***Application event distribution:*** Application events that occur on each client, and are vital for the interaction, must be distributed to all the clients that need them.

- ***Time-dependent application events:*** A time-dependent application event must be distributed in a timely manner, such that the interaction can continue in real-time.

- ***Low-latency requirements:*** Time-dependent application events have low-latency requirements to deliver them on time to all parties.

- ***Internet resources:*** Scarce Internet resources calls for an event-distribution that limits the resource consumption.

The following sections discuss each of these five requirements.

2.3.2 Join and leave groups of clients

Distributed interactive applications should have the functionality of joining clients to an ongoing session of real-time interaction, and also removing clients without disrupting the interaction. Parts of this functionality should be enabled by a membership management that handle incoming join and leave requests. The requirements to the membership management in distributed interactive applications is likely to vary depending on the number of client groups and their dynamics (client churn).

Managing one group of clients, where all clients receive the same events, is less complicated. In this flat group situation it suffices to form low-latency event-distribution paths in which all clients are reachable. In this case, a membership management system is enabled by distributed mechanisms for handling client churn [57].

The draw-back of a flat group is that each client in the network receives every event, even though a client may not be interested in large portions of these events. This consumes unnecessary link bandwidth in the client network. The membership management should therefore be enhanced such that it is able to divide the clients into subgroups, where each group has its own low-latency network for the events that they are interested in. When multiple dynamic subgroups are used to distribute events in an application, the approach poses membership management challenges related to how the groups are updated.

The membership management should be enhanced to include mechanisms that can search for and elect nodes to administrate clients that join and leave subgroups. Membership updates are achievable in a dynamic scenario, when a limited set of nodes handles the membership management. Enabling distributed interactive applications with multiple subgroups of clients require techniques for resilient and timely membership management.

2.3.3 Techniques for distributing application events

Due to the group communication features of distributed interactive applications, there is a need for distributing application events efficiently within client groups. Multicast is a mechanism that is designed to enable cost efficient and timely group communication. Enabling such cost efficient and timely group communication over large areas is important if distributed interactive applications are to become scalable and cheaper to deploy.

One implementation of multicast is IP Multicast. It can be used for distributed interactive applications, but it is not fully deployed in the Internet, and lacks features like address filtering and group membership control. The alternative is application layer multicast. It adds group membership control, and makes it easy to support high level functionalities. Compared to IP Multicast, application layer multicast is necessarily less efficient in terms of latency but is easier to deploy. Application layer multicast uses an overlay network of the clients and distributes messages much the same way as in IP Multicast, but on the application layer.

2.3.4 Time-dependent application events

Distributed interactive applications generate many time-dependent application events. One scalable approach to distribute such events, is through overlay networks using application layer multicast. Due to the time-dependent events, the overlay networks need to be constructed such that events can be efficiently distributed among groups of clients. The algorithms that construct the overlay networks should also be able to interpret specific event requirements.

Events may differ in terms of latency and consistency requirements; urgency and importance, where urgent events require low-latency delivery, and important events require delivery guarantees. Urgent events may benefit from a shallow low-latency overlay network, while less urgent events allow overlay optimization to save network cost, for example, to reduce the bandwidth consumption. The importance of an event is related to how critical it is for the participants to receive an event. Furthermore, the groups that are created may vary in life expectancy and group membership. It might be the case that a group is transient, but, in the other extreme, some groups may last for a very long time. Thus, distributed interactive applications need one or more algorithms supporting such requirements, e.g., provide overlays with resource efficient routes, optimized for single- or multiple sources and handle fast dynamic overlay updates.

2.3.5 Low-latency requirements

The latency requirements in distributed interactive applications are generally very strict compared to most other applications. One application-type with less strict latency requirements are applications that support audio/video streaming of live events. They typically buffer the stream for thirty seconds (or more) because it is more important that the audio/video stream runs smoothly, than if the audio/video is "exactly" live and appears jittery. For a video-conference application, on the other hand, it is not an option to buffer video for multiple seconds because this makes it very hard to achieve smooth interaction when there are many active participants discussing. Rather, the audio/video streams are strictly time-dependent and must be distributed among the participants through low-latency paths. This time-dependent requirement generally applies to all distributed interactive applications.

Obtaining the exact latency requirements in distributed interactive applications is a very hard problem, and has been generalized to obtaining approximate latency bounds based on user satisfaction. For example, in audio/video conferencing and voice over IP (VoIP) with real-time delivery of voice data, users start to become dissatisfied when the latency exceeds 150-200 milliseconds, although 400 milliseconds is acceptable in most situations [70]. The latency requirements of game traffic [27] were measured to be approximately 100 milliseconds for first-person shooter games, 500 milliseconds for role-playing games and 1000 milliseconds for real-time strategy games. Virtual-reality applications have latency requirements that fall into one or more multiplayer online game categories [27].

2.3.6 Resource management of data flows

It is envisioned that distributed interactive applications will support multiple traffic types, such as, several kinds of events, audio/video or 3D streams and instant text messages (chat). Many of these traffic types are bandwidth intensive data streams, therefore, it is vital that there are techniques enabling efficient dynamic resource management which is scalable.

Traffic flows in interactive virtual-reality applications may consist of small packets carrying event information like an avatar's position update, where latency is the main issue [27]. However, the same applications may support audio and video, and these traffic flows have high bandwidth and low-latency requirements, making it vital to apply efficient transcoding to compress the multimedia streams that require lots of resources. It is also important to enable efficient resource management by applying intelligent dynamic mechanisms that identify participants with high computational power. Current solutions for resource management of bandwidth-intensive environments are non-dynamic and rely on servers provided by a third party.

When the number of clients connected to a distributed interactive application increases, it is important for scalability reasons to apply some interest management to intelligently distribute application data to the clients that need it. The interest management in distributed interactive applications should adopt resource optimization techniques that reduces the bandwidth consumption both on the client and server side.

2.4 Research approaches and Internet characteristics

It is valuable at the start of a project to state precise research goals. However, there are also many important questions related to how the research goals are met, that is:

Which research approach should be used to find out what the better techniques are?

In the area of distributed interactive applications, it is natural to try to identify techniques through studies that take the current Internet as the starting point. And in the following sections, we introduce well-known research approaches for studying techniques or algorithms that are meant to be applied on the Internet, and explain their positive and negative sides. It is typical that such algorithms are evaluated with the basis in a few very basic Internet metrics. Therefore, the link latency, link bandwidth, and throughput metrics are also introduced. Then, we introduce the BRITE topology generator and a Zipfian distribution model for group dynamics. These are applied when simulation studies of distributed interactive applications are performed later in the thesis.

2.4.1 Studying algorithms for the Internet

Studying the main network characteristics in the Internet is important in order to improve the performance of Internet protocols and applications. Modeling and analysis of the Internet is a fundamental problem, and is attracting the attention of researchers and businesses world-wide. There are three main approaches to studying the Internet characteristics and effects, which are all important to improve the performance of Internet protocols and applications.

- ***Theoretical*** studies of algorithms often based in graph theory.
- ***Simulation*** studies of algorithms run on single computers.
- ***Experimental*** studies of algorithms in real networks.

In the theoretical approach it is likely that simplifying assumptions are made, and the Internet is studied through graph theory, mathemathics, etc. For these reasons, it is hard to capture the real-world cumulative or sporadic effects that occur in the Internet. Theoretical studies are often used for protocol and algorithmic verification, for example, proving that an algorithm never dead-locks, theoretical performance evaluations, etc.

The simulation-based approaches simulate Internet traffic on top of an Internet topology. The Internet-traffic is often generated using a discrete event simulator and the Internet topology is, for example, an undirected graph with vertices, and edges that have link latencies associated. Such simulation studies are cheap and they accelerate the studies, because they are typically performed on one machine. However, simulations of algorithms still cannot guarantee that the real-world properties are captured. For example, for mobile ad-hoc networks it is a hard problem to simulate the wireless-signals such that they act as the real thing.

Conducting real-world experiments in the Internet enhances the validity of the results. It is through Internet experimentation that algorithms and protocols are verified for performance. However, one problem is that Internet experiments require equipment, which in turn costs large sums of money. Mainly for this reason, *PlanetLab* was established as an alternative for research institutions around the world to conduct experiments across the Internet. PlanetLab has about 400 computers spread around the globe that are inter-connected through the Internet. Users are allowed to install their own software and conduct experiments using as many of the PlanetLab nodes they require. However, regardless of PlanetLab, there are certain problems that are almost impossible to study through real-world experiments, due to their immense complexity. For example, it is a hard problem to study large-scale MMOG games, their group dynamics and multicast algorithms, through real-world experiments. This is mainly because they have thousands of concurrent players in a highly complex virtual world.

All of these approaches study Internet characteristics in some form, and three of the most important and most studied Internet metrics are link *latency*, *bandwidth* and *throughput*. The following sections introduce these metrics and how they are measured in the Internet.

2.4.2 Link latency in the Internet

When data is transferred across the Internet it is delayed due to the complex infrastructure of the Internet. An example of a route is typically from the wireless local area network (LAN) at a client, through Internet name-servers, gateways, routers, many types of physical links, and then perhaps through the wireless LAN at the destination. The latency in this route is determined by the time it takes from the initial data is sent until it is received. Latency is an additive metric, and such metrics are measurable by tools that only need to be executed at the end-points, and not at every single hop across the Internet.

The latency in data-transfers across the Internet has been studied for many years, and many metrics are derived from such network latencies. For example, the end-to-end latency is the time it takes to transfer data from one end-point to the other end-point. What the end-to-end latency is, depends on how the end-point is defined. Therefore, it is typical to use round trip time, and one-way transmit time:

- The ***round-trip time (RTT)*** is the time it takes to transfer data from the source to the destination and then back again. The RTT is often used in the study of reliable transfer protocols, where the most typical is the transmission control protocol (TCP).
- The ***one-way transmit time (OWT)*** is the time it takes to transfer data from the source to the destination. The OWT is often used when studying unreliable transfer protocols, for example, the user datagram protocol (UDP).

There are readily available tools that measure a link's RTT, for example, ping and traceroute [74]. Tools for measuring OWT are also available, but they require access to both end-points [70].

2.4.3 Link bandwidth in the Internet

A fundamental property of an Internet connection is the bandwidth, which is a measure for the amount of data that a link is able to send in a time-slot, for example, bits-per-second (bps). Bandwidth is referred to as a min-max or concave metric [138], because the bandwidth of a given Internet route is determined by the network-hop that has the lowest bandwidth. The link bandwidth is essentially calculated in two different manners, the bottleneck bandwidth and the available bandwidth:

- The ***bottleneck bandwidth*** of an Internet connection is the maximum number of bits-per-second that the path can digest from source to destination, when there is no other traffic in the path. The bottleneck bandwidth is a static measure and assumes perfect conditions along the Internet route.

- The ***available bandwidth*** is the maximum rate that an Internet route can provide to a flow without reducing the rate of the cross-traffic. The cross-traffic on a route is traffic that originates from other sources and uses the available bandwidth on the links. The available bandwidth is dynamically changing depending on the amount of cross-traffic, and is important because it gives an image of the status of an Internet route.

There are many tools for measuring the bandwidth, and a few of them include one-packet technique, packet pair technique, and packet train technique [48].

2.4.4 Throughput in the Internet

The throughput is the average rate of successful message delivery over a communcation channel. On an Internet path, the throughput is usually meausured in terms of the number of bits-per-second or TCP/IP packets per time-slot that are successfully delivered. The bottleneck and available bandwidth on an Internet path, influences the throughput. However, it is also influenced by other factors such as link or router failure, choice of transport protocol, packet loss, etc. The throughput metric is used to research techniques on all layers in the OSI stack [122].

The throughput of an Internet path can be calculated in many different manners. The most common ones are *maximum throughput* and *average throughput* [74].

2.5 Thesis goals and target metrics

Section 2.2 and 2.3 introduced a range of requirements that distributed interactive applications have. Many of these requirements are now formulated into more specific research goals that the thesis tries to address. First, we define four research areas with one particular goal statement for each of them. Then the four research areas are discussed briefly along with the most important metrics for each of them.

2.5.1 Introducing the specific goals

The basic requirements to achieve a system that supports interactivity and dynamicity of clients from section 2.3 can be summarized as such: "Joining and removing clients from groups should be done within real-time latency bounds and not disrupt the service. Application events must be distributed to all clients such that their latency bounds are met. Internet resources are scarce, therefore, the events must also be distributed efficiently and cost-effective [131, 130, 125, 127, 128, 129, 60, 17, 126]."

The thesis aims to address these requirements through studies of network-related group communication techniques. A general goal of the thesis is therefore to:

Investigate multiple network-related group communication techniques with the goal of identifying those that enable distributed interactive applications, in which clients are able to join and leave ongoing sessions of real-time interaction.

This goal is very general and applies to very many research areas. Therefore, we subdivide this general goal into four specific reasearch goals that we aim to address through simulations and experimental studies in the course of the thesis. Before we state the research goals, we present the research areas:

- A ***membership management*** must ensure that clients are able to join and leave ongoing sessions of real-time interaction, in a timely fashion.
- A ***resource management*** must ensure that well-placed nodes are found that yield low latencies to groups of clients, such that they are available for management tasks.
- An ***overlay network management*** must ensure that clients are configured in overlay networks that yield sufficiently low-latencies for real-time interaction.
- A ***network information management*** must ensure that Internet path latencies between the interacting clients are available and sufficiently accurate.

These four research areas are now briefly presented, and a research goal is stated for each of them. In addition, specific performance metrics are introduced in some detail and then linked to the research goal.

2.5.2 Membership management

The membership management in a distributed interactive application must be able to join and remove clients from ongoing sessions of real-time interaction. The memebership management must carry mechanisms that support the management of a highly dynamic client base, where multiple subgroups are dynamically forming and changing. The concrete goal is therefore:

1) Identify techniques that enable an efficient and timely membership management of multiple dynamic subgroups of clients.

Generally, the membership management must have mechanisms that can execute incoming messages from clients that request to join and leave sub-groups of clients engaging in real-time interaction. The consistency in the membership management system must be at a level that enable any frequency of dynamically changing subgroups of clients.

It is important that the time it takes to execute a membership change request is sufficiently low to seemlessly update the sub-groups of clients. An important metric is therefore the *membership change execution latency*, which is the time it takes for a membership change request to complete. This latency should be bound sufficiently such that real-time membership changes are enabled.

2.5.3 Resource management

The resource management in a distributed application must enable the search for nodes in the network that yield a desirable property. Therefore, the resource management should include algorithms that are able to search for such nodes. In the thesis, the specific goal for enabling a desirable resource management is:

2) Identify techniques that enable a resource management to identify nodes in the (application) network that yield low pair-wise latencies to groups of clients.

Such nodes may be any node in the application network (server, proxy, member-node, etc), and is often referred to as core-nodes. When the resource management has identified such core-nodes, it is possible to appoint and execute management tasks on them. In particular, it is possible to execute the membership management from section 2.5.2.

As mentioned, a core-node should yield low pair-wise latencies to groups of clients. More specifically, a core-node should have a low *maximum one-way latency* to nodes in its group of clients. This is because, if a core-node executes management tasks, a low maximum one-way latency ensures that the group-nodes can communicate with the core-node through low latency paths. In addition, the *execution time* of the techniques that identify the nodes should be low to enable real-time search for core-nodes and appointing of management tasks.

2.5.4 Overlay network management

A distributed interactive application has events that occur on each client, and these events must be shared with the remaining clients that take part in the real-time interaction. Therefore, there is a need for techniques that configure event-distribution networks such that they are able to share time-dependent communication among multiple participants. In the thesis we want to enable an efficient event-distribution network management. Therefore, there is a need to:

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

A focal point of the thesis is to use application layer overlay multicast to distribute events among interacting clients. The main reason for this is that network layer multicast is not fully deployed in the Internet, and have a range of unsolved problems (section 3.2.2).

The overlay networks that are used to multicast events must be constructed by algorithms that take into account the pair-wise latencies among the clients. In particular, it is the *maximum pair-wise latency* that needs to be bounded, if all the clients are to achieve real-time interaction with a certain user satisfaction. The maximum pair-wise latency in an overlay network is defined by the longest shortest path. Furthermore, when nodes join and leave groups of clients, the overlay networks must be reconfigured such that the current member-nodes are still able to engage in real-time interaction. Therefore, the overlay network must be reconfigured such that the *reconfiguration time* is minimal. Moreover, the overlay should be reconfigured in a manner that minimizes the disruption to the active communication paths. Therefore, the *stability* of the overlay network across each reconfiguration should be acceptable. The stability can be measured in terms of the number of *edge changes* that occur across overlay network reconfigurations due to nodes changing groups. Average clients on the Internet currently have rather limited capacities, although this has improved significantly in recent years. Therefore, the overlay network should not contain nodes that have an unreasonble high *stress* level. The stress on a node is measurable in terms of how much resources it expends when distributing and forwarding application events in the overlay network. Finally, the overlay network should not add an unreasonable cost to the network. The cost may, for example, be in terms of the total bandwidth consumed. Therefore, the *total cost* of the overlay network should be bounded within reason. The previous metrics can be summarized to this goal:

Identify techniques that in a timely fashion can configure overlay networks for event-distribution based on incoming join and leave requests, such that the overlay:
1) yields a low maximum pair-wise latency,
2) yields a low average pair-wise latency,
3) does not add unreasonable cost to the network,
4) is reconfigured such that its stability is acceptable,
5) does not contain nodes with an unreasonable high stress level.

2.5.5 Network information management

The network information management in a distributed interactive application should be able to obtain sufficiently accurate Internet path characteristics among the application clients. These path characteristics may be used by the resource management in the search for nodes that have low pair-wise latencies to groups of clients, and also the application network management in the search for low-latency event distribution paths that inter-connect the clients. The goal is therefore to:

4) Identify techniques that are able to obtain accurate all-to-all Internet path latencies.

The network information management should contain techniques that obtain path latencies through active measurements, but also techniques that are able to estimate path latencies based on partial network information. In the case of active network monitoring, the *scalability* is very much a question as long as all-to-all Internet path latencies are required. Further, in the case of latency estimation techniques, the *latency estimates' accuracy* is a very important metric to determine their usability.

2.6 Evaluation methods

The previous goals may be studied in many different ways, but choices have to be made in order to study and compare group communication techniques properly. The following sections highlight how we approached the goals in the thesis.

2.6.1 Methods to reach the goals

In our research, we aim to study and identify techniques for a membership management, resource management, overlay network management and network information management. However, it is a hard problem to study such techniques through real-world experiments, when the context is large-scale distributed interactive applications. Nevertheless, in the thesis, we do study such techniques both through simulations and real-world experiments.

The real-world experiments are conducted on PlanetLab. However, PlanetLab currently only has a rather limited number of nodes available (much fewer nodes than in current MMOGs). Therefore, we mainly use the PlanetLab experiments to study whether it is possible to retrieve accurate all-to-all path latencies.

The simulations are performed using a group communication simulator (section 2.6.4) we developed. For the simulations, we used the BRITE tool (section 2.6.2) to generate Internet topologies, and the group dynamic model we used, was based on a Zipfian distribution model (section 2.6.3). These are now introduced in some detail.

2.6.2 BRITE Internet topology generator

When simulation studies of the Internet are performed, Internet topology generators are often used to generate close-to-real networks [73, 66]. These generated networks should include the most necessary information about Internet characteristics and its infrastructure: router information, link latency and bandwidth, authoritative system (AS) information, etc. The main motivation for using topology generators is due to the fact that the performance of a network protocol can be heavily influenced by the network topology. This may be the case even when a network protocol is independent of the underlying network topology [66].

Many topology generators exist [73, 148, 91] and their characteristics and topology generating models are outside the scope of the thesis. However, it has been argued that the topology generator BRITE [91] generates Internet-like topologies [73, 66]. Therefore, we used BRITE for the simulation studies in the thesis.

BRITE supports two basic models for topology generations: The Waxman model [139], and the Barabasi-Albert model [15]. Both models are commonly used in the literature, and more information regarding the pros and cons about these models can be found here [66]. In addition, BRITE can produce both flat router graphs, and hierarchical AS topologies, in which the node placement can be either random or heavy tailed.

2.6.3 Group dynamics and a Zipfian distribution model

Group dynamics is the study of groups, where a group is defined by two or more parties (nodes, clients, etc) that are connected to each other by some relationships [15]. Group dynamics are studied in many different fields, such as phsychology, sociology, and communication studies. In the context of the thesis, we use group dynamic models to study client dynamics and group communication in distributed interactive applications. Group dynamic models are also applied to other areas, for example, in mobile ad-hoc network studies to simulate mobile nodes that are roaming around and communicating through wireless links [132].

A group dynamics model for a distributed interactive application must consider many issues. For example, the groups that are available to join may vary depending on a user's current group. If the group dynamics in an MMOG is simulated, it is often the case that each group represents a virtual area in the game, which was previously defined as an area-of-interest (section 2.2.2). Therefore, a user controlling an avatar in the virtual world only has a number of adjacent virtual areas-of-interests ("next-group" options) it can direct the avatar to. Furthermore, the popularity of the groups in an application may also differ significantly, and depends on a number of application-related factors. For example, what services the group has to offer, where it is located in a virtual world, how it is accessed, etc.

There are a range of existing group dynamic models, where two common approaches are a random walk model and a Zipfian distribution model. A random walk model is the simplest. In this model users are randomly choosing a next group of users to join, based on its current. In the thesis we use a Zipfian distribution model for group dynamics, which is a model that follows Zipf's law [20]. Zipf's law states [55]: "Given some corpus of natural language utterances, the frequency of any word is inversely proportional to its rank in the frequency table. Thus the most frequent word will occur approximately twice as often as the second most frequent word, which occurs twice as often as the third most frequent word, etc." Similarly, we use Zipf's law

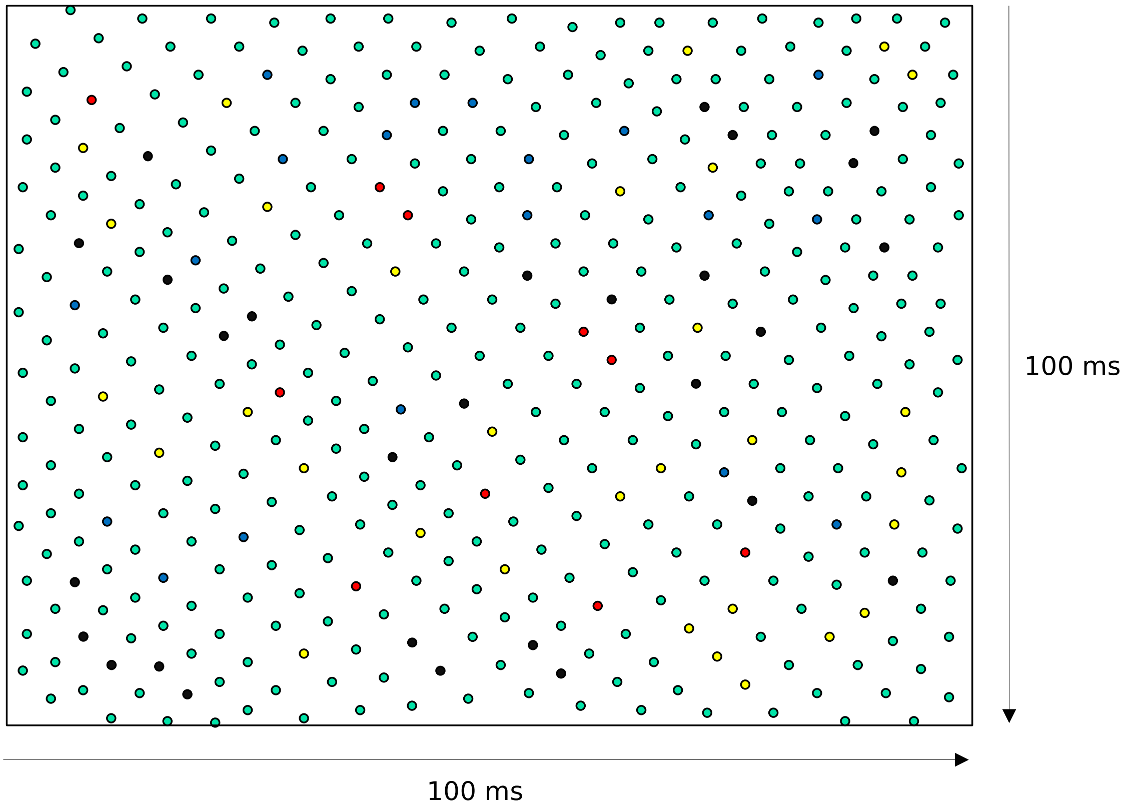


Figure 2.4: Illustration of the simulation layout used in the group communication simulator. Each circle is a client and each client can communicate with every other client through application layer overlay links (invisible in the figure).

to achieve a Zipfian distribution of the group popularity. We explain this further in section 2.6.4.

2.6.4 Group communication simulator

The group communication simulator we developed uses a fully meshed (complete) shortest-path graph to simulate group communication on the application layer, where all nodes are clients.

For our experiments, we used BRITE to generate flat undirected Waxman topologies [139] with 1000 routers. The router placement was random in a plane of size 100×100 milliseconds. The placement in the plane is used by BRITE to calculate the edge weights between the routers. The BRITE network graph is translated into an undirected fully meshed shortest-path graph on the application layer, where each router has exactly one client connected to it.

The nodes in the shortest-path graph are now clients, and the edges are shortest paths connecting each client pair. These shortest paths between clients include a varying number of routers that are invisible from the application layer. Figure 2.4 illustrates how the simulation layout appears from the application layer. In the figure, each circle is a client and each client can communicate with every other client through application layer overlay links (invisible in the figure).

Each edge in the shortest-path graph has two cost values associated to it. One is the edge latency, the other is the network layer hop-count. They are both calculated from the BRITE router network graph, by adding the edge weights on the shortest paths, and the resulting hop count. As mentioned, each router in the BRITE network graph was transformed to a client in the shortest-path graph.

Relaying packets through intermediate nodes contributes to the end-to-end delay. However, in our investigation we do not add relay penalties, because such penalties may vary very much depending on a node's computational power, traffic type, etc. In our studies, we rather highlight the hop-counts and from them it is straight forward to observe how a relay penalty will add to the latencies.

We assume that all of the clients are part-taking in a distributed interactive application, in which they are continuously changing groups. For example, each client may be controlling an avatar in an MMOG. These avatars roam in a virtual world, in which they enter and leave area-of-interests. The area-of-interests and the avatar dynamics are simulated using a Zipfian distribution model on a torus. The torus is divided into 1000 squares, one for each client, where each square represents an area-of-interest. At the start of a simulation, each area-of-interest contains one avatar.

In our simulations, the number of clients is fixed, while an MMOG has a varying number of clients playing at any given time. However, we still have unpredictable dynamics in group membership, and it is this group dynamics that we are aiming to simulate.

2.7 Conclusions and the research goals in summary

The previous sections introduced distributed interactive applications, in terms of their research challenges and specific requirements. Specifically, we presented a range of research challenges for MMOGs. Then, we discussed five basic requirements to distributed interactive applications. These requirements can be summarized as such: "Joining and removing clients from groups should be done within real-time latency bounds and not disrupt the service. Application events must be distributed to all clients such that their latency bounds are met. Internet resources are scarce, therefore, the events must also be distributed efficiently and cost-effective." From these requirements we defined a general research goal:

Investigate multiple network-related group communication techniques with the goal of identifying those that enable distributed interactive applications, in which clients are able to join and leave ongoing sessions of real-time interaction.

From these straight-forward requirements of interactivity and dynamicity we managed to describe more specific goals for the research in the thesis. These were four goals that are:

1) Identify techniques that enable an efficient and timely membership management of multiple dynamic subgroups of clients.

2) Identify techniques that enable a resource management to identify nodes in the (application) network that yield low pair-wise latencies to groups of clients.

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

4) Identify techniques that are able to obtain accurate all-to-all Internet path latencies.

Each of these goals are addressed later in the thesis. The first goal is addressed mainly in chapter 5. The second goal is mainly addressed in chapter 7. The third goal is addressed in chapters 8 through 14. Finally, the fourth goal is addressed in chapters 6, 7 and 15.

In the investigations we aim to research and identify techniques for each of these goals. However, it is a hard problem to study such techniques through real-world experiments when the context is large-scale distributed interactive applications. Nevertheless, in the thesis, we do study such techniques both through simulations and real-world experiments.

The real-world experiments are conducted on PlanetLab. However, PlanetLab currently only has a rather limited number of nodes available (much fewer nodes than in current MMOGs). Therefore, we mainly use the PlanetLab experiments to study whether it is possible to retrieve accurate all-to-all path latencies.

The simulations are performed using a group communication simulator (section 2.6.4) we developed. For the simulations, we used the BRITE tool (section 2.6.2) to generate Internet topologies, and the group dynamic model we used, was based on a Zipfian distribution model (section 2.6.3).

Both the experiments on PlanetLab and the group communication simulations were applied with the aim of identifying techniques that properly address our goals. We also study graph theoretical problems related to overlay network design, and the algorithms we tested were put through some theoretical scrutiny.

Chapter 3

Group communication: State-of-the-art and related work

The background and motivation chapter identified a range of research questions that need to be answered, and also introduced the concrete research goals that are important to achieve for the further development of distributed interactive applications. In the following sections we link the research goals to specific research areas. These research areas are briefly surveyed with a particular focus on identifying techniques and algorithms appropriate for distributed interactive applications.

In section 3.1, we discuss group communication and how low-latency overlay networks are investigated in different scenarios. We identify that overlay multicast is suited for content and event distribution through a network, and briefly highlight existing approaches to investigate overlay construction. We conclude that a distributed interactive application scenario needs overlays that are built such that all the clients can communicate through paths that have a sufficiently low maximum pair-wise latency (longest shortest path).

Section 3.2 introduces multicast in the Internet and many of its challenges. A thorough survey is given of a wide range of multicast approaches and protocols. Many of their properties are highlighted and critisized in terms of their applicability to distributed interactive applications. The survey concludes that no existing multicast protocol have all the properties that are required; support for constructing low-latency overlays and handling multiple dynamic subgroups of clients.

Section 3.3 introduces core-node selection algorithms as a way of identifying well-placed nodes in a network that yield low pair-wise latencies to groups of clients. We found that there are existing algorithms that are promising, but that needs further investigation in a dynamic scenario to verify their efficiency.

Section 3.4 surveys the area of spanning-tree algorithms, with the intention of highlighting existing work for creating low-latency overlay networks. We found that there are existing spanning-tree algorithms that construct desirable low-latency networks, but they have not been thoroughly examined and compared. It is especially uncertain how their behavior is in an environment of multiple dynamically changing subgroups. We do similar related-work surveys for Steiner-tree algorithms in section 3.6, and also subgraph (mesh) algorithms in section 3.7.

Section 3.5 surveys dynamic tree algorithms, which are algorithms that can insert and remove single nodes from existing trees. This property is desirable for distributed interactive applications, because they may have potentially many dynamically changing subgroups of clients. The survey highlights approaches that are suitable, but we did not find any study of these algorithms applied to an application scenario to investigate their cumulative effect on maximum pair-wise latencies in trees.

Section 3.8 presents some related-work for latency estimation techniques. We want to identify approaches that are able to obtain all-to-all link latencies, with the intention of building a graph such that graph algorithms can build low-latency overlay networks, and search for core-nodes. The survey concluded that there are existing techniques that are suited for all-to-all path measurements.

First up are introductions to group communication in the Internet and how low-latency overlay networks may be obtained and studied.

3.1 Low-latency overlay networks

Group communication in the Internet poses challenges related to asymmetry, heterogeneity, resource availability and latency. One way of modelling some of these challenges is to apply graph theory. In graph theory, a network may be modelled as a *directed* or *undirected* graph. Directed edges (arcs) allow asymmetric links, while undirected edges are symmetric. Currently, asymmetric links are the most common situation for clients in the Internet. However, the symmetric link assumption of undirected graphs is only considered unrealistic in highly asymmetric networks, and in applications that require high bandwidth and low latencies. Current multi-player online games support few if any such flows of content or events, rather, the streams are relatively thin [59].

It is possible to use overlay *multicast* to accomplish content and event distribution through a network. And, in that respect, we found in [127] that a connected acyclic graph (*tree*) has several advantages over a connected cyclic graph (*mesh*).A tree has small routing tables and saves network bandwidth. In addition, it has low administration costs when the membership is dynamic. However, the pair-wise latencies do increase, and if a non-leaf node is disconnected from a tree, the tree is also disconnected resulting in two subtrees. A mesh increases the node failure tolerance of the graph, because multiple paths to a node exist. Unless some path routing is applied to a mesh it introduces data redundancy because some nodes receive two copies of the same data. Multiple paths are also valuable in cases of fluctuating link costs and to reduce the pair-wise latencies. Generally, there are two main directions of mesh and tree (*overlay*) construction related to group communication in networks; overlays that are constructed for a *single source* and for *multiple sources*. However, a mesh is not used for single source scenarios unless data redundancy is necessary due to unstable network nodes and/or links, rather, a mesh is mostly used in a multiple source scenario.

For single source situations, a shortest-path tree [58] is often used. A shortest-path tree gives minimum latency routes between a pre-chosen source and all destinations. However, shortest-path trees are costly because they consume much of the network resources. Especially forwarding nodes close to the source may experience massive stress issues because they have a high *degree*, where the degree of a node is the number edges it is connected to in a graph. Alternatively, a minimum-cost spanning-tree [58] may be used. A minimum-cost spanning-tree is constructed such that the *total cost* is minimized, i.e., the sum of the edge weights. However, the latencies between the source node and destinations are higher than in shortest-path trees. Clearly, there is a tradeoff between the source destination latencies and the total cost of a tree. *Shallow-light* trees [80] are trees with a single source that simultaneously approximate well both a minimum-coast spanning-tree and a shortest-path tree. A shallow-light algorithm computes a tree that is at most a constant times heavier than the minimum-cost spanning-tree, and with the property that the diameter of the tree is at most a constant times the diameter of the input graph.

For multiple source situations, it is vital to reduce the pair-wise latency, such that every source is within a constant latency bound of every destination. That is, the *eccentricity* of the destinations should be minimized, where the eccentricity of a node is defined as the maximum pair-wise latency from the node to its farthest destination. In our group communication scenario, every node in the network is a source that distributes data. In situations where all nodes are sources, the overlay should be constructed as a *shared-overlay*. For such a shared-overlay scenario, it is not enough to only consider an overlay viewed from a single source or a set of sources. Every node is both a source and a receiver, thus the worst case eccentricity in a shared-overlay equals the diameter. Therefore, in a shared-overlay scenario, the diameter is a very important metric. One example of such overlay algorithms are *minimum-diameter* tree-algorithms, which aim to build a tree on an input graph that yield the minimum diameter [116]. Ho, Lee, Chang and Wong [68] proved that a minimum-diameter tree is achievable in polynomial time. Another example are *bounded-diameter minimum-cost* tree-algorithms, which aim to build a tree on an input graph that yield the minimum-diameter within a total cost bound [145]. This problem is NP-complete, but there are several polynomial time heuristics that approximate close-to-optimal solutions [131].

These and other algorithms are surveyed in more details in the following sections. First, we introduce a range of multicast protocols that exist in the literature.

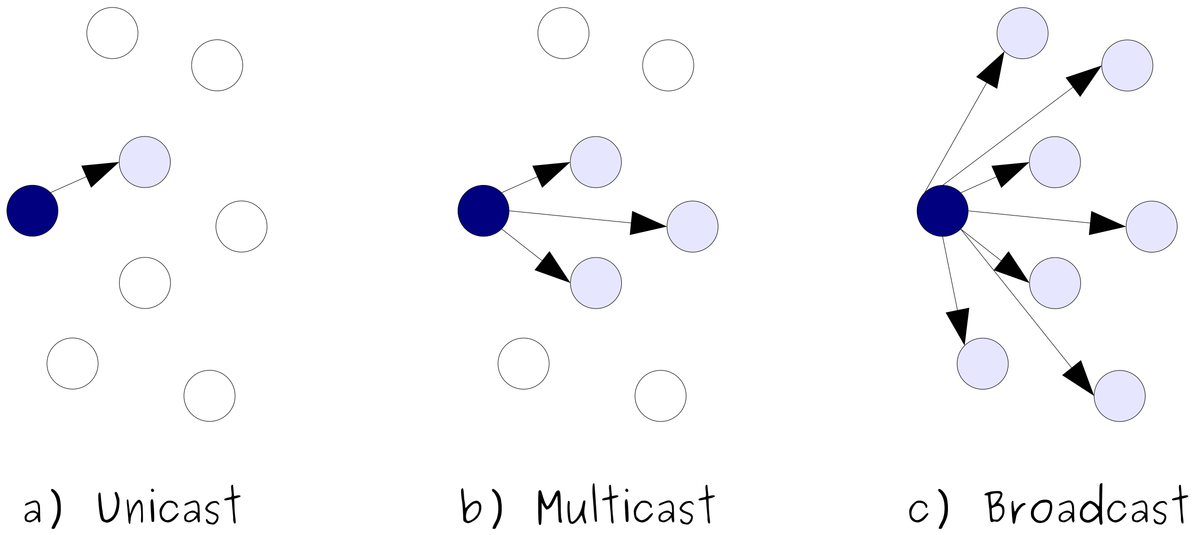


Figure 3.1: Unicast, multicast and broadcast.

3.2 Multicast communication in the Internet

The following sections survey the area of multicast communication in the Internet, with a range of existing approaches and protocols. We discuss general requirements to multicast protocols, and what specific types that have been developed. However, the focus is especially on identifying existing protocols that satisfy the strict requirements that distributed interactive applications have, for example, low-latency event distribution networks, low join latency and support for group membership dynamics.

3.2.1 Network communication in the Internet

Network communication is the act of transmitting data from one network entity to another, where a network entity may be a client's computer, a network layer router, etc. Generally, network communication across the Internet may be achieved in essentially three different ways [1] (see figure 3.1):

- *Unicast* is a one-to-one communication, where each packet is addressed to a single receiver.
- *Broadcast* is a one-to-all transmission. A broadcast packet is received by every participant of a network.
- *Multicast* is a one-to-many or many-to-many (selected nodes) transmission, where a multicast group receives the data.

[1] Other variations of these communication methods do exist; anycast, geocast, etc.

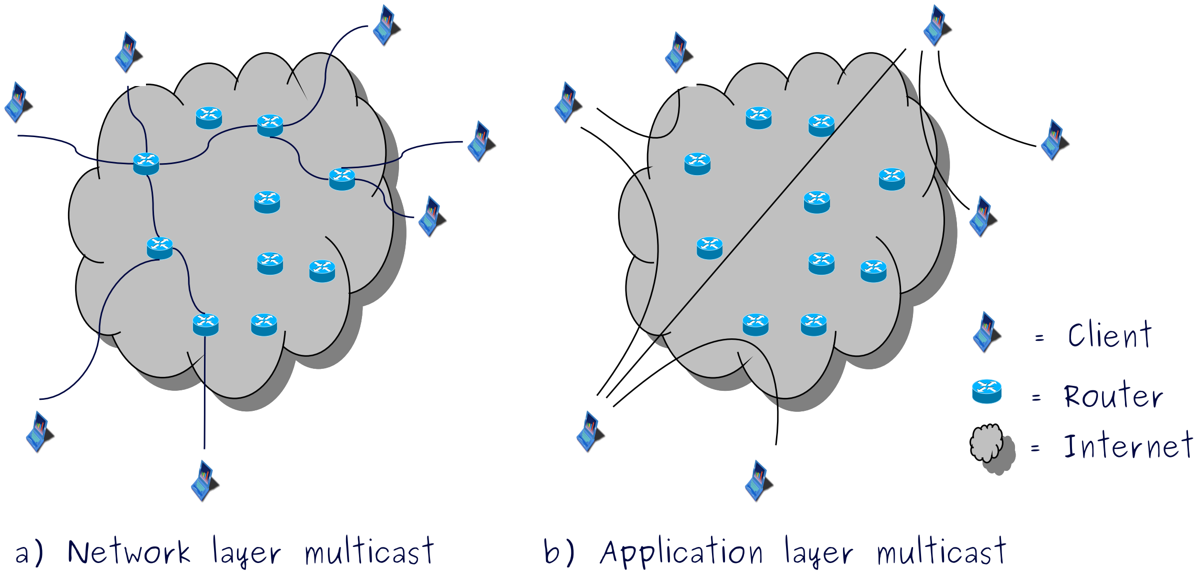


Figure 3.2: Application layer overlay multicast compared to network layer IP Multicast.

A client on the Internet has the option of using unicast and multicast, while broadcast cannot be used mainly due to security reasons. Unicast is by far the most common way of communicating across the Internet. However, in recent years, the number of applications that would benefit from using multicast has grown. This is mainly due to the increased bandwidth capacities and consequently increased throughput in the Internet. Figure 3.1 illustrates the differences between unicast, multicast and broadcast communication.

The target applications for the thesis are distributed interactive applications. These applications typically need to share events in order to achieve live interaction across the Internet, and multicast is a way of sharing events among groups of clients in an efficient manner. It is common to say that multicast is a way of achieving group communication in the Internet. There are interactive applications that are already using multicast, for example, video-conference systems and tele-education. However, the use is rather limited at present.

In the Internet today, multicast services may be available on the network layer and application layer:

- *IP Multicast* is offered at the network layer, but it is not widely deployed.
- *Application layer multicast* is end-system overlay multicast, which must be supported by the application.

Figure 3.2 illustrates the differences between IP Multicast on the network layer and application layer multicast. Network layer and application layer multicast are now introduced as options for achieving group communication in the Internet.

3.2.2 Network layer multicast

The network layer multicast protocol is called *IP Multicast* [19, 119, 40, 37], and is not widely deployed in the Internet today. IP Multicast is based on an open service model, which means that no mechanisms restrict the hosts or users from creating a multicast group, receiving data from a group, or sending data to a group [40]. Hosts join a multicast group by contacting their local router using the Internet Group Management Protocol [46] (IGMP). The multicast distribution tree between the routers is built and maintained by a multicast routing protocol, for example, DVMRP [135], MOSPF [94], CBT [12, 11] and PIM DM/SM [43, 2]. IP Multicast was heavily researched in the mid nineties, but it has not been the anticipated success. Today, IP Multicast is mostly used by educational institutions and commercial networks, for example, TV operators delivering high-speed video on-campus, and financial stock-ticker.

In summary, some of the issues in IPv4 Multicast are that it 1) is not supported by all Internet service providers, 2) cannot be used efficiently with TCP (which is frequently used by, for example, online games), 3) does not easily support frequent membership change, 4) cannot prevent snooping, and 5) has a rather limited address space available.

3.2.3 Application layer multicast

Multicast on the application layer is a less complex and cheaper alternative to multicast on the network layer [137]. Various names are given to multicast services at the application layer, for example, application layer multicast, overlay multicast and end-host/end-system multicast [26, 14]. We use these terms interchangeably.

Application layer multicast avoids the IP Multicast issues, mainly, because it is user managed and not network dependent. The data replication, multicast routing, group management, and other functions are all achieved at the application layer and implemented at the end-host. The multicast functionality is software based and often embedded in the application. Peer-to-peer applications are typical applications that use application layer multicast [124].

In application layer multicast the group members form an application layer multicast topology. The links in the multicast topology connect group members, and create an overlay where the network layer is invisible. The group members act as relay agents, and multicast is achieved through message forwarding among the members of the overlay using unicast across the underlying network (Internet). Therefore, application layer multicast incurs some delay and bandwidth penalties, furthermore, it is less stable (more prone to failure) because fault prone clients are replicating the data. Despite these drawbacks, application layer multicast is a more attractive (read: cheap) multicast approach than IP Multicast, until the issues in IP Multicast are solved (section 3.2.2).

Today, two general approaches are used to accomplish application layer multicast. One is peer-to-peer networks that are designed for file and information sharing, for example, BitTorrent [28] and Gnutella [29]. The peer-to-peer topology tends to be more random to the underlying physical topology which affects the service as the latency can be very high. The second approach focuses on improving the application layer multicast and providing group communications. Many of these application layer multicast protocols are topology aware and designed to achieve a lower latency and better bandwidth usage [83, 9].

3.2.4 Overlay multicast properties

Many overlay multicast protocols have been proposed [30, 116, 44, 32, 107, 35, 146, 124, 33, 39, 97, 85], but there is still much research that needs to be done, especially in the areas of supporting dynamically changing sub-groups of clients, and the subsequent reconfiguration of multicast topologies. Currently, there are two classes of overlay multicast techniques: Fixed nodes based multicast, and dynamic nodes based multicast.

Fixed nodes based multicast places a number of nodes strategically around the Internet. Then, according to the applications' requirements, these nodes autonomously form overlay multicast trees to provide multicast services [30]. Fixed nodes based multicast is stable, but it is not flexible, and needs ISP support. Furthermore, the fixed nodes are potential bottlenecks, such that some quality of service flow management may be necessary. Figure 3.3 illustrates a fixed nodes based multicast tree.

Dynamic nodes based multicast approaches work such that the nodes self-organize into an overlay multicast tree. Data duplication, multicast data forwarding, and group membership management and other functions are all achieved at the client side. Figure 3.2 b) illustrates a dynamic nodes based multicast tree. Dynamic nodes based multicast has two subclasses: Structured and unstructured overlay multicast.

Structured overlay multicast approaches typically impose constraints both on the topology of the overlay and on the data placement. Both of these constraints are applied to enable efficient discovery of data. Structured overlay multicast protocols are characterized by self-organization and fault-tolerance capabilities, where examples include CAN [105], SkipNet [63], Chord [118], Pastry [109], and Tapestry [149]. These systems provide an addressing scheme, which is independent of the actual network addresses, and is used to implement scalable and efficient application-level routing mechanisms that are adaptive to node joins and leaves. The addressing scheme is commonly implemeneted using distributed hash tables (DHTs). Many publish-subscribe systems are built on top of structured networks, where two examples are Scribe [23] and Bayeux [151]. However, due to the complicated structures in the DHTs, these systems are not easy to deploy in a heterogeneous network environment. Overlay protocols that use DHTs are appropriate for file sharing applications, but to this date, none of them fit to event sharing applications because they do not consider pair-wise latency requirements.

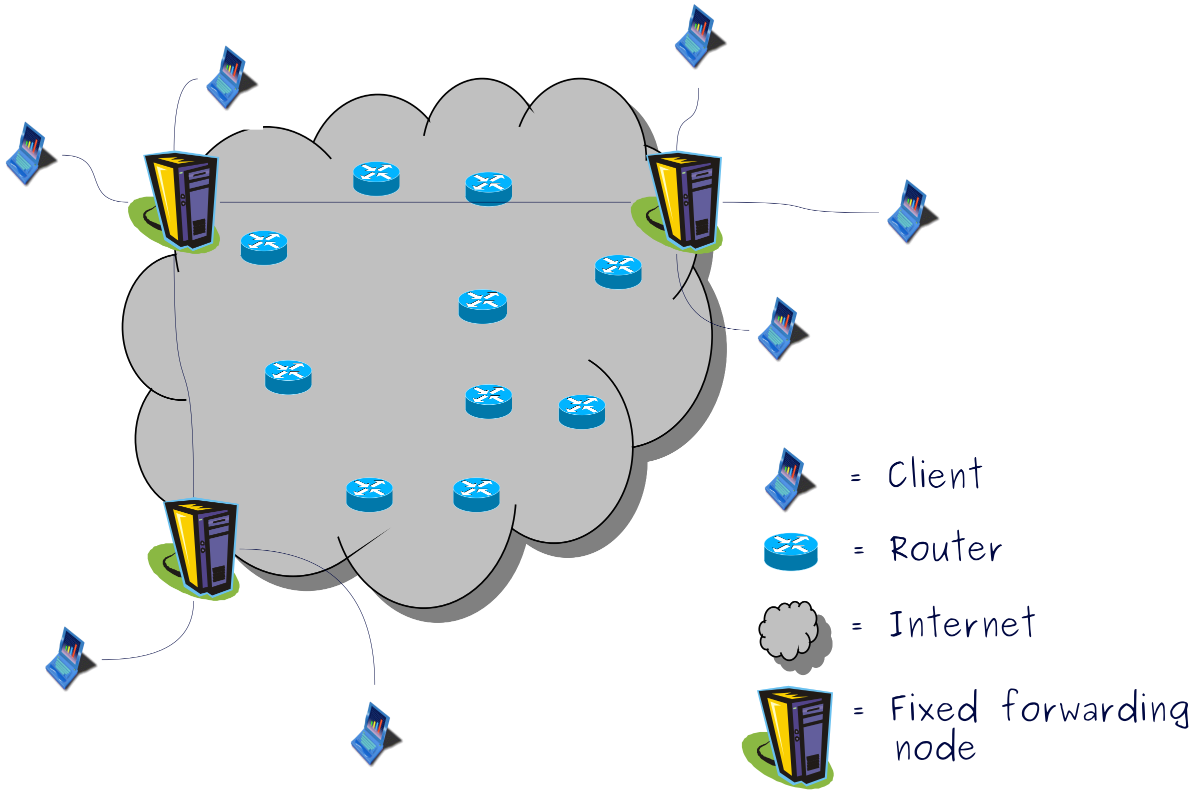


Figure 3.3: Fixed nodes based multicast has dedicated forwarding nodes.

Unstructured overlay multicast approaches incur looser constraints in the topology building and maintenance. The overlays are often created ad-hoc, and may also be created completely random in the extreme case, although this has become less common. There are two main strategies in the creation of unstructured overlays: Mesh-first and tree-first multicast protocols.

Mesh-first multicast approaches build overlays that have more than one path between the pair of nodes. After creating a mesh, the members often participate in a routing algorithm on the mesh topology. This is the case for both Narada [49] and Scattercast [25]. Because mesh-first overlay multicast protocols build such cyclic meshes that have multiple paths between participants, they yield a level of node failure resilience. While the advantage of having a mesh is resilience, the downside is that it may be necessary to run a routing algorithm for construction of loop free forwarding paths between the members. Another option, is to add a sequence number to the packets and broadcast them through the overlay.

Tree-first multicast approaches directly construct an overlay tree topology for data delivery. Examples of such approaches include Yoid [50], Overcast [71], and ALMI [101]. Trees have only a single path between any pair of nodes, and are thus sensitive to partitioning because they are acyclic graphs. For example, if any non-leaf member disconnects from the tree, the overlay is partitioned (broken). That is why tree-first overlay protocols often provide additional control links to allow quick recovery from member failures, which is the case for the centralized

	Topology construction	*Routing algorithm*	*Node join/leave*	*Topology constraints*	*Group size*	*Membership management*	*Quality of Service*
Narada	mesh-first	DVMRP	random/time-out	e2e del	small	distributed	-
Yoid	tree-first	root based	reconstruction	-	medium	rendevouz point	partition discovery
ALMI	mesh-first	MST	centralized	e2e del	small	sess. controller	neighbor latency
NICE	hierarchical	implicit	min-cost/new center	e2e del	large	rendevouz point	leader heart beat
Bullet	mesh-first	any root based	(N/A)	e2e bw	large	distributed	-
Tmesh	mesh-first	root based	partition recovery	e2e del	large	distributed	tree shortcuts
TAG	tree-first	root based	min-cost/(N/A)	hop cnt	large	root controls	topology aware
BTP	tree-first	spanning/root	(N/A)	e2e del	large	distributed	switch tree
AMcast	tree-first	MDDL-heurist.	(N/A)	e2e del	large	distributed	-

Table 3.1: Overlay multicast table

ALMI [101]. However, for the reason of being an acyclic graph, tree overlays have no routing loops and do not need a routing algorithm other than neighbor information. Tree-first based overlay multicast protocols include approaches that construct source trees, spanning trees, core based trees, etc [130].

Hybrid mesh and tree multicast approaches first construct a mesh topology, and then use standard shortest path tree algorithms to establish a minimum distribution tree/mesh. Such approaches typically include many of the mesh-first multicast approaches (mentioned above), where one example include Narada [49]. Tmesh [137] is another example that first builds a tree, and then enhances the tree with a number of single edges to reduce the pair-wise lantecies.

Topology aware multicast approaches construct overlays based on knowledge of the underlying Internet topology. As mentioned previously, overlay networks are built from the application layer. Therefore, data may traverse the IP network quite inefficiently if the overlay multicast protocol is unaware of the Internet topology. Topology aware overlay multicast protocols aim to improve the data delivery efficiency in terms of reduced latency and resource usage (network cost, bandwidth) [83, 9].

3.2.5 Group management techniques

Group management of dynamically changing sub-groups of clients is important to enable a variety of scalable distributed interactive applications. Essentially, there are three different approaches to group management; centralized, distributed and a hybrid approach named hierachical management. Which to choose is an important design choice, and influences the target applications, in terms of their scalability, robustness, etc.

In a *centralized* approach, a node is assigned to control the membership information, and assist the application's group members to form an overlay multicast topology. ALMI [101] is a centrally managed application-level group communication middleware, tailored towards the support of relatively small multicast groups with many-to-many semantics. Centralized approaches avoid many consistency issues, but the scalability may suffer. Other typical centralized issues are, single point of failure, potential bottleneck problems and resulting slower management. However, a multitude of fault tolerance and quality of service mechanisms are available that reduces the chance of loosing valuable data [149].

In a *distributed* approach the overlay and membership information is dynamically distributed to the members. Many file-sharing and video-streaming peer-to-peer applications use this distributed approach [124, 33, 39, 97, 85]. However, these applications do not support interactivity, and due to this there is no need to create sub-groups of clients. Generally, such completely distributed overlay applications are scalable, but they do have inherent consistency issues that must be handled in order to support interactivity.

A *hierarchical* approach aims to address the centralized scalability issues, and distributed consistency issues. In a hierarchical approach the members form a hierarchical structure and assign specific tasks/roles to the group members. This way, the group management is distributed among a few nodes, which effectively decreases the multicast tree control traffic. AMcast [115] is such an approach, which uses a set of distributed multicast service nodes (MSN). The authors focus on optimizing the access bandwidth of the MSN's interfaces and end-to-end delay, and propose several new centralized tree algorithms, for example, the compact-tree and bounded compact-tree algorithm. These spanning-tree algorithms are evaluated in chapter 9.

3.2.6 Overlay multicast protocols

The following includes a more in-depth introduction to a number of overlay multicast protocols that exist in the literature. These were found to have interesting properties that are useful in interactive settings, although none of them address more than a small sub-set of the requirements (of distributed interactive applications). In the presentations we try to answer:

Do the protocols have support for constructing low-latency overlays and handling multiple dynamic sub-groups of clients?

We list the strengths and weaknesses of the overlay multicast protocols in terms of this question.

End-system multicast (Narada) is one of the early proposed overlay multicast protocols, and was designed for small to medium sized groups [26]. *Narada* [49] is the *distributed* protocol, by which the multicast group members self-organize into an overlay structure.

Narada is a *mesh-first* protocol and uses a two step process to build and refine a *source-specific* multicast tree. First, it distributedly constructs a mesh while it tries to ensure that the mesh has low pair-wise latencies. Second, reverse shortest paths between each recipient and the source are created, using the DVMRP algorithm [135]. The efficiency of the resulting tree depends on the quality of the mesh (from step one).

Group dynamics are addressed as each member maintains a list of all other members in the group and periodically shares its group information with neighbor nodes in the mesh. However, the overhead of such a protocol is tremendeous for larger dynamic groups. A new member *joins* the mesh by randomly selecting peers from a list of connected members obtained through an out-of-band mechanism and requests to be their neighbors. A *leaving* member is removed by an approach similar to detecting a node *failure*: When a host stops receiving refresh messages from a neighbor, it probes the neighbor, and if the neighbor is dead, it propagates this information through the mesh.

Banana Tree Protocol (BTP) [67] is a *tree-first* protocol, which was designed especially for peer-to-peer applications.

BTP supports *source* and *spanning* tree optimizations, and uses reconfigure and optimization algorithms, called *switch-tree* algorithms, to improve the tree. The switch-tree algorithms allow any node to switch parent to its nearby sibling or grandparent, to reduce the tree cost or latency. The switch-tree algorithms obey a user-set *degree constraint*. BTP avoids tree loops by applying two basic switching rules: 1) the potential parent can not simultaneously attempt a switch, 2) the potential parent is still the node's sibling or grandparent.

Group dynamics are not addressed by the authors.

Your Own Internet Distribution (Yoid) [50] is a DARPA-sponsored general overlay network based content distribution toolkit. It addresses applications like netnews, streaming broadcasts, and bulk email distribution. Yoid includes a full framework for overlay multicast implementation, where the main goal is to address all the aspects of multi-peer transmissions (connectivity, flow-control, reliability, etc.).

Yoid *distributedly* and separately constructs two topologies that are a *mesh* and a *tree*. The tree is optimized for efficiency (cost or low-latency), the mesh for robustness. The mesh topology is mainly used for control messages, tree-partition discovery and recovery, etc.

Group dynamics are addressed such that each group has one or more *rendevouz-points* associated. When a host wants to *join* a group, it contacts its rendevouz-point and receives an ID and a list of some of the current members. The node chooses one of the members and connects. Node *liveness* is explicitly checked by the rendevouz-point. When needed, the tree is refined for improved *latency and loss rate*.

NICE (is the Internet Cooperative Environment) [13] is a cooperative framework to scale multi-party applications.

NICE is based upon a *hierarchical* clustering of the members and can be used to produce a number of different data delivery trees. The members are self-organized into a layered topology, with nodes organized at each layer into clusters. Only the leader of each cluster is part of the next layer (also organized in clusters). Layer 0 contains all the nodes, while the last layer contains only one host, the rendevouz-point. For any host wishing to send data, its delivery path is a *source-specific* tree implicitly defined by forwarding each message to all other members of the cluster(s) it belongs to, except to the host the message was received from. The layered design helps it scale better, with a worst-case state and control overhead for any member of $O(logN)$.

Group dynamics are handled, such that each host maintains soft state about the other members of the cluster(s) it belongs to. A new member *joins* the group by, first, contacting the rendevouz-point, which replies with a list of the members that are part of the highest layer of the hierarchy. The joining host probes each of them, and selects the "closest" one in terms of *latency*. It contacts this host which replies with a list of the nodes in its cluster. This process repeats until the joining node finds the closest node to connect to. Slight modifications and improvements to this procedure are suggested. When a node *leaves* or *fails*, each of the remaining members of the cluster independently selects a new leader, based on a core-node selection algorithm [75].

Application Level Multicast Infrastructure (ALMI) [101] is a *centralized* overlay multicast protocol targeted for applications with a large number of groups, each with relatively few members.

ALMI uses a rendevouz-point or *session controller* to calculate a *minimum spanning tree*. It ensures the efficiency of the multicast tree by periodically calculating a minimum spanning tree based on the measurement updates received from the members. These measurements are performed by members that each monitor a set of other members and report back to the session controller.

Group dynamics are handled by the session controller, which controls the member registration and maintains the multicast tree. A node that wishes to *join*, contacts the session controller and receives the address of the parent node in the tree. Similarly, when a node *leaves* it contacts the session controller. Node failure is not discussed.

Bullet [81] is a multicast protocol designed for peer-to-peer streaming applications of audio and video.

Bullet constructs a *source-specific mesh-first* based data dissemination overlay. The authors argue that tree-based dissemination overlays may not be the best when high bandwidth is desired. Overlay trees limit the bandwidth, are hard to optimize and lead to high overhead due to probing. Therefore, Bullet uses a mesh instead of a tree. Bullet starts with any root-based overlay tree. The sender selects a (pseudo) random subset of its children and transmit disjoint data sets to them. The same process is done by the children as the data propagates down the tree. Nodes then obtain the missing parts of the data from other peers. Bullet only works when the transport protocol is TCP-friendly.

Group dynamics are not addressed.

Topology Aware Grouping (TAG) [83] is a *distributed* overlay multicast protocol targeted for applications with a large number of members. The authors state that TAG works best with applications which regard delay as a primary performance metric and bandwidth as a secondary metric.

TAG is *source-specific* and a *tree-first* multicast protocol, which aims to construct trees that are optimized for network delay with loose bandwidth constraints. Furthermore, it is *topology aware* and contains algorithms that detect the underlying router topology, and the multicast-tree is optimized with this in mind.

Group dynamics are handled by fairly naive mechanisms. Each new member determines the path from the root of the session to itself, and uses path overlap information to determine its parent and children. Whenever a node *leaves* or *fails*, this information is used to re-connect the tree.

Tmesh [137] is a *distributed* overlay multicast protocol that adds a level of resilience to the overlays it creates. The applications Tmesh targets are latency sensitive applications that require low pair-wise latencies.

Tmesh is a *tree-first* protocol and uses the *source-specific* Dijkstra's shortest path algorithm to discover *shortcuts* and add edges to the tree, which is then transformed to a *mesh*. The shortcuts are chosen to reduce the relative delay penalty and also the average pair-wise latencies. In addition, Tmesh yield a configurable level of resilience to the meshes.

Group dynamics is not discussed in a large extent. However, Tmesh uses a mesh to distribute events, and needs only recover from mesh partitions, but the authors do not address the mesh partitioning problem.

AMcast [116, 115] uses and assumes that it has a set of distributed multicast service nodes. The paper focuses on optimizing the access bandwidth of the multicast service node's interfaces and end-to-end delay, and proposes several new tree algorithms.

The tree algorithm types AMCast introduces are: 1) The compact tree algorithm is a minimum diameter degree limited spanning tree heuristic. 2) The balanced compact tree algorithm is a limited diameter residual balanced spanning tree heuristic, and is a slight adjustment of the compact tree algorithm. 3) The paper also introduces several algorithms to achieve best possible residual degree balance.

The proposed algorithms perform well in terms of creating low-latency overlay networks, but suffer due to somewhat high time complexity. The algorithms are implemented as centralized algorithms, and the paper does not discuss how they can be distributed.

Group dynamics are not addressed.

3.2.7 Conclusions, findings and shortcomings

These proposed multicast protocols are starting points, but they use overlay network construction algorithms that are either shortest-path or minimum-spanning trees. Hence, it is not sufficient to address the latency demands in distributed interactive applications.

One exception is AMcast [116,115], which (as mentioned) uses a set of distributed multicast service nodes. In addition, they propose two diameter optimizing tree algorithms, compact tree and balanced compact tree that are both evaluated in chapter 9. Furthermore, the multicast service node placement problem is strongly linked to selecting core-nodes using a core-node selection algorithm, which is evaluated in chapter 7. Another exception is Tmesh [137], which adds shortcut edges to pre-constructed trees. Chapter 11 evaluates a range of such shortcut selection strategies.

Furthermore, few, if any, protocols are able to maintain subsets of a larger set of nodes. An approach that looks at the maintenance of subgroups within a larger set of overlay nodes is PartyPeer [87]. This system creates subgroups by forming overlay multicast groups as subtrees of a tree that covers the entire set. However, the approach taken results in poorer performance, because subgroups are always created as subtrees of a single tree for the entire application.

Some of the multicast protocols support dynamic tree manipulations [145]. Typical operations include insert and remove, and some allow online rearrangement of the multicast tree [86]. An insert operation is typically performed using the shortest path [16, 4], or the delay constrained minimum cost path [103,6]. This is cheaper than tearing down the tree and re-building it from scratch, but it will probably lead to a larger tree diameter. Chapter 12 investigates a wide range of such insert and remove (dynamic) tree algorithms.

ACTIVE is a system [86] that distinguishes between clients that are actively interacting and those who are passive spectators. The ACTIVE system uses tree structures for communication among peer nodes, where the communication structure adapts itself depending on how active a peer is. For example, active clients that communicate have a smaller latency between each other than passive spectators that do not communicate. If a peer changes from an active client to a passive spectator role and back again, it also means a reconfiguration of the communication structure.

In summary, there is a considerable body of work on overlay multicast protocols and efficient tree construction and maintenance. However, current approaches do not address frequent group membership changes and resource limitations of a node (degree) while at the same time minimizing the diameter for latency-bound communication.

3.3 Core-node selection algorithms

An important group management technique is to elect one or more nodes to administrate clients that join and leave groups. When a limited set of nodes handles the membership management, it simplifies membership updates, and applications that have highly dynamic groups require fast and simple group management. Names given to such nodes include leaders, core-nodes or rendevouz-points [123]. Typically, the core-node of a group is contacted for each membership change, such that it always has the up-to-date (membership) information. Protocols with core-nodes are often defined as core-based multicast protocols [75, 47].

Core-based multicast protocols work on the assumption that one or more core-nodes are selected as group management and forwarding nodes. These protocols need some core-node selection algorithms to select the core-nodes. Several core-node selection algorithms have been proposed, and a comprehensive study is given by Karaman and Hassanein [75]. An overall goal is to select core-nodes on the basis of certain node properties, such as, bandwidth and computational power. Distributed interactive applications benefit particularly on core-node selection algorithms that consider latency, but also the degree limitations of the available core nodes.

There are existing core-node algorithms that find well-placed nodes in networks that yield low pair-wise latencies among some set of target-nodes [17]. However, we have not found any study that investigates their efficiency in a dynamic scenario, where sub-groups of clients are dynamically changing.

3.4 Spanning-tree algorithms

Spanning-tree algorithms build trees that span all the nodes in a network. A tree is a connected acyclic network, typically built on top of a larger network. A spanning-tree is a sub-network of the larger network that spans all the nodes. The advantages of a tree are that it has small routing tables and saves network bandwidth. In addition, it yields low administration costs when the membership is dynamic. However, the pair-wise latencies do increase, and if a non-leaf node is disconnected from a tree, the tree is also disconnected resulting in two subtrees.

The most famous and oldest spanning-tree algorithms are the shortest path tree (SPT) and minimum spanning tree (MST) algorithms. An SPT algorithm constructs a tree that has p shortest paths from a given source node to the destinations, where p is the number of destinations. The most famous SPT algorithm is the $O(n^2)$ Dijkstra's SPT algorithm [58]. An MST algorithm constructs a tree of minimum total cost, where the total cost is the sum of all the link weights in the tree. MST algorithms are proposed by Prim, Kruskal and Sollin [58, 93], all of which are $O(n^2)$. Figure 3.4 A) and B) illustrates an SPT and an MST on a network. A more recent algorithm is the minimum diameter spanning tree (MDST) algorithm. An MDST algorithm builds a spanning-tree in which the diameter is minimized, where the diameter is the longest shortest

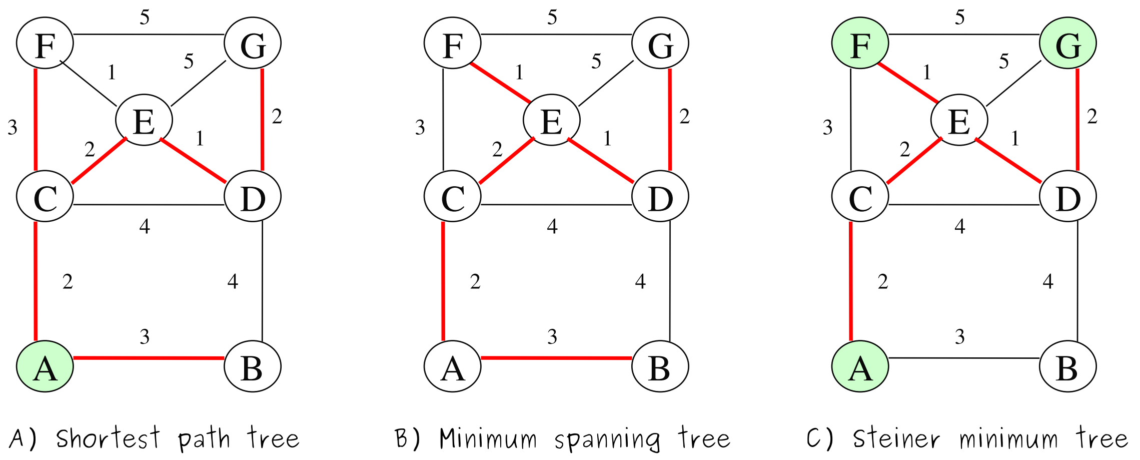


Figure 3.4: Examples of a shortest path tree, minimum-cost spanning tree and a Steiner minimum-cost tree.

path in the tree, also called the maximum pair-wise latency. Ho, Lee, Chang and Wong [68] proved that an optimal MDST has one or two nodes that are connected to the remaining nodes.

Many spanning tree algorithms add constraints to the tree construction [144], where the most common constraints are variations of delay bounds and degree limitations [80, 64]. The delay bounds are typically added in algorithms that optimize for the total cost [110,78,77]. The delay bounds may be from a given source, in which case, the tree is considered a source-tree [80, 64]. The delay bound may also express the maximum allowed diameter in the tree [116]. Degree limitations are added to bound the (forwarding) stress on each node in the tree [95].

The constraints are important to control and tune the tree construction. For example, a delay bound ensures that the latencies are controlled while the total cost is minimized, and degree limitations ensure a controlled stress level on each node. However, generally the constraints make the tree construction harder to solve, such that the tree construction takes longer. Many constrained tree construction problems are NP-complete [131].

From these algorithms wee observe that there are existing spanning-tree algorithms that construct desirable low-latency networks, but they have not been thoroughly examined and compared. It is especially uncertain how their behavior is in an environment of multiple dynamically changing subgroups.

3.4.1 Degree limited algorithms

The degree limited algorithms limit the number of incident edges to a node. Finding an optimal degree limited tree is NP-complete for both MST and SPT. Hence, polynomial time heuristics are needed for practical implementations. d-MST algorithms found in the literature often employ Prim's algorithm as the algorithm base. Knowles and Corne [79] listed three methods that Prim's algorithm could be used: 1) Use an alteration of Prim's algorithm so that it does not add edges that violate the degree constraint, 2) A dual simplex approach, begin with an MST and alternate until there are no degree violations, 3) Perturb Prim's algorithm through the use of an input solution vector, so that it does not always choose the minimum weight edge.

3.4.2 Delay constrained algorithms

The delay constrained algorithms optimize the network hop-count, and search for source-destination paths within the input delay limit. A delay constrained path may not always exist if the delay constraint is too strict/stringent. Three main solutions can be found in the literature today: 1) Renegotiate delay constraint, and rebuild tree, 2) Iteratively increase delay constraint until all nodes are covered, 3) Include uncovered nodes using unconstrained paths.

1 and 2 do not guarantee a solution on the first delay constraint increase. Solution 1 rebuilds the tree for each increase, hence, it may be forced to rebuild several times. Solution 2 is much faster, it increases the delay constraint and continues to build the tree. However, solution 3 is the fastest, because it includes all uncovered nodes using unconstrained paths.

3.5 Dynamic tree algorithms

The client dynamics in distributed interactive applications require algorithms that support online configurations of event distribution paths. Dynamic tree algorithms comprise such algorithms that reconfigure an existing tree by inserting and removing nodes [145], and also algorithms that only reconfigure the tree. Waxman and Imase were the first to address the problem of updating a multicast tree online, and referred to the problem as the online multicast tree problem [140]. The main goals of the dynamic tree algorithms in distributed interactive applications are to reduce the tree reconfiguration time, ensure that few links are changed during a tree update, and ensure that the latency requirements are met.

In the literature an insert operation is typically performed using the shortest path [44, 18, 16, 4] or the delay constrained minimum cost path [6, 103, 8]. A leaf node is trivially removed, and deleting a node with a degree of 2 results in two subtrees that may be reconnected using the least cost path [140] or the least cost delay bounded path [6]. Existing literature has not, in any particular degree, addressed the removal of nodes with a degree > 2. Such nodes are often kept in the tree for routing purposes, however, this may degrade the trees [60]. Furthermore, it is critical that tree reconfiguration is fast, in particular node insert operations, because all nodes must receive the data on time. It is also vital that remove operations keep the multicast tree intact for all remaining nodes [103]. Rearranging the tree while the data is flowing may cause members to loose data if the operation is not handled appropriately.

Many of these dynamic-tree algorithms have not been properly compared against each other. In adddition, as far as we can see there are no study that investigates the long-term effects of applying dynamic-tree algorithms to a scenario in which there are multiple dynamically changing subgroups of clients.

3.6 Steiner-tree algorithms

Section 3.4 introduced spanning-tree algorithms to be algorithms that build trees that span all the nodes in the network. Steiner-tree algorithms, on the other hand, applies two alternative node-states to the nodes in the network. The node-states have been given different names in the literature. For example, if a node is taking part in a multicast session, the node has one of these states: terminal, member or z-node. If it is not in the multicast session, it has one of these states: non-terminal, non-member or Steiner point. The Steiner-points in a network are nodes that forward data without reading it or contributing with new data. They are there to help the Steiner-tree algorithm achieve its optimization goal. For example, the Steiner minimum-cost tree (SMT) algorithms aim to reduce the total cost of the network. Hence, an SMT algorithm differs from an MST algorithm in that it can choose to use Steiner-points in the networks to minimize the total cost of the tree. Figure 3.4 C) illustrates an SMT on a network.

3.6.1 Steiner minimum-cost tree in networks

The problem of finding a Steiner minimum-cost tree in networks (SMT) is an NP-complete problem that was originally formulated independently by Hakimi and Levin [142] in 1971. Several exact algorithms and heuristic have been suggested, implemented and compared [134, 69]. In the following we introduce four Steiner-tree heuristic classes called spanning-heuristics, path-heuristics, tree-heuristics and vertex-heuristics [142].

The simplest SMT-heuristics are the *spanning-heuristics* [36]. They apply an MST algorithm on a network, and then remove (prune) Steiner-points with degree one (leaves). The MST algorithm generates a minimum-cost spanning tree of a network graph, spanning all the nodes. An approximate Steiner-tree is then obtained by removing subtrees containing only Steiner-points, from the MST.

A Steiner-tree *path-heuristic* starts from a pre-chosen source and includes the member-nodes one by one, until the tree spans all member-nodes. Typically, the tree grows based on the addition of (shortest) paths between member-nodes in the tree and member-nodes not yet in the tree. SPH is an SMT path-heuristic and was suggested by Takahashi and Matsuyama [120].

The Steiner-tree *tree-heuristics* are based on the idea of constructing an initial tree that spans all member-nodes, and then optimizing it towards a close-to-optimal SMT. Commonly, a minimum spanning tree variant is used to obtain the initial tree. DNH is an SMT tree-heuristic and was suggested, among others, by Kou, Markowsky and Berman [82].

The general idea behind Steiner-tree *vertex-heuristics* is to identify "good" non-member-nodes (Steiner points). It has been shown [142] that one big difficulty of the SMT problem is to identify non-member-nodes that belong to an SMT. Once the Steiner points are given, the SMT is an MST for the subnetwork induced by the member-nodes and selected Steiner points [142]. ADH is an SMT vertex-heuristic and was suggested by Rayward-Smith [106].

3.6.2 Steiner-tree algorithm variations

In the literature, there are numerous Steiner-tree heuristics that are only slight variations of the SMT heuristics presented in section 3.6.1. Generally, the SMT-heuristics [82, 106, 120] do not meet the latency requirements of distributed interactive applications. However, many similar heuristic variations exist that obey a latency bound from a given source, while optimizing for the total cost [45,72,103,100,145,7,10,16,102]. These variations often base the edge selection on a mix between path latency from the source and minimum cost edges.

Few [16, 111] of these proposals discuss how the source trees are to be applied in an interactive application. One way is to have one source-tree for each source, which increases the network cost, another, is to create a tree as a shared-tree that is shared by all the nodes to both send and receive data. A few proposals have addressed the requirements of a shared-tree. These heuristics have attempted to reduce the diameter in trees, where the diameter expresses the maximum path latency [65, 3, 150, 21]. The proposals use Steiner-tree heuristics that uses shortest paths between the nodes. However, they do not take into account the inherent fully meshed application layer network made of shortest paths.

3.7 Mesh-construction algorithms

A mesh construction algorithm differs from a tree construction algorithm in that it is not bound to constructing a tree, it is rather allowed to produce a connected cyclic graph (mesh). Compared to a tree, a mesh increases the node failure tolerance of the graph, because multiple paths to a node exist [90]. Unless some path routing is applied to a mesh it introduces data redundancy because some nodes receive two copies of the same data. Multiple paths are also valuable in cases of fluctuating link costs and to reduce the pair-wise latencies.

A mesh is mostly used in a multiple source scenario. For a single source scenario a mesh is only used if data redundancy is necessary because of unstable network nodes and/or links.

Existing mesh construction algorithms focus on reducing the pair-wise latencies by adding a configurable amount of edges to the mesh. The mesh construction algorithms can be divided into the three categories: A) Interleaved-trees, B) Enhanced tree, and C) Edge pruning (removal and addition) algorithms.

Interleaved-trees algorithms compute multiple connected trees and merge them into one mesh. The interleaved-trees algorithms may also be referred to as k-trees algorithms. Young et al. [147] described such an algorithm called k-MST, which computes k minimum spanning trees that are merged into one mesh.

Enhanced tree algorithms apply a tree algorithm to an input graph, and then add single edges to the tree based on some criteria. Wang et al. [137] described such strategies, and proposed an overlay protocol called Tmesh, which adds "short-cut" edges to a pre-constructed tree. There are also edge-selection strategies that aim at reducing the pair-wise distances in an overlay. The average pair-wise distance of a node is the sum of the shortest path distances between it and every other node, divided by the number of nodes. Both Narada [49] and Tmesh [137] focus on reducing a node's pair-wise distance.

Edge pruning algorithms include strategies that remove edges from an input graph based on some goal, and also algorithms that pick single edges from an input graph and constructs a mesh. These two approaches are essentially different, however, the algorithms share the same goal. Consequently, we call all of them edge pruning algorithms. The simplest edge pruning algorithm is to add a number of edges randomly to the mesh. Yoid [50] applies this method, with the added step of applying a routing protocol atop of the mesh. In the Narada [49] protocol a node joins a random peer and then slowly moves to more favorable peers as they are discovered.

Although there are many approaches that use meshes, especially multicast protocols. There are few studies that focus on algorithmic properties of mesh construction, and how to efficiently construct meshes of low diameters, while still preserve a low network cost of the mesh. In addition, we did not find a complete comparison of such algorithms in a dynamic scenario.

3.8 Latency estimation techniques

The round trip times (RTTs) between hosts in the Internet may be obtained by using readily available tools like Ping and Traceroute. These tools estimate a single path's end-to-end latency, and are well-suited for smaller groups of hosts. However, an all-to-all measurement incurs an $O(n^2)$ traffic growth [9], and in our target applications this poses a scalability problem because the number of clients participating may be in the thousands. The clients in these applications may form dynamic subgroups, which makes it important to retrieve the link latencies among the clients, in order to organize the subgroups intelligently in an overlay network.

In such scenarios, it is currently not scalable to use Ping or Traceroute for active probing of link latencies. Instead, latency estimation techniques that reduce the probing overhead may be applied. Estimation techniques have been classified into three classes in [42], 1) Landmarks-based latency estimation, 2) Multidimensional-scaling based latency estimation and 3) Distributed network latency database.

Landmarks-based latency estimation techniques give each client a point in a metric space, and aim to predict the latency between any two clients. They use landmark nodes, which are a set of nodes used by the remaining nodes as measurement references for their relative position in the network. Such techniques include Netvigator [113], NetForecast, Global Network Positioning (GNP) and Practical Internet Coordinates [41].

Multidimensional-scaling based latency estimation techniques do not involve landmark nodes. They use statistical techniques for exploring similarities and dissimilarities in data. For example, a matrix of item-item similarities is used to assign a location for each item in a low-dimensional space [31]. Examples of such techniques include Vivaldi [34] and Big Bang Simulation technique [41].

Distributed network latency database techniques use active measurements to build a knowledge base about the underlying network. These approaches have been designed to efficiently answer queries of the form: Who is the closest neighbor to node A in the network? Since these schemes are based on direct measurements they have better accuracy, however they do also inject more traffic into the network compared to the landmarks-based and multidimensional-scaling based techniques. Meridian [143] is such a technique.

Among these techniques we find that the landmarks-based and multidimensional-scaling based latency estimation techniques are approprite for finding all-to-all path latencies. Distributed network latency database techniques, on the other hand, are not desirable for our target area, because they are not designed to retrieve all-to-all link latencies. They are, however, desirable for leader election scenarios, and discovering the closest neighbor for a node.

3.9 Summary of the main points

The previous surveys aimed at identifying existing work that address some or preferrably all of the problems in distributed interactiv applications. To do this we surveyed a range of research areas and brief conclusions were given for each survey.

We found that in the area of overlay multicast protocols there are many approaches that have interesting ideas that are worth adopting and investigating in a dynamic application scenario. For example, two latency optimizing spanning-tree algorithms from the authors of AM-cast [116, 115], compact tree and balanced compact tree, are both evaluated in chapter 9. Furthermore, ideas from Tmesh [137], which investigated including shortcut edges to pre-constructed trees, are evaluated in chapter 11. The group management approach from the centralized ALMI [101] may also prove valuable in a scenario with multiple dynamically changing subgroups, and chapter 5 evaluates similar group management approaches through empirical studies and experiments.

For an overlay network construction we found several suitable spanning-tree and Steiner-tree algorithms [116, 100, 16]. However, we were unable to find large comparative studies that identified algorithms that are suitable for distributed interactive applications. Although Shi et al. [116] did propose and study some spanning-tree algorithms that are likely to be suitable, they did not fully study the effects in a dynamic application scenario. Chapter 9 and 10, respectively, give comparative studies of a wide range of spanning-tree and Steiner-tree algorithms. A similar study for subgraph (mesh) construction algorithms is conducted in chapter 11.

We also found several dynamic-tree algorithms that support dynamic tree manipulations [145], where typical operations included insert and remove of single nodes, and also online rearrangement of multicast trees [86]. This functionality is exactly what is needed in a dynamic application scenario. However, we did not find studies that investigated how the dynamic tree algorithms perform when the group sizes vary and multiple reconfigurations are conducted. Chapter 12 performs such a study and includes a wide range of dynamic tree algorithms, and chapter 13 investigates dynamic subgraph algorithms. Furthermore, chapter 14 investigates algorithms for online rearrangements that combine dynamic tree algorithms and Steiner-tree algorithms, and also dynamic subgraph algorithms and Steiner subgraph algorithms.

Furthermore, to retrieve all-to-all path latencies we need measurement or estimation techniques, and in our survey we found such latency estimation techniques [113, 34]. Evaluations are found in chapter 6. However, we did not find any work that investigates the interoperability between latency estimation techniques and a centralized group communication approach that includes core-node selection algorithms and spanning-tree and dynamic tree algorithms. Chapter 7 and 15 aims to highlight some of the unanswered questions regarding their cooperation.

In summary, the main drawback of existing related work is the lack of large comparative studies that investigate the effects of algorithms in a dynamic application scenario. In addition, many studies do not take into account all the necessary performance measurements in order to quantify their results sufficiently. This thesis aims to present such comparative studies using the performance measurements that are necessary for a sufficient study.

Before the comparative studies, we present a wide range of graph theoretical overlay network design problems, and also some algorithmic terms and performance measures.

Chapter 4

Overlay network design: Problems in graph theory

Chapter 2 introduced the interactivity and dynamicity as features of distributed interactive applications that require low-latency networks for application-specific event distribution. The problem of constructing such low-latency networks has been studied for many years using graph theory.

Graph theory is the study of graphs, which are mathematical structures that consist of objects from a given collection and their relational characteristics.

The origin of graph theory can be traced back to Euler's work on the Konigsberg-bridges problem (1735), which subsequently led to the concept of an Eulerian graph [52]. The concept of a tree, a connected graph without cycles, appeared implicitly in the work of Gustav Kirchhoff (1824-87), who employed graph-theoretical ideas in the calculation of currents in electrical networks or circuits. Later, Arthur Cayley (1821-95), James J. Sylvester (1806-97), George Polya (1887-1985), and others use 'tree' to enumerate chemical molecules.

In our work, we use low-latency overlay networks to multicast application events through. Therefore, we introduce a range of overlay network design problems found in graph theory that address low-latency issues. Generally, graph theoretical problems are used to formally describe problems that are applicable to real-world issues. The problem-area in the thesis is distributed interactive applications for real-time interaction, but graph theory can be used in many other application settings.

There are many advantages by formally describing real-world problems using graph theory. One advantage is that it makes it easier to discuss and compare algorithms when they can be linked to specific graph theoretical problems that are independent of the algorithms solving it. In addition, many graph theoretical problems are NP-complete, that is, no polynomial time algorithm exists. Therefore, it is important to identify these inherently difficult problems, such that alogrithm designers can focus on solving them approximately.

These observations are the motivation for why we present the overlay network design problems in this chapter. They can be used by overlay network designers that are interested in identifying how hard their problems are, and whether there are similar problems that are simpler to solve or approximate. In the context of the thesis, we use the overlay network design problems to discuss and compare the algorithms we evaluate in later chapters.

The rest of the chapter is organized in the following manner. Section 4.1 introduces the graph theoretical terms and symbols that we use in the thesis. Section 4.2 presents relevant algorithmic terms and a few graph algorithmic metrics. Section 4.3 introduces some of the most basic graph algorithms for graph search and overlay construction, namely, depth-first search, breadth-first search, Dijkstra's shortest-path tree and Prim's minimum-cost spanning-tree. Section 4.4 introduces a range of graph search problems found in graph theory. Then, we introduce a wide range of graph theoretical overlay network design problems from section 4.5 to 4.12. The focus is particularly on graph theory problems for constructing trees and subgraphs that are optimized for minimum cost and latency. The problem areas are spanning-tree and -subgraph problems, Steiner-tree and -subgraph problems, dynamic-tree and -subgraph problems and various other graph theory problems that address overlay network design issues.

4.1 Graph theoretical terms and symbols

Graph theoretical terms and symbols are introduced in the following. The terms and symbols are used in later chapters to introduce specific algorithms, and explain their behavior.

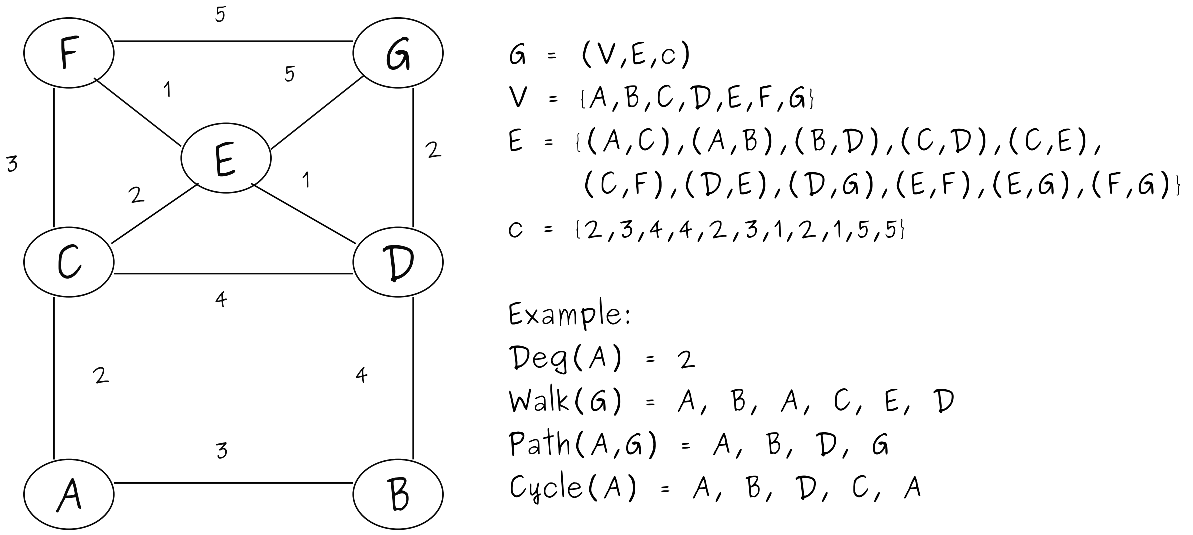
4.1.1 The graph structure

Graph theoretical algorithms operate on a graph structure G holding a set of nodes (vertices) V and links (edges) E. The graph structure may, for example, be a virtual picture of a network, with vertices scattered around and edges connecting them.

Definition 1 ***Graph structure:*** *A graph $G = (V, E)$ contains a finite non-empty set V of vertices and a set E of unordered pairs of distinct vertices of G called edges. An edge $e \in E$ which holds the vertex pair $u, v \in V$ makes $e = (u, v)$.*

A graph G can be either *directed* or *undirected*. If G is directed it is common to refer to E as a set of *arcs*, where an arc is an edge with a single direction. If G is undirected the set E contains a set of undirected edges, which is equal to each edge being bi-directional (goes both ways).

Definition 2 ***Vertex degree:*** *The degree, $deg(v)$, of a vertex $v \in V$ is the number of graph edges $e \in E$ that touch v. $deg(v) = |E_{sub}|$, where E_{sub} holds the edges $e = (v, u) \in E$ that includes $v \in V$.*



G = (V,E,c)
V = {A,B,C,D,E,F,G}
E = {(A,C),(A,B),(B,D),(C,D),(C,E),
(C,F),(D,E),(D,G),(E,F),(E,G),(F,G)}
c = {2,3,4,4,2,3,1,2,1,5,5}

Example:
Deg(A) = 2
Walk(G) = A, B, A, C, E, D
Path(A,G) = A, B, D, G
Cycle(A) = A, B, D, C, A

Figure 4.1: A graph structure $G = (V,E,c)$, and examples of degree, walk, path and cycle.

Two vertices are adjacent if they share a common edge, for example, $u, v \in V$ are adjacent if $e = (u,v) \in E$. The set of neighbours, $N(v)$, of a vertex v is the set of vertices which are adjacent to v. The degree of a vertex is also the cardinality of its neighbour set.

Definition 3 ***Walk, trail, path and cycle:*** *A walk in a graph $G = (V,E)$ is a sequence of vertices $v_0, v_1, \dots v_n$, $0 < i < n$, such that (v_i, v_{i+1}) are all edges in G. The length l of a walk is $l = n-1$. A trail is a walk in which no edge is repeated. A path is a walk in which no vertex is repeated. A cycle is a walk in which $v_0 = v_n$ and no other vertex is repeated.*

A *walk* is an alternating sequence of vertices and edges, with each edge being incident (attached) to the vertices immediately preceeding and succeeding it in the sequence. A *trail* is a walk with no repeated edges. A *path* is a walk with no repeated vertices. A walk is closed if the initial vertex is also the terminal vertex. A *cycle* is a closed trail with at least one edge and with no repeated vertices except that the initial vertex is the terminal vertex. The length of a walk is the number of edges in the sequence defining the walk. Thus, the length of a path or cycle is also the number of edges in the path or cycle.

Definition 4 ***Weighted graph structure:*** *A weighted graph $G = (V,E,c)$, contains a set V of vertices, a set E of edges, and an edge cost function $\mathrm{c} : E \rightarrow \mathbb{R}^+$ (only positive reals are used throughout the thesis). Each edge $e \in E$, has an associated cost $\mathrm{c}(e)$.*

Figure 4.1 shows a graph $G = (V,E,c)$ and illustrates what G consists of. In addition, the figure has a few examples of degree, walk, path and cycle.

4.1.2 Graph distances and locations

There are many different ways for defining distances, locations, costs, etc, in graphs. The next descriptions are formal definitions of the ones that are used in the thesis.

Definition 5 ***Total (graph) cost:*** *The cost of a weighted graph G is the sum of all the edge-costs in the graph, $cost(G) = \sum_{e \in E} c(e)$.*

Definition 6 ***Distance:*** *The distance between $u \in V$ and $v \in V$ is denoted $dist(u,v)$, and is the minimum cost of any path from u to v.*

Definition 7 ***Average (pair-wise) distance:*** *The average (pair-wise) distance is given by the arithmetic mean of all distances in the graph $average(G) = \frac{1}{(|V|^2-|V|)} \times \sum_{u \neq v \in V} dist(u,v)$.*

Definition 8 ***Eccentricity:*** *The eccentricity of $u \in V$ is the maximum value of $dist(u,v)$, for all $v \in V$, and it is defined $ecc(u) = max\{dist(u,v) : v \in V\}$.*

Definition 9 ***Radius:*** *The radius of G is the minimum eccentricity among the vertices of G, and it is defined $radius(G) = min\{ecc(u) : u \in V\}$.*

Definition 10 ***Diameter:*** *The diameter of G is the maximum eccentricity among the vertices of G, and it is defined $diameter(G) = max\{ecc(u) : u \in V\}$.*

Definition 11 ***Center:*** *The center of G is the set of vertices of eccentricity equal to the radius, and it is defined $center(G) = \{u \in V : ecc(u) = radius(G)\}$.*

For any graph G the diameter is at least the radius, and at most double the radius.

A tree has a center set of size one or two. If it contains one vertex, the tree is called monopolar or central. If it has two vertices, the tree is called dipolar or bicentral. For a general graph, there may be several centers and a center is not necessarily on a diameter.

Definition 12 ***Absolute 1-center:*** *The absolute 1-center of G is the point u minimizing the function $f(x) = max_{v \in V} dist(u,v)$.*

The *absolute 1-center* may be any *point* in the graph. Different from the graph model usually used, any interior point in addition to the two endpoints (vertices) of an edge is referred to as a point in the graph.

Definition 13 ***Median:*** *The median vertex of G is the vertex with minimum total distance to all vertices. The median vertex minimizes the function $f(v) = \sum_{u \in V} dist(v,u)$.*

Figure 4.2 illustrates some examples of distances, locations, costs, etc, for an example graph $G = (V,E,c)$.

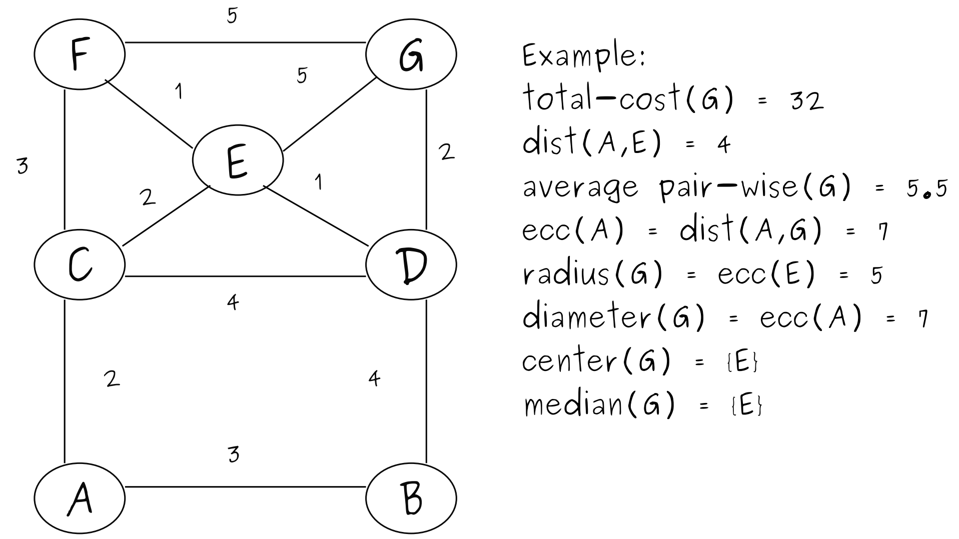


Figure 4.2: A graph structure $G = (V, E, c)$, and examples of distances.

4.1.3 Graph connectivity and topologies

Graph connectivity is one of the basic concepts of graph theory, and is closely related to the theory of network flow problems [136]. A graph's connectivity is an important measure of its robustness as a network.

Definition 14 ***Connected vertices:*** *Two vertices $u, v \in V$ are connected if there is a path from u to v in G.*

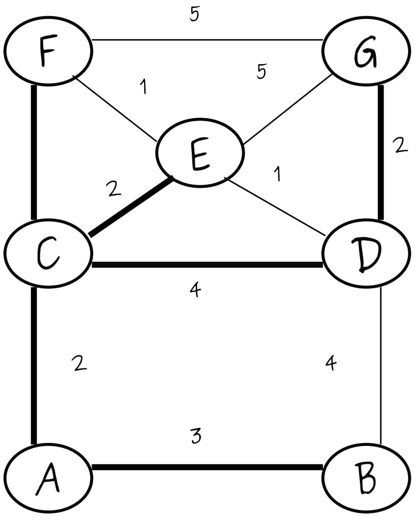
Definition 15 ***Connected graph:*** *A graph is connected if every pair of distinct vertices in G are connected.*

Definition 16 ***Connected acyclic graph:*** *A connected acyclic graph is a connected graph in which there are no cycles.*

Definition 17 ***Connected cyclic graph:*** *A connected cyclic graph is a connected graph in which there are cycles.*

Definition 18 ***Vertex-cut and vertex-connectivity:*** *A vertex cut of a graph $G = (V, E)$ is a set of vertices whose removal renders G disconnected. The vertex-connectivity $K_V(G)$ is the size of a smallest vertex cut.*

Definition 19 ***Edge-cut and edge-connectivity:*** *An edge cut of a graph $G = (V, E)$ is a set of edges whose removal renders G disconnected. The edge-connectivity $K_E(G)$ is the size of a smallest edge cut.*



Given a graph G=(V,E,c) (see figure 4.1).
G is a connected cyclic graph.
G is a connected spanning mesh.
All vertices in V are connected vertices.
Removal of C and D renders G disconnected.
Removal of e={C,D} renders G disconnected.

Given a subgraph G'=(V',E').
E'={(A,B),(A,C),(C,D),(C,E),(C,F),(E,G).
G' is a connected acyclic graph.
G' is a spanning tree.

Figure 4.3: A graph structure $G = (V, E, c)$, and examples of connectivity.

Definition 20 ***Connected spanning subgraph:*** *A connected spanning subgraph $G' = (V, E')$ on $G = (V, E)$ is a connected graph in which there is a minimum of one path between all $v \in V$, and $E' \subseteq E$.*

Definition 21 ***Connected spanning tree:*** *A connected spanning tree $T = (V, E')$ on $G = (V, E)$ is a connected acyclic graph in which there is exactly one path between all $v \in V$, and $|E'| = |V| - 1$.*

Definition 22 ***Connected spanning mesh:*** *A connected spanning mesh $M = (V, E')$ on $G = (V, E)$ is a connected cyclic graph, in which there are paths between all $v \in V$, and $E' \subseteq E$.*

Definition 23 ***Complete graph:*** *A graph G is a complete graph (also called clique, and a full mesh) if every pair of vertices in V are adjacent.*

Definition 24 ***k-Connected graph:*** *A graph G is k-connected if for every proper subset Y of V with fewer than k elements, $G - Y$ is connected.*

A connected graph G has connectivity k if G is k-connected but not $(k + 1)$-connected. A complete graph on $k + 1$ vertices is defined to have connectivity k. A connected spanning tree is a 1-connected graph, in which the removal of any single inner node disconnects the graph. It is common to use the terms *sparse* and *dense* when describing the connectivity of a graph. For example, a connected acyclic graph is the most sparse graph possible, similarly, a complete graph graph is the most dense graph possible.

Figure 4.3 illustrates some examples of connectivity, etc, for the graph $G = (V, E, c)$ from figure 4.1.

4.2 Graph algorithmic terms and symbols

The performance of an algorithm $\mathscr{A}$ can be measured in terms of the quality of the outcome (result), the time it took and the space that was used. The performance of an algorithm is heavily dependent on its complexity in relation to the problem it is aiming to solve.

4.2.1 Algorithm complexity

The *complexity* of an algorithm $\mathscr{A}$ is an important indication of its execution costs, and includes time and space complexity. The *time* complexity is, for example, the number of steps n it takes to solve an instance of a problem as a function of the size of the input, using the most efficient algorithm. The *space* complexity is, for example, the amount of memory (space) the algorithm requires to finish.

Generally, a *Big O* notation is used to describe the upper-bound of the complexity, for example, $O(n)$, $O(n^2)$, $O(n^3)$, etc. For a specific algorithm, the Big O notation is often said to describe its worst-case complexity. In addition to the upper-bound, there is also the lower-bound, Big Ω notation, and the tight-bound, Big Θ notation. The lower-bound, Big Ω, is often used in theoretical studies of algorithms. It signifies the absolute lowest complexity that an algorithm can have in order to solve a given problem. However, this does not mean that such an algorithm exists. The tight-bound, Big Θ, is used to describe how the complexity will "normally" be. It is often used to describe how the complexity of an algorithm is in the average-case. When all of the complexity bounds are well-defined for a given algorithm, it is clearer how the algorithm behaves, and the algorithm is said to have *tight* behavioral bounds. If, for example, only the worst-case behavior is defined, the algorithm is defined with a *loose* behavioral bound.

The two most famous complexity classes are P and NP. If a problem is in P it is also in NP, but problems in NP are not always in P. Generally the relation is considered to be $P \subseteq NP$, but, one of the most famous open problems is whether $P = NP$. A problem belonging to complexity class P is solved in *polynomial time*, that is, an efficient algorithm exists. The complexity class *NP-complete* is a subset of NP. An NP-complete problem is considered to be a tough problem to solve, but it can be verified quickly. For example, an NP-complete problem is said to be solvable by a non-deterministic machine in polynomial time, in addition to being verifiable in polynomial time. Problems that are in NP but not in P, are yet to be solved by a polynomial time algorithm, but rather have *exponential time* complexity, that is, only inefficient algorithms exist. *Heuristics* are algorithms that are generally used to approximate close-to-optimal solutions for problems that are in NP but not in P.

4.2.2 Algorithm types

A *deterministic* algorithm is an algorithm for which every step of the algorithm, the next step can be deterministically predicted. A *non-deterministic* algorithm is an algorithm for which every step of the algorithm, multiple different next steps may be taken, and no exact specification for which one to take exists. Non-deterministic algorithms are often used in theoretical algorithm design as specifications for algorithms that do not care how they get to a valid solution, as long as they get there. Deterministic algorithms are often the practical implementations of problem specifications. However, exceptions do exist, for example, genetic algorithms [108].

Centralized algorithms are algorithms that run on a single entity, and the execution does not rely on cooperating with other entities to complete. For example, the single entity may be a computer that run and complete the algorithm "stand alone". *Distributed* algorithms, on the other hand, are run on multiple entities and must cooperate with other entities to complete. For example, distributed algorithms are often run on each (entity) computer in a computer network. Typically, the computers are triggered to start in an initial state and then each of them compute sub-solutions that are exchanged with their neighbors. The state of the sub-solution evolves based on the information in incoming messages and how the algorithm continues the calculation.

Both centralized and distributed algorithms may be implemented as a *sequential* or a *parallel* algorithm. Sequential algorithms are run in sequence such that the sub-solutions are merged deterministically as the algorithm evolves. Parallel algorithms run in parallel on multiple entities such that the sub-solutions are potentially merged non-deterministically as the algorithm evolves.

4.2.3 Algorithm choices

A graph algorithm makes choices at each step of the algorithm. The choices may, for example, be based on some calculated values from the current sub-solution.

Random algorithms do not calculate anything from the sub-solutions, they only base the decisions on the validity of the next step. There may be algorithms that are pseudo-random, in the sense that they have sub-routines that are random, and others that do some calculations to increase the probability for the random sub-routines to randomly choose a "good" next step.

Greedy algorithms always choose the local-optimum to be the next sub-solution. Greedy algorithms are fast and works well both as centralized and distributed algorithms. The property of choosing the local-optimum often lends itself to finding the global-optimum, for example, the well known tree problems minimum-cost spanning tree problem, and shortest-path tree problem (see section 4.5).

Dynamic programming algorithms break the problem into smaller subproblems, and solve these problems optimally, and then construct an optimal solution from the original problem.

These algorithms are often quite complex, because, figuratively, they calculate every local-optimum to retrieve the global-optimum.

4.2.4 Algorithm performance metrics

If an algorithm always solves a given problem, it is considered to be *exact*. However, many problems are very hard to solve, such that algorithms solve a problem *approximately*. Graph algorithms that return optimal solutions, given a problem, are only comparable by time and space complexity, because the solutions have identical properties. The algorithm performance metrics related to time and space complexity are described as:

- The *execution time* of an algorithm is the time it takes before a solution is available. It is important in time-dependant situations.
- The *execution space* of an algorithm is the space it requires to reach a solution. It is important when algorithms are executed in entities with limited memory.

Approximation algorithms are often called heuristics of a given problem, and are comparable in terms of the quality of the result. The *competetive ratio* C_R is a measure to how well a heuristic performs in terms of any given metric, in comparison to the optimal solution. For example, given an optimal solution T_{OPT} and a sub-optimal solution T, then the competetive ratio is $C_R = T/T_{OPT}$.

It is also common to compare the solutions of algorithms that address slightly different but related problems. For example, tree algorithms produce a tree from an input graph, but there are many different tree algorithm problems, thus, the resulting trees vary in terms of graph metrics.

There are quite a few performance metrics for graphs, where the most common are related to cost and degree. Cost is a collective (abstract) term used for edge weights in a graph $G = (V, E, c)$. It is often mapped from network latency or bandwidth, hop-count and euclidian-length. The cost term in graphs is mostly related to spanning-costs or path-costs.

- *Spanning-cost:* A spanning-cost is a graph metric which is calculated by summarizing edge-weights in no relative order, and are typically related to the graph cost.
- *Path-cost:* A path-cost is a graph metric which is calculated by summarizing edge-weights in a relative order, and are typically certain paths in a graph that exhibit a property.

Many common graph performance metrics are categorized as either a path-cost or a spanning-cost metric. In the following, we introduce many of the graph performance metrics that are addressed in the thesis.

- The *total cost* of a graph is the sum of all the edge weights. It is a spanning-cost and is used as a metric in algorithms that aim to minimize the network costs.

- The *diameter* of a graph is the maximum pair-wise cost between any two vertices. It is a path-cost and is used as a metric in algorithms that aim to minimize the network latencies.

- The *radius* of a graph is the minimum pair-wise cost between any two vertices. It is a path-cost and is used as a metric in algorithms that aim to minimize the source destination latencies/costs.

- The *average pair-wise cost* of a graph is the is the sum of all shortest paths among the nodes in a graph, divided by the number of nodes. It is a path-cost and is used as a metric in algorithms that aim to minimize the pair-wise latencies.

- The *shortest-path* cost is the shortest possible path between two vertices in a graph. It is a path-cost and is used as a metric in algorithms that minimize the source destination latencies/costs.

- The *relative delay penalty* is the cost/latency penalty of a sub-optimal path compared to the optimal path. Relative delay penalty is a path-cost metric and is common in routing problems, and subgraph problems.

- The *stress* (degree) of a vertex is the number of edges to or from a single vertex. It is common to measure the average degree of non-leaf nodes in a graph, and also the maximum degree of any node in a graph.

Algorithms that address problems of evolving graphs based on node- and edge-addition/removal have other metrics specifically related to the problems. For example, upon node- or edge-addition/removal the total cost of the graph changes.

- The *stability* in a graph is the number of edge changes that occur across node- and edge-addition/removal. It is used as a metric in scenarios in which there is client dynamics (churn).

- The *degradation* of a graph is related to the competetive ratio, and is how much a given graph metric differs from the optimal. It is used as a metric to study non-optimal heuristics.

4.2.5 Algorithm optimization goals

Graph algorithms take is input a graph G, does computation on it and tries to optimize towards a given algorithm optimization goal. A graph algorithm that construct a subgraph on a graph G often optimizes towards a minimum-cost subgraph, a shortest-path-cost subgraph, or a minimum-diameter subgraph. The problem of computing a minimum-cost subgraph is well known and one of the oldest problems in graph theory. One example is the compuation of minimum-cost spanning tree. Computing a shortest-path-cost subgraph is similarly well known and old, where one example is the computation of a shortest-path-cost tree. Other optimization goals include a minimization of distances between different sets of nodes in a graph. Later in the chapter, we introduce a wide range of graph theory problems that define specific optimization goals that must be addressed by algorithms.

4.2.6 Algorithm constraints

Algorithms that take several optimization metrics into account often do this by choosing one metric as its optimization goal, and then address the remaining metrics by adding constraints. In general, adding constraints to an algorithm increases the algorithm complexity if an optimal solution is targeted. The most common constraints for subgraph construction are degree, spanning-cost, and path-cost constraints:

- A *degree constraint* is related to how many edges a vertex is allowed to have attached to it in a graph.
- A *spanning-cost constraint* is related to how high a spanning-cost (for example, total cost) is allowed to become in a graph.
- A *path-cost constraint* is related to how high a path-cost (for example, diameter) in a graph is allowed to become.

A degree limited subgraph algorithm should build subgraphs where all vertices have a degree of less than or equal to its degree limit. Path-cost constraints are, for example, diameter and radius, while, spanning-cost constraints mostly is the total cost. When degree, spanning-cost and path-cost constraints are combined in one algorithm, it makes it complicated for an algorithm to find a solution that does not violate the constraints. For example, many constrained subgraph heuristics cannot guarantee that a constrained subgraph is found.

The *success rate* of an algorithm is the rate of which it is able to solve a problem given a graph instance. The success rate of an algorithm depends on the constraint and the input graph. If the constraints are *strict*, it is harder to solve a given problem, than if the constraints are *loose*. For some constraints, it is even impossible to solve a problem, for example, finding a spanning subgraph G' on a graph G such that the $diam(G') < diam(G)$ is impossible. Therefore, one major problem related to any constrained overlay network design problem is how the constraints are determined.

4.3 Basic graph algorithms

There are graph algorithms that are fundamental in their behavior, and have been sources of influence and adaptation through many years. The next few sections introduce some basic graph search algorithms and subgraph construction algorithms that are among the most basic graph algorithms in the literature.

4.3.1 Basic algorithms for graph search

The standard and most basic graph search algorithms are depth first search and breadth first search. Many other graph search and tree algorithms extend these algorithms.

Breadth first search (BFS) is an algorithm for traversing a graph $G = (V, E)$. Informally, the algorithm starts from a single source vertex $s \in V$ and explores the neighbors. Then for each neighbor, it explores the unexplored neighbors. The algorithm terminates when every reachable vertex in G from v has been explored. The time complexity of BFS is also $O(|V| + |E|)$. The breadth-first search algorithm is so named because it discovers all vertices at distance k from the source vertex s before discovering any vertices at distance $k + 1$.

The BFS algorithm is usually employed to find shortest paths and distances from a given source, but only in graphs with uniform edge weights. BFS disregards graph edge weights, but rather treats every edge as a single hop. Hence, it finds the shortest paths in terms of hops. The BFS algorithm also finds the hop-eccentricity of a node, and if BFS is executed from each node, it finds the hop-diameter.

Depth first search (DFS) is an algorithm for traversing a graph $G = (V, E)$. Informally, the algorithm starts from a single source vertex $s \in V$ and explores as far as possible along each branch, before backtracking. The algorithm terminates when every reachable vertex from in G from v has been explored. The time complexity of DFS is $O(|V| + |E|)$.

The DFS algorithm is usually employed as sub-routines in another algorithm. For example, the DFS is very efficient for finding the maximum shortest-path and distance (eccentricity) of a node in acyclic graphs (trees). Other applications, include topological sorting for directed graphs and finding strongly connected components in a graph [58].

Diameter search algorithm in graphs (cyclic and acyclic)

Algorithms that search for the diameter in graphs differ depending on whether the graph is cyclic or acyclic (mesh or tree). If the graph is cyclic, a diameter algorithm is more complex than if the graph is acyclic, that is, a tree. For cyclic graphs the diameter may be obtained by running Dijkstra's SPT from each node. For acyclic graphs the diameter is obtained by, first, finding the eccentricity path (longest shortest path) of a random node $v \in V$ using DFS, then, DFS is started from the node u that is farthest away from v. The eccentricity of u is equal to the diameter of the tree [144]. Algorithm 1 presents the algorithm for finding the diameter of trees.

Algorithm 1 TREE-DIAMETER(T):

```
In: A tree $T_r = (V, E, c)$ rooted at $r$.
Out: The diameter of $T$.
Root $T$ at an arbitrary vertex $r$.
Use DFS to find the vertex $v$ farthest from $r$. {eccentricity-path's end-point}
Root $T$ at $v$.
Use DFS to find the eccentricity of $v$.
return eccentricity of $v$; {as the diameter of $T$.}
```

4.3.2 Basic algorithms for overlay construction

Overlay construction algorithms build a subgraph on an input graph that contain a subset of the edges and vertices of the input graph. Most overlay construction algorithms are based on one or more fundamental graph algorithms, for example, breadth first search, that are adapted and given some additional logic. The following algorithms are presented because they are the the algorithmic foundation for the evaluated overlay construction algorithms in the thesis.

Three basic spanning-tree algorithms are Prim's minimum-cost spanning-tree (MST) algorithm, Kruskal's MST algorithm, and Dijkstra's shortest-path tree (SPT) algorithm [58]. These algorithms are the foundations of a wide range of subgraph construction algorithms. Prim's MST builds the tree starting from a given source, and for each iteration, it connects a vertex through the minimum cost path to the tree. Algorithm 2 presents an algorithm for Prim's MST. Dijkstra's SPT works similarly, it builds a shortest-path tree from a given source, and adds the next vertex that has the shortest path to the source. Algorithm 4 presents an algorithm for Dijkstra's SPT. Prim's MST algorithm and Dijkstra's SPT algorithm [58] use similar ideas to breadth first search. Kruskal's MST algorithm starts with a forest of trees that are merged through minimum-cost edges until there is only one left. Algorithm 3 presents an algorithm for Kruskal's MST. Chapter 9 introduces many spanning-tree algorithms that are based on ideas from Prim's, Dijkstra's and Kruskal's algorithms.

4.4 Graph search problems

A graph search problem typically includes a search criterion that should be fulfilled by searching in a graph G. The search criterion may be to search for one or more nodes or paths that exhibit certain properties, for example, return the graph diameter, return the node with lowest eccentricity, etc. Graph search problems are important in the search for well-placed nodes in

Algorithm 2 PRIM-MINIMUM-SPANNING-TREE(G, s):

```
In: A connected graph $G = (V, E, c)$, and a source $s \in V$.
Out: A connected spanning-tree $T = (V, E')$.
for each $u \in V$ do
    distance[$u$] = infinity
    parent[$u$] = NIL
end for
distance[$s$] = 0
$Q = V(G)$ {Minimum priority queue}
while $Q \neq \emptyset$ do
    $u$ = ExtractMin($Q$)
    for each $v \in V$ adjacent to $u$ do
        if $v \in Q$ and $dist(u, v) < distance[v]$ then
            parent[$v$] = $u$
            distance[$v$] = $dist(u, v)$
        end if
    end for
end while
return createTree($G$, parent, distance)
```

Algorithm 3 KRUSKAL-MINIMUM-SPANNING-TREE(G):

```
In: A connected graph $G = (V, E, c)$.
Out: A connected spanning-tree $T = (V, E')$.
$E' = \emptyset$
for each $u \in V$ do
    Make-Set($u$)
end for
sort the edges $e \in E$ into nondecreasing order by weight $c(e)$
for each edge $e = (u, v) \in E$, taken in nondecreasing order by weight $c(e)$ do
    if Find-Set($u$) $\neq$ Find-Set($v$) then
        $E' = E' \cup \{u, v\}$ {only merge disjoint sets}
    end if
end for
return $T$
```

overlay networks. Several core-node selection algorithms that address these problems are introduced and evaluated in chapter 7. The core-nodes in that chapter are used as administrators for groups of clients. Therefore, it is important to reduce the latencies from the clients to the core-nodes. The following graph search problems are related to vertex search in graphs.

Definition 25 ***Minimum k-center problem:*** *Given a weighted graph $G = (V, E, c)$, and a $k > 0$. Find a k-center set, which is a subset $C \subseteq V$ with $|C| = k$, such that the maximum distance from a vertex to its nearest center $\max_{v \in V} \min_{u \in C} dist(v, u)$ is minimized.*

Algorithm 4 DIJKSTRA-SHORTEST-PATH-TREE(G, s):

In: A connected graph $G = (V, E, c)$, and a source $s \in V$.

Out: A connected spanning-tree $T = (V, E')$.

```
for each $u \in V$ do
    distance[$u$] = infinity
    parent[$u$] = NIL
end for
distance[$s$] = 0
$Q = V(G)$ {Minimum priority queue}
while $Q \neq \emptyset$ do
    $u$ = ExtractMin($Q$)
    for each $v \in V$ adjacent to $u$ do
        if $v \in Q$ and $distance[u] + dist(u, v) < distance[v]$ then
            parent[$v$] = $u$
            distance[$v$] = $distance[u] + dist(u, v)$
        end if
    end for
end while
return createTree($G$, parent, distance)
```

The NP-complete minimum k-center problem, is the problem of finding a subset $C \subseteq V$ of k vertices, such that the maximum distance from any remaining vertex in $V - C$ to a node in C is minimized. The problem is applicable to settings in which it is important to reduce the latency from application clients to, for example, a back-bone of core-nodes that are inter-connected with high-capacity links.

Definition 26 ***Minimum k-median problem:*** *Given a weighted graph $G = (V, E, c)$, and an integer $k > 0$. Find a k-median set, which is a subset $V' \subseteq V$ with $|V'| = k$, such that the sum of the distances from each vertex to its nearest median $\sum_{v \in V} \min_{w \in V'} dist(v, w)$ is minimized.*

The NP-complete minimum k-median problem, is similar to the minimum k-center problem, and is the problem of finding a subset $V' \subseteq V$ of k vertices, such that the sum of the distances from each vertex $v \in V$ to to its nearest median vertex $w \in V'$ is minimized.

Definition 27 ***k-minimum-eccentricity problem:*** *Given a weighted graph $G = (V, E, c)$, and an integer $0 < k < |V|$. Find a set $D \subset V$ of size k, such that the sum of the eccentricities yielded by the vertices $v \in D$ is the smallest.*

The k-minimum-eccentricity problem is the problem of finding a vertex set $D \subset V$ of size k, such that these nodes $d \in D$ have the lowest eccentricities among the vertices $v \in V$. The problem differs from the minimum k-center and k-median problems, in that the core-node set D is not chosen such that a non-core-node is close to one core-node. The core-node set is rather chosen such that all core-nodes are within minimum distances to the non-core-nodes.

Definition 28 ***k-minimum-pairwise problem:*** *Given a weighted graph $G = (V, E, c)$, and an integer $0 < k < |V|$. Find a set $D \subset V$ of size k, such that the sum of the distances from the vertices $u \in D$ to all nodes $v \in V$ is the smallest.*

The k-minimum-pairwise problem, is similar to the k-minimum-eccentricity problem, and is the problem of finding a subset $D \subset V$ of k vertices, such that the average pair-wise distances from the vertices $u \in D$ are the smallest of all the nodes $v \in V$.

4.5 Spanning-tree problems

A spanning-tree is applicable to many scenarios, and in the distributed interactive applications the intention is to use the spanning-tree as an overlay network and multicast events in the overlay tree. Chapter 3.4 gave some background and advances in spanning-tree algorithms, and chapter 9 evaluates a wide range of spanning-tree algorithms that address the spanning-tree problems that are introduced in the following.

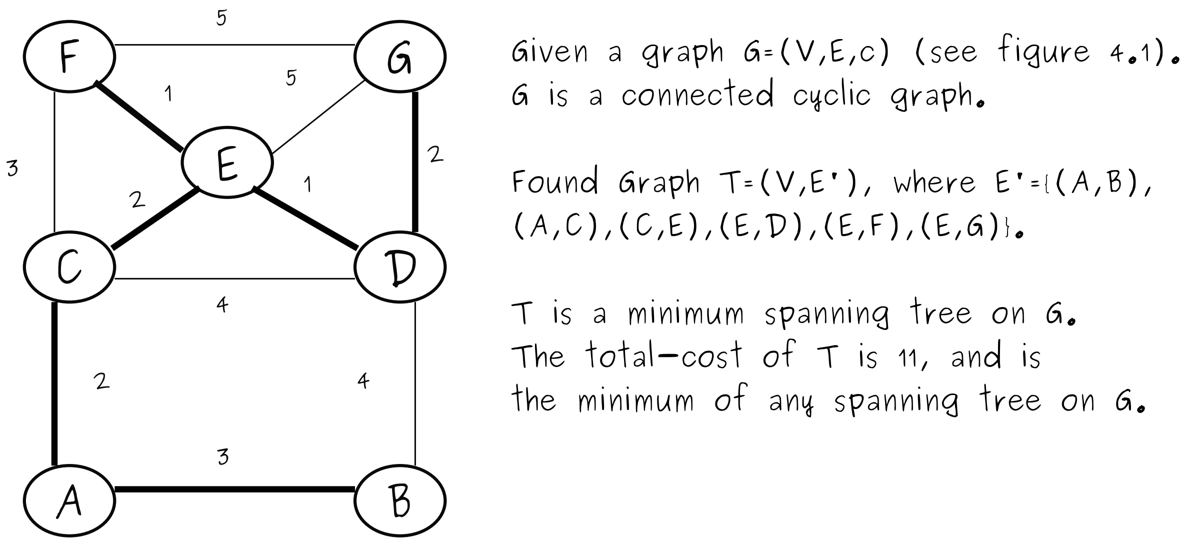
Definition 29 ***Spanning-tree problem:*** *Given a connected undirected weighted graph $G = (V, E, c)$, where V is the set of vertices, E is the set of edges, and $c : E \rightarrow \mathbb{R}$ is the edge cost function. Find a connected undirected subgraph (tree) $T = (V, E_T)$, without cycles.*

The basic spanning-tree problem requires that an acyclic graph (tree) is constructed that covers all the vertices in a given input graph. However, this basic spanning-tree problem lacks a cost-related optimization goal, and is therefore rarely used. One typical optimization goal is to construct a minimum-cost spanning-tree.

The following spanning-tree problems address various low-latency and low-cost requirements, and the ultimate goal is to try and identify problem that fit the application requirements identified in chapter 2.3. First, spanning-tree problems that require minimum total-cost solutions are introduced. The total-cost of T is the sum of the edge weights in T. Secondly, a range of spanning-tree problems are introduced that address diameter requirements in T. The diameter of T is defined as the longest of the paths in T among all the pairs of nodes in V. Finally, spanning-tree problems that address the radius are given. The radius is the minimum eccentricity in T and is associated to one node. The radius problems are used as source-tree problems, such that, the radius is the maximum shortest path from a given source node. Next, the introduction of the spanning-tree problems.

4.5.1 Spanning-tree problems and the minimum-cost

The spanning-tree problems that require minimum-cost spanning-trees are among the oldest spanning-tree problems. The minimum-cost requirement is important to reduce the network



Given a graph G=(V,E,c) (see figure 4.1). G is a connected cyclic graph.

Found Graph T=(V,E'), where E'={(A,B), (A,C),(C,E),(E,D),(E,F),(E,G)}.

T is a minimum spanning tree on G. The total-cost of T is 11, and is the minimum of any spanning tree on G.

Figure 4.4: The minimum-cost spanning-tree of $G = (V, E, c)$.

cost, and fits for application types that do not yield stringent latency requirements. Chapter 2.5 includes a discussion of the minimum-cost metric.

Definition 30 ***Minimum-cost spanning-tree problem (MST):*** *Given G, find a spanning-tree $T = (V, E_T)$, where the $\sum_{e \in E_T} c(e)$ is minimized.*

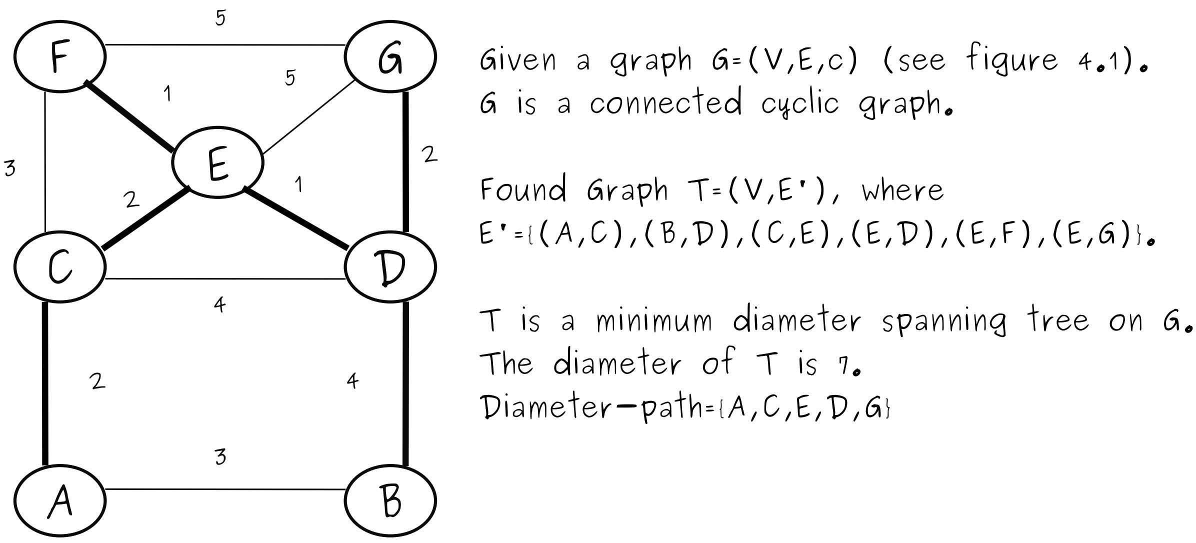
Informally, an MST algorithm computes a tree that contains the least cost edges in E connecting every vertex in V, definition 30. MST was first discovered by Boruvka [58], and later refined or rediscovered by several authors, most famously, Prim, Kruskal and Sollin [58, 93]. Figure 4.4 illustrates the minimum-cost spanning-tree of an example G.

Definition 31 ***Degree-limited minimum-cost spanning-tree problem (d-MST):*** *Given G, a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$; find a spanning-tree T, where the $\sum_{e \in E_T} c(e)$ is minimized, subject to the constraint that $deg_T(v) \leq deg(v)$.*

A d-MST algorithm builds a minimum-spanning-tree T in which each vertex is subject to a degree limit. d-MST heuristics found in the literature often employ Prim's MST algorithm. We have implemented a d-MST heuristic that we call dl-MST [95] (chapter 9).

4.5.2 Spanning-tree problems and the diameter

Spanning-tree problems that address diameter requirements are important to applications that require low-latency networks for distribution of data. The diameter-reducing spanning-tree problems are many and they are typically some combination of diameter, total-cost and degree requirements and constraints. Many of them (diameter-reducing spanning-tree problems) are



Given a graph G=(V,E,c) (see figure 4.1). G is a connected cyclic graph.

Found Graph T=(V,E'), where E'={(A,C),(B,D),(C,E),(E,D),(E,F),(E,G)}.

T is a minimum diameter spanning tree on G. The diameter of T is 7. Diameter−path={A,C,E,D,G}

Figure 4.5: The minimum diameter spanning-tree of $G = (V, E, c)$.

also NP-complete, such that there are no polynomial time algorithm that solve the problems exact. Chapter 9 include polynomial time heuristics that address the following diameter-related spanning-tree problems.

Definition 32 ***Minimum diameter spanning-tree problem (MDST):*** *Given G, find a spanning-tree T of G such that the maximum weight shortest path $p \in T$, $\sum_{e \in p} W(e)$ is minimized.*

An MDST-algorithm builds a tree of minimum *diameter*, and is solvable in polynomial time. Figure 4.5 illustrates a minimum diameter spanning-tree. Ho, Lee, Chang and Wong [68] considered the case in which the graph G is a complete Euclidian graph induced by a set of points in the Euclidian plane. They call this special case the *geometric* MDST problem. They prove that there is an optimal tree in which either one or two vertices in V are connected to the remaining vertices. The result extends to any complete graph whose edge lengths satisfy the triangle inequality, which holds for overlay networks that are built from shortest path links from the network layer.

For general graphs, a Steiner minimum diameter tree algorithm is to search for the absolute 1-center of a graph (chapter 4.1.1) and connect the nodes to that point. The absolute 1-center is the one point that may be located at any point in the graph (including edges), that has the lowest radius. For a complete graph, finding a simple heuristic for building a close-to-optimal MDST reduces to finding a single node located close to the center of the graph that connects to the remaining nodes through shortest paths (direct links). The topology of the resulting tree T is that of a star. Consequently, the work-load (stress) of the center node becomes significant as the degree increases. The degree is the number of incident edges a node has. Thus, a solution is not viable unless the center node has a considerable amount of resources.

Definition 33 ***Bounded diameter minimum-cost spanning-tree problem (BDMST):*** *Given G, and a bound $D > 0$. Find a minimum-cost spanning-tree T on G, where $\sum_{e \in E_T} c(e)$ is minimized and the diameter of which does not exeed D.*

A BDMST-algorithm builds a tree of minimum *total cost* within a diameter bound. An advantage of the NP-complete BDMST problem over MDST is that it is possible to tune the tree diameter while minimizing the *total cost*. However, one problem with BDMST remains the potentially high node degree of central nodes in the tree when the diameter bound D is stringent.

Definition 34 ***Minimum diameter degree-limited spanning-tree problem (MDDL):*** *Given G, a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$; find a spanning-tree T of G of minimum diameter, subject to the constraint that $deg_T(v) \leq deg(v)$.*

An MDDL-algorithm builds a tree of minimum diameter while obeying the degree limits. MDDL is NP-complete, nevertheless, it is relevant because it avoids the problems with stress that beset spanning tree problems that do not have limitations on degree. One issue with the MDDL problem is that it is a minimization of the maximum diameter within a degree limit, which increases the complexity of a heuristic.

Definition 35 ***Bounded diameter degree-limited minimum-cost spanning-tree problem (BDDLMST):*** *Given G, a diameter bound $D > 0$, and a degree bound $deg(v) \in \mathbb{N}$ for each $v \in V$; Find a minimum-cost spanning-tree T on G, where $\sum_{e \in E_T} c(e)$ is minimized, subject to the constraint that the diameter does not exceed D, and $deg_T(v) \leq deg(v)$.*

The NP-complete BDDLMST problem is identical to BDMST, but with degree limits for each vertex. A BDDLMST-heuristic is able to produce trees with a diameter that is in accordance with the diameter bound it received. This property is vital in cases of lighter application requirements, because the time complexity of the BDDLMST-heuristic decreases with looser diameter bounds.

Definition 36 ***Bounded diameter residual balanced tree problem (BDRB):*** *Given an undirected weighted graph G, a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$, and a diameter bound $D \in R$; Find a spanning-tree T of G with diameter $\leq D$ that maximizes $res_T(v) = deg(v) - deg_T(v)$, subject to the constraint that $deg_T(v) \leq deg(v)$.*

A BDRB-algorithm builds the "most balanced" tree that satisfies an upper bound on the diameter. The most balanced tree is any tree that maximizes the smallest *residual degree*. However, a balanced tree does not have an optimal diameter, instead, all nodes suffer.

4.5.3 Spanning-tree problems and the radius

Spanning-tree problems that consider the radius of trees are similar to their diameter counterparts. These radius problems do not explicitly consider the diameter, but are often approximable using faster spanning-tree heuristics.

Definition 37 ***Shortest-path tree problem (SPT):*** *Given a weighted graph G, find a spanning-tree $T = (V, E_T)$, starting from a root node $s \in V$, where, for each $v \in V - s$ the path $p = (v, \ldots, s)$ minimizes $\sum_{v_i \in p} c(e(v_i, v_{i+1}))$, where $e \in E_T$.*

Informally, an SPT algorithm computes a tree that contains the shortest paths from a single vertex $v \in V$ to $V - v$, definition 37. The most famous algorithm solving SPT was proposed by Dijkstra [58]. It was solved by Dijkstra and has a worst-case time complexity of $O(n^2)$. A shortest-path tree is actually a simple MDST heuristic if the source vertex is selected on the basis of its location in relation to the other nodes. Remember that in a complete graph, the topology of a close-to-optimal MDST is a star, where the issue was the degree (stress) on the center vertex. Hence, a degree limit is needed when using a shortest-path tree.

The following spanning-tree radius problems are used as source specific problems, such that, the radius is the maximum shortest path from a given source node.

Definition 38 ***Minimum radius spanning-tree problem (MRST):*** *Given G, a source s, find a spanning-tree T of G such that the eccentricity of s is minimized.*

The SPT problem can also be written as the MRST problem and solves the exact same problem.

Definition 39 ***Bounded radius minimum-cost spanning-tree problem (BRMST):*** *Given G, a source s, and a bound $R > 0$. Find a minimum-cost spanning-tree T on G, where $\sum_{e \in E_T} c(e)$ is minimized and the radius of which does not exeed R.*

A BRMST-algorithm builds a tree of minimum *total cost* within a radius bound. BRMST is an NP-complete problem. An advantage of BRMST over MRST is that it is possible to tune the tree radius while minimizing the *total cost*. However, one problem with BRMST remains the potentially high node degree of central nodes in the tree when the radius bound D is stringent. If the source s is close to the center of G, a BRMST-algorithm is approximating the BDMST-problem.

Definition 40 ***Degree-limited shortest-path tree problem (d-SPT):*** *Given G, a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$; find a spanning-tree T, starting from a root node $s \in V$, where, for each $v \in V$ the path $p = (v, \ldots, s)$ minimizes $\sum_{p_i \in p} c(p_i)$. subject to the constraint that $deg_T(v) \leq deg(v)$.*

A d-SPT algorithm builds a shortest-path tree in which each vertex is subject to a degree limit. A d-SPT algorithm combined with a centrally located source-node is able to compete with an MDDL algorithm.

Definition 41 ***Minimum radius degree-limited spanning-tree problem (MRDL):*** *Given G, a source s, and a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$; find a spanning-tree T of G such that the eccentricity of s is minimized, subject to the constraint that $deg_T(v) \leq deg(v)$.*

The d-SPT problem is identical to the MRDL-problem. An MRDL-algorithm builds a tree of minimum radius while obeying the degree limits. MRDL is NP-complete, nevertheless, it is relevant because it avoids the problems with stress that beset spanning tree problems that do not have limitations on degree. If the source s is close to the center of G, a MRDL-algorithm is approximating the MDDL-problem.

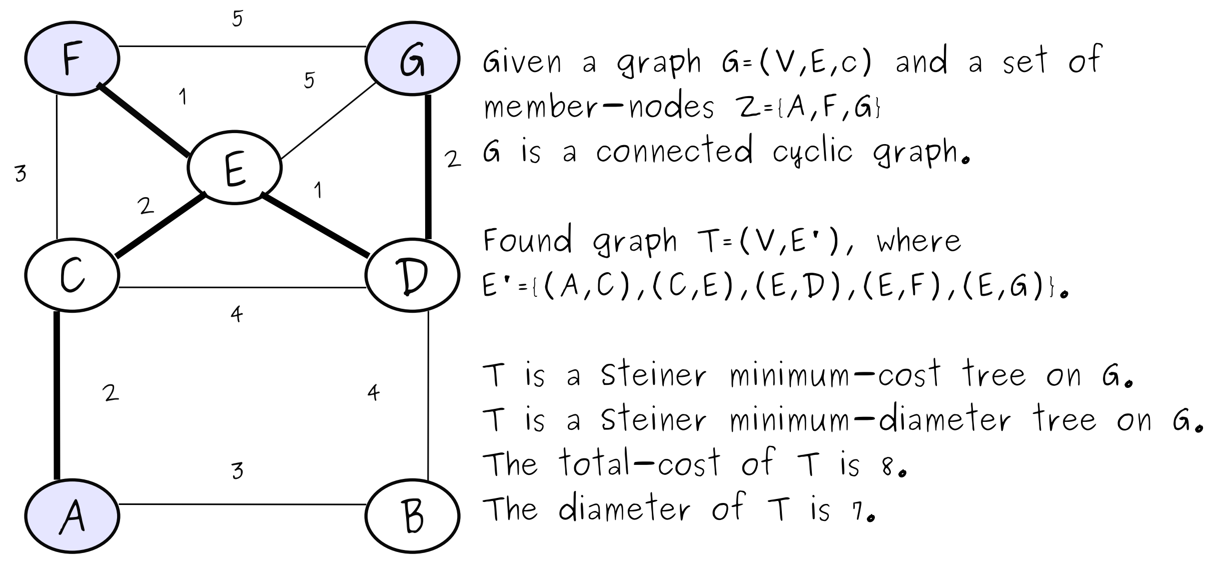
Definition 42 ***Bounded radius degree-limited minimum-cost spanning-tree problem (BRDLMST) :*** *Given G, a source s, a radius bound $R > 0$, and a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$; Find a minimum-cost spanning-tree T on G, where $\sum_{e \in E_T} c(e)$ is minimized, subject to the constraint that the radius does not exceed R, and $deg_T(v) \leq deg(v)$.*

The NP-complete BRDLMST problem is identical to BRMST, but with degree limits for each vertex. A BRDLMST-heuristic is able to produce trees with a radius that is in accordance with the radius bound it received. This property is vital in cases of lighter application requirements, because the time complexity of the BRDLMST-heuristic decreases with looser radius bounds. If the source s is close to the center of G, a BRDLMST-algorithm is approximating the BDDLMST-problem.

4.6 Steiner-tree problems

A Steiner-tree differs from a spanning-tree in that it may use non-member-nodes (Steiner-points) to enhance the efficiency of the tree. A Steiner-tree is applicable to similar scenarios as a spanning-tree, and the intention is to use Steiner-trees as overlay networks to multicast application events. In an application layer multicast scenario, the Steiner points may, for example, be application-provided proxies or servers.

Section 3.6 gave some background material and recent advances in Steiner-tree algorithms. In chapter 10, a wide range of Steiner-tree heuristics are introduced and then evaluated. The evaluated Steiner-tree heuristics address the Steiner-tree problems that are formally introduced in the following.



Given a graph G=(V,E,c) and a set of member-nodes Z={A,F,G}
G is a connected cyclic graph.

Found graph T=(V,E'), where E'={(A,C),(C,E),(E,D),(E,F),(E,G)}.

T is a Steiner minimum-cost tree on G.
T is a Steiner minimum-diameter tree on G.
The total-cost of T is 8.
The diameter of T is 7.

Figure 4.6: The Steiner minimum-cost tree on $G = (V, E, c)$.

Definition 43 ***Steiner-tree problem:*** *Given a connected undirected weighted graph $G = (V, E, c)$, where V is the set of vertices, E is the set of edges, and $c : E \rightarrow \mathbb{R}$ is the edge cost function, and there is a set $Z \subset V$ of member-nodes. Find a connected undirected tree $T_Z = (V_Z, E_Z)$ of G, where $V_Z \subseteq V$, $Z \subseteq V_Z$ and $E_Z \subseteq E$, such that there is a path between every pair of member-nodes.*

A Steiner-tree T on a graph G spans all the member-nodes (terminals) in G, and may also span non-member-nodes (non-terminals), which are called Steiner-points in the literature. Typically, a Steiner-tree algorithm includes non-member nodes, if they help achieve an optimization goal. The next sections introduce Steiner-tree problems that optimize for minimum-cost, diameter and radius while applying various constraints.

4.6.1 Steiner-tree problems and the minimum-cost

The problem of finding a Steiner minimum-cost tree in networks (SMT) is an NP-complete problem that was originally formulated independently by Hakimi and Levin [142] in 1971. Several exact algorithms and heuristic have been suggested, implemented and compared [134].

Definition 44 ***Steiner minimum-cost tree in networks (SMT):*** *Given an undirected weighted graph G, and a subset $Z \subseteq V$ of member-nodes. Find a connected undirected tree T_Z, such that there is a path between every pair of member-nodes, and $\sum_{e \in E_Z} c(e)$ is minimized.*

An SMT-algorithm finds a least cost tree connecting a set of member nodes ($Z \subseteq V$). The tree may contain a subset ($V - Z$) of non-member-nodes (Steiner points) that reduces the cost of the tree. SMT-heuristics are often applied to network layer multicast, where the routers are Steiner points and the clients are member-nodes. Figure 4.6 illustrates a Steiner minimum-cost tree on an example graph G.

Definition 45 ***Degree limited Steiner minimum-cost tree in networks (*** d***-SMT):*** *Given G and Z, a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$; Find an SMT T_Z of G and Z of minimum total cost, subject to the constraint that $d_{T_Z}(v) \leq deg(v)$.*

A d-SMT-algorithm finds a least cost tree (like SMT-algorithms), where each node obeys a given degree limit. The degree limit provides a degree or stress control to the nodes. The d-SMT problem has been studied by, for example, Bauer and Varma [16]. They introduced several heuristics that are directly derived from SMT heuristics.

From these Steiner-tree problems that yield least cost trees, many other Steiner-tree problems have been derived. The next sections focuses particularly on Steiner-tree problems that construct low-latency overlays.

4.6.2 Steiner-tree problems and the diameter

The following gives an introduction to Steiner-tree problems that address diameter requirements in Steiner-trees. The diameter of T_Z is defined as the longest of the paths in T_Z among all the pairs of nodes in V_Z. The study also includes Steiner-tree problems that optimize for the total-cost, that is, the sum of the edge weights in T_Z, while obeying a given diameter bound. Other studies of similar problems can be found in [65, 3, 150, 21].

Most Steiner-tree problems are NP-complete, and such are also many of the *diameter*-related Steiner problems. Algorithms that solve NP-complete problems are exponential time algorithms, and not practical if time is important. Chapter 10 introduces a range of polynomial time Steiner-tree heuristics that address the following Steiner-tree problems.

Definition 46 ***Steiner minimum diameter spanning-tree problem (Steiner-MDST):*** *Given G and Z; Find a Steiner spanning-tree T_Z of G and Z such that the maximum weight shortest path (diameter) $p \in T$, $\sum_{e \in p} W(e)$ is minimized.*

A Steiner-MDST algorithm builds a tree of minimum *diameter*. Ho, Lee, Chang and Wong [68] considered the case in which the graph G is a complete Euclidian graph induced by a set of points in the Euclidian plane. They call this special case the *geometric* Steiner-MDST problem. They prove that there is an optimal tree in which a single Steiner-point is connected to all the vertices in Z, and that it is solvable in polynomial time. The result extends to any complete graph whose edge lengths satisfy the triangle inequality,

Hence, for a complete graph, a simple heuristic for building a close-to-optimal Steiner-MDST is to find a single node located close to the center of the graph that connects to the remaining nodes through shortest paths. The topology of the resulting tree T_Z is that of a star. Consequently, the work-load (stress) of the center node becomes significant as the degree increases. Thus, a solution is not viable unless the center node has a considerable amount of resources.

Definition 47 ***Bounded diameter Steiner minimum-cost tree problem (BDSMT):*** *Given G and Z, and a diameter bound $D \in \mathbb{R}$; Find a Steiner minimum-cost tree T_Z on G and Z, where $\sum_{e \in E_{T_Z}} c(e)$ is minimized and whose diameter does not exeed D.*

A BDSMT-algorithm builds a tree of minimum *total cost* within a diameter bound. An advantage of BDSMT over Steiner-MDST is that it is possible to tune the tree diameter while minimizing the *total cost*. This property is vital in cases of lighter application requirements, because the time complexity of the BDSMT-heuristic decreases with looser diameter bounds. However, one problem with BDSMT remains the potentially high node degree of central nodes in the tree when the diameter bound D is stringent.

Definition 48 ***Steiner minimum diameter degree limited spanning-tree problem (Steiner-MDDL):*** *Given G and Z, a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$; Find a Steiner-tree T_Z of G and Z of minimum diameter, subject to the constraint that $d_{T_Z}(v) \leq deg(v)$.*

A Steiner-MDDL-algorithm builds a tree of minimum diameter while obeying the degree limits. The Steiner-MDDL problem avoids the stress issue that beset spanning-tree problems that do not have limitations on degree. One issue with the Steiner-MDDL problem is that it is a minimization of the maximum eccentricity (diameter) within a degree limit, which increases the complexity of a heuristic.

Definition 49 ***Bounded diameter degree limited Steiner minimum-cost tree problem (BDDLSMT):*** *Given G and Z, a diameter bound $D \in \mathbb{R}$, and a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$; Find a Steiner minimum-cost tree T_Z on G and Z, where $\sum_{e \in E_{T_Z}} c(e)$ is minimized, subject to constraints that the diameter does not exeed D, and $d_{T_Z}(v) \leq deg(v)$.*

A BDDLSMT-algorithm builds a tree of minimum *total cost* within a diameter bound, subject to a degree limit. Like the BDSMT heuristics, a BDDLSMT-heuristic is able to produce trees with a diameter that is in accordance with the diameter bound it received. In addition, the BDDLSMT problem solves the stress issues of BDSMT by using degree limits for each vertex. However, the added degree constraint makes BDDLSMT a harder problem to solve than BDSMT.

4.6.3 Steiner-tree problems and the radius

In the following, we introduce problems that bound the *radius* of T_Z, where the radius is defined as the longest of the paths to a pre-defined source node in V. We believe that heuristics of the following problems combined with a well-placed source (core) node may be able to compete with Steiner-tree problems that explicitly consider the diameter.

Definition 50 ***Steiner minimum radius spanning-tree problem (Steiner-MRST):*** *Given G and Z; Find a Steiner-tree T_Z on G and Z, starting from a root node $s \in V$, where, for each $z \in Z$ the path $p = (z, \ldots, s)$ minimizes $\sum_{v_i \in p} c(e(v_i, v_{i+1}))$, where $e \in E_{T_Z}$.*

A Steiner-MRST algorithm builds a tree of minimum *radius* and is solvable in polynomial time. The optimal Steiner-MRST-algorithm is equal to connecting the root node $s \in V$ to the member-nodes in Z through shortest-paths. If the root node $s \in Z$, and the input graph is a full mesh of shortest paths, the optimal Steiner-MRST is always Dijkstra's SPT. In addition, if the root node s is located close to the center of G, a Steiner-MRST heuristic and a Steiner-MDST heuristic would produce trees with similar diameter and radius.

Definition 51 ***Bounded radius Steiner minimum-cost tree problem (BRSMT):*** *Given G and Z, a root node $s \in V$ and a radius bound $R \in \mathbb{R}$. Find a Steiner minimum-cost tree T_Z on G, where, for each $z \in Z$ the path $p = (z, \ldots, s)$, with weight $\sum_{p_i \in p} c(p_i) \leq R$.*

A BRSMT-algorithm builds a tree of minimum *total cost* within a radius bound from a given root node s to all destinations. The BRSMT-problem is NP-complete, and is similar to the shallow-light tree problem (section 3.1). It is applicable to a shared-tree environment if the root node s is close to the center of G, since it is then approximating a BDSMT.

Definition 52 ***Steiner minimum radius degree limited spanning-tree problem (Steiner-MRDL):*** *Given G and Z, a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$; find a Steiner-tree T_Z on G and Z, starting from a root node $s \in V$, where, for each $z \in Z$ the path $p = (z, \ldots, s)$ has minimum cost and is subject to the constraint that $d_{T_Z}(v) \leq deg(v)$.*

A Steiner-MRDL algorithm builds a Steiner-tree of minimum radius in which each vertex is subject to a degree limit. The Steiner-MRDL problem is NP-complete. One Steiner-MRDL-heuristic is to connect the root node $s \in V$ to the member-nodes in Z through shortest paths. If the root node s is located close to the center of G, a Steiner-MRDL heuristic and a Steiner-MDDL heuristic would produce trees with similar diameter and radius.

Definition 53 ***Bounded radius degree limited Steiner minimum-cost tree problem (BRDLSMT):*** *Given G and Z, a root node $s \in V$, a radius bound $R \in \mathbb{R}$. and a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$; Find a Steiner minimum-cost tree T_Z on G, where, for each $z \in Z$ the path $p = (z, \ldots, s)$, with weight $\sum_{p_i \in p} c(p_i) \leq R$, and $d_{T_Z}(v) \leq deg(v)$.*

Like the BRSMT-algorithm, a BRDLSMT-algorithm builds a tree of minimum *total cost* within a radius bound from a given source to all destinations. The BRDLSMT problem is NP-complete, and solves the stress issues of BRSMT by using degree limits for each vertex. BRDLSMT is approximating a BDDLSMT when the root node s is close to the center of G. The advantage is that a BRDLSMT-heuristic is often less complex than a BDDLSMT-heuristic.

4.6.4 Miscellaneous Steiner-tree problems

There are many different Steiner-tree problems in the literature that address slightly different problems than the ones introduced before. Here are two Steiner-tree problems that are of relevance for our application scenario.

Definition 54 ***Terminal Steiner-tree problem:*** *Given an undirected weighted graph $G = (V,E,c)$, and a subset $Z \subseteq V$ of member-nodes. Find a connected undirected tree T_Z, such that every member-node is a leaf and there is a path between every pair of member-nodes.*

The objective of the terminal Steiner-tree problem is to find a subtree of G in which every member-node is a leaf-node. This problem has importance when member-nodes are not allowed to, or cannot communicate directly with each other.

Some distributed interactive applications may face consistency issues, cheating problems, etc, such that the application-provider needs to control the application data in its back-bone infrastructure of servers and proxies. The terminal Steiner-tree problem is easily re-defined to address minimum-cost, minimum-diameter, minimum-radius, etc.

Chapter 8 introduces graph reduction algorithms that force the member-nodes to be leaf-nodes. Similarly, in the cases of degree-limited Steiner-tree heuristics, it is possible to reduce the degree-limits to one on each member-node. Chapter 10.3 includes an evaluation of the graph reduction algorithms.

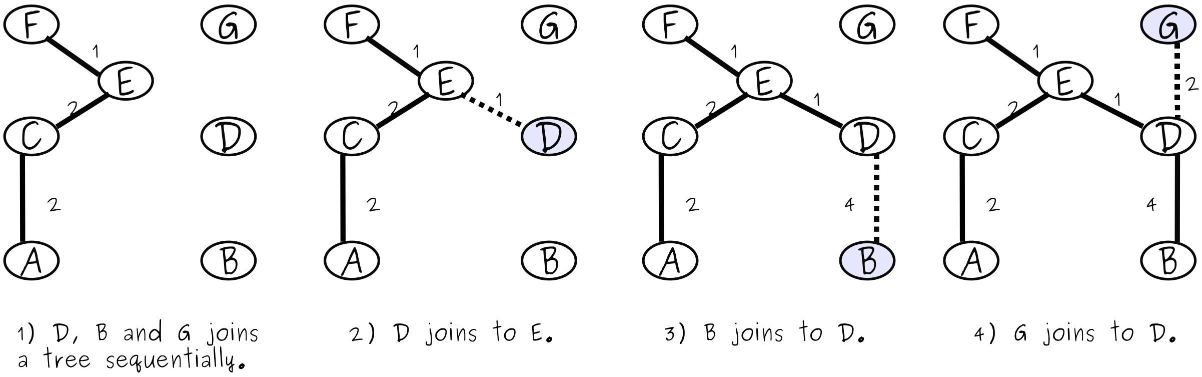
Definition 55 ***Group Steiner-tree problem:*** *Given an undirected weighted graph $G = (V,E,c)$, and subsets of vertices that are called groups $g_1, g_2 \dots g_k \in V$. Find a connected undirected tree $T = (V', E', c)$, such that $V' \cap g_i \neq$ for all $i \in 1 \dots k$.*

The objective of the group Steiner-tree problem is to find a subtree of G that contains at least one vertex from each given group. This problem is important in cases when groups of clients in a distributed interactive application are merged to one group. For example, instead of merging k groups to one group and then rebuild the merged group from scratch, it is possible to connect the k groups through single edges to form one merged group. The group Steiner-tree problem is easily re-defined to address minimum-cost, minimum-diameter, minimum-radius, etc [88].

Chapter 12 introduces dynamic tree algorithms for removal of nodes in trees. When a non-leaf node $v \in V$ is removed from a tree T, the tree is disconnected creating $deg_T(v)$ subtrees, which are equal to the groups in the group Steiner-tree problem. The reconnection of the subtree groups is exactly the group Steiner-tree problem.

4.7 Dynamic tree problems

Dynamic tree problems address the client dynamicity that is a part of the application scenario outlined in chapter 2. The dynamic-tree problems define requirements to situations where single



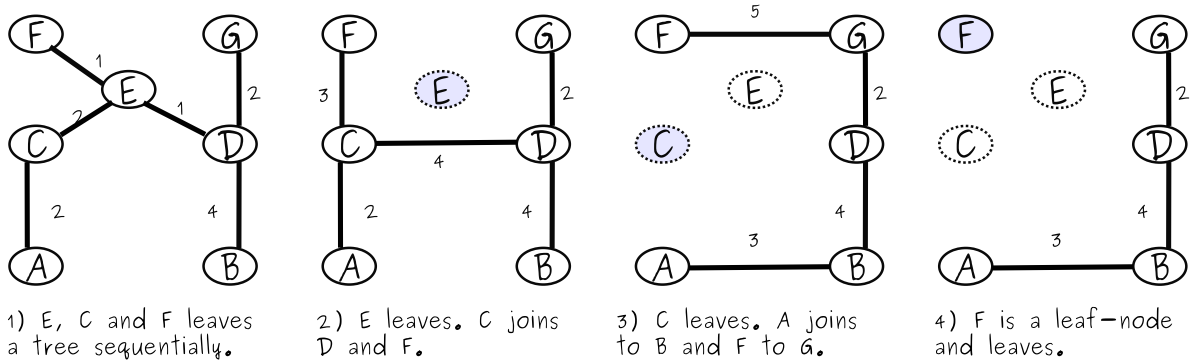
1) D, B and G joins a tree sequentially.

2) D joins to E.

3) B joins to D.

4) G joins to D.

Figure 4.7: Examples of tree dynamics, when nodes join a tree.



1) E, C and F leaves a tree sequentially.

2) E leaves. C joins D and F.

3) C leaves. A joins to B and F to G.

4) F is a leaf-node and leaves.

Figure 4.8: Examples of tree dynamics, when nodes leave a tree.

nodes are inserted and removed from an existing tree, based on incoming insert and remove requests. It is very relevant for dynamic group communication where clients may join and leave ongoing sessions of real-time interaction. Figure 4.7 illustrates an example where nodes join an existing tree. Similarly, figure 4.8 illustrates an example where nodes leave an existing tree.

Chapter 12 introduces and evaluates a range of dynamic tree algorithms for inserting and removing nodes from trees. Dynamic tree problems are introduced next, specifically dynamic Steiner-tree problems, and dynamic spanning-tree problems.

4.7.1 Dynamic Steiner-tree problems

Definition 56 ***Dynamic Steiner-tree problem (DST):*** *An instance of DST includes a graph $G = (V, E, c)$, a set $Z_i \subseteq V$, and a queue of requests $Q_r = \{r_0, r_i, \dots, r_k\}$, where each r_i is a pair (v_i, ρ_i), $v_i \in V$, $\rho_i \in \{insert, remove\}$. The set $Z_i = \{v | (v, insert) = r_j \text{ for some } j, 0 \leq j \leq i$ and $(v, remove) \neq r_l$ for all $l, j < l \leq i\}$, where Z_i is the terminal set at step i which are to be connected with a Steiner-tree after request r_i.*

The dynamic Steiner-tree (DST) problem was first introduced by Waxman and Imase [140]. An instance of DST includes a series of requests r_i that contains a node v_i to be inserted or removed from Z_i. A DST algorithm finds a Steiner-tree connecting each terminal set Z_i without knowledge of request r_j for any $j > i$. The optimization goal of the original DST problem was minimum-cost, but it is applicable to most optimization goals.

This basic DST problem formulation is equivalent to a Steiner-tree problem for each request r_i, therefore the DST problem must be extended with some conditional restrictions for it to be of practical significance.

Definition 57 ***Non-reconfigurable dynamic Steiner-tree problem (N-DST):*** *Given an instance $I = (G,,cost,Q_r)$, find a sequence of trees $\{T_1, T_2, \ldots, T_k\}$ satisfying the following conditions C_N and minimizing a function of $\{cost(T_i)|i = 1,2\ldots k\}$. $C_N = \{$ 1. Each T_i spans Z_i, 2. If r_i is an insert request, T_i includes T_{i-1} as a subgraph, and 3. If r_i is a remove request, T_{i-1} includes T_i as a subgraph. $\}$.*

The N-DST [140] problem is a special case of the DST problem. Conditions 2 and 3 in C_N imply that edges and nodes are inserted to a tree only for an insert request, and removed only for a remove request. In practice it means that, upon an insert request a node is added to the tree as a leaf-node, and a non-leaf node cannot be removed, but rather remains in the tree as a Steiner point until it is a leaf node (possibly never).

It is evident that the N-DST problem results in tree degradation in cases where many Steiner-points remain in the Steiner-tree forever. The conditions C_N, although practical and simplistic, are unrealistic in a real application where clients join and leave applications continuously.

Definition 58 ***Reconfigurable dynamic Steiner-tree problem (R-DST):*** *Given an instance $I = (G,,cost,Q_r)$, find a sequence of trees $\{T_1, T_2, \ldots, T_k\}$ where each T_i spans Z_i and minimizes a function of $\{cost(T_i)|i \leq k\}$. while not exceeding an upper bound B on the number of rearrangments.*

The R-DST [140] problem allows reconfigurations to the edge and node sets, regardless of r_i being an insert or remove request. However, if the number of reconfigurations allowed is unlimited, the R-DST is equivalent to the Steiner-tree problem for each instance (G,cost,Z_i). Therefore, a bound B is defined to limit the number of rearrangements that are allowed for each request r_i.

The R-DST problem fits an application scenario in which trees may experience frequent insert and remove requests. Tree stability then becomes important, and the bound B sets a worst-case limit to the number of rearrangements. One issue of the R-DST problem is that it does not consider the degradation of the Steiner-tree. Rather, the problem is satisfied as long as the number of rearrangments is equal to or below B.

Definition 59 ***Minimum-reconfiguration dynamic Steiner-tree problem (M-DST)*** [1]***:*** *Given an instance $I = (G,,cost,Q_r)$, find a sequence of trees $\{T_1, T_2, \ldots, T_k\}$ where each T_i spans Z_i and minimizes the number of rearrangements, while not violating the cost function $\{cost(T_i)|i \leq k\}$.*

The M-DST problem minimizes the number of rearrangements needed to fulfill a given optimization goal. The optimization goal may, for example, provide an upper bound to the diameter of the tree T_i.

Chapter 2 introduced the stringent latency requirements of different distributed interactive applications. In these cases, the M-DST problem requires that the maximum effort is done to meet the latency requirements. One issue is that in the worst case, the optimization goal cannot be met, and a complete reconfiguration is conducted.

4.7.2 Dynamic spanning-tree problems

Definition 60 ***Dynamic spanning-tree problem*** [2]***:*** *An instance of the dynamic spanning-tree problem includes a graph $G = (V, E, c)$, and a queue of requests $Q_r = \{r_0, r_i, \ldots, r_k\}$, where each r_i is a pair (v_i, ρ_i), $v_i \in V$, $\rho_i \in \{insert,remove\}$. There is a subgraph $G_i = (V_i, E_i, c)$, in which $V_i = \{v \in V|(v,insert) = r_j \text{ for some } j, 0 \leq j \leq i \text{ and } (v,remove) \neq r_l \text{ for all } l, j < l \leq i\}$, where V_i is the node set after step i, and $E_i = \{e \in E|e = (u, v), \text{ such that } u \in V_i, v \in V_i\}$, where E_i interconnects all nodes in V_i. For each request r_i, construct a spanning-tree T_i on G_i.*

The dynamic spanning-tree problem is similar to the dynamic Steiner-tree problem. Only, in the dynamic spanning-tree problem a tree T_i cannot include nodes that are Steiner-points (non-member nodes, non-terminals). This influences the problem definition in the cases where the request r_i is a remove request and the node v (to be removed) is a non-leaf node. These cases force a reconfiguration to the tree T_i. Therefore, the dynamic spanning-tree problem does not have a non-reconfigurable dynamic spanning-tree problem. Rather, dynamic spanning-tree problems must be defined such that any node can be removed regardless of its degree in T_i.

Definition 61 ***Restricted reconfigurable dynamic spanning-tree problem (RR-DST):*** *Given an instance $I = (G,,cost,Q_r)$, find a sequence of trees $\{T_1, T_2, \ldots, T_k\}$ where each T_i spans V_i and minimizes a function of $\{cost(T_i)|i \leq k\}$, while not exceeding an upper bound B_i on the number of rearrangments. B_i is determined such that if r_i is an insert request then $B_i = 1$, if r_i is a remove request then $B_i = (|d_{T_i}(v_i)| * 2) - 1$.*

The restricted reconfigurable dynamic spanning-tree problem minimizes the number of rearrangements of T_i to a minimum for each r_i. For each insert request, a node is added to the

[1] Not found in the literature

[2] Not found in the literature

tree through a single edge, and for each remove request, the neighbors of the removed node are reconnected. Note that, when a node $v_i \in T_i$ has been removed and created $deg_{T_i}(v_i)$ subtrees, the reconnection of these subtrees is equivalent to the group Steiner-tree problem that was introduced in definition 55. The RR-DST problem requires that the tree remains a spanning-tree of member-nodes, and that the reconfigurations are minimum upon insert and remove requests.

Definition 62 ***Reconfigurable dynamic spanning-tree problem:*** *Given an instance $I = (G,, cost, Q_r)$, find a sequence of trees $\{T_1, T_2, \ldots, T_k\}$ where each T_i spans V_i and minimizes a function of $\{cost(T_i)|i \leq k\}$, while not exceeding an upper bound B_i on the number of rearrangments. B_i is determined such that if r_i is an insert request then (trivially) $B_i \geq 1$, if r_i is a remove request then $B_i \geq (|d_{T_i}(v_i)| * 2) - 1$.*

The reconfigurable dynamic spanning-tree problem allows reconfigurations to T_i bounded by an integer B_i. If the bound $B_i > |V_i| * 2$ the reconfigurable dynamic spanning-tree problem is equivalent to the spanning-tree problem (definition 29) for each instance (G,cost, Z_i). Furthermore, for a remove request, if the bound $B_i < (|d_{T_i}(v_i)| * 2) - 1$, then a solution does not exist. Therefore, the bound B_i must be determined for each request r_i, such that, for a remove request, it is greater than the degree of v_i in T_i.

The reconfigurable dynamic spanning-tree problem requires that the number of rearrangements of the Steiner-tree is $\leq B_i$ when a request r_i has been completed. However, it may be difficult to determine this bound B_i for each request r_i. In practice, it may be wise to use a percentage of additional reconfigurations above the necessary.

Definition 63 ***Minimum-reconfiguration dynamic spanning-tree problem:*** *Given an instance $I = (G,, cost, Q_r)$, find a sequence of trees $\{T_1, T_2, \ldots, T_k\}$ where each T_i spans V_i and minimizes the number of rearrangements, while not violating the cost function $\{cost(T_i)|i \leq k\}$.*

The minimum-reconfiguration dynamic spanning-tree problem minimizes the number of rearrangments needed to achieve a given cost function. The cost function may, for example, provide an upper bound to the diameter of the tree T_i.

4.8 Dynamic tree insert- and remove-node problems

The dynamic-tree problems introduced in section 4.7 defined requirements to tree-updates when single nodes are inserted and removed, for example, concerning bounds on the number of rearrangments, bounds on the cost of the tree, etc. In practice, a dynamic tree algorithm is comprised of one insert-strategy, which inserts a node into a tree, and one remove-strategy, which removes a node from a tree. The strategies must be paired to create one dynamic tree algorithm to address the dynamic-tree problems. However, one issue is that the insert and remove strategy may belong in different dynamic tree problems. Existing dynamic tree problem definitions do not take these situations into account. This is a drawback when dynamic tree algorithms are analyzed, because the dynamic tree problems are too general and inaccurate. The goal is therefore to:

Define formal insert and remove-node problems that more accurately categorizes the insert and remove strategies that are needed to address the dynamic tree problems.

In this regard, it is especially the number of rearrangements that are allowed upon inserting or removing a node that influences an algorithm's performance and defines a tree's stability. For example, a remove strategy that is only allowed to change one edge, only removes leaf-nodes, and is therefore fast and keeps trees stable, but may suffer in regards to many other graph performance metrics. Moreover, if there is no bound, the remove-node operation may then be performed by simply removing a node from the input graph, and running a Steiner-tree or spanning-tree algorithm on it, which results in an optimal tree but the tree stability is less. However, even though a complete reconfiguration has a worst-case situation of exchanging every edge, it may still perform much better on average. But, it is obvious that the execution-time is much larger in a complete reconfiguration, than simply removing leaf-nodes.

By defining accurate bounds on the number of rearrangements that are allowed, we aim to categorize the insert and remove strategies such that it is clearer to see which strategies require more logic to complete, and which do not.

The observation is that more algorithmic logic very often leads to a larger execution time, and bounding the execution time is important for time-dependent applications. In conclusion, the ultimate goal is to identify insert and remove strategies that are stable and fast.

Following are definitions of a wide range of insert-node to tree and remove-node from tree problems that exhibit the accuracy that is desirable for describing the insert and remove strategies in dynamic tree algorithms.

4.8.1 Insert-node to tree problems

Insert strategies insert a new member m to a tree and assure that m is connected. Formally, an insert strategy works like this:

Given $G = (V, E, c)$, a tree $T = (V_T, E_T)$, a set of members $Z_T \subseteq V_T$, and a new member $m \in V$. Update T, such that $Z_T \cup \{m\}$ are connected.

The new member m may be inserted into T in many different manners. We have devised several insert strategies that bound the size of the reconfiguration set R. The reconfiguration set R contains the edges that are changed between reconfigurations of a tree. It is possible to insert m to T through a single edge, while other strategies use the degree limit as a bound.

Definition 64 ***Insert vertex edge-addition:*** *Given G, a connected tree T, and a joining member $m \in V$, Update T, such that $Z_T \cup \{m\}$ are connected, and $|R| = 1$.*

The insert vertex edge-addition problem connects (inserts) a node m to the tree using a single edge. Insert strategies belonging to this problem typically search for the one edge that, for example, is the minimum-cost edge, or the edge that results in the minimum eccentricity of the inserted node.

Definition 65 ***Insert vertex and degree limited reconfiguration set:*** *Given G, a connected tree T, and a joining member $m \in V$, Update T, such that $Z_T \cup \{m\}$ are connected, and $|R| \leq deg(m) * 2$.*

The insert vertex and degree-limited reconfiguration set restricts the size of the reconfiguration set R to be less than the double the current degree-limit of m. This allows for an insert strategy to, for example, connect m to T as an intersection node and then ensure that T is acyclic by removing unwanted edges.

Definition 66 ***Insert vertex and unlimited reconfiguration set:*** *Given G, a connected tree T, and a joining member $m \in V$, Update T, such that $Z_T \cup \{m\}$ are connected, and $|R| \leq |E_T| * 2$.*

The insert vertex and unlimited reconfiguration set problem put no restrictions on the size of the reconfiguration set R upon connecting m to T. If an insert strategy fits to this insert problem, it is clear that its worst-case behavior is that every edge in T is exchanged when m is inserted. However, the insert strategy may still not be equal to a Steiner-tree or spanning-tree algorithm if the average-case size of R is rather small.

4.8.2 Remove-node from tree problems

Remove strategies remove a member m from the multicast tree while assuring that the members stay connected. Formally, a remove strategy works like this:

Given $G = (V, E, c)$, a tree $T = (V_T, E_T)$, a set of members $Z_T \subseteq V_T$, and a member $m \in V$. Update T, such that $Z_T \setminus \{m\}$ are connected.

The number of direct neighbor nodes of m (its degree) in T influences the necessary actions. If the degree $deg_T(m) = 1$, m is a leaf that is simply removed along with the edge to its only neighbor. If it is greater than 1, a removal of m partitions the tree if no additional steps are taken, and $deg_T(m)$ unconnected subtrees would be the result. The basic goal of remove strategies is the reconnection of subtrees into a single tree when $deg_T(m) \geq 2$.

Similar to the insert strategies, we divide the remove strategies into the worst case sizes of the reconfiguration set R. The minimum reconfiguration set when removing m consists of its immediate neighbors in T. We have a number of remove strategies that bound the size of the reconfiguration set to the worst case number of edges it may change when disconnecting and reconnecting the neighbors of m. Moreover, it is possible to reconfigure a larger portion of T while removing m, and we also include such remove strategies.

Definition 67 ***Remove vertex and edge-removal:*** *Given G, a connected tree T, and a leaving member $m \in V$, Update T, such that $Z_T \setminus \{m\}$ are connected, and $|R| \leq 1$.*

The remove vertex and edge-removal is the most restricted remove problem, and only allows the removal of leaf non-member nodes. All other non-member-nodes are kept in the tree until they are leaf nodes.

Definition 68 ***Remove vertex and degree limited reconfiguration set:*** *Given G, a connected tree T, and a leaving member $m \in V$, Update T, such that $Z_T \setminus \{m\}$ are connected, and $|R| \leq deg(m) * 2$.*

The remove vertex and degree-limited reconfiguration set bounds the size of the reconfiguration set to include, for example, a disconnection of m and then reconnecting the neighbors of m. It is also possible, for a remove strategy to include a Steiner-point to reconnect the neighbors of m. More advanced reconfigurations are also possible that does not include exchanging edges near m, but these are not addressed in the thesis.

Definition 69 ***Remove vertex and unlimited reconfiguration set:*** *Given G, a connected tree T, and a leaving member $m \in V$, Update T, such that $Z_T \setminus \{m\}$ are connected, and $|R| \leq |E_T| * 2$.*

The remove vertex and unlimited reconfiguration set problem put no restrictions on the size of the reconfiguration set R upon disconnecting m from T, and then reconnecting T. If a remove strategy fits to this remove problem, it is clear that its worst-case behavior is that every edge in T is exchanged when m is removed. However, the remove strategy may still not be equal to a Steiner-tree or spanning-tree algorithm if the average-case size of R is rather small.

4.9 Spanning subgraph problems

Definition 20 defined a connected spanning subgraph $G' = (V, E')$ on G to be a connected graph in which there is a minimum of one path between all nodes $v \in V$. By this definition a connected spanning-tree and a connected spanning-mesh are both defined to be a connected spanning subgraph. (definition 21 and 22). Therefore, a connected spanning subgraph may be both cyclic and acyclic (tree and mesh). A cyclic graph may be advantegous to increase the resilience in fault prone networks and reduce the pair-wise distances, however, it does inflict a higher network cost, for example, increased bandwidth consumption. An acyclic graph yield on average lower network costs, but is more fault prone and generally has higher pair-wise distances. Chapter 11 introduces and evaluates a range of practical spanning subgraph algorithms, especially towards the graph metrics, diameter, pair-wise distances and total-cost.

In the following, the class of spanning subgraph problems are formally introduced using definitions and explanations.

Definition 70 ***k-vertex connected spanning subgraph problem:*** *Given $G = (V, E, c)$, and an integer $0 < k < |V|$. Find a k-vertex connected spanning subgraph $M = (V, E_M)$ in which a removal of any $k - 1$ nodes leaves the subgraph in a connected state.*

The NP-complete k-vertex connected spanning subgraph problem requires that a spanning subgraph is built in which the removal of any $k - 1$ nodes leaves the subgraph in a connected state, and such that there are at least $k - 1$ paths between the remaining nodes.

k-vertex connected spanning subgraphs are desirable for the targeted application scenarios, in which fault prone clients dynamically join and leave applications. The problem ensures a configurable level of resilience in the computed subgraph, defined by an input integer $k > 0$.

Definition 71 ***k-edge connected spanning subgraph problem:*** *Given $G = (V, E, c)$, and an integer $k > 0$. Find a k-edge connected spanning subgraph $M = (V, E_M)$ of G, such that M is not disconnected by removing $k - 1$ edges.*

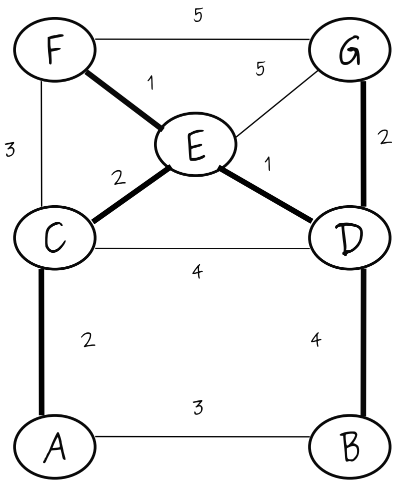
The NP-complete k-edge connected spanning subgraph problem requires that a spanning subgraph is built in which the removal of any $k - 1$ edges leaves the subgraph in a connected state, and such that there are at least $k - 1$ paths between the remaining nodes. It is similar to the k-vertex connected spanning subgraph problem (definition 70).

k-edge connected spanning subgraphs are important in fault prone networks in which pairs of clients often loose their connection. k-edge connectivity ensures a configurable level of link resilience in the computed subgraph, defined by an input integer $k > 0$.

On the Internet, the network layer address the link resilience (k-edge connectivity) issues, while the client crashes must be handled by the application. Therefore, the following k-connected problem definitions address *k-vertex connectivity*. Figure 4.9 has an example of a spanning subgraph and its properties.

4.9.1 Spanning subgraph problems and the minimum-cost

The minimum-cost requirement to subgraphs is introduced to minimize the network cost, for example, bandwidth consumption. It is very relevant when bandwidth intensive traffic flows like audio/video are distributed in a network.



Given a graph G=(V,E,c) (see figure 4.1).
G is a connected cyclic graph.
G is a 2-vertex connected graph.
G is a 2-edge connected graph.

Found Graph T=(V,E'), where
E'={(A,C),(B,D),(C,E),(E,D),(E,F),(E,G)}.

T is the minimum diameter spanning tree on G.
T is the minimum diameter spanning subgraph on G.

Figure 4.9: Graph $G = (V, E, c)$ and spanning subgraph properties.

Definition 72 ***k-connected minimum-cost spanning subgraph problem:*** *Given G, and an integer $k > 0$. Find a k-connected spanning subgraph $M = (V, E_M)$ of minimum cost.*

The k-connected minimum-cost spanning subgraph problem is NP-complete for $k \geq 2$ for any graph [76]. There has been a lot of work on designing approximation algorithms for the k-connectivity minimum-cost problem. Most of these algorithms are centralized algorithms which are quite sophisticated and their main goal is to obtain polynomial time algorithms with the best possible approximation ratio.

Definition 73 ***k-connected degree-limited minimum-cost spanning subgraph problem:*** *Given G, and an integer $k > 0$. Find a k-connected spanning subgraph $M = (V, E_M)$ of minimum cost, where each vertex $v \in V$ does not violate a degree bound $deg(v) \in \mathbb{N}$.*

The k-connected degree-limited minimum-cost spanning subgraph problem is NP-complete for $k \geq 1$ for any graph [76]. The added degree-limits makes the problem more configurable in terms of node-stress, but it also enhances the complexity of finding an optimal solution. Few approximation algorithms have been proposed to address this problem.

4.9.2 Spanning subgraph problems and the diameter

The spanning-tree problems that address the diameter are similar to the spanning subgraph problems that address the diameter. However, a connected spanning subgraph is allowed to include as many edges as possible. Therefore, a minimum-diameter spanning subgraph problem without constraints does not make sense. A bound on the network cost is necessary, and the diameter should be minimized in the subgraph while not violating this cost bound [96].

Definition 74 ***Minimum diameter spanning subgraph problem:*** *Given G, and a cost bound $B > 0$. Find a connected spanning subgraph $M = (V, E_M)$ such that the diameter(M) = diameter(G), subject to the constraint that the total cost $\sum_{e \in E_T} c(e)$ is below B.*

The objective of the NP-complete minimum diameter spanning subgraph problem is to find a subgraph M on G, that has a diameter equal to that of G. In this process the subgraph M must not violate a total cost bound B. This problem is desirable because it does add resilience to the computed subgraph, and the total cost is monitored. Two problems related to any overlay network design problem are, first, how the total cost bound is obtained, and second, when it is determined, can a subgraph be found that does not violate it.

Definition 75 ***Minimum average pair-wise latency spanning subgraph problem:*** *Given G, and a cost bound $B > 0$. Find a connected spanning subgraph $M = (V, E_M)$ such that the average-pairwise-latency(M) = average-pairwise-latency(G), subject to the constraint that the total cost $\sum_{e \in E_T} c(e)$ is below B.*

The NP-complete minimum average pair-wise latency spanning subgraph problem is a variation that does not address a minimization of the maximum path latency, which is the diameter. The objective is rather to find a subgraph that minimizes the average pair-wise latency, which includes every node in the computed subgraph and not only the endpoints on the diameter path.

Definition 76 ***k-connected minimum-diameter spanning subgraph problem:*** *Given G, an integer $k > 0$, and a total cost bound $B > 0$. Find a k-connected spanning subgraph $M = (V, E_M)$ of G such that the diameter(M) = diameter(G), subject to the constraint that the total cost $\sum_{e \in E_T} c(e)$ is below B.*

The NP-complete k-connected minimum diameter spanning subgraph problem is similar to the minimum-diameter spanning subgraph problem. However, the objective is here to ensure k-connectivity and minimum diameter within a total cost bound. Hence, a level of resilience is required for it be a solution. The resilience level is configurable by an integer $k > 0$.

Definition 77 ***Minimum diameter degree-limited spanning subgraph problem:*** *Given G, a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$, and a cost bound $B > 0$. Find a connected spanning subgraph $M = (V, E_M)$ such that the diameter(M) = diameter(G), subject to the constraint that $deg_T(v) \leq deg(v)$, and the total cost $\sum_{e \in E_T} c(e)$ is below B.*

The objective of the NP-complete minimum diameter degree-limited spanning subgraph problem is to find a subgraph M on G, that has a diameter equal to that of G. In this process the subgraph M must not violate a total cost bound, and each vertex $v \in V$ must not violate a degree limitation. This problem avoids the stress issue of the minimum diameter spanning subgraph problem (definition 74), and exhibit similar benefits and drawbacks; a subgraph that has added resilience, but is hard to compute.

Definition 78 ***k-connected minimum-diameter degree-limited spanning subgraph problem:*** *Given G, an integer $k > 0$, and a total cost bound $B > 0$. Find a k-connected spanning subgraph $M = (V, E_M)$ of G such that the diameter$(M) =$ diameter(G), subject to the constraint that $deg_T(v) \leq deg(v)$, and the total cost $\sum_{e \in E_T} c(e)$ is below B.*

The NP-complete k-connected minimum diameter degree-limited spanning subgraph problem adds degree limitations to the stress beset k-connected minimum-diameter spanning subgraph problem. The benefit is still a configurable level of resilience, which is defined by an integer $k > 0$. But, one drawback is that it is a hard problem to approximate.

4.10 Steiner subgraph problems

A sub-class of the spanning subgraph problems from section 4.9 are the Steiner subgraph problems. Steiner-subgraph problems allow non-member-nodes (Steiner points) in their computed Steiner-subgraph A Steiner-tree is also a Steiner subgraph, the difference is that the Steiner subgraph problems consider graphs with cycles as valid solutions, if the other requirements are met. A cyclic connected graph may be advantegous to increase a network's resilience in fault prone networks. The drawback is that it does inflict a higher network cost, for example, through an increased bandwidth consumption.

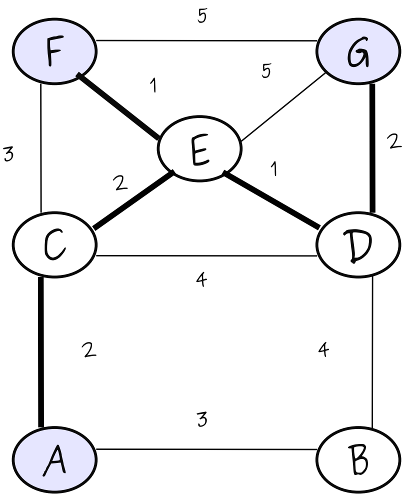
Chapter 11 introduces and evaluates a range of practical Steiner subgraph algorithms. In the following, the class of Steiner subgraph problems are formally introduced.

Definition 79 ***k-vertex connected Steiner subgraph problem:*** *Given G, a set $Z \subset V$ of member-nodes, and an integer $0 < k < |V|$. Find a k-vertex connected Steiner subgraph $M_Z = (V_Z, E_Z)$ where $V_Z \subseteq V$, $Z \subseteq V_Z$ and $E_Z \subseteq E$, in which a removal of any $k - 1$ nodes leaves the subgraph M_Z in a connected state.*

The NP-complete k-vertex connected Steiner subgraph problem requires that a Steiner subgraph is built that exhibits the property of being k-vertex connected among the nodes in Z. This means that the removal of any $k - 1$ nodes leaves the subgraph in a connected state, in which there is at least $k - 1$ paths between the remaining Z nodes.

This problem is important for the targeted application scenarios, in which fault prone clients dynamically join and leave applications. The reason is that the problem ensures a configurable level of resilience in the computed subgraph, defined by an input integer $k > 0$.

Definition 80 ***k-edge connected Steiner subgraph problem:*** *Given G, a set $Z \subset V$ of member-nodes, and an integer $0 < k < |V|$. Find a k-edge connected Steiner subgraph $M_Z = (V_Z, E_Z)$ where $V_Z \subseteq V$, $Z \subseteq V_Z$ and $E_Z \subseteq E$, in which a removal of any $k - 1$ edges leaves the subgraph M_Z in a connected state.*



Given a graph G=(V,E,c) and a set of
member-nodes Z={A,F,G}
G is a connected cyclic graph.
G is a 2-vertex connected graph.

Found graph T=(V,E'), where
E'={(A,C),(C,E),(E,D),(E,F),(E,G)}.

T is the Steiner minimum-cost tree on G.
T is the Steiner minimum-diameter tree on G.
T is the Steiner minimum-diameter subgraph on G.

Figure 4.10: Graph $G = (V, E, c)$ and Steiner-subgraph properties.

The NP-complete k-edge connected Steiner subgraph problem requires that a Steiner subgraph is built that exhibits the property of being k-edge connected among the nodes in Z. This means that the removal of any $k-1$ edges leaves the subgraph in a connected state, in which there is at least $k-1$ paths between the remaining Z nodes. It is similar to the k-vertex connected Steiner subgraph problem (definition 79).

This problem is important for applications that are executed in fault prone networks in which pairs of clients often loose their connection. k-edge connectivity ensures a configurable level of link resilience in the computed subgraph, defined by an input integer $k > 0$.

On the Internet, the network layer address the link resilience (k-edge connectivity) issues, while the client crashes must be handled by the application. Therefore, the following k-connected problem definitions address *k-vertex connectivity*. Figure 4.10 has an example of a Steiner subgraph and its properties.

4.10.1 Steiner subgraph problems and the minimum-cost

The minimum-cost requirement to subgraphs is introduced to minimize the network cost, for example, bandwidth consumption. It is very relevant when bandwidth intensive traffic flows like audio/video are distributed in a network.

Definition 81 ***k-connected minimum-cost Steiner subgraph problem:*** *Given G, and an integer $k > 0$. Find a k-connected Steiner subgraph $M_Z = (V_Z, E_Z)$ of minimum cost (theorem 24).*

The k-connected minimum-cost Steiner subgraph problem is NP-complete for $k \geq 2$ for any graph [76]. There has been a lot of work on designing approximation algorithms for the k-connectivity problem. Most of these algorithms are centralized algorithms which are quite sophisticated and their main goal is to obtain polynomial time algorithms with the best possible approximation ratio.

Definition 82 ***k-connected degree-limited minimum-cost Steiner subgraph problem:*** *Given G, and an integer $k > 0$. Find a k-connected Steiner subgraph $M_Z = (V_Z, E_Z)$ of minimum cost, where each vertex $v \in V$ does not violate a degree bound $deg(v) \in \mathbb{N}$.*

The k-connected degree-limited minimum-cost Steiner subgraph problem is NP-complete for $k \geq 1$ for any graph [76]. The added degree-limitations makes it more configurable in terms of node-stress, but it also increases the complexity of finding an optimal solution. Few approximation algorithms have been proposed to address this problem.

4.10.2 Steiner subgraph problems and the diameter

The Steiner-tree problems that address the diameter are similar to the Steiner subgraph problems that address the diameter. However, a connected Steiner subgraph is allowed to include as many edges as possible. Therefore, a minimum-diameter Steiner subgraph problem without constraints does not make sense. A bound on the network cost is necessary, and the diameter should be minimized in the subgraph while not violating this cost bound [96].

Definition 83 ***Minimum diameter Steiner subgraph problem:*** *Given G, and a cost bound $B > 0$. Find a connected Steiner subgraph $M_Z = (V_Z, E_Z)$ such that the diameter(M_Z) = diameter(G), subject to the constraint that the total cost $\sum_{e \in E_T} c(e)$ is below B.*

The objective of the NP-complete minimum diameter Steiner subgraph problem is to find a subgraph M_Z on G, that has a diameter equal to that of G. In this process the subgraph M_Z must not violate a total cost bound. This problem is desirable because it does add resilience to the computed subgraph, and the total cost is monitored. Two problems related to any overlay network design problem are how the total cost bound is obtained, and when it is determined, can a subgraph be found that does not violate it.

Definition 84 ***Minimum average pair-wise latency Steiner subgraph problem:*** *Given G, and a cost bound $B > 0$. Find a connected Steiner subgraph $M_Z = (V_Z, E_Z)$ such that the average-pairwise-latency(M_Z) = average-pairwise-latency(G), subject to the constraint that the total cost $\sum_{e \in E_T} c(e)$ is below B.*

The NP-complete minimum average pair-wise latency Steiner subgraph problem is a variation that does not address a minimization of the maximum path latency, which is the diameter. The objective is rather to find a subgraph that minimizes the average pair-wise latency, which includes every node in the computed subgraph and not only the endpoints on the diameter path.

Definition 85 ***k-connected minimum-diameter Steiner subgraph problem:*** *Given G, an integer $k > 0$, and a total cost bound $B > 0$. Find a k-connected Steiner subgraph $M_Z = (V_Z, E_Z)$ of G such that the diameter(M) = diameter(G), subject to the constraint that the total cost $\sum_{e \in E_T} c(e)$ is below B.*

The NP-complete k-connected minimum diameter Steiner subgraph problem is similar to the minimum-diameter Steiner subgraph problem. However, the objective is here to ensure k-connectivity and minimum diameter within a total cost bound. Hence, a level of resilience is required for it be a solution. The resilience level is configurable by an integer $k > 0$.

Definition 86 ***Minimum diameter degree-limited Steiner subgraph problem:*** *Given G, a degree bound $deg(v) \in \mathbb{N}$ for each vertex $v \in V$, and a cost bound $B > 0$. Find a connected Steiner subgraph $M_Z = (V_Z, E_Z)$ such that the diameter(M) = diameter(G), subject to the constraint that $deg_T(v) \leq deg(v)$, and the total cost $\sum_{e \in E_T} c(e)$ is below B.*

The objective of the NP-complete minimum diameter degree-limited Steiner subgraph problem is to find a subgraph M on G, that has a diameter equal to that of G. In this process the subgraph M must not violate a total cost bound, and each vertex $v \in V$ must not violate a degree limitation. This problem avoids the stress issue of the minimum diameter Steiner subgraph problem (definition 74), and exhibit similar benefits and drawbacks. A computer subgraph that has added resilience, but is hard to compute.

Definition 87 ***k-connected minimum-diameter degree-limited Steiner subgraph problem:*** *Given G, an integer $k > 0$, and a total cost bound $B > 0$. Find a k-connected Steiner subgraph $M_Z = (V_Z, E_Z)$ of G such that the diameter(M) = diameter(G), subject to the constraint that $deg_T(v) \leq deg(v)$, and the total cost $\sum_{e \in E_T} c(e)$ is below B.*

The NP-complete k-connected minimum diameter degree-limited Steiner subgraph problem adds degree limitations to the stress beset k-connected minimum-diameter Steiner subgraph problem. The benefit is still a configurable level of resilience, which is defined by an integer $k > 0$. But, one drawback is that it is a hard problem to approximate.

4.10.3 Miscellaneous Steiner-subgraph problems

There are many different Steiner-subgraph problems in the literature that address slightly different problems than the ones introduced before. Here are two Steiner-subgraph problems that are of relevance for our application scenario.

Definition 88 ***Terminal Steiner subgraph in networks problem:*** *Given an undirected weighted graph $G = (V, E, c)$, and a subset $Z \subseteq V$ of member-nodes. Find a connected undirected subgraph M_Z, such that every member-node is a leaf and there is a path between every pair of member-nodes.*

The objective of the terminal Steiner-subgraph problem is to find a subgraph of G in which every member-node is a leaf-node. This problem has importance when member-nodes are not allowed to, or cannot communicate directly with each other.

Some distributed interactive applications may face consistency issues, cheating problems, etc, such that the application-provider needs to control the application data in its back-bone infrastructure of servers and proxies. The terminal Steiner-subgraph problem is easily re-defined to address k-connectivity problems of minimum-cost and minimum-diameter.

Chapter 8 introduces graph reduction algorithms that force the member-nodes to be leaf-nodes. Similarly, in the cases of degree-limited Steiner-subgraph heuristics, it is possible to reduce the degree-limits to one on each member-node. Chapter 11.3 includes an evaluation of the graph reduction algorithms.

Definition 89 ***Group Steiner subgraph in networks problem:*** *Given an undirected weighted graph $G = (V, E, c)$, and subsets of vertices, which are called groups $g_1, g_2 \dots g_k \in V$. Find a connected undirected subgraph $M = (V', E', c)$, such that $V' \cap g_i \neq$ for all $i \in 1 \dots k$.*

The objective of the group Steiner-subgraph problem is to find a subgraph of G that contains at least one vertex from each given group. This problem is important in cases when groups of clients in a distributed interactive application are merged to one group. For example, instead of merging k groups to one group and then rebuild the merged group from scratch, it is possible to connect the k groups through single edges to form one merged group. The group Steiner-subgraph problem is easily re-defined to address k-connectivity problems of minimum-cost and minimum-diameter.

Chapter 13 introduces and evaluates dynamic subgraph algorithms for removal of nodes in subgraphs. When a non-leaf node $v \in V$ is removed from a subgraph T, the subgraph is disconnected creating $deg_T(v)$ subgraphs, which are equal to the groups in the group Steiner subgraph problem. The reconnection of the subgraph groups is exactly the group Steiner-subgraph problem.

4.11 Dynamic subgraph problems

Dynamic subgraph problems address the client dynamicity that is a part of the application scenario outlined in chapter 2. The dynamic-subgraph problems define requirements to situations where single nodes are inserted and removed from an existing subgraph, based on incoming insert and remove requests. The dynamics subgraph problems are a refinement of the dynamic tree problems (section 4.7), and allow both cyclic and acyclic connected graphs. Cyclic graphs add a level of resilience to the connected subgraph, but the drawback is added network cost, for example, increased bandwidth consumption. The problems are very relevant for dynamic group communication scenarios where fault-prone clients join and leave ongoing sessions of real-time interaction.

Chapter 13 introduces and evaluates a range of dynamic subgraph algorithms for inserting and removing nodes from subgraphs. Dynamic subgraph problems are introduced next, specifically dynamic Steiner-subgraph problems, and dynamic spanning-subgraph problems.

4.11.1 Dynamic Steiner subgraph problems

Definition 90 ***Dynamic Steiner-subgraph problem*** [3]***:*** *An instance of the dynamic Steiner-subgraph problem includes a graph* $G = (V,E,c)$*, a set* $Z_i \subseteq V$*, and a queue of requests* $Q_r = \{r_0, r_i, \dots, r_k\}$*, where each* r_i *is a pair* (v_i, ρ_i)*,* $v_i \in V$*,* $\rho_i \in \{insert,remove\}$*. The set* $Z_i = \{v | (v,insert) = r_j \, for\ some\ j, 0 \leq j \leq i\ and\ (v,remove) \neq r_l \, for\ all\ l, j < l \leq i\}$*, where* Z_i *is the terminal set at step* i *which are to be connected with a Steiner-subgraph after request* r_i*.*

An instance of the dynamic Steiner subgraph problem includes a series of requests r_i that contains a node v_i to be inserted or removed from Z_i. A dynamic Steiner subgraph algorithm finds a Steiner-subgraph connecting each terminal set Z_i without knowledge of request r_j for any $j > i$. The dynamic Steiner subgraph problem is applicable to most optimization goals.

This basic dynamic Steiner subgraph problem formulation is equivalent to a Steiner-subgraph problem for each request r_i, therefore the dynamic Steiner subgraph problem must be extended with some conditional restrictions for it to be of practical significance.

Definition 91 ***Non-reconfigurable dynamic Steiner-subgraph problem:*** *Given an instance* $I = (G,,cost,Q_r)$*, find a sequence of subgraphs* $\{M_1, M_2, \dots, M_k\}$ *satisfying the following conditions* C_N *and minimizing a function of* $\{cost(M_i) | i = 1,2\dots k\}$. $C_N = \{$ *1. Each* M_i *spans* Z_i*, 2. If* r_i *is an insert request,* M_i *includes* M_{i-1} *as a subgraph, and 3. If* r_i *is a remove request,* M_{i-1} *includes* M_i *as a subgraph.* $\}$.

The non-reconfigurable dynamic Steiner subgraph problem is a special case of the dynamic Steiner subgraph problem. Conditions 2 and 3 in C_N imply that edges and nodes are inserted to a subgraph only for an insert request, and removed only for a remove request. In practice it means that, upon an insert request a node is added to the subgraph as a leaf-node, and a non-leaf node cannot be removed, but rather remains in the subgraph as a Steiner point until it is a leaf node (possibly never).

It is evident that the non-reconfigurable dynamic Steiner subgraph problem face problems of subgraph degradation in cases where many Steiner-points remain in the Steiner-subgraph forever. The conditions C_N, although practical and simplistic, are unrealistic in a real application where clients join and leave continuously.

[3] Not found in the literature

Definition 92 ***Reconfigurable dynamic Steiner subgraph problem:*** *Given an instance $I = (G,,cost, Q_r)$, find a sequence of subgraphs $\{M_1, M_2, \ldots, M_k\}$ where each M_i spans Z_i and minimizes a function of $\{cost(M_i)|i \leq k\}$. while not exceeding an upper bound B on the number of rearrangments.*

The reconfigurable dynamic Steiner subgraph problem allows reconfigurations to the edge and node sets, regardless of r_i being an insert or remove request. However, if the number of reconfigurations allowed is unlimited the reconfigurable dynamic Steiner subgraph problem is equivalent to the Steiner-subgraph problem for each instance (G,cost, Z_i). Therefore, a bound B is defined to limit the number of rearrangements allowed for each request r_i.

The reconfigurable dynamic Steiner subgraph problem problem fits an application scenario in which subgraphs may experience frequent insert and remove requests. Tree stability then becomes important, and the bound B sets a worst-case limit to the number of rearrangements. One issue of the reconfigurable dynamic Steiner subgraph problem problem is that it does not consider the quality of the Steiner-subgraph when the number of rearrangments has equalled B.

Definition 93 ***Minimum-reconfiguration dynamic Steiner subgraph problem:*** *Given an instance $I = (G,,cost, Q_r)$, find a sequence of subgraphs $\{M_1, M_2, \ldots, M_k\}$ where each M_i spans Z_i and minimizes the number of rearrangements, while not violating the cost function $\{cost(M_i)|i \leq k\}$.*

The minimum-reconfiguration dynamic Steiner subgraph problem minimizes the number of rearrangments needed to fulfill a given optimization goal. The optimization goal may, for example, provide an upper bound to the diameter of the subgraph M_i.

Chapter 2 introduced the stringent latency requirements of different distributed interactive applications. In these cases, the minimum-reconfiguration dynamic Steiner subgraph problem problem ensures that the maximum effort is done to meet the latency requirements. One drawback is that in the worst case, the optimization goal cannot be met, and a complete reconfiguration is conducted.

4.11.2 Dynamic spanning-subgraph problems

Definition 94 ***Dynamic spanning-subgraph problem:*** *An instance of the dynamic spanning-subgraph problem includes a graph $G = (V, E, c)$, and a queue of requests $Q_r = \{r_0, r_i, \ldots, r_k\}$, where each r_i is a pair (v_i, ρ_i), $v_i \in V$, $\rho_i \in \{insert,remove\}$. There is a subgraph $G_i = (V_i, E_i, c)$, in which $V_i = \{v \in V|(v,insert) = r_j \text{ for some } j, 0 \leq j \leq i \text{ and } (v,remove) \neq r_l \text{ for all } l, j < l \leq i\}$, where V_i is the node set after step i, and $E_i = \{e \in E|e = (u, v)$, such that $u \in V_i, v \in V_i\}$, where E_i interconnects all nodes in V_i. For each request r_i, construct a spanning subgraph M_i on G_i.*

The dynamic spanning-subgraph problem is similar to the dynamic Steiner-subgraph problem. Only, in the dynamic spanning-subgraph problem a subgraph M_i cannot include nodes that are Steiner-points (non-member nodes, non-terminals). This influences the problem definition in the cases where the request r_i is a remove request and the node v (to be removed) is a non-leaf node. These cases force a reconfiguration to the subgraph M_i. Therefore, the dynamic spanning-subgraph problem does not have a non-reconfigurable dynamic spanning-subgraph problem. Rather, dynamic spanning-subgraph problems must be defined such that any node can be removed regardless of its degree in M_i.

Definition 95 ***Restricted reconfigurable dynamic spanning-subgraph problem:*** *Given an instance $I = (G,,cost,Q_r)$, find a sequence of subgraphs $\{M_1, M_2, \ldots, M_k\}$ where each M_i spans V_i and minimizes a function of $\{cost(M_i)|i \leq k\}$, while not exceeding an upper bound B_i on the number of rearrangments. B_i is determined such that if r_i is an insert request then $B_i = 1$, if r_i is a remove request then $B_i = (|d_{M_i}(v_i)| * 2) - 1$.*

The restricted reconfigurable dynamic spanning-subgraph problem minimizes the number of rearrangements of M_i to a minimum for each r_i. For each insert request, a node is added to the subgraph through a single edge, and for each remove request, the neighbors of the removed node are reconnected. Note that, when a node $v_i \in M_i$ has been removed and created $deg_{M_i}(v_i)$ sub-subgraphs, the reconnection of these sub-subgraphs is equivalent to the group Steiner-subgraph problem that was introduced in definition 89. The restricted reconfigurable dynamic spanning-subgraph problem requires that the subgraph remains a spanning-subgraph of member-nodes, and that the reconfigurations are minimum upon insert and remove requests.

Definition 96 ***Reconfigurable dynamic spanning-subgraph problem:*** *Given an instance $I = (G,,cost,Q_r)$, find a sequence of subgraphs $\{M_1, M_2, \ldots, M_k\}$ where each M_i spans V_i and minimizes a function of $\{cost(M_i)|i \leq k\}$, while not exceeding an upper bound B_i on the number of rearrangments. B_i is determined such that if r_i is an insert request then (trivially) $B_i \geq 1$, if r_i is a remove request then $B_i \geq (|d_{M_i}(v_i)| * 2) - 1$.*

The reconfigurable dynamic spanning-subgraph problem allows reconfigurations to M_i bounded by an integer B_i. If the bound $B_i > |V_i| * 2$ the reconfigurable dynamic spanning-subgraph problem is equivalent to the spanning-subgraph problem (definition 20) for each instance $(G$,cost, $Z_i)$. Furthermore, for a remove request, if the bound $B_i < (|d_{M_i}(v_i)| * 2) - 1$, then a solution does not exist. Therefore, the bound B_i must be determined for each request r_i, such that, for a remove request, it is greater than the degree of v_i in M_i.

The reconfigurable dynamic spanning-subgraph problem requires that the number of rearrangements of the Steiner-subgraph is $\leq B_i$ when a request r_i has been completed. However, it may be difficult to determine this bound B_i for each request r_i. In practice, it may be wise to use a percentage of additional reconfigurations above the necessary.

Definition 97 ***Minimum-reconfiguration dynamic spanning-subgraph problem:*** *Given an instance $I = (G,,cost,Q_r)$, find a sequence of subgraphs $\{M_1,M_2,\ldots,M_k\}$ where each M_i spans V_i and minimizes the number of rearrangements, while not violating the cost function $\{cost(M_i)|i \leq k\}$.*

The minimum-reconfiguration dynamic spanning-subgraph problem minimizes the number of rearrangments needed to achieve a given cost function. The cost function may, for example, provide an upper bound to the diameter of the subgraph M_i.

4.12 Dynamic subgraph insert- and remove-node problems

The dynamic-subgraph problems introduced in section 4.11 defined requirements to subgraph-updates when single nodes are inserted and removed, for example, concerning bounds on the number of rearrangments, bounds on the cost of the subgraph, etc. Like a dynamic-tree algorithm, a dynamic subgraph algorithm is also comprised of one insert-strategy, which inserts a node into a subgraph, and one remove-strategy, which removes a node from a subgraph. The strategies must be paired to create one dynamic subgraph algorithm to address the dynamic-subgraph problems. However, one issue is that the insert and remove strategy may belong in different dynamic subgraph problems. Therefore, similar to dynamic-tree algorithms (section 4.8) we:

Define formal insert and remove-node problems that more accurately categorizes the insert and remove strategies that are needed to address the dynamic subgraph problems.

In this regard, it is especially the number of rearrangements that are allowed upon inserting or removing a node that influences an algorithm's performance and defines a subgraph's stability.

Following are definitions of a wide range of insert-node to subgraph and remove-node from subgraph problems that exhibit the accuracy that is desirable for describing the insert and remove strategies in dynamic subgraph algorithms.

4.12.1 Insert-node to subgraph problems

Insert strategies insert a new member m to a subgraph and assure that m is connected. Formally, an insert strategy works like this:

Given $G = (V,E,c)$, a subgraph $M = (V_M,E_M)$, a set of members $Z_M \subseteq V_M$, and a new member $m \in V$. Update M, such that $Z_M \cup \{m\}$ are connected.

The new member m may be inserted into M in many different manners. We have devised several insert strategies that bound the size of the reconfiguration set R. The reconfiguration set R contains the edges that are changed between reconfigurations of a subgraph. It is possible to insert m to M through a single edge, while other strategies use the degree limit as a bound.

Definition 98 ***Insert vertex edge-additions:*** *Given G, a connected subgraph M, an integer $k > 0$, and a joining member $m \in V$, Update M, such that $Z_M \cup \{m\}$ are connected, and $|R| = k$.*

The insert vertex edge-additions problem connects (inserts) a node m to the subgraph using k edges. Insert strategies belonging to this problem typically search for the k edges that, for example, are the minimum-cost edges, or the edges that results in the least eccentricity for the inserted node.

Definition 99 ***Insert vertex and limited reconfiguration set:*** *Given G, a connected subgraph M, an integer $k > 0$, and a joining member $m \in V$, Update M, such that $Z_M \cup \{m\}$ are connected, and $|R| \leq deg(m) * (2+k)$.*

The insert vertex and limited reconfiguration set restricts the size of the reconfiguration set R to be less than the $k+2$ times the current degree-limit of m. This allows for an insert strategy to, for example, connect m to M as an intersection node and then re-connect using k edges from m.

Definition 100 ***Insert vertex and unlimited reconfiguration set:*** *Given G, a connected subgraph M, and a joining member $m \in V$, Update M, such that $Z_M \cup \{m\}$ are connected, and $|R| \leq |E_M| * 2$.*

The insert vertex and unlimited reconfiguration set problem put no restrictions on the size of the reconfiguration set R upon connecting m to M. If an insert strategy fits to this insert problem, it is clear that its worst-case behavior is that every edge in M is exchanged when m is inserted. However, the insert strategy may still not be equal to a Steiner-subgraph or spanning-subgraph algorithm if the average-case size of R is rather small.

4.12.2 Remove-node from subgraph problems

Remove strategies remove a member m from the multicast subgraph while assuring that the members stay connected. Formally, a remove strategy works like this:

Given $G = (V, E, c)$, a subgraph $M = (V_M, E_M)$, a set of members $Z_M \subseteq V_M$, and a member $m \in V$. Update M, such that $Z_M \setminus \{m\}$ are connected.

The number of direct neighbor nodes of m (its degree) in M influences the necessary actions. If the degree $deg_M(m) = 1$, m is a leaf that is simply removed along with the edge to its only neighbor. If it is greater than 1, a removal of m partitions the subgraph if no additional steps are taken, and $deg_M(m)$ unconnected subsubgraphs would be the result. The basic goal of remove strategies is the reconnection of subsubgraphs into a single subgraph when $deg_M(m) \geq 2$.

Similar to the insert strategies, we divide the remove strategies into the worst case sizes of the reconfiguration set R. The minimum reconfiguration set when removing m consists of its immediate neighbors in M. We have a number of remove strategies that bound the size of the reconfiguration set to the worst case number of edges it may change when disconnecting and reconnecting the neighbors of m. Moreover, it is possible to reconfigure a larger portion of M while removing m, and we also include such remove strategies.

Definition 101 ***Remove vertex and edge-removal:*** *Given G, a connected subgraph M, and a leaving member $m \in V$, Update M, such that $Z_M \setminus \{m\}$ are connected, and $|R| \leq 1$.*

The remove vertex and edge-removal is the most restricted remove problem, and only allows the removal of leaf non-member nodes. All other non-member-nodes are kept in the subgraph until they are leaf nodes. In highly connected subgraphs, there are few leaf-nodes, such that, an algorithm of this problem is not able to remove non-member-nodes, which results in higly degraded graphs.

Definition 102 ***Remove vertex and degree limited reconfiguration set:*** *Given G, a connected subgraph M, and a leaving member $m \in V$, Update M, such that $Z_M \setminus \{m\}$ are connected, and $|R| \leq deg(m) * 2$.*

The remove vertex and degree-limited reconfiguration set bounds the size of the reconfiguration set to include, for example, a disconnection of m and then reconnecting the neighbors of m. It is also possible, for a remove strategy to include a Steiner-point to reconnect the neighbors of m. More advanced reconfigurations are also possible that does not include exchanging edges near m, but these are not addressed in the thesis.

Definition 103 ***Remove vertex and unlimited reconfiguration set:*** *Given G, a connected subgraph M, and a leaving member $m \in V$, Update M, such that $Z_M \setminus \{m\}$ are connected, and $|R| \leq |E_M| * 2$.*

The remove vertex and unlimited reconfiguration set problem put no restrictions on the size of the reconfiguration set R upon disconnecting m from M, and then reconnecting M. If a remove strategy fits to this remove problem, it is clear that its worst-case behavior is that every edge in M is exchanged when m is removed. However, the remove strategy may still not be equal to a Steiner-subgraph or spanning-subgraph algorithm if the average-case size of R is rather small.

4.13 Summary of the main points

The chapter introduced the research area of overlay network design. The motivation was to introduce relevant graph theory problems that address the issues faced by developers of distributed interactive applications. We also use the formal definitions of the problems when we introduce and evaluate algorithms in later chapters.

More specifically, we presented the graph theoretical terms and symbols that are addressed during the research in the thesis. Moreover, we gave a brief introduction to a few important graph algorithmic terms related to graph algorithm complexity, algorithm types and some of the metrics that are typically addressed by graph algorithms. We identified the overlay construction algorithms that often are the algorithmic foundation for other more recent algorithms: Dijkstra's SPT, Prim's MST and Kruskal's MST.

A wide range of graph theoretical problems related to graph search and overlay construction were also presented. It is clear that not all of the graph theoretical problems are equally relevant. However, we do believe that by providing such a comprehensive study it will make it easier to distinguish and identify the more relevant problems for specific applications. In the course of the thesis, we shall evaluate algorithms that address almost every graph theoretical problem. By doing this and also by providing references to the problems the algorithms address, we hope it will make it more clear how the algorithms behave.

Chapter 5

Distributed interactive system: Group management techniques

The motivation for this chapter is found in chapter 2, which introduced a range of requirements and design issues for distributed interactive applications. There it was identified that clients in these applications should have the possibility of joining and leaving an ongoing session of real-time interaction. When clients join and leave, the real-time interaction must not be disrupted, and the joining clients must be included to the applicaction in a timely fashion such that they can start interacting instantaneously. From these motivational points, it is clear that:

If a distributed interactive application is to support real-time interactivity and group dynamics, it yields great challenges to the distributed interactive system's design.

The following sections introduce specific requirements to distributed interactive systems, and also 4 proposed sub-systems: *membership management*, *resource management*, *overlay network management* and *network information management*. These sub-systems include different techniques and algorithms, many of which were surveyed in the state-of-the-art and related work in chapter 3. The goal of the chapter is to address the requirements and design issues motivated by chapter 2, and describe specific teqhniques and algorithms that together are able to adress them.

More specifically, we evaluated 3 *membership management* variations theoretically and through experiments on PlanetLab. From these observations, we concluded that a centralized approach is most fitting in a dynamic and interactive scenario. Moreover, we proposed that the *resource management* should use core-node selection algorithms, to find nodes in the network that yield low pair-wise latencies to groups of clients (chapter 7). Furthermore, we proposed that the *overlay network management* consists of graph manipulation algorithms that both enhance and reduce complete group graphs, and also overlay construction algorithms that use the group graphs to build low-latency overlay networks for event distribution. Generally, we deduced that distributed interactive applications have such strict lantency requirements that the resource and overlay management should mainly consist of centralized graph algorithms. The graph theoretical problems for all of these graph algorithms are found in chapter 4. Finally, we proposed that a *network information management* should contain latency estimation techniques that obtain accurate all-to-all path latencies such that centralized graph algorithms can use them.

The rest of the chapter is organized in the following manner. Section 5.1 introduces a few of the requirements that a distributed interactive system has and also the specific research areas that are the focus of the thesis: resource management, membership management, overlay network management and network information management. Section 5.2 evaluates 3 approaches for membership management in distributed interactive applications. Section 5.3 introduces the resource management techniques that are evaluated in chapter 6 and 7. Section 5.4 introduces the overlay management techniques that are evaluated in chapter 8 through chapter 14. Section 5.5 briefly introduces the network information management, along with the latency estimation techniques. They are evaluated later in chapter 6. Finally, section 5.6 gives a brief summary of the main points.

5.1 Requirements to a distributed interactive system

We discussed in chapter 2 that an application that is to support distributed real-time interaction must use a distributed system that handles the interactivity and dynamics of clients. An application developer requires a range of basic techniques and algorithms to enable a distributed system that supports clients to join and leave ongoing sessions of real-time interaction. Generally, we observed that:

When clients join and leave, efficient mechanisms should ensure that the service to the remaining clients is not disrupted, and that the joining clients are included such that they, in a timely fashion, can start the real-time interaction with the clients online.

These straight-forward requirements of interactivity and dynamics together impose great challenges that must be enabled by basic mechanisms and then adapted to a specific system's design. The interactivity poses requirements to the latency, and the dynamics pose requirements to the configuration of the event-distribution paths. Together, the interactivity and dynamics should be handled by a system that support configuration of low-latency networks for event-distribution. In other words, the clients should be within a latency bound to each other in the member network. The system support for interactivity and dynamics must be enabled by basic mechanisms for network configurations and management.

Section 2.5 introduced 4 specific research areas and goal statements that we intend to address in the thesis. In the following, we re-state the 4 identified research areas and then delve into the techniques and algorithms that are required to address them.

- A ***membership management*** must ensure that clients are able to join and leave ongoing sessions of real-time interaction, in a timely fashion (section 5.2).

- A ***resource management*** must ensure that well-placed nodes are found that yield low latencies to groups of clients, such that they are available for management tasks (section 5.3).

- An ***overlay network management*** must ensure that clients are configured in overlay networks that yield sufficiently low-latencies for real-time interaction (section 5.4).

- A ***network information management*** must ensure that Internet path latencies between the interacting clients are available and sufficiently accurate (section 5.5).

These approaches are now introduced and put in the right context with references to the algorithms and techniques evalutated in the thesis.

5.2 Membership management

In section 2.5.2, we described the goal for the membership management to be:

1) Identify techniques that enable an efficient and timely membership management of multiple dynamic subgroups of clients.

To achieve this goal, the membership management must include efficient techniques that can process incoming join and leave requests. Such requests are sent from interacting clients in the application, and their functionality can be summarized such:

*A **join request** is sent by a client that wishes to join a group of clients and interact with them, and a **leave request** is sent by a client that wishes to leave a group of clients and stop interacting with them.*

After processing the requests, the membership management must then be in a state where the membership views in the network are sufficiently consistent, such that it is able to continuously service join and leave request in a timely manner.

5.2.1 Requirements to the membership management

The requirements to the membership management in distributed interactive applications vary depending on the number of groups and their dynamics (client churn). However, the requirements and complexity of the methods for such membership management systems is envisioned to vary especially depending on the number of active groups in the application.

- ***Single group*** interactive applications exhibit looser requirements to the membership management.
- ***Multiple subgroups*** of interacting clients in an application exhibit stricter requirement to the membership management.

In a single (flat) group situation it suffices to form low-latency event-distribution paths in which all clients are reachable. A membership management system is therefore enabled by efficient distributed mechanisms for handling client churn. The drawback of having a single flat group is that every client in the network receive every event, even though a client may not be interested in large portions of these events. This consumes unnecessary link bandwidth in the client network. The distributed interactive system should therefore be enhanced such that it is enabled to divide the clients into subgroups, where each group has its own low-latency network for the events they are interested in.

Applying multiple subgroups to distribute events in an application, pose membership management challenges related to how the groups are updated and how the event distribution overlays are constructed.

The distributed interactive system should be enhanced to include mechanisms that can search for and elect nodes to administrate clients that join and leave subgroups. In that respect, the resource management (section 5.3) has methods to identify well-placed core-nodes that may be elected to administrate and execute the membership management. Having a limited set of nodes to handle the membership management, makes membership updates achievable in a dynamic scenario.

5.2.2 Membership management techniques

Group management of dynamically changing sub-groups of clients is important to enable a variety of scalable distributed interactive applications. Essentially, there are three different approaches to group management; centralized, distributed and a hybrid approach named hierachical management. Which to choose is an important design choice, and influences the target applications, in terms of their scalability, robustness, etc.

In a *centralized* approach, a node is assigned to control the membership information, and assist the application's group members to form an overlay multicast topology. ALMI [101] is a centrally managed application-level group communication middleware, tailored towards the support of relatively small multicast groups with many-to-many semantics. Centralized approaches avoid many consistency issues, but the scalability may suffer. Other typical centralized issues are, single point of failure, potential bottleneck problems and resulting slower management. However, a multitude of fault tolerance and quality of service mechanisms are available that reduces the chance of loosing valuable data [149].

In a *distributed* approach the overlay and membership information is dynamically distributed to the members. Many file-sharing and video-streaming peer-to-peer applications use this distributed approach [124, 33, 39, 97, 85]. However, these applications do not support interactivity, and due to this there is no need to create sub-groups of clients. Generally, such completely distributed overlay applications are scalable, but they do have inherent consistency issues that must be handled in order to support interactivity.

A *hierarchical* approach aims to address the centralized scalability issues, and distributed consistency issues. In a hierarchical approach the members form a hierarchical structure and assign specific tasks/roles to the group members. This way, the group management is distributed among a few nodes, which effectively decreases the overlay network control traffic. AMcast [115] is such an approach, which uses a set of distributed multicast service nodes.

The *membership management architecture* in the thesis is not linked explicitly to any architecture, but it does rely on having a *membership manager* that is in charge of a group's membership management. The membership manager executes join and leave requests and updates the group view accordingly. It is also possible to elect multiple managers, and dynamically migrate tasks between the managers [17].

5.2.3 Membership management variations

There are many possible membership management variations that are based on appointing tasks to single nodes for each group. Following are some variations of membership management architectures that are envisioned to fit particularly well to enable support for dynamically changing subgroups of clients. In all of the variations, a manager-node has been appointed to service incoming join and leave requests from a group of clients.

Membership management actions

For interactive applications that enable multiple sub-groups of clients, it is envisioned that join and leave requests often are coupled to one *membership-change request*. The result of the membership-change request is that a client is removed (left-out) from a group, and joined to another. Upon reception of a membership-change request the membership management must act such that the affected groups of clients are updated.

There are a number of actions that need to be taken to service a membership-change request. The sequence of these actions depend on the system, but the actions themselves share important commonalities. The basic sequence of actions that occur when a manager recieves a membership-change request are that:

*The membership-change **request** should be **accepted** by the membership management before it is **initiated**, and at the end, the membership management should be notified upon **completion**.*

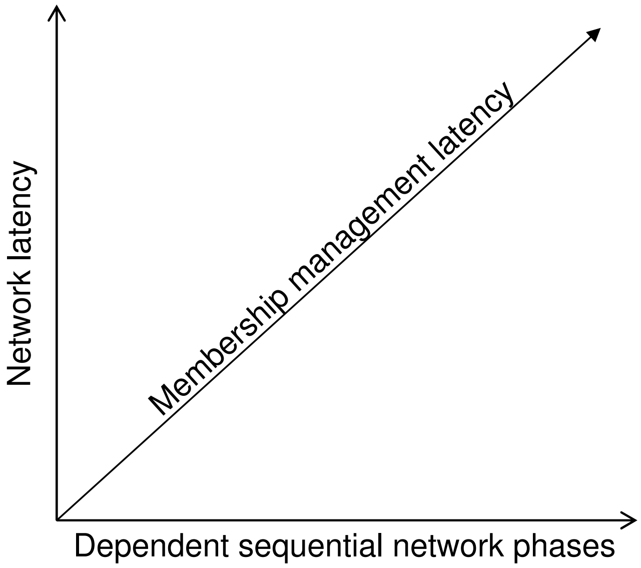


Figure 5.1: The membership management latency is influenced by the number of dependent sequential network phases (steps).

In the following, three membership management variations are presented with the purpose of evaluating their applicability for use in diststributed interactive applications. For each of them, the membership consistency between the manager and the group members is discussed in some detail. Furthermore, the number of required steps in a system influences the latency of each successful completion of a request, and this is put in context with each of the membership management variations. Figure 5.1 illustrates how the membership management latency is influenced by the number of dependent sequential network phases (steps). For example, two dependent sequential network steps occur when a client sends a membership change request to the manager, who accepts it and sends a reply back.

A) Distributed sequential membership management

The distributed sequential management (Figure 5.2 A) is an architecture in which each client pulls membership data from the server when it requires to change a group. The clients then use this membership information to complete the membership change request, and reports the result back to the server. It is an architecture that may be implemented using a *release consistency* model [122]. A system that is implementable using release consistency requires synchronized read and write operations on shared data.

In more details, clients in the distributed sequential membership management sends a membership change *request* to a manager, in which it is entered into a request queue. The request queue may be implemented in several different manners, but it must follow these rules:

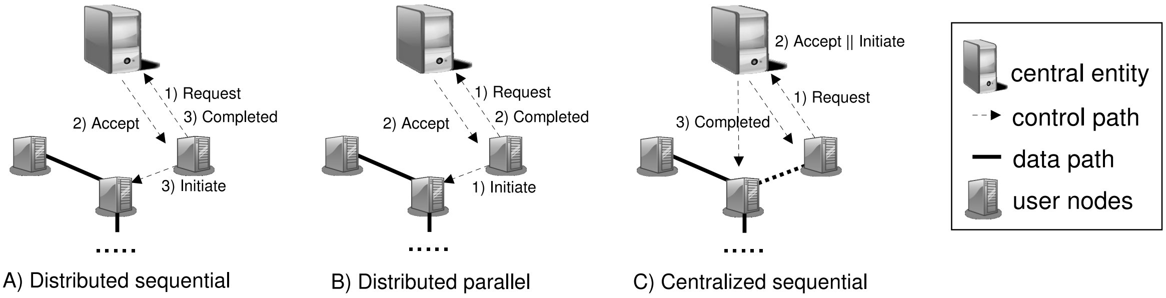


Figure 5.2: Group management techniques with control and data paths

Each request from a single source must be ***accepted*** *in the order they were received, and each request on a group must be* ***completed*** *such that the membership view on the server is consistent with the sub-view on the clients.*

Upon servicing a membership change request in the queue, the server sends an *accept* message to the client, which contains the latest data on a specific group's membership that is available on the server. Upon reception of the accept message, the client *initiates* the membership change by updating the membership data. The client should now inform the group members that it has entered, while it in parallel notifies the server that it has *completed* the membership change.

The system has three steps that are executed sequentially and require cooperation over an overlay network link in the Internet. These steps are: 1) client sends membership change request to the server, 2) server sends accept back to the client, 3) client sends request completed to the server and the group members. The drawback and latency bottleneck of this approach lies in these three network steps and that there is a potential of delayed requests due to the consistency requirements. There is actually no computation on membership data in the first two steps of a membership change. It is in step three that the group membership data is being updated. The upside is the consistency and the load balancing the system yields.

B) Distributed parallel membership management

The distributed parallel membership management (Figure 5.2 B) is an architecture in which each client pushes membership updates to the server when it has already decided to change its group. Therefore, all clients should have sufficient global knowledge of the current group memberships, such that each client can change memberships on their own (without consulting a server). One way to achieve this, is that the server continouosly pushes its entire membership database to every member in all groups for each group update it receives. This is an architecture that most likely may be implementable using a weak consistency model [122]. In a system implemented using a weak consistency model, the shared-state can only be counted on to be consistent after a synchronization is done. The membership views on the server and the clients do not have to be consistent with each other as long as each client can join the group it requires in a timely fashion.

In more details, a client in the distributed parallel system changes its group by sending a membership change *request* to a manager, and at the same time *initiate* the membership change by using its latest group membership data (assumed to be present). The server executes the membership change request upon reception and sends an *accept* to the client (regardless). In parallell, the client *notifies* the server when it has completed the membership change.

This system only requires one connect step to enter a group if the membership database on the client is up-to-date. Unfortunately, it is expected that the membership database may be out-dated and additional steps may be needed. However, this does depend on two factors: frequency of group changes (dynamics) and the number of dynamically changing groups in the system. If the group dynamics is very low, the distributed parallel approach is expected to perform well. If it is high, the performance may suffer due to potential inconsistencies, and the number of membership database messages propagated to the clients in the network also escalates. Therefore, a distributed parallel membership management does most likely fit small to medium sized distributed interactive applications with fairly static groups.

C) Centralized sequential membership management

The centralized sequential membership management (Figure 5.2 C) is an architecture in which each client sends a request to the server that executes it locally and then pushes the membership updates to the affected clients. This is a typical centralized architecture that may be implemented straight forward using a *release consistency* model. The membership database on the server is the only copy being updated, and these updates are pushed to the clients that accept the latest received update as the current.

In more details, clients in the centralized sequential system send a membership change *request* to a manager, in which the request is entered into a request queue. The request queue may be parallelized to any extent, as long as it follows the rules introduced in the distributed sequential membership management paragraph. The server *accepts* and *initiates* the membership change locally, and notifies the affected clients upon *completion*.

This centralized approach requires two steps accross overlay network links to complete a membership change request, which is the minimum of any centralized system. A typical drawback of centralized systems is the added load on the server that executes every incoming request locally. It is also a single point of failure, but fairly simple fault tolerance mechanisms may be added to increase the resilience, where one approach is to replicate data to other servers. To the best of our knowledge, a centralized architecture is the most common approach to be used by current distributed interactive applications (section 5.2.2).

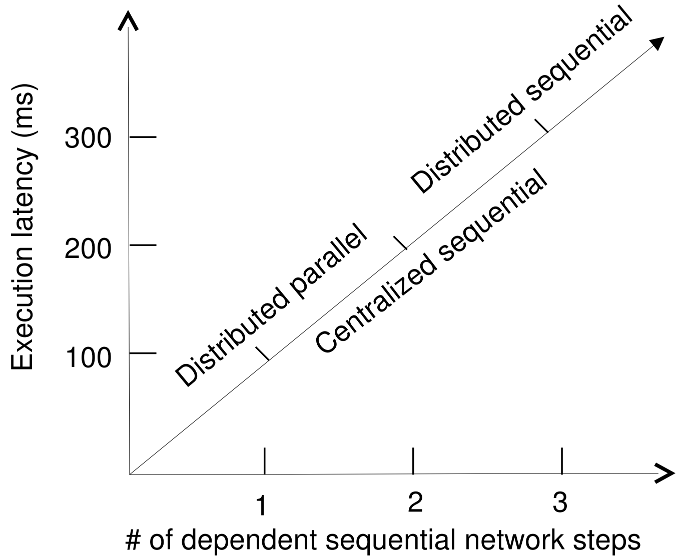


Figure 5.3: The execution latency of a single membership change request for the membership management variations. The execution latencies are based on average PlanetLab link latencies.

5.2.4 Membership management evaluations and experiments

The 3 membership management variations are now evaluated theoretically and through experiments conducted on PlanetLab. The evaluation results in a choice being made upon which membership management variation is better suited for distributed interactive applications.

Membership management goals and metrics

A membership management should handle dynamically changing sub-groups of clients, and enable them to communicate in the sub-groups. Therefore, a membership change request should be executed in a timely manner, while leaving the memberhip database on the manager sufficiently consistent with the sub-views on the clients. The most important important metric is the *membership change execution latency*, which is the time it takes for a membership change request to complete. The membership change execution latency may be influenced by the *membership change frequency*, if the system variation employs a request queue.

Membership change execution latency of a single request

Through experiments on PlanetLab we found an average link latency of approximately 100 milliseconds. We use this average link latency to estimate the average membership change execution latency for each membership management variation. This is possible because each of the 3 evaluated variations has a fixed number of control messages exchanged through overlay links for each membership change request. Distributed sequential has 3 control messages, centralized sequential has 2 control messages, and distributed parallel has only 1 control message. Hence,

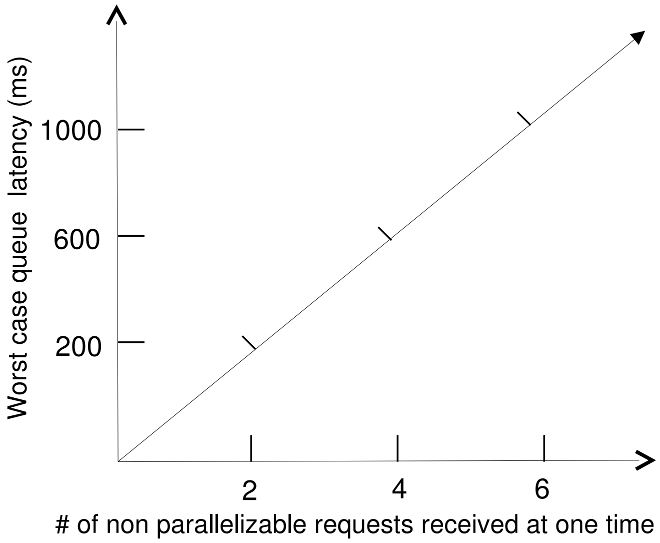


Figure 5.4: Distributed sequential variation: The worst-case queue-latency for one membership change request, when a number of non-parallelizable membership change requests are received at one instance. The execution latencies are based on average PlanetLab link latencies.

on PlanetLab these membership variations finish, on average, one membership change request respectively in 300 milliseconds, 200 milliseconds and 100 milliseconds. Figure 5.3 illustrates the average execution latency for each variation, based on an average link latency on PlanetLab.

Membership change frequency and the execution latency

Studying figure 5.3 it looks like the difference in the membership change execution latency between the 3 variations is fairly marginal. However, one factor that may affect this, is the membership change frequency.

For the latency, the most critical step in the membership management is the accept and initiate steps. If the execution of these two steps are seperated by a large delay, the membership management may suffer in cases of frequent membership changes. Such a large delay is present in the distributed sequential membership management, in which the accept and initiate steps are seperated by two overlay links (average latency of 200 milliseconds). The distributed sequential approach uses a request queue to ensure consistency in the system. When the membership change frequency is high, the request queue is likely to grow. Hence, this puts extra importance on handling the request queue in an extremely parallel manner.

Figure 5.4 shows how the worst-case queue-latency for distributed sequential membership management is affected by how many non-parallelizable requests are received at one time. For example, if 6 non-paralellizable requests are received at one time, the worst-case queue-latency is 1 second for one of the requests. It takes 300 milliseconds to complete a membership change request with no queue-latency. Therefore, we observe that the this variation cannot receive more than one membership change request every 300 milliseconds if it is to avoid queue-latency.

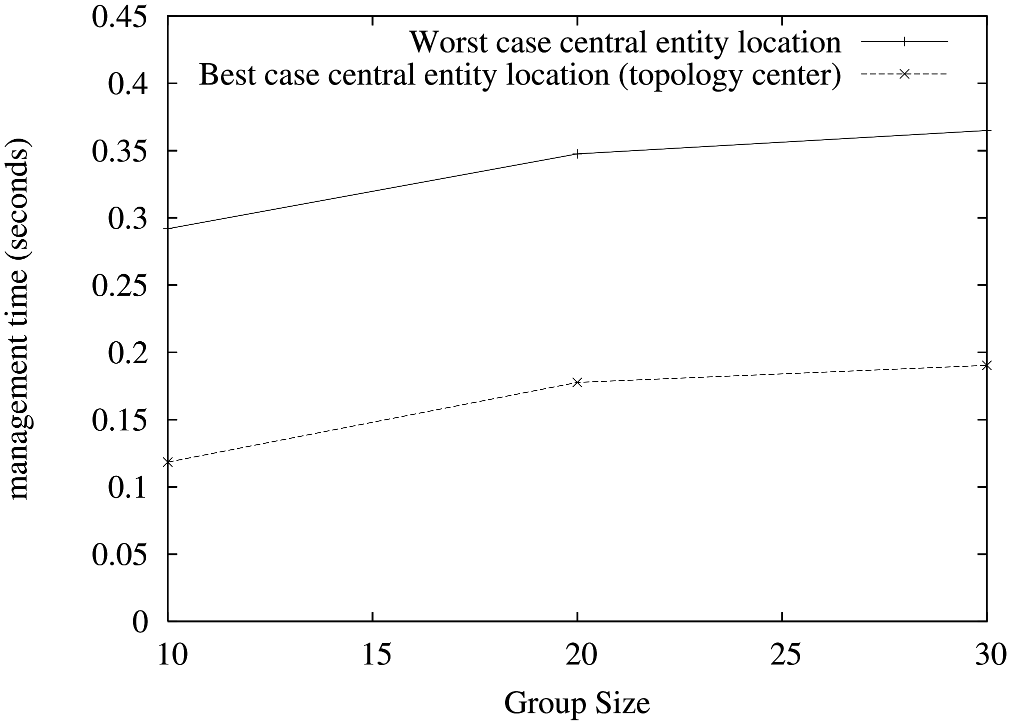


Figure 5.5: Average membership management times for the centralized sequential management. The central entity (manager) was chosen by a core-selection algorithm (topology center) in the resource management, and compared to worst case manager selection.

In the distributed parallel variation, the accept and initiate steps are concurrent, and it is assumed that each client can change membership and connect to a group based on its own membership database. However, the distributed parallel variation cannot guarantee that a client's membership database is up to date. Therefore, it cannot ensure that the membership request is completed in one step. For this reason, the distributed parallel variation is not reliable in situations where the membership change frequency is high.

In the centralized sequential variation, the accept and initiate steps are executed locally on the manager. Therefore, the membership execution time is more dependent on the manager's computational capacity. Based on the previous observations, the choice is made to use the *centralized sequential membership management.*

Centralized membership management experiments

The advantage of a centralized approach is the consistency it gives. However, the drawback may lie in the administration latencies involved in membership and group updates. Especially the membership change execution latency has been identified as an issue. Figure 5.5 shows the average membership change execution latencies for the centralized sequential variation. The experiments included 100 PlanetLab nodes that dynamically changed groups by contacting a membership manager node. The latencies are consistently just below 200 milliseconds, and

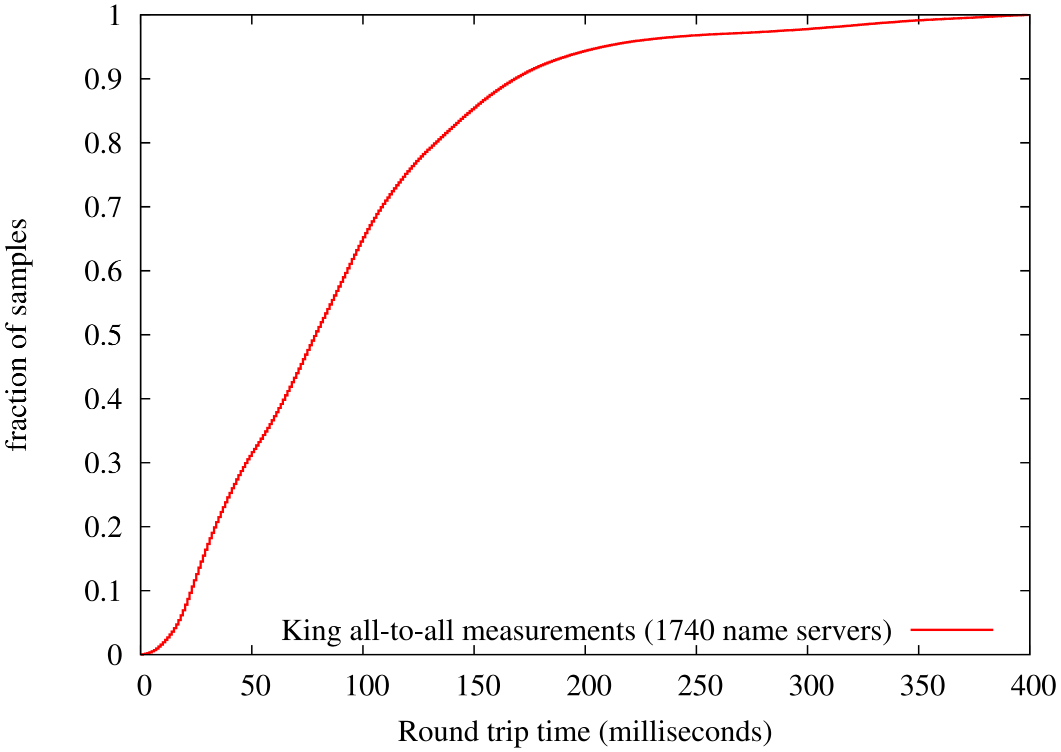


Figure 5.6: Round trip times obtained by King. Measuring all-to-all path latencies between 1740 name servers.

confirm the observations made in the previous sections regarding membership change latency.

In the experiments, the resource management (section 5.3) has used a core-node selection algorithm (chapter 7) and chosen a well-placed central entity (manager) to handle the membership management. It is quite clear that the core-node selection strategy (topology center) ensures that a central entity is chosen that has low pair-wise latencies to the clients in the member network. The worst case central entity placement may be a valid situation in cases where a game provider has a static central entity that serves the entire world. A static central entity cannot be in the current topological center at all times, because as the night and day passes the clients in a game dynamically shift from continent to continent.

King round-trip time measurements

It is clear that the average link latency of 100 milliseconds on PlanetLab may not be the general average in the Internet. Figure 5.6 plots the cumulative distribution function (CDF) of the round-trip times (RTTs) between 1740 Internet name servers. The plot is obtained from the publicly available P2PSim King data set [98], which are all-to-all measurements as obtained by King [61]. From the plot we see that the average RTT is about 100 milliseconds, which is about half of what we found in our PlanetLab measurements. The larger PlanetLab latencies are likely due to heavily loaded PlanetLab nodes, and that PlanetLab measurements are application

Management Technique	*Group dynamics*	*Number of groups*	*Consistency*	Network *steps*	*Drawback*	*Upside*
Distributed parallel	low	low	low or none	1 step	network cost	fast
Centralized sequential	high	high	achieved	2 steps	server load	medium fast
Distributed sequential	low	high	achieved	3 steps	slow	load balancing

Table 5.1: Properties of the membership management systems.

layer latencies. The King measurements, on the other hand, are network layer latencies between name servers that are by en large very well connected.

It is expected that application layer latencies are larger than network layer latencies [24]. Although 50 milliseconds may seem like too big a difference, it does not change our conclusions pertaining the centralized membership management system.

Conclusions and relevance

Empirical results and experiments on PlanetLab show that the centralized sequential variation is a very good approach for membership management of multiple dynamically changing subgroups of clients. An average membership execution latency of just below 200 milliseconds on PlanetLab is acceptable. However, we also observed that a bad choice of manager increases the membership execution latency significantly. Therefore, it is clear that a resource management (section 5.3) that finds well-placed managers must be present for membership management systems that rely on manager nodes. Table 5.1 summarizes the observations regarding the membership management variations.

5.3 Resource management

In section 2.5.3, we described the goal for the resource management to be:

2) Identify techniques that enable a resource management to identify nodes in the (application) network that yield low pair-wise latencies to groups of clients.

Such nodes are often referred to as core-nodes, where, essentially, a core-node may be any node in the application network (server, proxy, member-node, etc). Once the resource management has identified well-placed core-nodes, it is possible to appoint and execute management tasks on them. Therefore, one application area for well-placed core-nodes is to use them to execute parts of a distributed interactive system and as such act as a *manager* for groups of clients.

Low-latency paths to a core-node is the paramount requirement if the core-node is used for management tasks that are time-dependent. A secondary goal is to identify core-nodes that have high computational power and bandwidth capacity, in addition to low pair-wise latencies. Bandwidth capacity is especially important if a distributed interactive application supports bandwidth intensive streams, because the core-nodes may be used to forward such data-streams. However, in the thesis, the focus is on achieving the primary goal: low pair-wise latencies to a selected group of clients. Therefore, the resource management consists of:

Core-node selection techniques *that use all-to-all path latencies to search for core-nodes in the network that yield low pair-wise latencies to selected groups of clients.*

It is the network information management (section 5.5) that provides the all-to-all path latencies. The evaluated core-node selection techniques in the thesis, are 5 core-node selection algorithms, which are are thoroughly introduced in chapter 7. These algorithms address the graph theoretical problems introduced in section 4.4.

5.4 Overlay network management

In section 2.5.4, we described the goal for the overlay network management to be:

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

In the thesis, we use application layer overlay multicast to distribute events among interacting clients. Therefore, overlay network construction techniques are needed to configure efficient multicast overlays. The overlay network construction is assisted by many different types of graph algorithms, many of which are presented in the state-of-the-art and related work (chapter 3). The proposed overlay network management consists of these techniques:

Graph manipulation techniques *that reduce and enhance complete group graphs such that the overlay construction techniques build desirable low-latency overlays.*

Overlay construction techniques *that use the group graphs to construct low-latency overlay networks for event distribution.*

5.4.1 Overlay management and the techniques

The graph manipulation techniques consist of graph algorithms whose primary task is to manipulate a group's complete graph such that it enables the overlay construction algorithm to execute fast and build desirable overlay networks. Application layer overlay networks are inherently complete graphs (fully meshed), and such complete overlay networks can be manipulated and optimized in terms of identifying well-placed nodes and reducing the number of links that are available to an overlay algorithm. Chapter 8 introduces and evaluates a range of graph manipulation algorithms.

The overlay construction techniques consist of overlay construction algorithms whose main task is to construct overlay networks for distribution of application events. An overlay construction algorithm takes as input a group graph and constructs a subgraph on it that is especially designed for timely event distribution. Overlay networks should meet the event distribution requirements, which is especially related to creating low-latency overlay networks. The overlay construction algorithms that are evaluated are spanning-tree and Steiner-tree algorithms, spanning-subgraph and Steiner-subgraph algorithms, and dynamic-tree and -subgraph algorithms. These algorithms are introduced and evaluated from chapter 9 through 14, and the graph theoretical problems they adress are described in chapter 4.

Both the graph manipulation techniques and the overlay construction techniques use all-to-all path latencies, which are obtained by the network information management (section 5.5).

5.4.2 Centralized overlay management

If a centralized architecture is used, the overlay network management is run on a manager node along with the centralized membership management. Each time the centralized membership management receives a membership change *request* the overlay management is *initiated* to perform the necessary overlay construction work for a group. Therefore, the efficiency of a centralized group management approach heavily relies on the efficiency of the overlay management.

Section 5.2 identified that the centralized sequential membership management is the most suitable approach due to the low average latency in a membership execution and the achieved consistency. During the evaluation, the latency of the overlay network management was neglected, but we shall see that in most cases, centralized graph algorithms are still the preferred choice. The graph algorithms use the all-to-all link latencies from the network information management in the search for low-latency overlay networks.

If a graph algorithm is distributed and requires more than 4 steps on a single node to complete, where each step requires communication across overlay links, it is already too much for distributed interactive applications. In the case of 4 sequential steps, it adds up to at least 400 milliseconds for the overlay construction. With the added 200 milliseconds in the membership management it takes at least 600 milliseconds to complete the membership change request. However, for distributed interactive applications, the latency requirement are often below this (chapter 2). For many online games, the latency requirements are less than 400 milliseconds [70]. Therefore, the only distributed graph algorithms for overlay construction that may be evaluated are algorithms that require less than 2 sequential overlay network steps to complete. However, every known distributed graph algorithm for overlay construction have a worst case message complexity that vastly supercedes this number. Studies have shown that distributed overlay construction algorithms may use up to 30 seconds to build a spanning-tree with 50 nodes [147]. This is not acceptable for the time-dependent event distribution scenarios that are considered. Figure 5.7 illustrates the overlay construction latencies of distributed al-

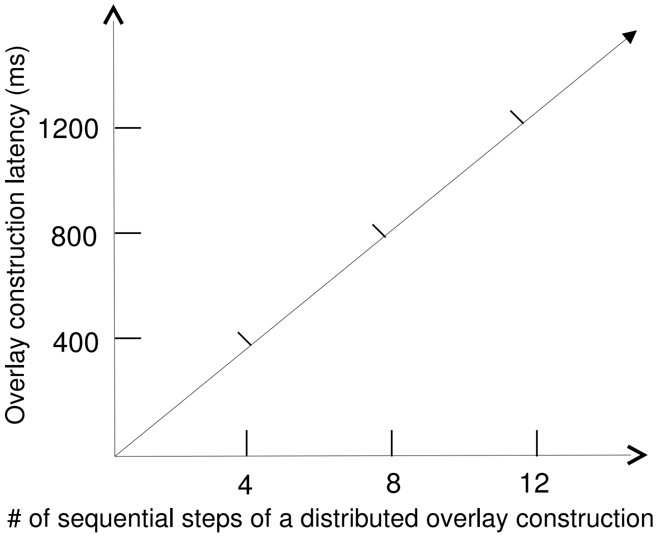


Figure 5.7: Overlay construction latency of distributed overlay construction algorithms in terms of the worst case sequential steps of a distributed algorithm on a single node (each step requires communication across overlay links). The overlay construction latencies are based on average PlanetLab link latencies (100 milliseconds).

gorithms in terms of their number of sequential steps over overlay links on PlanetLab (average 100 millisecond links).

Many graph theoretical problems that address the construction of low-latency networks are also inherently difficult to solve. In chapter 4 we introduced many such problems, and we found that problems addressing pair-wise latencies and have constraints on the node-degree, total-cost, etc, are for the most part NP-complete. Such problems are hard to approximate distributedly, and often rely on sequential algorithms, which suffer from large construction latencies [22]. To the best of our knowledge, there are few if any distributed parallell algorithms for constructing low-latency overlay networks.

On the basis of these observations, we decide to use *centralized graph algorithms* in the remainder of the thesis. Many of centralized graph algorithms we evaluate are approximation algorithms for the graph theoretical overlay construction problems in chapter 4. Generally, all of the algorithms need all-to-all path latencies to be available, and the network information management provides these (section 5.5).

5.4.3 Types of evaluated overlay construction algorithms

The overlay construction algorithms that are chosen for the investigation have desirable properties for the target applications. The investigation include algorithms that are pseudo-random, greedy and based on dynamic programming (see section 4.2.3 for a brief introduction). Pseudo-random algorithms are very fast, but the quality of the solution often suffers. Greedy algorithms are also fast, but many NP-complete problems are still too hard to approximate well using greedy choices. The algorithms that use dynamic programming are slower, but the solutions are often considerably better than pseudo-random and greedy approaches.

5.4.4 Algorithm constraints and dynamic relaxation

Section 2.5.4 introduced the target metrics that overlay construction algorithms should address. Generally, overlay construction algorithms that take several target metrics into account often do this by choosing one metric as its optimization goal, and then address the remaining metrics by adding constraints. Many constrained overlay construction heuristics cannot guarante that a constrained overlay is found. Tree heuristics with degree limits as only constraint, always find a tree when given a fully meshed graph and a degree limit > 1. A tree has a minimum of two leaf nodes at all times, therefore, in a fully meshed graph, the leaf node always has an available degree and a link to all the other member-nodes. Heuristics with latency bounds, on the other hand, do face situations in which it is impossible to find a tree within the bound. When a heuristic fails to find a constrained overlay, the options are to:

1) rebuild the overlay from scratch with relaxed constraints.
2) abandon the constraints and add the remaining member-nodes through some shortest paths.
3) relax the constraints dynamically while building the overlay.

In our application scenario it is not an option to rebuild the overlay from scratch, as it may potentially take a very long time. Furthermore, we do not want to completely abandon the constraints, because they are among our target metrics. Rather, we relax the constraints dynamically whenever a heuristic cannot continue the overlay construction process. Section 10.3.2 introduces these constraints issues applied to Steiner-tree heuristics.

Algorithm 5 OVERLAY-CONSTRUCTION(G):

```
In: A graph $G = (V, E, c)$.
Out: An overlay $G' = (V, E')$.
VertexSet constraintIsViolated
Start:
for each $v \in V \notin G'$ do
    Try to add edge $(u, v)$ to $G'$, where $u \in V$, such that a constraint cost $c(G, u, v)$ is not violated
    Add vertices $u$ or $v$ for which the constraint cost is violated to constraintIsViolated
end for
if $V(G') \neq V(G)$ then
    Relax constraints of vertices in constraintIsViolated
    Goto Start:
end if
```

Algorithm 5 describes a general overlay construction algorithm, with dynamic relaxation of some generic constraints. If the overlay construction algorithm cannot add a vertex to the overlay, due to the constraints are violated, the algorithm relaxes the constraints for the selected vertices and tries to continue the construction. The algorithm ensures that the overlay construction finishes with a connected overlay, and in the process gracefully relaxes the constraints. The constraints may be degree-limits, latency bounds, total cost bounds, etc.

5.5 Network information management

In section 2.5.5, we described the goal for the network information management to be:

4) Identify techniques that are able to obtain accurate all-to-all Internet path latencies.

In general, a network information management should retrieve any network-related metric that is useful for centralized algorithms. For a resource management that identifies nodes, this may be metrics describing a node's bandwidth, computational powers, etc. In an overlay network management that constructs overlay networks for event-distribution, it may be similar, but also other metrics like packet-loss, throughput, etc. The network information techniques that are addressed in the thesis are:

Latency estimation techniques *that identify all-to-all overlay link latencies to construct a complete graph of the application's members.*

This complete graph of the member network have nodes (vertices) that are the clients, and links (edges) that are overlay connections with the estimated or measured latency. These latencies assist the core-node selection techniques (section 5.3) to find well-placed nodes. Moreover, they assist the graph manipulation techniques and overlay construction techniques to construct low-latency overlay networks for event distribution.

Chapter 6 presents and evaluates 2 latency estimation techniques through experiments on PlanetLab.

5.6 Summary of the main points

We proposed a group management approach that consists of 4 main parts: membership management, resource management, overlay network management and network information management.

We evaluated 3 *membership managment* variations towards expected latency in a membership change request, and the expected consistency of the variation. We deduced that a centralized membership management approach is most fitting to a scenario in which there are multiple dynamic subgroups of clients. The distributed variations (parallel and sequential) may be applicable, but they have drawbacks related to consistency (parallel) and latency (sequential) (see table 5.1).

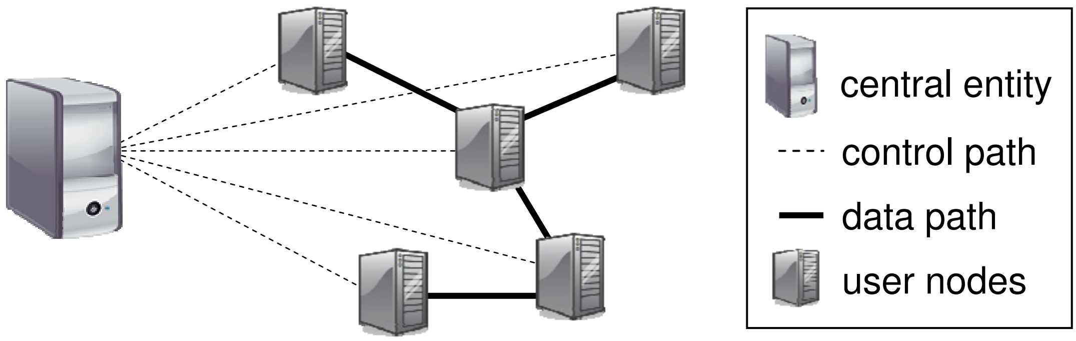


Figure 5.8: Central entity executes the group management teqhniques (resource management, membership management and overlay management).

The *resource management* consists of core-node selection techniques for identifying well-placed core-nodes that yield low pair-wise latencies to the clients in the application. Such well-placed core-nodes are typically suitable for time-dependent management tasks. The core-node selection techniques are evaluated chapter 7.

The *overlay network management* consists of graph manipulation teqhniques that operate on complete group graphs to reduce the overlay construction time and generally assist the overlay construction techniques (chapter 8). The group graphs are used by overlay construction teqhniques to construct overlay networks for event distribution (chapter 9 through 14). We found that centralized overlay construction algorithms are most likely better suited for distributed interactive applications than distributed algorithms. This is mainly due to the large overlay construction time of distributed algorithms.

The *network information management* has latency estimation techniques for identifying all-to-all link latencies in the application's member network (chapter 6). These all-to-all latencies assist the centralized graph algorithms in the resource management and overlay network management.

From these observations, we choose a centralized approach to the group management. A central entity executes the group management, and the control-messages are sent to the central entity. The data-paths, on the other hand, are overlay networks in which the application data is multicast among groups of clients (figure 5.8).

Chapter 6

Characteristics of overlay networks: Latency estimation techniques

The investigation in the thesis includes many centralized graph algorithms that operate on complete application layer overlay networks. For example, the resource management and overlay management introduced in chapter 5, apply centralized algorithms for finding core-nodes (chapter 7) and constructing low-latency overlay networks (chapter 9 through chapter 14). Such centralized graph algorithms require network characteristics to be available at the executing node, and the focus in this thesis is on link latency as the most important network characteristic.

Link latencies may be obtained through active probing and monitoring of the entire application network, however, this is not scalable due to a quadratic traffic growth. Instead, latency estimation techniques that reduce the probing overhead are applicable. Such techniques do not perform all-to-all measurements, but rather they do measurements on a sub-set of the links, and then estimate the remaining links based on the measurements. Though such techniques are scalable, the penalty lies in their latency estimation accuracy, which is likely to suffer because less measurements are performed. This chapter evaluates two latency estimation techniques, Vivaldi [34] and Netvigator [113], towards their accuracy and applicability to distributed interactive applications. The results showed that Netvigator yields accurate latency estimtates, while Vivaldi is more inaccurate but still usable. Netvigator is harder to setup than Vivaldi, but they are both likely candidates for use in distributed interactive applications.

The rest of the chapter is organized in the following manner. Section 6.1 introduces several latency estimation techniques along with a few requirements of distributed interactive applications. Section 6.2 describes the latecy estimation techniques that are evaluated. These are latency estimation techniques that, in our scenario, are applicable to estimate the link latency between clients currently participating in a distributed interactive application. Section 6.3 presents several real-world experiments that were performed on PlanetLab using the latency measurement tool ping, and the latency estimation techniques Vivaldi and Netvigator.

6.1 Obtaining overlay network latencies

Chapter 2 discussed the strict latency requirements that distributed interactive applications have. It was identified that centralized graph algorithms are desirable and necessary to meet these latency requirements. Centralized graph algorithms that require link latencies to work, motivates the need for accurate latency estimation techniques and measurement tools. From the observations a goal was formulated:

4) Identify techniques that are able to obtain accurate all-to-all Internet path latencies.

The following sections have brief introductions to latency estimation techniques and measurement tools, and discusses them in terms of ther applicablity to estimate all-to-all path latencies.

6.1.1 Latency measurement tools

It is straight forward to obtain a link's round-trip-time (RTT) in the Internet using measurement tools like ping and traceroute.

Ping works by sending ICMP "echo request" packets to the target host and listening for ICMP "echo response" replies. Ping estimates the round-trip time (milliseconds), records any packet loss, and prints a statistical summary when it is done.

Traceroute works by increasing the time-to-live (TTL) value of each successive batch of packets sent. The first packets sent have a TTL value of one (implying that they are not forwarded by the next router and make only a single hop), the second TTL value of two, the third three, and so on. When a packet passes through a router, the router decrements the TTL value, and forwards the packet to the next router. When a packet with a TTL of one reaches a router, the router discards the packet and sends an ICMP time exceeded packet to the sender. Traceroute uses these returning packets to produce a list of traversed routers in the route to the destination. The timestamp values returned for each router along the path are the latency values (milliseconds) for each packet in the batch.

The drawback with Ping, is that it does not return any measurements if the target host is unreachable. Traceroute, on the other hand, returns latency measurements for each hop in the route, as far as it gets. This may be valuable for some latency estimation techniques if they control network routers, and if the end-to-end reachability is limited.

6.1.2 Latency estimation techniques

Application layer overlay networks are complete graphs (full meshes) for which the number of links increases exponentially when nodes are added. Therefore, all-to-all measurements in overlay networks that may hold hundreds or thousands of clients participating in a distributed interactive application, incur a massive overhead [9]. Achieving full, up-to-date knowledge of the network requires monitoring and is not scalable for a large number of nodes because the monitoring traffic grows quadratically with the number of nodes. This scalability problem is addressed by techniques that estimate link latencies, but the trade-off is their accuracy. In general, a latency estimation technique probes a number of links using a measurement tool (or real traffic) to retrieve their link latencies, and then attempts to estimate the remaining links based on these probes. For example, by using network coordinate systems to place the network nodes relative to each other.

There are a multitude of latency estimation techniques. King [61] is a tool that enables a node A to retrieve a latency estimate between two arbitrary hosts in the Internet, for example, B and C. King works by estimating the latency between B's authoritative name server and C's authoritative name server. The assumption is that both B and C are close to their respective authoritative name server. Recently, an improved King was proposed that reduced the error that this assumption may incur [84]. Some latency estimation techniques rely on landmark nodes. Such techniques include Netvigator, NetForecast, Global Network Positioning (GNP) and Practical Internet Coordinates [41]. Other latency estimtion techniques do not need any infrastructure and works in any distributed setting, for example, peer-to-peer. Such techniques include Vivaldi [34] and Big Bang Simulation technique [41]. Many of the latency estimation techniques are likely to be usable in a distributed interactive application setting. However, the main comparative metric is whether or not the estimations are accurate enough.

We find that in the scenarios of large-scale distributed interactive applications, it is currently not scalable to use measurement tools like ping or traceroute to actively monitor entire application layer overlay networks for their link latencies. Instead, latency estimation techniques that reduce the probing overhead should be applied. Such estimation techniques may be classified into three classes [42]:

- *Landmarks-based latency estimation* techniques assigns each node a point in a metric space, and aim to predict the latency between any two nodes. They use landmark nodes, which are a set of nodes used by the remaining nodes as measurement references for their relative position in the network. Netvigator [113] is such a technique.

- *Multidimensional-scaling based latency estimation* techniques do not involve landmark nodes. They use statistical techniques for exploring similarities and dissimilarities in data. For example, a matrix of item-item similarities is used to assign a location for each item in a low-dimensional space [31]. Vivaldi [34] is an example of such a technique.

- *Distributed network latency database* techniques use active measurements to build a knowledge base about the underlying network. These approaches have been designed to efficiently answer queries of the form: Who is the closest neighbor to node A in the network? Since these schemes are based on direct measurements they have better accuracy,

Technique	*Measurement Overhead*	*Requires*	*Churn recovery*	Infrastructure dependability
Vivaldi	$O(g*N+N)$	Inter-nodes traffic	yes	no
Netvigator	$O(L*N+N)$	Traceroute	no	yes

Table 6.1: Properties of the latency estimation techniques.

however they do also inject more traffic into the network compared to the landmarks-based and multidimensional-scaling based techniques. Meridian [143] is a technique that uses a distributed network latency database.

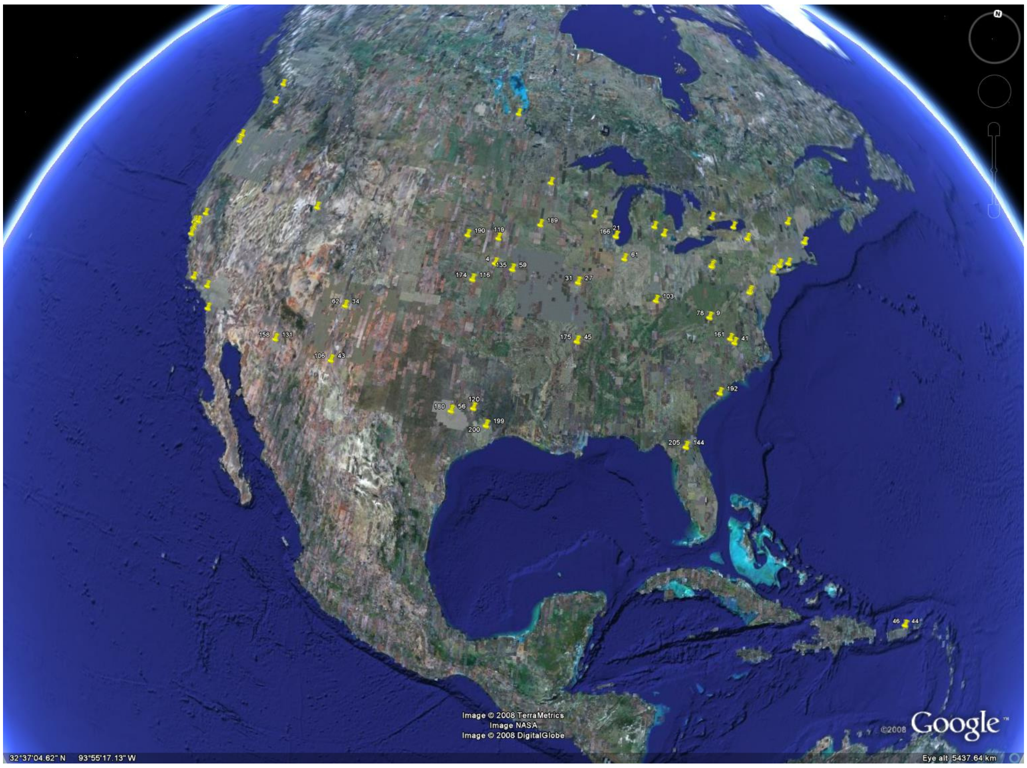
From these observations we deduce that distributed network latency database techniques are not desirable for our target area, because they are not designed to retrieve all-to-all link latencies. They are, however, desirable for leader election scenarios, and discovering the closest neighbor for a node. Instead, we focus on landmarks-based and multidimensional-scaling based latency estimation techniques, and evaluate Netvigator and Vivaldi as valid representatives.

6.2 Evaluated latency estimation techniques

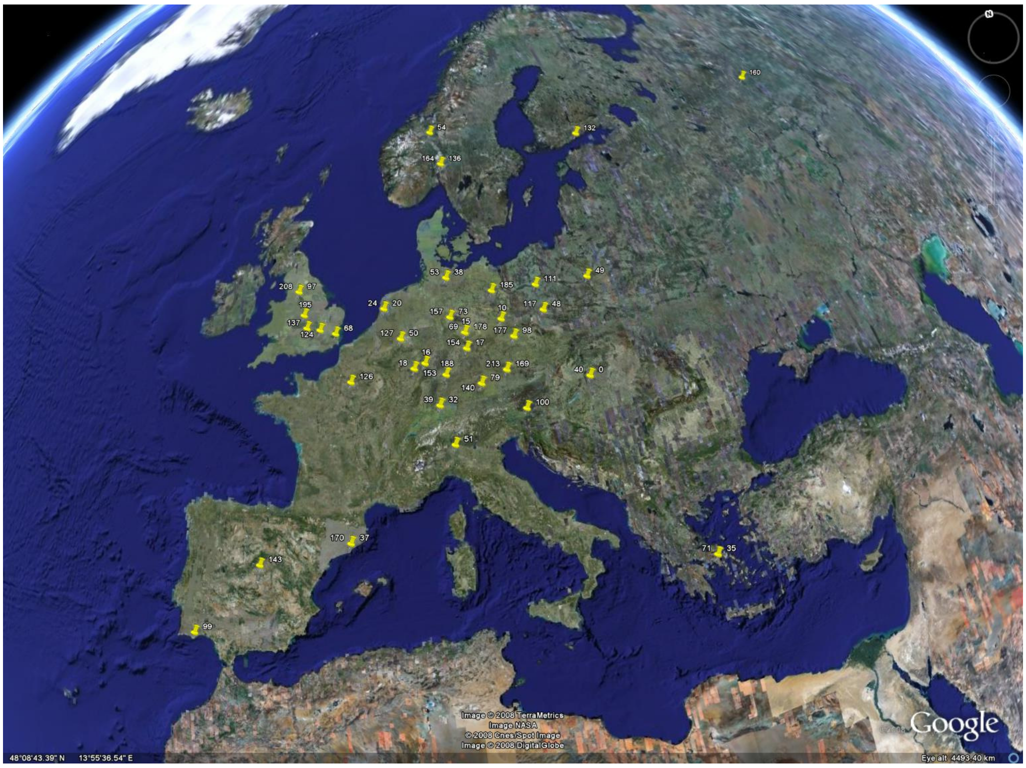
Netvigator [113] is a landmarks based estimation technique, where a set of landmark nodes L are probed asynchronously by N nodes using Traceroute (L*N probes). Each node reports its measurements to a repository (typically a server node), which estimates a global graph with latencies. Netvigator was originally designed for proximity estimation, that is, to rank nodes according to proximity to any given node.

Vivaldi [34] is a multidimensional scaling technique and is based on spring embedding, which models network nodes as masses connected by springs (links) and then relaxes the spring length (energy) in an iterative manner to reach the minimum energy state for the system. All nodes N joining the system are placed at the origin, and starts sharing Vivaldi information among a selected group g of nodes, for example, piggybacked on application level data. The Vivaldi information includes its coordinates, confidence estimations and the measured latency. If a global graph is desired, each node can report its Vivaldi information to a repository that does some calculations and inserts the node in a tow-dimensional plane where the euclidian distance equals the estimated latencies.

Several others latency estimation techniques have been proposed and evaluated [42]. However, Netvigator and Vivaldi are two highly valued techniques in their respective latency estimation technique classes [42]. Vivaldi has the advantage that it recovers from node churn (nodes joining and leaving), and does not depend on any infrastructure. Netvigator, on the other hand, needs landmark nodes and does not (easily) recover from churn. Table 6.1 summarizes the comparisons.



(a) Nodes in North-America.



(b) Nodes in Europe.

Figure 6.1: A few selected PlanetLab nodes and their location.

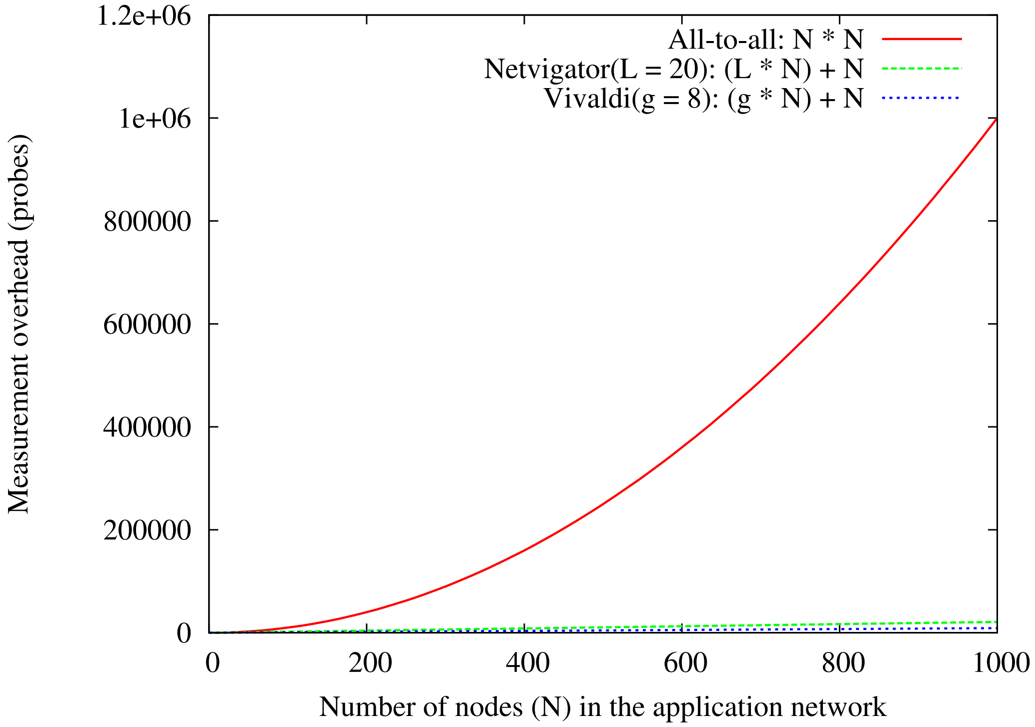


Figure 6.2: Measurement overhead in terms of the number of probes of Netvigator (L=20) and Vivaldi (g=8) compared to all-to-all measurements.

Figure 6.2 visualizes the measurement overhead that Netvigator and Vivaldi incur on the network, compared to all-to-all measurements. We observe that, when the number of application members reaches 1000, the overhead of the all-to-all measurements is 1 million probes. It is quite clear that performing all-to-all measurements in large-scale applications is not a scalable approach.

6.3 PlanetLab experiments

PlanetLab was used to measure the accuracy of the latency estimation techniques Netvigator and Vivaldi. Their accuracy is determined by comparing the estimates to real all-to-all ping measurements. In the following, the latency estimation metrics, experiment configurations and results are presented.

6.3.1 Experiment configurations

To study the performance of Vivaldi and Netvigator, we performed latency estimation experiments repeatedly over a period of 10 days and included 215 PlanetLab nodes (the total number of nodes we were able to access). Figure 6.1 shows a few selected nodes and their locations.

Descriptions	*Configurations*
Group sizes	g = 4, 8, 12 clients
RTT measures	*tcpinfo*, *ping*
Packet rates	high (100 packets/sec.), low (2 packets/sec.)
Log times	t = 4, 8, 12, 16, 20 minutes

Table 6.2: Vivaldi experiment configurations.

The plots, in the following, represent the results from applying Netvigator and Vivaldi using the same nodes over the same time interval. The RTTs that are used by Netvigator and Vivaldi are obtained both from ping and tcpinfo. The reference for our latency estimations was established by measuring latencies between all pairs of nodes using ping once per minute.

For the Netvigator experiments, we used publicly available estimates performed on PlanetLab. Netvigator is currently a running PlanetLab service that estimates the link latencies between nearly every PlanetLab node. The results of the measurements are available from the S^3 website[1], which is updated every four hours with the most recent estimations. We used these measurements in our experiments. However, the Netvigator configuration is currently a black box for us.

For the Vivaldi experiments, we used a wide variety of configurations, investigating all combinations of the parameters in table 6.2. To clarify what is actually happening in the Vivaldi experiments we give an example. A PlanetLab node is assigned to a random group of nodes of size g. Then the PlanetLab node starts performing *RTT* measurements to the $g - 1$ neighbor nodes, while sharing its own Vivaldi coordinates. Upon reception of a neighboring node's Vivaldi coordinates and the links RTT measurement, it updates its own Vivaldi coordinates. Then, in certain timed intervals t the PlanetLab nodes *log* their current (Vivaldi) network coordinates to a central entity that estimates a complete graph with link latencies.

We varied the Vivaldi configurations, and used *group sizes* up to 12 nodes (more neighbors makes more measurements and better estimations [34]). Groups are created to limit the amount of measurements done in the network. The RTT measurements were obtained in two different manners. Once from the *tcpinfo* structure, which is updated for each open TCP connection and that can be used easily in passive measurements. Then by active measurements using *ping*. The *packet rate* was varied because a higher rate follows the actual latency development more closely, while it is also consuming more bandwidth itself, at least in the active measurements. The *log times* (t) parameter determined for how long the Vivaldi information was collected until its estimations were used for identification decisions.

[1] http://networking.hpl.hp.com/cgi-bin/scubePL.cgi

6.3.2 Latency estimation metrics

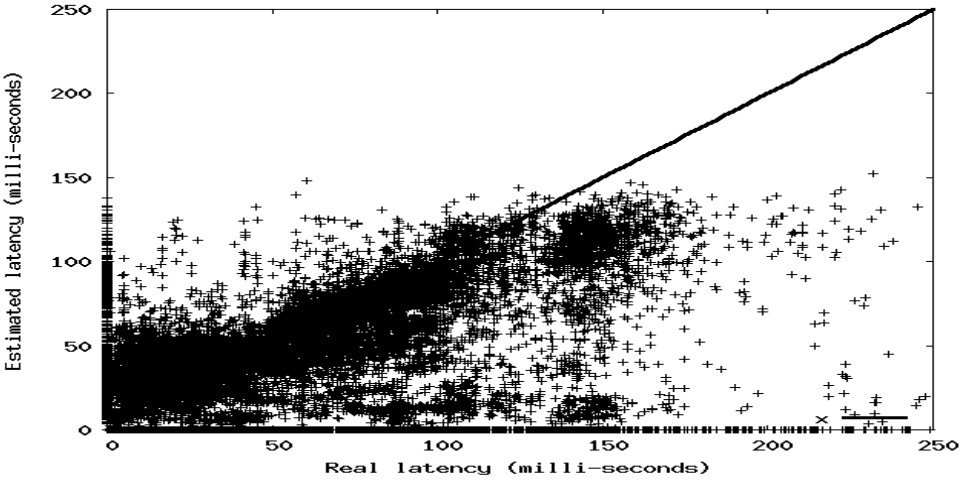
The evaluation includes a number of metrics to measure the performance of Vivaldi and Netvigator [42]:

- The *relative error* measures the difference between the real network latency from all-to-all ping measurements, and the estimated latency, for each pair of PlanetLab nodes.
- The *directed relative error* measures the directed difference between the real network latency from all-to-all ping measurements, and the estimated latency for each pair of PlanetLab nodes.
- *Relative rank loss (RRL)* [89] expresses how well the relative closeness of the PlanetLab nodes is maintained when latency estimates are used instead of all-to-all ping measurements. Each PlanetLab node checks if its relative rank is preserved in the latency estimates by checking every pair of (neighboring) PlanetLab nodes, and then compare the latency estimates and the all-to-all ping measurements. For example, if A is closer to B (than C) in the all-to-all ping measurements, then A should be closer to B in the latency estimates. RRL is computed for every PlanetLab node.
- *Closest neighbor loss significance (CNLS)* [42] is calculated for each PlanetLab node. Conceptually, a PlanetLab node first checks if its closest neighbor in the all-to-all ping measurements is still the closest neighbor in the latency estimates. If yes, then the CNLS is zero. If no, it means a different PlanetLab node is the closest in the latency estimates. Exemplified, the CNLS is computed as the absolute difference between the distance A to the estimated closest neighbor C, divided by the absolute distance between A and the real closest neighbor B (in the ping measurements).

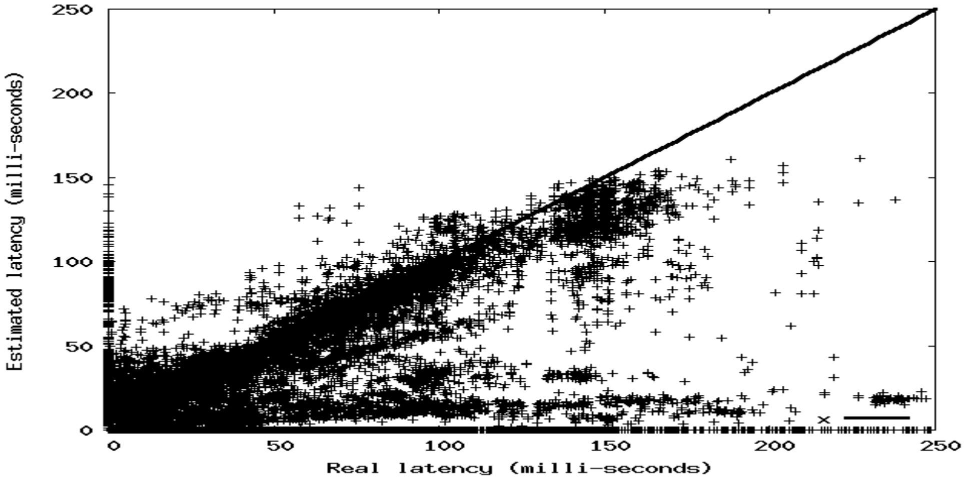
6.3.3 Experiment results

Figure 6.3 visualizes the discrepancy between ping-measured latency and the estimated latencies in greater detail. It is a scatterplot of all pairs of nodes that compares the absolute measured RTT with the absolute estimated RTT [2]. Measurements are sorted by ping-measured RTTs on the X-axis, the latency estimates for the same pairs of nodes are Y-values (the ideal line is $y = x$). Vivaldi results are shown for two configurations to see the effects of the packet rate. The figures show that Netvigator is very accurate in its estimations, closely following the ideal line whereas Vivaldi estimates has more variation. They also show that Vivaldi, and to some degree also Netvigator, overestimates RTTs for the smaller actual RTTs, while it underestimates

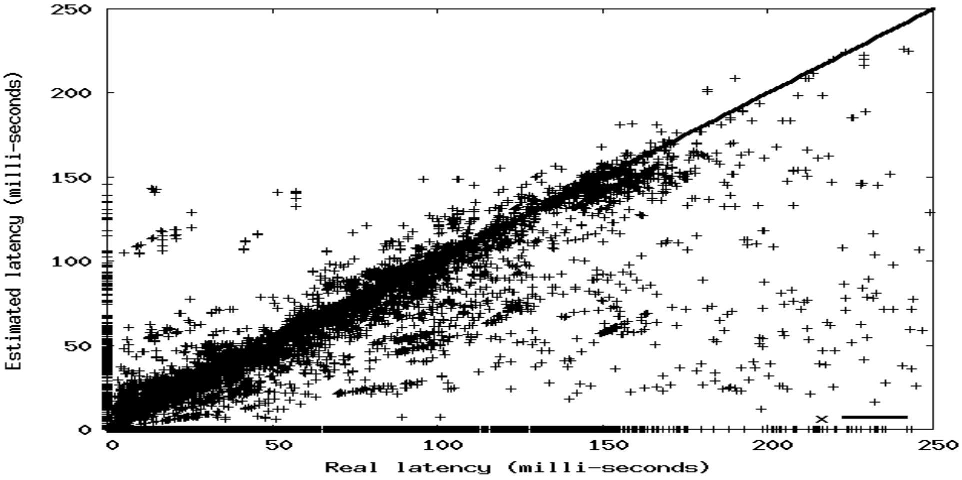
[2] Note that there are several points in the plots that have 0-values. This is due to the fact that not all of the 215 nodes could communicate with each other, and the reason why we use only those 100 nodes in chapter 7 and 15.



(a) Vivaldi, ping, low packet rate

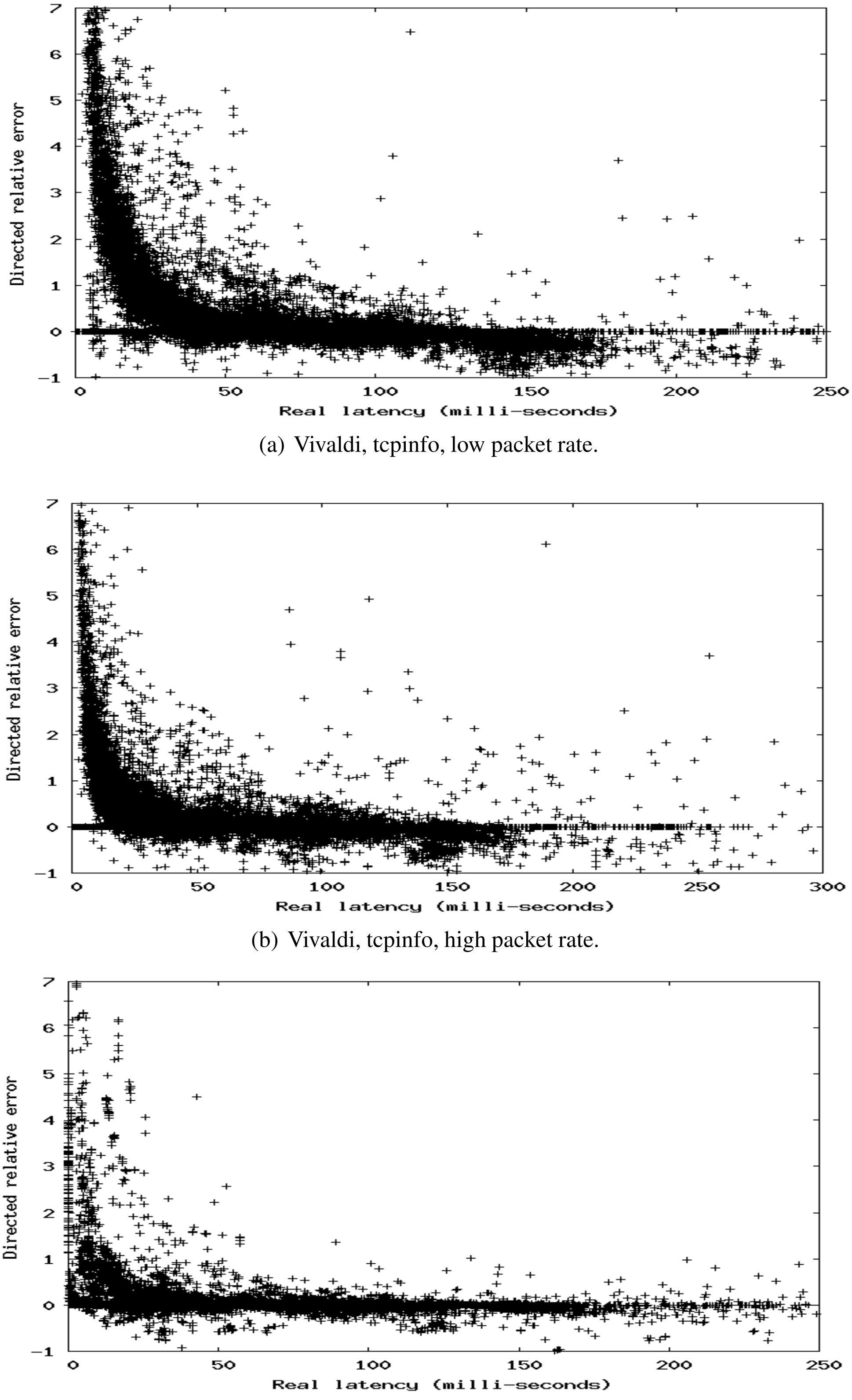


(b) Vivaldi, ping, high packet rate.



(c) Netvigator.

Figure 6.3: Real versus estimated latency.



(a) Vivaldi, tcpinfo, low packet rate.

(b) Vivaldi, tcpinfo, high packet rate.

(c) Netvigator.

Figure 6.4: Directed relative error of latency.

longer distances. When the actual RTTs are very small, the overestimations are relatively high, but the absolute deviation may still be acceptable for many applications.

Figure 6.4 is a scatterplot of the the directed relative error of the ping-measured RTTs compared to Netvigator and Vivaldi latency estimates. In the figures, Vivaldi uses tcpinfo RTTs with high and low packet rates. The figures show that Vivaldi yields more inaccurate latency estimates than Netvigator. We also see that a high packet rate yields better estimations for Vivaldi after 4 minutes. The difference between tcpinfo RTTs and Ping RTTs is not very visible.

Figure 6.5(a) plots the CDF of the relative error of the estimated latencies. We observe that Netvigator is the best, and yields 80 % of the estimations within a 15 % relative error. Vivaldi using RTT from pings and a high packet rate is the best among the Vivaldi configurations. This configuration yields 80 % of the estimations within a 50 % relative error after 4 minutes. RTTs from tcpinfo only performs slightly worse than ping RTTs, and a lower packet rate requires more time to reduce the relative error satisfactory.

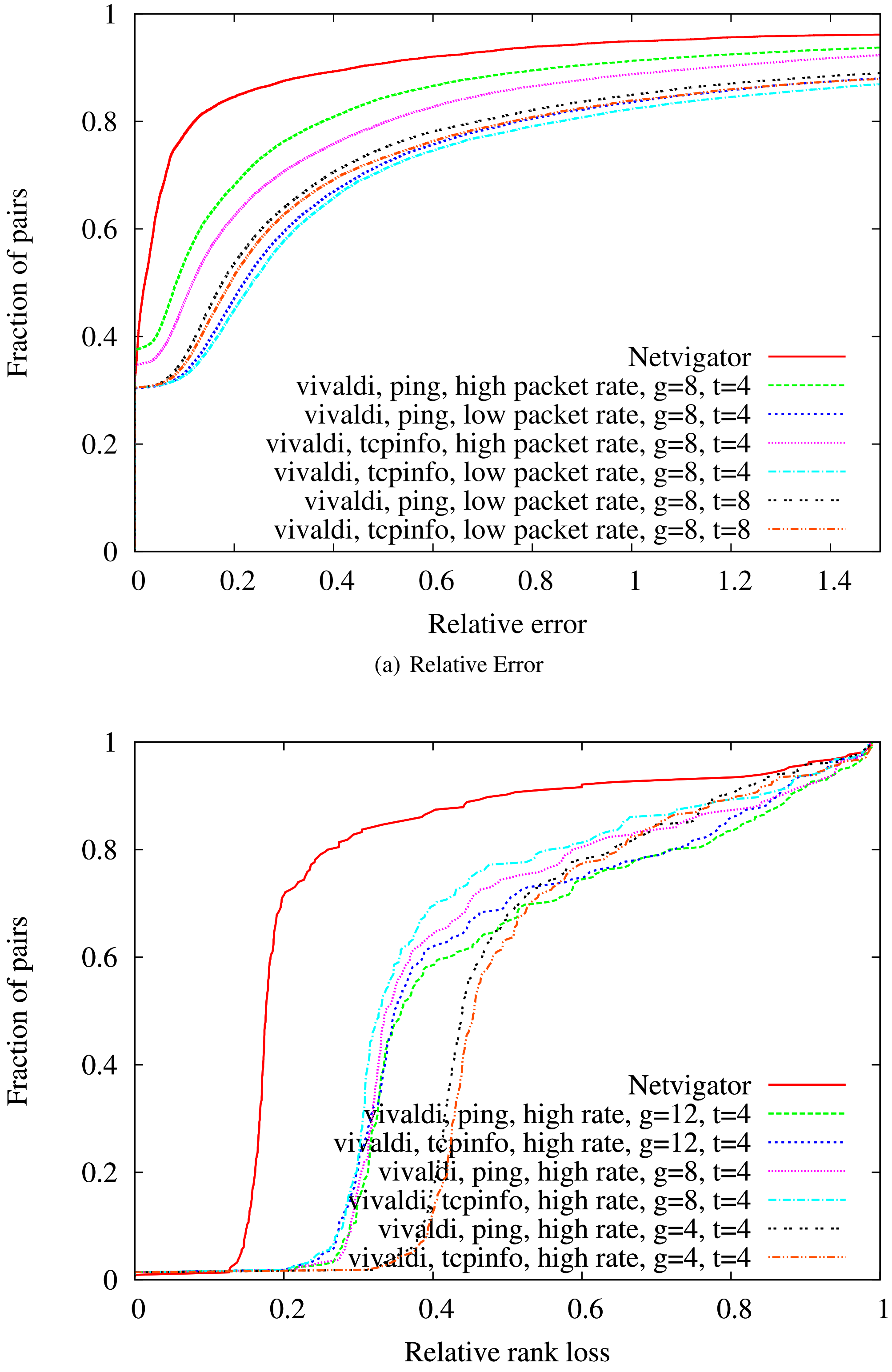
Figure 6.5(b) plots the CDF of the RRL in the estimates. Netvigator has the lowest RRL, where 80 % of the estimates have a relative rank loss of 15 %. The Vivaldi configurations perform very similar with regards to RRL with the exception of group sizes. Intuitively, larger group sizes should produce a lower relative rank loss. However, a group size of 8 actually performs better than a group size of 12. We suspect this is due to the lack of computational power on PlanetLab nodes, because they cannot handle the traffic. However, identifying the main reason remains for future work.

Figure 6.6 plots the CDF of the CNLS. Netvigator outperforms Vivaldi on this metric. The authors of Netvigator listed proximity estimation (i.e., rank nodes according to proximity to any given node) as the prime focus of Netvigator, and the results from CNLS shows that it works very well.

To summarize, Netvigator yields better estimations, but is difficult to set up due to its dependance on landmark nodes. Vivaldi performs worse, but has the advantage of easy deployment and churn recovery. For Vivaldi, the best configuration was a group size of 8 (and above) and high packet rates. A low packet rate reduces the estimation accuracy, and requires 8 minutes to stabilize in contrast to 4 minutes at high packet rates. Finally, RTTs from ping-based estimations performed slightly better than tcpinfo-based ones.

6.4 Summary of the main points

The chapter motivated the need for retrieving all-to-all link latencies in the context of centralized graph algorithms that operate on complete application layer overlay networks. The difference between latency monitoring and estimation is scalability, because achieving full, up-to-date knowledge of the link latencies in an overlay network requires continuous monitoring,



(a) Relative Error

(b) Relative Rank Loss

Figure 6.5: CDFs of Netvigator and Vivaldi performance. Vivaldi plots with different configurations.

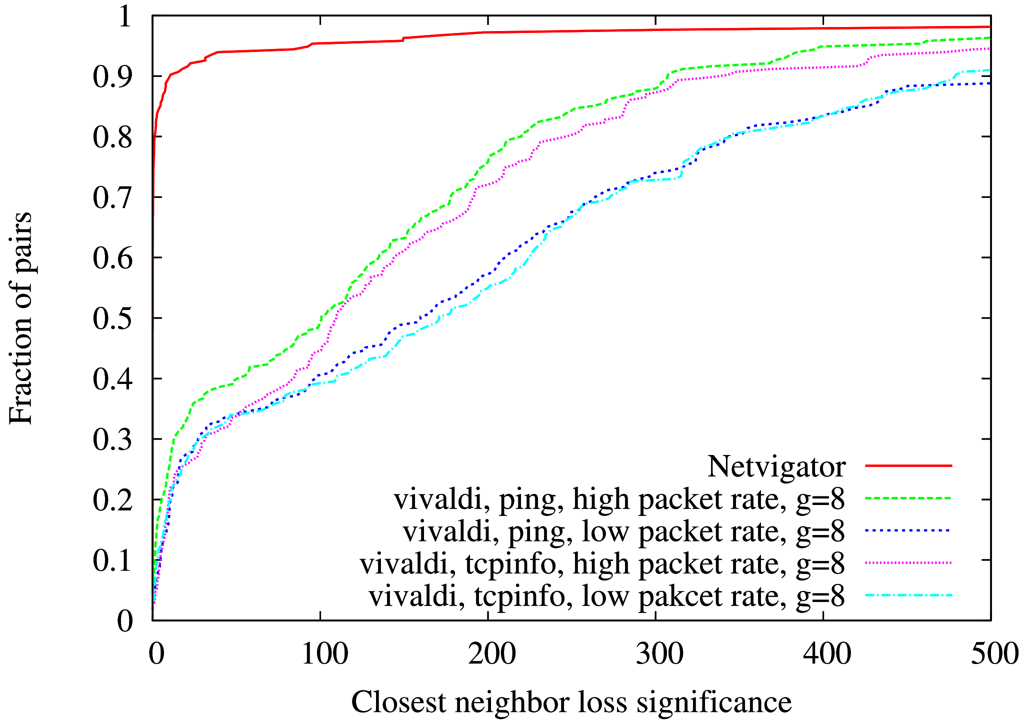


Figure 6.6: CDFs of closest neighbor loss significance (CNLS) for Vivaldi and Netvigator.

for which the number of probes grows quadratically with the number of nodes. This scalability problem is addressed by techniques that estimate link latencies, but the trade-off is their accuracy, which again can dramatically reduce the correctness of centralized graph algorithms that use the estimates. Chapter 7 and 15 evaluates the influence the latency estimates have on centralized graph algorithms.

We performed an experimental analysis of the latency estimation techniques Vivaldi and Netvigator. We evaluated the latency estimation quality and found that Netvigator yields estimations that are very close to optimal. The estimations from Vivaldi were not as good, but we found them to be usable in our application scenario. Vivaldi's advantages are that it is very easy to deploy in a peer-to-peer fashion and it handles membership dynamics. Netvigator, on the other hand, needs an infrastructure (landmark nodes). However, a game provider that controls a number of proxies could use Netvigator as it is the better alternative.

Chapter 7

Managers in overlay networks: Core-node selection algorithms

We have previously identified that managing multiple dynamic subgroups makes the group management hard, especially in time-dependent distributed interactive applications (chapter 2). Therefore, an important problem for distributed interactive applications is to identify nodes in the network that have desirable properties that make them suitable for some managing responsibility. In this chapter, we are considering core-node selection algorithms to find nodes in the network that yield low pair-wise latencies to groups of clients.

In the group management discussions in chapter 5 we identified that a centralized membership management is fitting to distributed interactive applications. A centralized membership management is dependent on appointing management tasks to a limited set of nodes. Hence, one important problem is to identify nodes that are suitable for group management. It is the resource management (section 5.3) that is responsible for identifying core-nodes. The core-node identification techniques discussed in the thesis include a class of algorithms called core-node selection algorithms [75,47]. These algorithms use the all-to-all latency esitmates and measurements from chapter 6 to search for well-placed core nodes that yield low pair-wise latencies to subgroups of clients in the application.

An overall goal is to select core-nodes on the basis of certain node properties, such as, bandwidth and computational power. However, a prime consideration is the minimization of the group management time, which is heavily reliant on the location of the group's *manager* node. Hence, the goal is to minimize the group management time by minimizing the latency to the group's manager. To address this goal, the observations from section 4.4 are used. There, several graph theoretical problems were identified as being relevant for our scenario, namely, searching for well-placed core-nodes.

The following introduces some basic core-node selection algorithm choices and also a range of core-node selection algorithms for single core-node and multiple core-node selections. We evaluate both the multiple and single core-node selection algorithms in terms of their ability to find well-placed core nodes that yield low pair-wise latencies to groups of nodes. The results clearly show that core-node selection algorithms are able to do this.

The chapter is organized in the following manner. Section 7.1 introduces the goal that we address in the chapter. Section 7.2 provides explanations to the workings of a core-node selection algorithm. Section 7.3 introduces the evaluated multiple core-node selection algorithms. Section 7.4 introduces the evaluated single core-node selection algorithms. Section 7.5 evaluates the core-node selection algorithms through simulations and experiments. Section 7.6 applies the core-node selection algorithms to a multiplayer online game scenario. Section 7.7 evaluates the usability of the all-to-all latency estimates from chapter 6 when they are used by a core-node selection algorithm. Finally, section 7.8 gives a brief summary of the main points.

7.1 Core-node selection algorithm goals

There are many different core-node selection algorithms, and they may have very different goals. However, the functionality may be stated such:

A core-node selection algorithm should identify one or more nodes in a network that yield a desired property.

The desired property that the core-node selection algorithms search for in this investigation, are core-nodes that yield low pair-wise latencies to groups of clients. The main reason for this is that we aim to execute the centralized group management from chapter 5 on the core-nodes. It is therefore important that clients in the application have low average pair-wise latencies to the core-nodes. In that respect, section 2.5.3 introduced the resource management and formulated a goal of the thesis:

2) Identify techniques that enable a resource management to identify nodes in the (application) network that yield low pair-wise latencies to groups of clients.

To achieve this goal, we use core-node selection algorithms that consider relational latencies in a network of nodes to establish which core-nodes are desirable. Generally, such core-node selection algorithms may be implemented as centralized or distributed algorithms (see section 4.2 for a brief introduction).

A *centralized* core-node selection algorithm requires that the overlay network link latencies are available at the executing node. The link latencies may be found by the latency estimation techniques from chapter 6, and are then used by the core-node selection algorithm as search metrics to find well-placed core-nodes. A *distributed* core-node selection algorithm solves the core-node selection distributedly in the application's client network. Network characteristics are still required, and the execution time is generally very much higher than centralized algorithms [75].

In our time-dependent application scenario it is desirable that the core-node selection algorithm returns a solution swiftly. Centralized algorithms are therefore the natural choice for the core-node selection algorithms. As mentioned, centralized algorithms require that a node is appointed to execute the algorithm, which may simply be a server given by the service provider. Nodes that appoint a centralized execution may also be identified by a core-node selection algorithm. In many peer-to-peer scenarios it is common to use boot-strapping methods [75] that are spontaniously initiated to find client nodes that may act as, for example, super-nodes because they have high bandwidth capacity. Such boot strapping algorithms are often distributed in peer-to-peer applications. In this chapter, we assume that some initial boot-strapping algorithm has found a server that executes a centralized core-node selection algorithm.

7.2 Core-node selection sets

In the context of this chapter, the main task of a core-node selection algorithm is to identify k core-nodes that are to act as group manager(s) for groups of member-nodes (clients). These core-nodes should be well-placed in the proximity of the member-nodes. To achieve this task, a core-node selection algorithm must consider the core-node set and the member-node set:

- ***Core-node set:*** The set of nodes that are available for core-node selection. They may be found among the application members or application-provided proxies/servers.

- ***Member-node set:*** The set of nodes that require a core-node to manage the dynamic group membership. They are the active application members.

The core-node(s) may be found among the member-nodes (clients) that are currently participating in the application, or among servers and proxies in a bac$k$-bone architecture administrated by the application provider. It is possible to only search for core-nodes among member-nodes in specific groups. However, the group membership may be highly dynamic, and may force frequent core-node switches. Appointing a new core-node is time-consuming and should be avoided if it is not necessary.

A core-node must be able to execute the group management techniques from chapter 5, and manage the group dynamics for a group of member-nodes. During core-node selection the member-nodes are used as the "reference" nodes for the core-node selection algorithm. The core-node selection algorithm chooses a core-node from the core-node set that is most fitting to be a manager for the member-node set, that is, a core-node with low pair-wise latencies to the member-nodes.

Core-node selection algorithms differ by the number of core-nodes they are trying to find. Single core-node selection algorithms are generally easier than multiple core-node selections. A wide range of multiple core-node selection problems were introduced in section 4.4, and identified to be NP-complete. In the following, both the multiple core-node and single core-node graph theoretical problems are addressed and linked to the scenario of distributed interactive applications. In this scenario, the core-node selection is divided into two separate steps:

First, the group managers (core-nodes) are identified by a ***multiple core-node*** *selection algorithm. Second, a group manager is selected for a group of clients by a* ***single core-node*** *selection algorithm.*

7.3 Multiple core-node selection algorithms

The group managers in an application must be identified by some multiple core-node selection algorithm that take into account a manager's placement in relation to the member-nodes. It is possible for the core-node selection to be a continuous process, in which group managers are gradually identified and entered as available group managers (core-nodes). If there are servers or proxies available, it is natural that these are made available as group managers straight away. In a peer-to-peer setting, it is more likely that the group manager set is dynamically changing (due to churn). However, it is certainly possible that an application may have both a static set and a dynamically changing set of group managers. On the basis of these observations, the multiple core-node selection raises a question:

Where should the group managers be placed, such that the network latency to it, is minimized for all members?

It is important that group manager-nodes yield low pair-wise latencies to groups of clients. Further, the group manager-nodes should also be spread out such that they may serve a dynamically changing client base. Chapter 4.4 introduced several problems that address these issues. For example, the minimum k-center and minimum k-median problem (definitions 25 and 26) aim to find k core-nodes, such that the distance from each member-node to an available core-node is minimized. Two related problems were also introduced, namely the k-minimum-eccentricity and the k-minimium-pairwise problem (definitions 27 and 28). They take a slightly different approach to the problem and, loosely translated, aim to minimize the sum of distances from the core-node set to the member-nodes. The effect is that the core-node sets (for these) contain nodes that are more centrally located than in the minimum k-center and minimum k-median problems.

The following sections introduce a number of multiple core-node selection algorithms that aim to find k core-nodes from a larger set of candidate nodes. These multiple core-node selection algorithms are divided into two categories:

- ***Full-network*** core-node selection algorithms use the full network to find the core-nodes.
- ***Split-network*** core-node selection algorithms split the network into more and more pieces and finds a number of core-nodes for each piece.

7.3.1 Full-network core-node selection algorithms

The full-network core-node selection algorithms base the core-node selection on path and node characteristics from the perspective of the entire network. The selected core-nodes typically exhibit certain path-related properties, where typical examples are minimum eccentricity (radius), median vertex, center vertex, etc (section 4.1.1). Algorithms in this category may, for example, address the k-minimum eccentricity and k-minimum pairwise problems.

Following are two algorithms that fit to this category. They are algorithms of the minimum k-minimum pairwise problems and k-minimum eccentricity (definitions 28 and 27).

k-Center core-node selection algorithm

The k-Center core-node selection algorithm finds k core-nodes that are the k nodes with the lowest maximum distance (eccentricity) to a node in the member-node set Z. It solves the k-minimum-eccentricity problem as defined in definition 27. The k-Center algorithm has a time-complexity of $O(n^2)$, when run on a complete graph. Algorithm 6 gives the pseudo-code.

Algorithm 6 k-CENTER(G):

```
In: An integer $k > 0$, a graph $G = (V, E, c)$. Sets $Z \subset V$ and $X \subset V$ .
Out: A set $C \in X$ of core-nodes.
map<id, eccentricity> mapIdEcc
for each $x \in X$ do
    $T$ = ShortestPathTree($x$, $G$)
    v = maxEccentricityNode($T$, $Z$)
    ecc = getEccentricity(v, $T$, $Z$)
    mapIdEcc.insert(x, ecc)
end for
C = kLowestEccentricities(mapIdEcc)
```

k-Median core-node selection algorithm

The k-Median core-node selection algorithm finds k core-nodes that are the k nodes with the lowest average pair-wise distances to the nodes in the member-node set. The algorithm solves the k-minimum-pairwise problem from definition 28 Algorithm 7 gives the pseudo-code for the k-Median algorithm, which has a time-complexity of $O(n^2)$ on any graph.

Algorithm 7 k-MEDIAN(G):

```
In: An integer $k > 0$, a graph $G = (V, E, c)$. Sets $Z \subset V$ and $X \subset V$.
Out: A set $C \in X$ of core-nodes.
map<id, pair-wise> mapIdPairwise
for each $x \in X$ do
    $T$ = ShortestPathTree($x$, $G$)
    v = maxEccentricityNode($T$, $Z$)
    pairwise = getPairwiseDistances(x, $T$)
    mapIdPairwise.insert(x, pairwise)
end for
C = kLowestPairwise(mapIdPairwise)
```

7.3.2 Split-network core-selection algorithms

Split-network core-node selection algorithms split the network into more and more pieces and finds a number of core-nodes for each piece. It is typical that split-network algorithms use a full-network core-node selection algorithm to find core-nodes inside one network piece. Algorithms in this category may, for example, address the minimum k-median and minimum k-center problems.

Following are three algorithms that fit inside this category. They are heuristics of the minimum k-median and minimum k-center problems (definitions 26 and 25).

k-Tailed core-node selection algorithm

The k-Tailed core-node selection algorithm is a heuristic of the minimum k-median and minimum k-center problems. It bases the core-node selection on using node-coordinates in a euclidian space (for example, 2-dimensional (x, y) coordinates). The node-coordinates may be obtained by, for example, using the latency estimation techniques introduced in chapter 6. The algorithm continuously divides the euclidian space into smaller and smaller rectangles and chooses a core-node to be the one closest to the middle of the current most populated rectangle (algorithm 8). The k-Tailed algorithm has a time-complexity of $O(n^2)$, on any graph.

k-Broadcast-Walk core-node selection algorithm

The k-Broadcast-Walk core-node selection algorithm is a heuristic of the minimum k-median problem. It does a semi-random walk through a network of nodes and divides the network into smaller pieces to find appropriate core-nodes. The algorithm iterates k times, and each iteration starts off from an unvisited node s in the network. s sends out a broadcast message that traverses (walks) the network until a number w of unvisited nodes are covered and entered to member-node set W. Then, a single core-node is found using this subnetwork W of nodes as the member-node set (introduced above). The algorithm that finds the single core-node is k-

Algorithm 8 k-TAILED(G):

```
In: An integer $k > 0$, a graph $G = (V, E, c)$, where each $v \in V$ has an $(x, y)$ coordinate. Sets
    $Z \subset V$ and $X \subset V$.
Out: A set $C \in X$ of core-nodes.
enum Direction {X = 0, Y}
struct Rect { (x,y), (x,y), (Z' = ∅) }
multimap<int, Rect> mapRect
mapRect.insert( sizeof(Z), Rect( (maxXpos(Z), 0), (0, maxYpos(Z)), Z) )
Direction Dir(X = 0)
for $i = 0$; $i < k$; $i$++ do
    Rect maxRect = mapRect.end() {maxRect has largest node set size}
    c = findNodeClosestToCenter(maxRect, Dir, C)
    Rect newRect = split(maxRect at $c$)
    mapRect.insert(sizeof(newRect.Z'), newRect)
    Dir = (Dir + 1 mod 2) {Dir is alternating between X- and Y-direction}
end for
```

Median($k = 1$). The nodes in the member-node set W are then marked as visited (algorithm 9). The k-Broadcast-Walk algorithm has a time-complexity of $O(n^3)$, on any graph.

Algorithm 9 k-BROADCAST-WALK(G):

```
In: An integer $k > 0$, a graph $G = (V, E, c)$. Sets $Z \subset V$ and $X \subset V$.
Out: A set $C \in X$ of core-nodes.
w = $|Z|/k$ {Subnetwork size}
M = ∅ { M contains the visited nodes }
for $i = 0$; $i < k$; $i$++ do
    s = findStartNode(Z - M)
    broadcast-network $G$ from s and insert unvisited neighbors to $W$ until $|W| \geq w$
    M = M + $W$
    $k$-Median(G,$k = 1$, $W$, C)
end for
```

k-Unicast-walk core-node selection algorithm

The k-Unicast-Walk core-node selection algorithm is almost identical to the k-broadcast-walk (introduced above). It too does a semi-random walk through a network of nodes and divides the network into smaller pieces to find appropriate core-nodes. The algorithm iterates k times, and each iteration starts off from an unvisited node s in the network. s now sends out a unicast message that traverses (walks) the network until a number w of unvisited nodes W are covered. The unicast message may, for example, be sent to a random neighbor, or the closest neighbor, etc. Then, (like k-broadcast-walk) a single core-node is found using this subnetwork W of nodes as the member-node set (introduced above). The algorithm that finds the single core-node is k-Median($k = 1$). The nodes in the member-node set W are then marked as visited (algorithm 10). The k-unicast-walk algorithm lends itself to a distributed implementation quite easily. The k-Unicast-Walk algorithm has a time-complexity of $O(n^3)$, on any graph.

Algorithm 10 k-UNICAST-WALK(G):

```
In: An integer $k > 0$, a graph $G = (V, E, c)$. Sets $Z \subset V$ and $X \subset V$.
Out: A set $C \in X$ of core-nodes.
w = $|Z|/k$ {Subnetwork size}
M = $\emptyset$ { M contains the visited nodes }
for $i = 0$; $i < k$; $i$++ do
    s = findStartNode(Z - W)
    unicast-network $G$ from s and insert unvisited neighbors to W until |W| $\geq$ w
    M = M + W
    $k$-Median(G,$k = 1$, $W$, C)
end for
```

7.4 Single core-node selection algorithms

The group manager of a single node must be identified by a single core-node selection algorithm, that take into account the pair-wise latencies from the manager to the group of clients. A group's core-node should be dynamically selected based on the current locations of the group nodes. However, some tradeoff between frequency of core-node switching should be included, for example, core-node latency bound.

In the context of the thesis, a group manager is assigned to run the group management system from chapter 5. A core-node may be assigned as a group manager for one or several groups. As mentioned, a group manager may be statically assigned to a group, or it may dynamically change along with the group dynamics. If the group manager is statically assigned, it is very much similar to a centralized architecture in which a server manages the participants. This architecture is straight forward to setup, but does probably increase the group management latencies. In a dynamic approach, a group manager is assigned to a group based on the current distances (latencies) to the group members. This approach reduces the group management latency, but it does require more advanced mechanisms, for example, state migration and consistency mechanisms to avoid inconsistent membership views.

A single core-node selection algorithm searches for a single node in the network that exhibit a desirable property. Following is a brief introduction of the single core-node selection algorithms:

- **Median vertex algorithm:** Returns a core-node that has the lowest average pair-wise distances to the nodes in the member-node set [75]. The algorithm is identical to running k-Median(k=1) (see algorithm 7).

Algorithm	*Optimization*	*Complexity*	*Problem*
k-Center	eccentricity	$O(n^2)$	k-minimum-eccentricity, definition 27
k-Median	average pair-wise latency	$O(n^2)$	k-minimum-pairwise, definition 28
k-Tailed	coordinates	$O(n^2)$	minimum k-center, definition 25
k-Broadcast-Walk	average pair-wise latency	$O(n^3)$	minimum k-median, definitions 26
k-Unicast-Walk	average pair-wise latency	$O(n^3)$	minimum k-median, definitions 26

Table 7.1: Core-node selection algorithms.

- **Center vertex algorithm:** Returns a core-node that has the lowest maximum distance (eccentricity) to a node in the member-node set [75]. The algorithm is identical to running k-Center(k=1) (see algorithm 6).

7.5 Core-node selection simulations and experiments

Core-node selection simulations and experiments were conducted to evaluate the core-node selection algorithms that have been introduced. Table 7.1 summarizes the core-node selection algorithms.

7.5.1 Group communication simulator

We implemented the core-node selection algorithms in a simulator that mimics group communication in a distributed interactive application using a preselected central entity to handle the core-node selection. In the experiments, we assume that some latency estimation technique from chapter 6 identifies a complete graph (full mesh) graph where all edges have an associated latency.

The BRITE [91] topology generator was used to generate Internet-like router networks. We simulated an application layer overlay network, so the network graph was translated into an undirected fully-meshed shortest-path graph, where each router had one client associated to it. Furthermore, the central entity dynamically divides the users into groups, such that each group has a fully meshed group graph. We here present results from simulations using networks with 1000 nodes. The network layout is a square world with sides equal to 100 milli-seconds, which corresponds to a geographical area of approximately Europe.

In the simulations, a multiple core-node selection algorithm identifies a set of suitable core-nodes among the 1000 nodes. All the nodes join and leave groups throughout the simulation, causing group membership to be dynamic. When a join or leave request is received by the central entity, a single core-node selection algorithm then identifies a suitable core-node that handles the group management for the groups. The group popularity is distributed according to a Zipf distribution [20].

For a real-world reference, we also performed similar group communication experiments on 100 PlanetLab nodes. During the experiments, the network was monitored by active all-to-all ping measurements, once each second. The RTT measurements were reported back to the central entity that built a fully meshed application layer graph. The remaining configuration was identical to the group communication simulations.

7.5.2 Target metrics

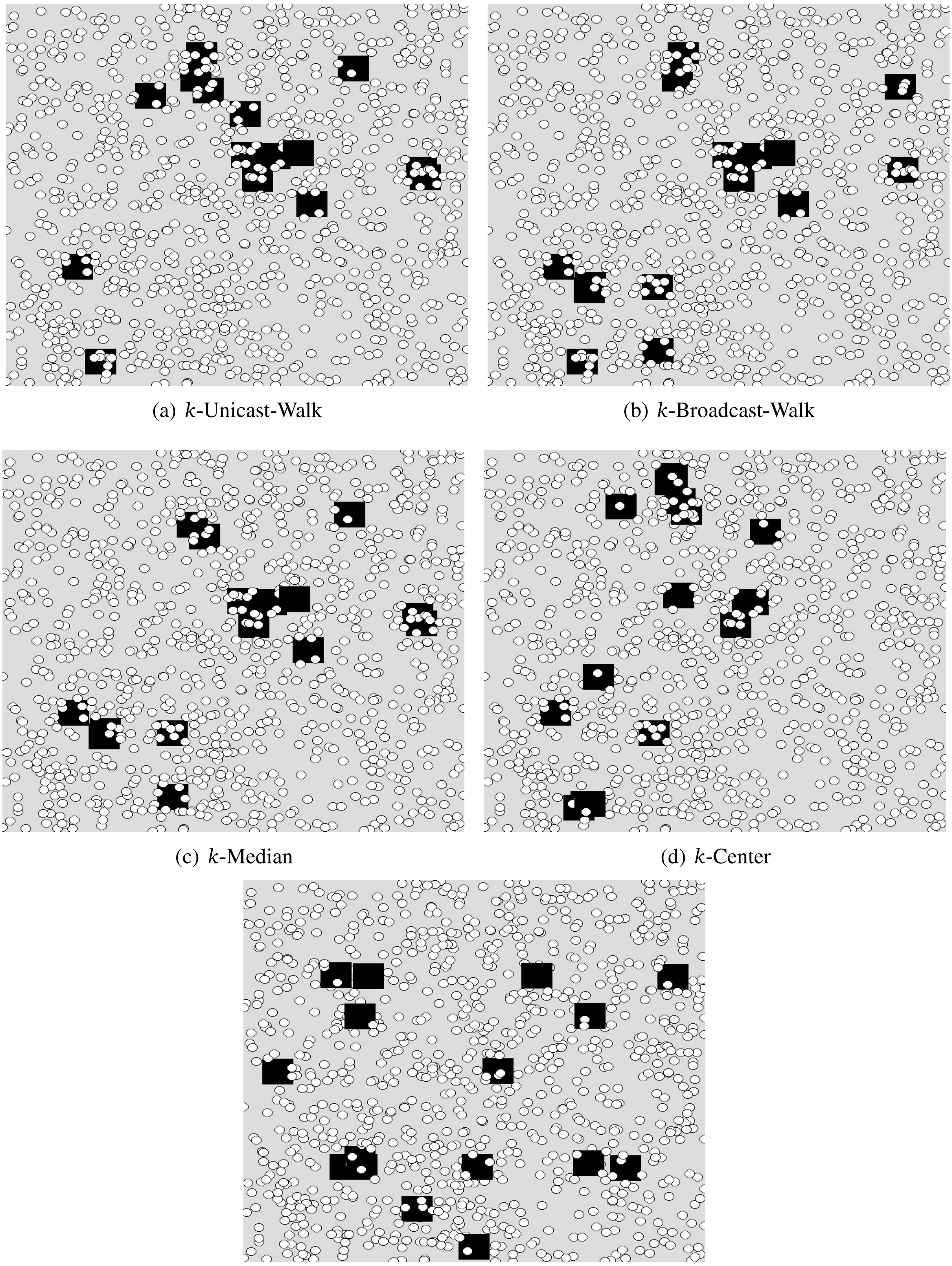
A multiple core-node selection algorithm is considered good if it is able to find a number of well placed core-nodes in the member network. These core-nodes must be placed such that a single core-node selection algorithm is able to find one core-node among them that yield low pair-wise latencies to a group of clients. These groups of clients are dynamically changing, therefore, the core-nodes that are made available by the multiple core-node selection algorithm should be placed out among the member-nodes.

We evaluate multiple core-node selection algorithm in terms of their ability to find core-nodes that are spread out such that the average *maximum one-way latency* from a client to a core-node is low. Similarly, a single-core is also evaluated in terms of their ability to find one core-node that minimizes the maximum one-way latency to the core-node.

7.5.3 Multiple core-node selection algorithms

Figure 7.1 illustrates the core-node selection done by the multiple core-node selection algorithms from section 7.3. From looking at the pictures, it may look as if the k-Tailed algorithm manages to choose the better selection, as they are spread fairly evenly out. The remaining algorithms look to miss certain areas of the canvas. However, the main optimization goal is latency in relation to the member-nodes that are to be managed, and the following results reveal which core-node selection algorithms actually find well-placed nodes.

Figure 7.2(a) plots the CDF of the average maximum latency from a client in the application to one of $k = 32$ core-nodes selected by a multiple core-node selection algorithm. We observe that k-Unicast-Walk and k-Tailed are the ones that yield the highest maximum latency to one core-node (among the k core-nodes). k-Unicast-Walk does a pseudo-random unicast walk in a network to find core-nodes, while the k-Tailed algorithm divides the network into more and more squares to find core-nodes based on their global coordinates, and not based on relative node proximity. k-Median, k-Broadcast-Walk and k-Center do all perform very similarly. Figure 7.2(b) evaluates k-Median and k-Center for $k = 4, 8, 16$ and 32. We observe that the maximum one-way latency does decrease step by step as the k is increased. k-Median and k-Center do perform very similarly, regardless of size of k. However, we observe that k-Center yield a smaller fraction of larger latencies, while k-Medien yield a larger fraction of smaller latencies. Figure 7.3 plots the CDF of the average maximum latency from a PlanetLab node to



(a) k-Unicast-Walk

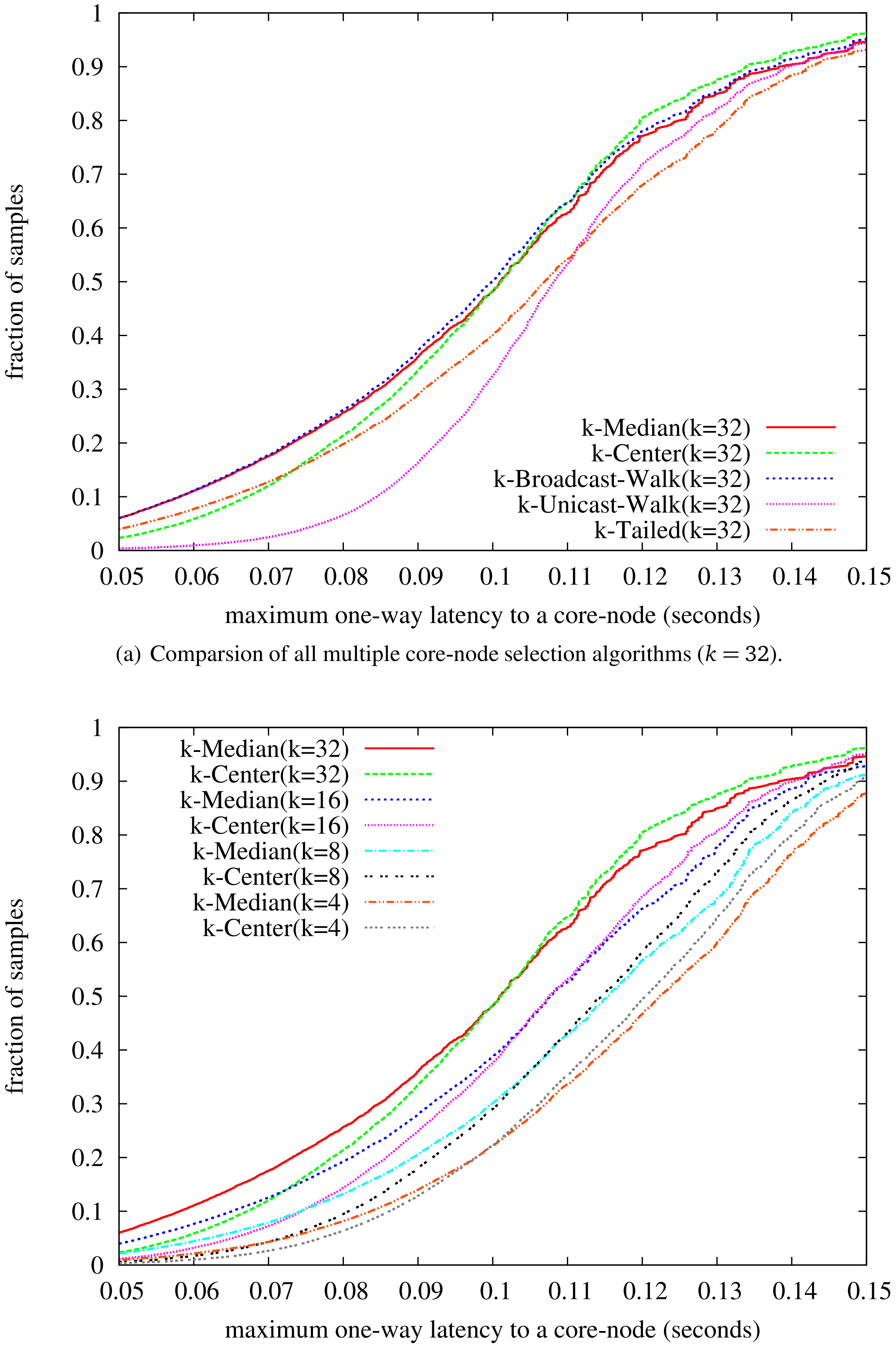
(b) k-Broadcast-Walk

(c) k-Median

(d) k-Center

(e) k-Tailed

Figure 7.1: 16 core-nodes as selected by the multiple core-node selection algorithms. 16 black squares are the core-nodes, white circles are the member-nodes.



(a) Comparsion of all multiple core-node selection algorithms ($k = 32$).

(b) Comparison of k-Median and k-Center ($k = 4, 8, 16, 32$).

Figure 7.2: Group communication simulations: Average maximum latency from a client to one core-node among k core-nodes, selected by a multiple core-node selection algorithm.

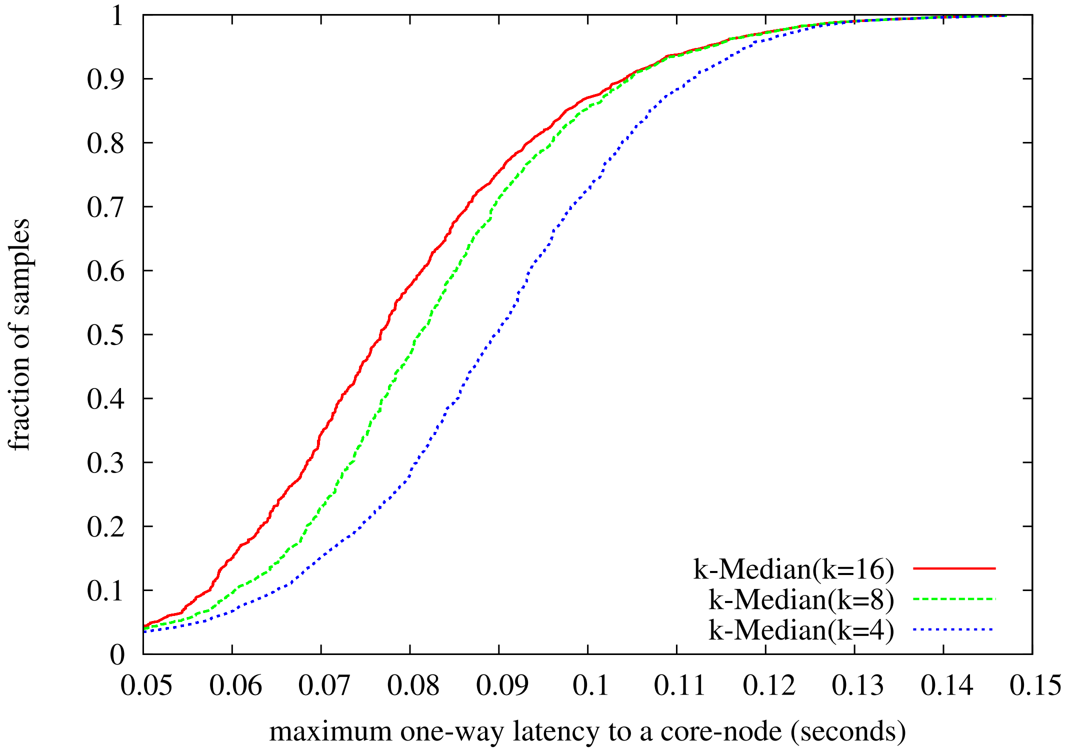


Figure 7.3: PlanetLab group communication experiments: Average maximum latency from a node to one core-node among k core-nodes, selected by k-Median.

one PlanetLab core-node. k-Median is used to find the core-nodes, with $k = 4, 8$ and 16. We observe that a similar pattern and performance emerges on these real-world measurements.

From these results, we see that the multiple core-node selection algorithms are able to select core-nodes that are well-placed and distributed across the network. The maximum one-way latency to one core-node among the k core-nodes is relatively low, and the latency decreases with an increased k.

7.5.4 Combining multiple and single core-node selection algorithms

The following results are from applying a single core-node selection algorithm ($k = 1$) to select one core-node among the $k \geq 1$ core-nodes selected by the multiple core-node selection algorithms. For these results, we measure the maximum one-way latency to this single core-node.

Figure 7.4 plots the maximum one-way latency from a group member-node to its core-node selected by the single core-node selection algorithm k-Median. Different multiple core-node selection algorithms are used to identify the available core-nodes in the network, and there is an increasing number of core-nodes to choose from. It is quite clear that the k-Tailed algorithm suffers because it does not consider the actual latencies on the core-nodes it picks as

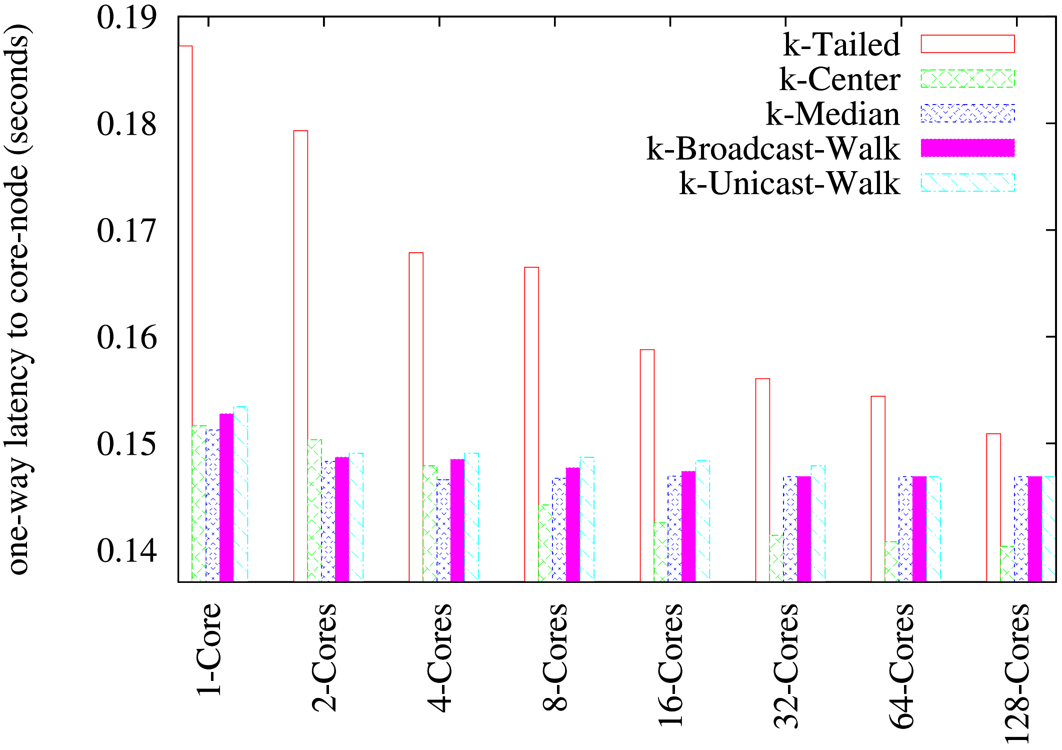
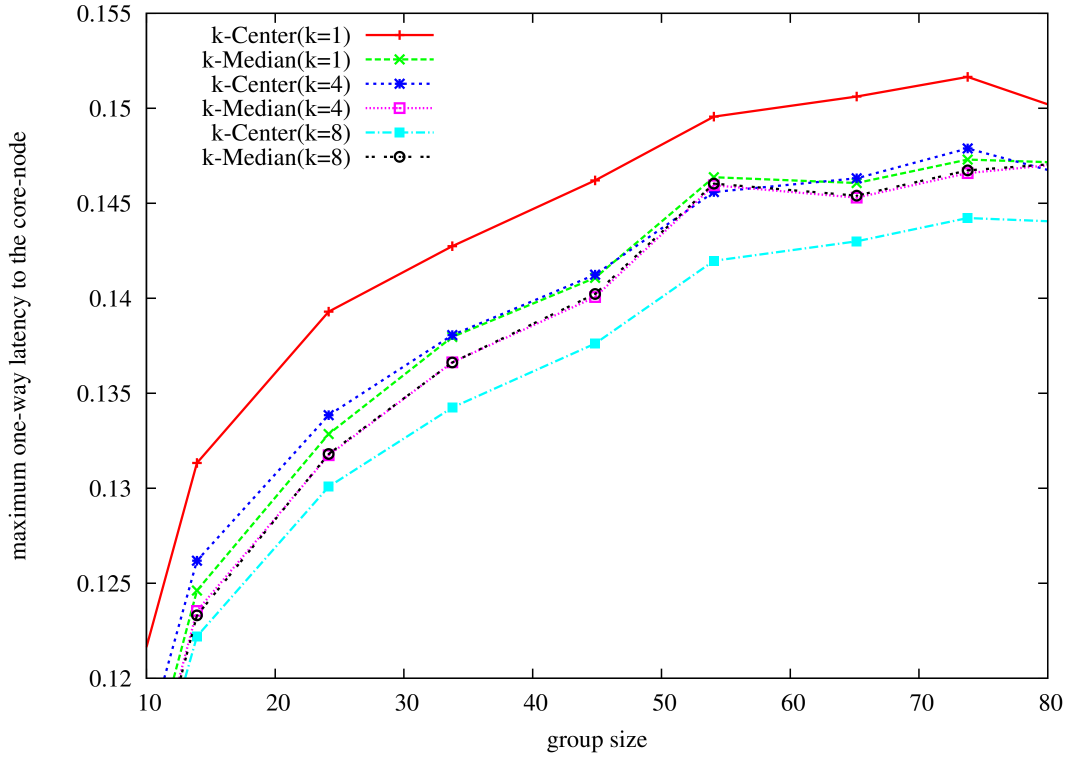


Figure 7.4: Group communication simulations: Average maximum latency from a member-node in a group, to the group's selected core-node (manager-node) as selected by a single core-node selection algorithm. The core-node set is selected by the multiple core-node selection algorithms, with different core-node set sizes.

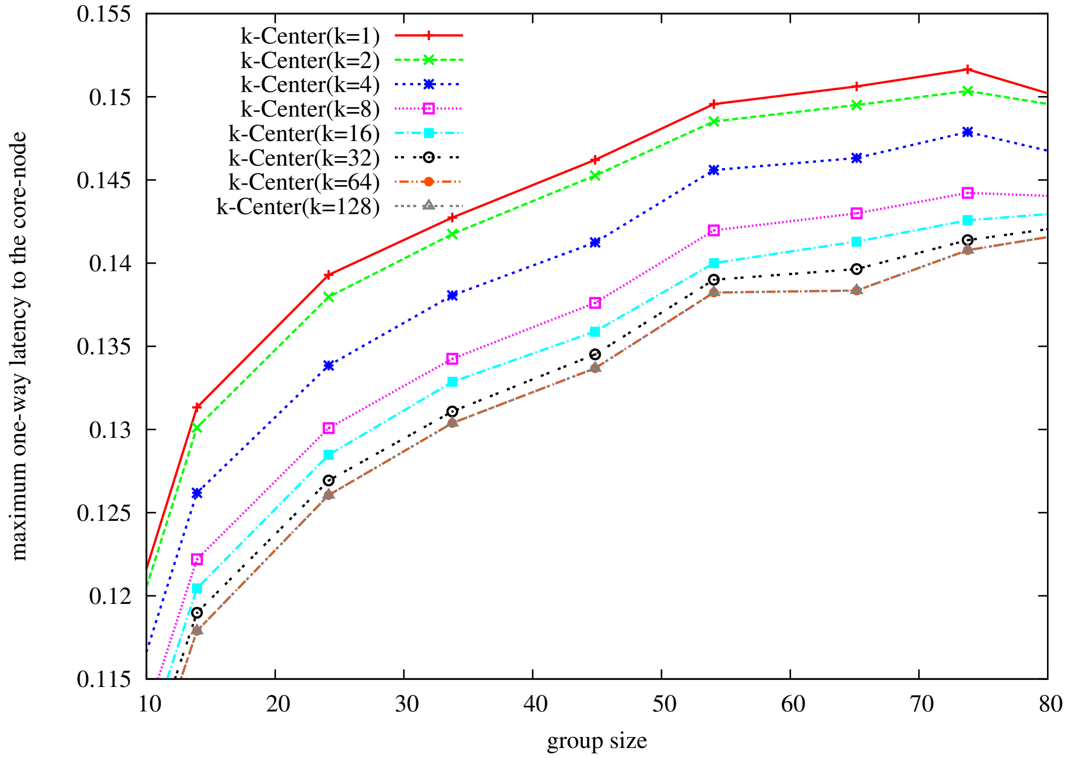
manager-nodes. The remaining multiple core-node selection algorithms perform very similarly. However, the tendency is in favor of k-Center, which consistently yield the lowest latency.

As mentioned, the k-Center is a heuristic of the minimum k-center problem (definition 25), and finds the core-nodes with minimum longest shortest path (minimum eccentricity) to the member nodes in the network. While, the k-Median is a heuristic of the minimum k-median problem (definition 26), and finds the core-nodes with minimum average pair-wise latencies to the member nodes in the network. These two goals are somewhat similar, but for the group communication simulations k-Center performs better because it finds the core-node with a minimum maximum latency. The k-Unicast-Walk and k-Broadcast-Walk perform very similar to k-Median. These two split the network in k smaller pieces and run k-Median($k = 1$) for each identified network piece. The split-network heuristics (k-Unicast-Walk, k-Broadcast-Walk and k-Tailed) do not perform as good in this scenario because the group communication simulations do not address the change in user mass as the day shifts to night across the real world. Generally, a different group dynamics model may have an impact on the performance of the core-node selection algorithms.

Figure 7.5(b) plots the maximum latency to a group's core-node (manager) selected by the



(a) k-Center and k-Median ($k = 1, 4$ and 8) finds the available core-nodes.



(b) k-Center ($k = 1, 2, 4, 8, 16, 32, 64, 128$) finds the available core-nodes.

Figure 7.5: Group communication simulations: Maximum one-way latency from a member-node in a group, to the group's selected core-node, selected by k-Median($k = 1$).

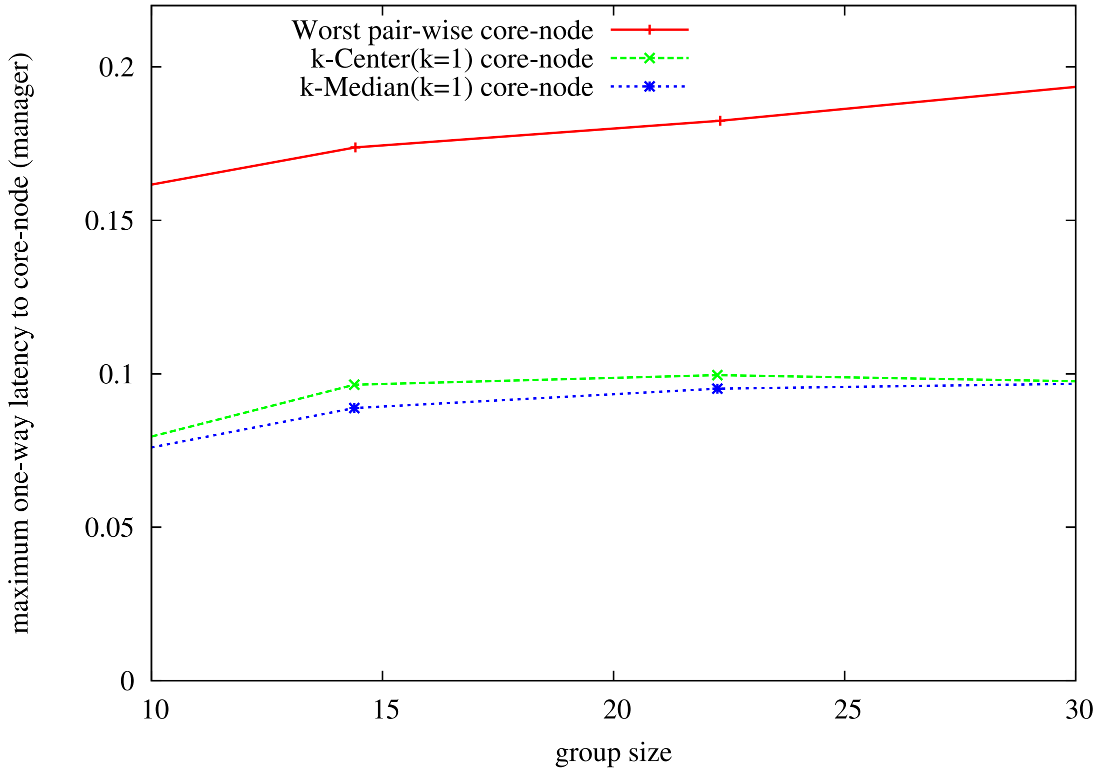


Figure 7.6: Group communication experiments on PlanetLab and single core-node selection: Maximum one-way latency to a core-node selected by a single core-node selection algorithm. The core-nodes are chosen freely from 100 PlanetLab nodes.

k-Center($k = 1$). The core-nodes are selected using the multiple core-node k-Center with different set sizes ($k = 1, 2, 4, 8, 16, 32, 64, 128$). It is clear that more available core-nodes enables the single core-node selection algorithm k-Center to find more well-placed manager nodes, which reduces the maximum latency to the member-nodes. However, the one way gain is only about 15 milliseconds from 1 core-node to 32 core-nodes, and is about 10 milliseconds from 1 core-node to 8 core-nodes. It is clear that only a few available core-nodes are necessary to reduce the maximum one-way latency to a group's manager-node.

Figure 7.5(a) compares the k-Center and k-Median algorithms with $k = 1, 4$ and 8. The results show that the k-Center algorithm struggles when $k = 1$, but performs better for $k > 1$. The results also show that k-Median is not able to perform better when the core-node set is ≥ 8. While, the k-Center algorithm continues to improve the maximum one-way latencies, the k-Median cannot. From these observations, the conclusion is that k-Center selection is the better alternative to identify the set of available core-nodes (managers) in a network ($k > 1$).

7.5.5 Results from single core-node selection algorithms

As mentioned, a single core-node selection algorithms should select a core-node that is well-placed related to a group of clients. The k-Median($k = 1$) and k-Center($k = 1$) are the two single core-node selection algorithms that are evaluted. The k-Median choose a core-node that has lowest average pair-wise latencies to a group of nodes, while the k-Center choose a core-node that has the minimum eccentricity (radius) between it and the nodes in the group. Figure 7.6 plots the maximum latency from a member-node to the core-node, selected by k-Median and k-Center selection algorithms. The worst-case is plotted as a reference point. The results are obtained from group communication experiments performed on PlanetLab, where the central entity continuously executed the single core-node selection algorithm to identify the group's manger-node. We observe that the single core-node selection algorithms perform very similarly. Their complexities are both $O(n^2)$, hence, they are both good alternatives for finding single well-placed core-nodes. But, the k-Median does perform slightly better, as was also observed in figure 7.5(a).

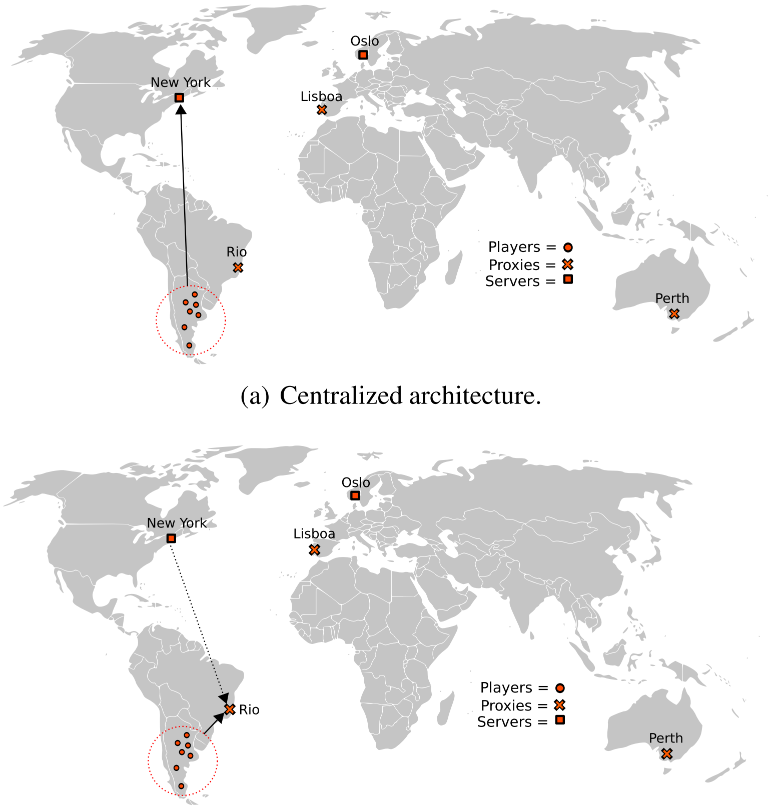
The worst case central entity placement may be a valid situation in cases where an application has a static central entity that serves the entire world. A static central entity cannot be in the current topological center at all times, because as the night and day passes the clients in a game dynamically shifts from continent to continent.

7.6 Managers in massively multiplayer online games

The previous evaluations focused on a fairly general scenario in which the goal was to find well-placed core-nodes for groups of clients. In the following, we highlight how the core-node selection algorithms are applicable for use in a distributed interactive application type called massively multiplayer online games (MMOGs). In this scenario, the core-node selection algorithms are used to select well-placed proxies to migrate game-state to.

Most current MMOGs rely on a centralized architecture, where the entire game state is stored on a central server. Such a client/server model makes it easy to manage the global game state, but it has drawbacks. The server is a potential bottleneck, both in terms of computing and bandwidth capacity, and the latency heavily depends on the physical distance from each individual client to the server. Figure 7.7(a) illustrates an example where a centralized server stores the game state and clients located anywhere in the world have to communicate through the server. Clearly, there is a latency penalty.

To improve these gaming scenarios, the previous core-node selection algorithms are used in combination with proxy technology to achieve better scalability and reduce the pair-wise latencies among interacting clients. The pair-wise latencies may be reduced by migrating game state to an appropriate proxy-server (core-node), that yield lower pair-wise latencies to a given



(a) Centralized architecture.

(b) Proxy architecture

Figure 7.7: Importance of server location: Centralized architecture vs. proxy architecture.

group of players (see chapter 2 for an introduction to proxy technology). Figure 7.7(b) gives an example where a central server has migrated game state to a proxy that is closer to a group of clients. Clearly, the latency is reduced.

7.6.1 Core-node selection in a proxy architecture

In this scenario, we evaluate proxy technology because it allows a trade-off between client/server and peer-to-peer advantages and disadvantages. However, a mix of client/server and peer-to-peer communication styles may be used for different traffic types fitting to these models.

In the group communication simulations, the k-Median core-node selection is used to select single core-nodes and k-Center to find the set of core-nodes (proxies) in the game network. The core-node selection algorithms use these defined approaches to search for core-nodes:

- *Topology Center:* k-Median($k = 1$) finds a (static) central entity (server) that yields the lowest pair-wise latency to all the clients in the application. The topology center server never changes regardless of group.

- *Group Center:* k-Median($k = 1$) finds a (dynamically changing) manager (core-node) for a group that yields the lowest pair-wise latency to the clients in the group. The group center dynamically changes depending on the group.

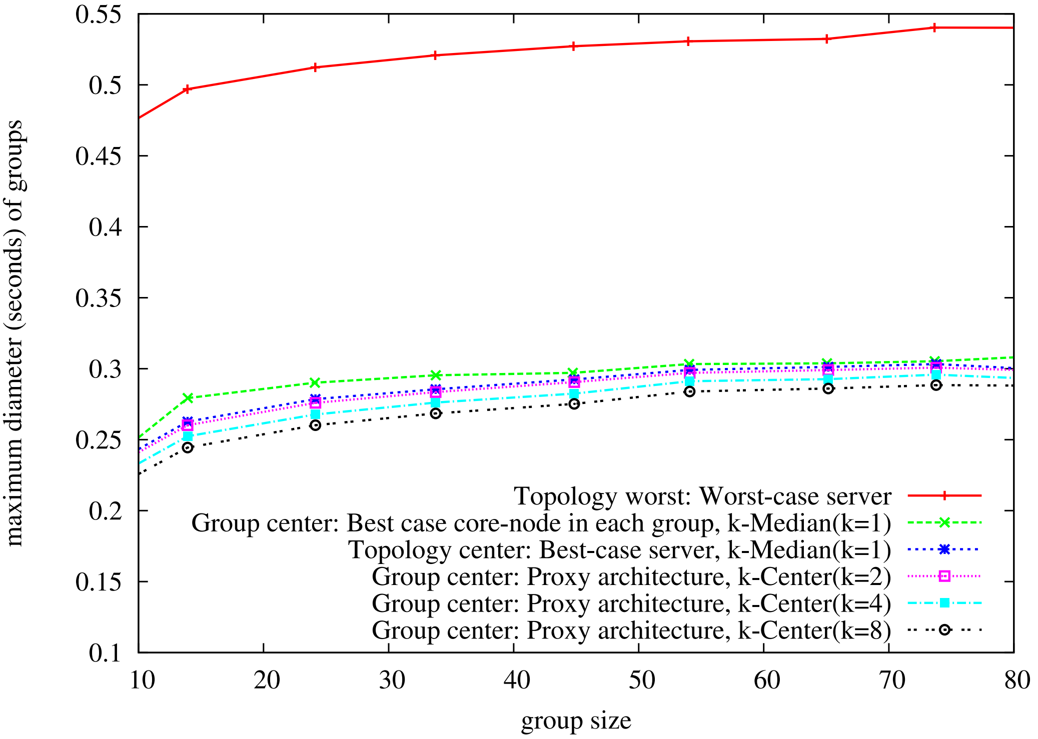


Figure 7.8: Group communication simulations: Diameter (seconds) of groups using a proxy architecture with different *k* number of proxies.

These simple approaches form powerful techniques in the search for suitable core-nodes that yield low pair-wise latencies.

7.6.2 Simulations and experiments

The goal is to illustrate the significance of server location and how it influences the latencies. Having a proxy architecture with a limited amount of proxies can dramatically reduce the overall latencies. The group's diameter expresses the worst-case pair-wise latency between two clients in a group, and it is desirable that the group diameter is as low as possible.

Figure 7.8 plots the average group diameter for which a single core-node selection algorithm have chosen a manager-node as the root of the group tree (all communication flows via the root). We see that choosing the manager-node (server) to be in the topology center does significantly reduce the group diameter. It is also clear that having a limited amount of proxies placed around the world can reduce the group diameter. And as expected, increasing the number of proxies does incrementally decrease the diameter. The worst-case placement of a server is not an invalid situation, because if a single server administrates the entire world the worst-case placement of a server will happen for large amounts of the clients as the day passes.

Figure 7.9 plots the group tree diameter based on experiments done on 100 Planetlab nodes. We observe similar results as from the group communication simulations that were based on

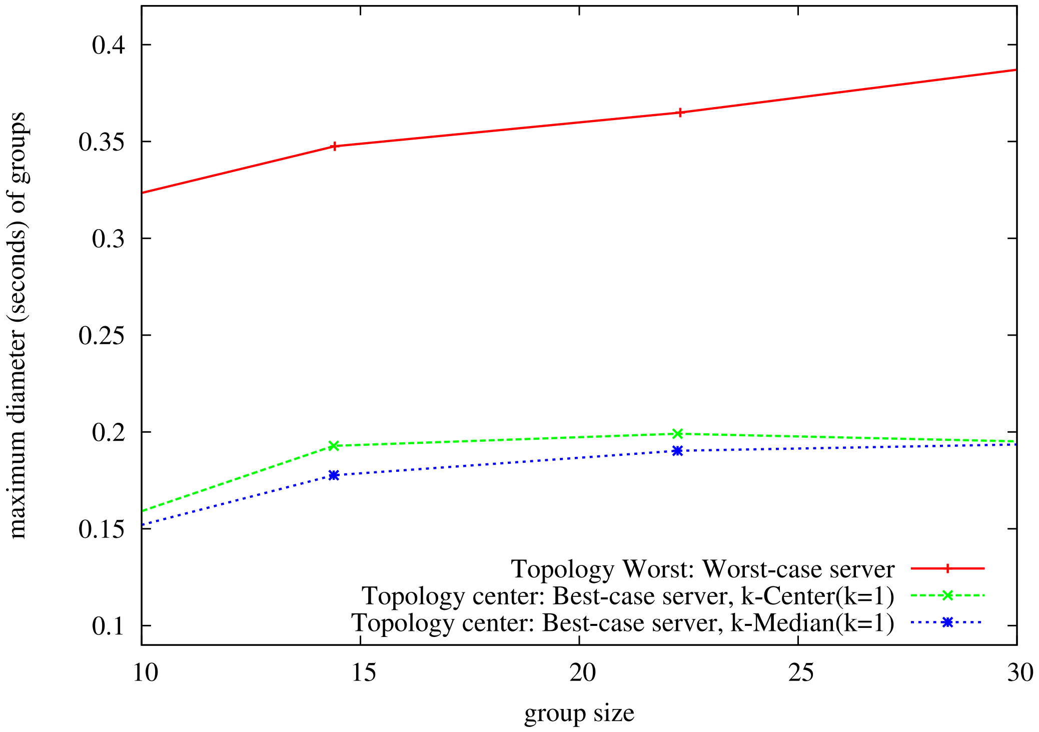


Figure 7.9: PlanetLab experiments: Diameter (seconds) of groups using a single core-node selection algorithm.

BRITE [91] generated graphs. The conclusion is that the core-node selection algorithms do work in practice, and they are able to identify nodes in the network that yield low pair-wise latencies to groups of clients.

7.7 PlanetLab experiments: Latency estimates applied to core-node selection

Chapter 6 introduced the latency estimation techniques Vivaldi and Netvigator. These techniques were evaluated in terms of their latency estimation accuracy in comparison to the real all-to-all ping measurements. It was observed from the results that Netvigator yields the better latency estimates. Vivaldi is less accurate, but the implementation lends itself more easily to a peer-to-peer setting.

One important issue related to centralized core-node selection algorithms is that they need all the required network information to be available at the executing node. Therefore, latency estimation techniques are important to enable centralized core-node selection algorithms in large-scale applications. When latency estimates are available, the issue then becomes how these latency estimates affect the performance of the core-node selection algorithms.

The following results are from applying the Netvigator and Vivaldi latency estimates to the

Descriptions	*Configurations*
Group sizes	g = 4, 8, 12 clients
RTT measures	*tcpinfo*, *ping*
Packet rates	high (100 packets/sec.), low (2 packets/sec.)
Log times	t = 4, 8, 12, 16, 20 minutes

Table 7.2: Vivaldi experiment configurations.

core-node selection algorithm k-Median. For comparison to the close-to-real latencies, the real all-to-all ping measurements are applied to k-Median as the point of reference.

7.7.1 Group communication experiment configurations

For the group communication experiments on PlanetLab, we used 100 nodes. This limited number was due to many unstable PlanetLab nodes and end-to-end reachability issues. The 100 nodes dynamically joined and left groups throughout the experiments by sending membership change (join and leave) requests to a central entity, which was chosen using the topology center heuristic (section 7.6.1).

The central entity applied the multiple core-node selection algorithm k-Median to find k core-nodes that are made available as well-placed manager-nodes. The algorithm used a complete graph of the application network, which was built using the latency estimation techniques introduced previously in chapter 6. We also used all-to-all ping measurements for comparison. The group popularity was distributed according to a Zipf distribution. For the Vivaldi estimates, we allowed a period of 4 minutes to let the node coordinates stabilize. The tests were run each day for a 10 day period. Table 7.2 gives the Vivaldi configurations.

7.7.2 Core-node selection metrics

We use the following metrics to highlight the quality of Netvigator and Vivaldi estimations when used for core-node search:

- The *core-node selection hit ratio* measures the ratio of optimal core-node selection hits when using estimated link latencies. In the context of the experiments, a core-node "hit" occurs for each time the k-Median finds the same core-node using first the estimated latencies and then the real all-to-all ping measurements.

- The *core-node density* is a measure for the maximum latency (eccentricity) between the core-nodes. The maximum-latency is obtained by applying the core-nodes found using the latency estimates to the all-to-all ping measurements. The core-node density value is the real eccentricity latency from the all-to-all ping measurments.

- The *minimum core-node error* (min) is the minimum latency between the core-node found using the estimated latencies, and the core-node found using the real all-to-all ping measurements. In the context of the experiments, the core-node error is zero if the k-Median algorithm found the same core-node using first the latency estimates and then the real all-to-all ping measurements.

- The *maximum core-node error* (ecc) is the maximum latency between a core-node found using the estimated latencies, and the core-node found using the real all-to-all ping measurements. In the context of the experiments, the maximum core-node error is used to highlight the maximum latencies (eccentricities) between the core-nodes found using the latency estimates, and the core-nodes found using the all-to-all ping measurements.

In addition, we measure the relative error and directed relative error of the one-way latencies (eccentricities) between the core-nodes and the group members. The eccentricity of a core-node is the maximum one-way latency (shortest path) to a group member. To avoid confusion, we define reported eccentricity, real eccentricity and optimal eccentricity:

- The *reported eccentricity* is the core-node eccentricity obtained when k-Median uses the estimated latencies to find cores-nodes.

- The *real eccentricity* is obtained when the core-nodes that k-Median found using the estimated latencies are applied to the all-to-all ping measurements.

- The *optimal eccentricity* is the core-node eccentricity obtained when k-Median uses the real all-to-all ping measurements to find core-nodes.

From an application point of view that uses a latency estimation technique, the reported eccentricity is what the application is reported the core-node eccentricities to be, based on the latency estimates. The real eccentricity, on the other hand, represents the real-world core-node eccentricities based on measured network latency (which the application does not know). The optimal eccentricity is only obtained if the application uses latency measurement tools (ping and traceroute), to do all-to-all measurements.

7.7.3 Group communication experiment results

We evaluate the core-node selection results towards the metrics introduced previously. The k-Median algorithm searches for $0 < k \leq 25$ nodes among the 100 PlanetLab nodes that yield the smallest average pair-wise latencies.

Figure 7.10 plots the core-node selection hit ratio when k-Median is applied to Netvigator and Vivaldi (ping RTT, low packet rate, g=8), and k varies between 1 and 25. It is clear that

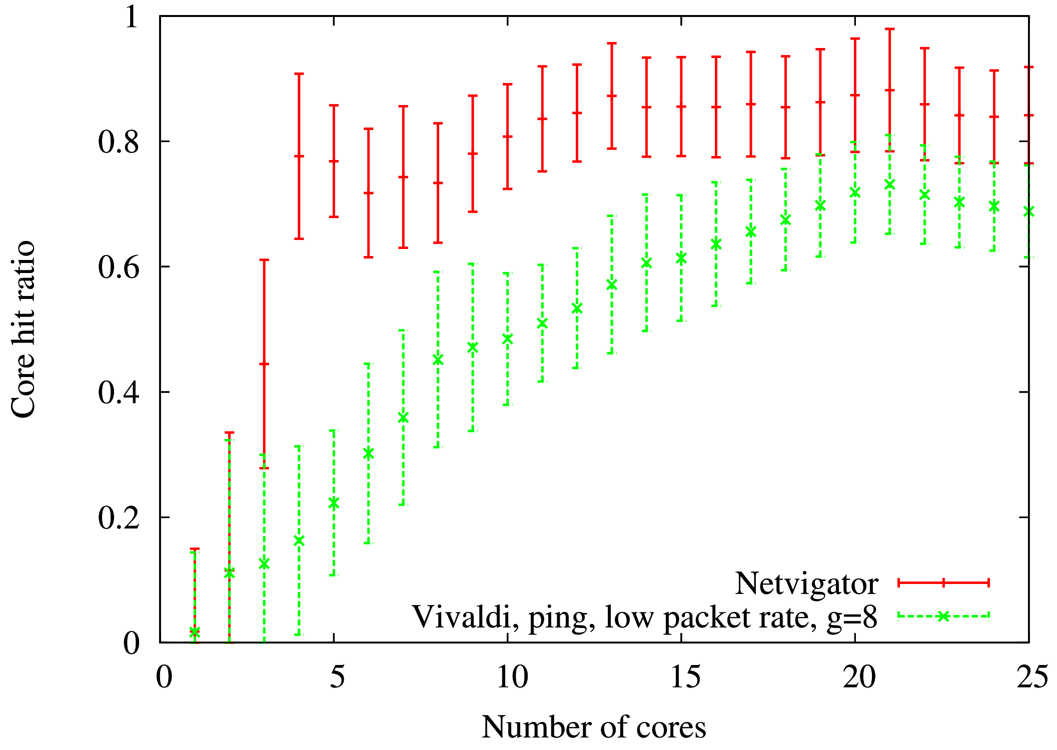
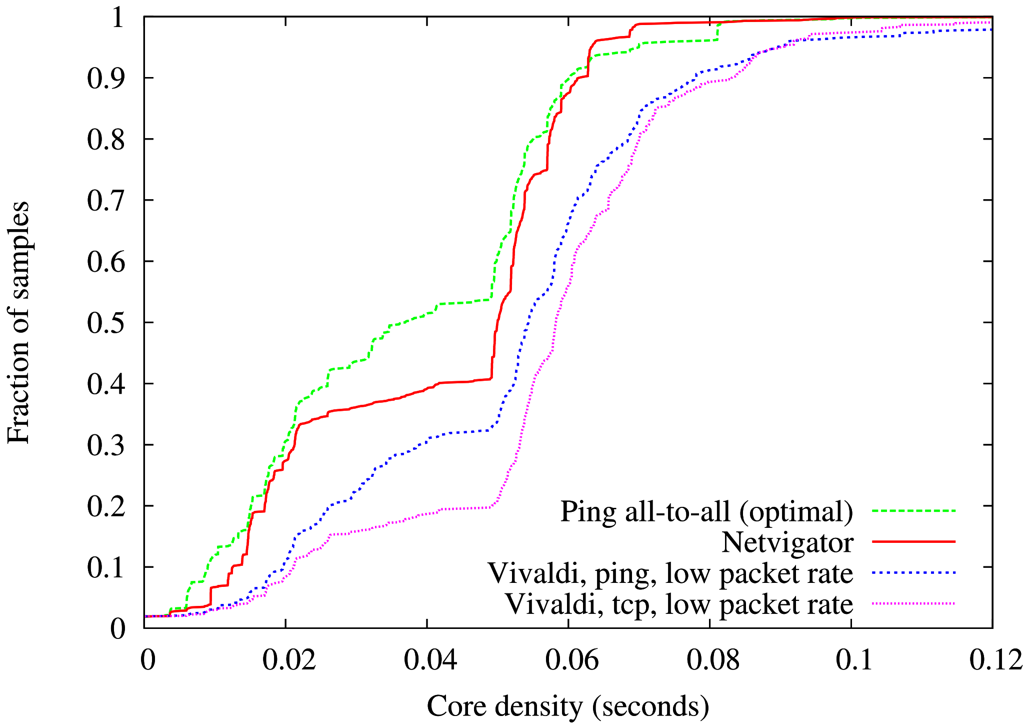


Figure 7.10: Ratio of optimal core hits when k-Median is applied to Vivaldi and Netvigator estimates.

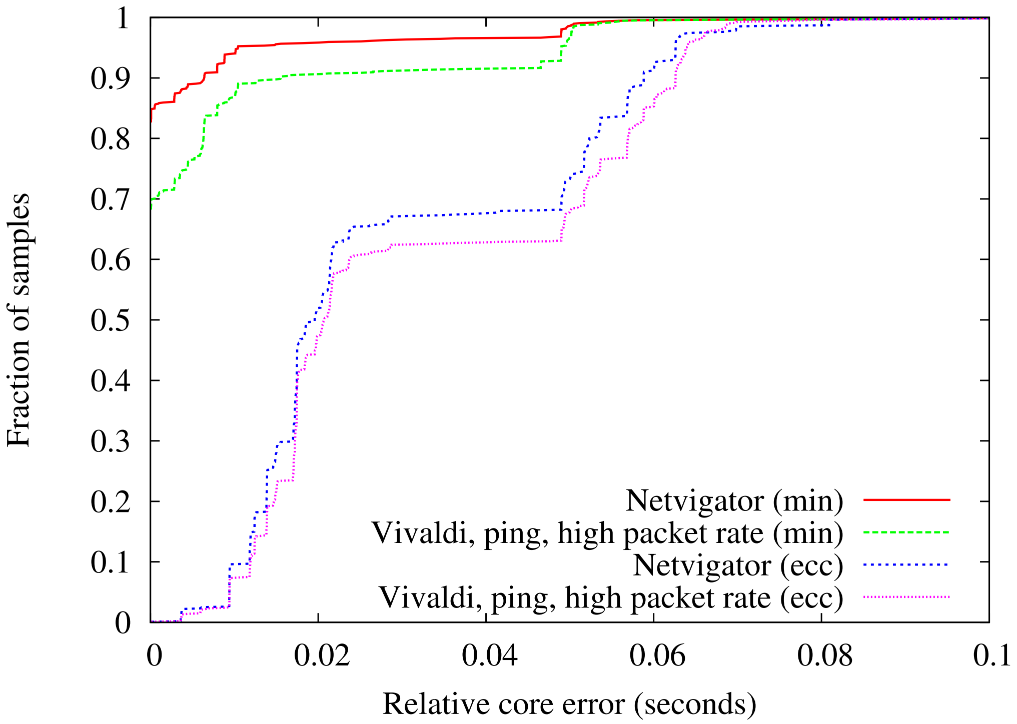
Netvigator yields better estimates for use in core-node search, and stabilizes around 80 % core-node selection hit ratio quickly. The hit-ratio is much lower when the k-Median algorithm is applied to Vivaldi estimates, especially when the number of core-nodes $k < 10$.

The core-node density, plotted in figure 7.11(a), compares the optimal eccentricity and the real eccentricity between the core-nodes found using k-Median. We see that Netvigator yields the best latency estimates, where the core-node eccentricities are very close to the optimal eccentricity (obtained from all-to-all ping measurements). The Vivaldi estimates are not as good, but still most of the core-nodes are within 20 milliseconds of the optimal core-nodes.

We observed that Vivaldi struggled to find the optimal core-nodes compared to Netvigator. Figure 7.11(b) plots the CDF of the minimum (min) core-node error between the core-nodes found using latency estimates and the (optimal) core-nodes found using the real all-to-all ping measurements. If the core-node error is zero, the k-Median algorithm found the same core-nodes. As expected, k-Median is most accurate when it uses Netvigator, with 95 % of the core-nodes within 10 milliseconds of an optimal core-node. The Vivaldi estimates makes k-Median return more inaccurate results, with 85 % of the core-nodes within 10 milliseconds of an optimal core-node. Figure 7.11(b) also plots the CDF of the maximum latency (ecc) between the core-nodes found using latency estimates and the (optimal) core-nodes found using the real all-to-all ping measurements. Both the minimum (min) core-node error and the maximum latency (ecc) illustrates that although Vivaldi yield poorer estimates for use in core-node selection, the k-Median is still able to find core-nodes that are close-to-optimal.



(a) CDFs of core density. Eccentricity between cores.



(b) CDFs of relative core error. Minimum (min) latency and eccentricity (ecc) for cores from estimates to real cores.

Figure 7.11: Netvigator and Vivaldi performance in core selection. Searching entire network.

Similar to section 6.3.3, we visualize the discrepancy between the reported eccentricities and the real eccentricities in figure 7.12. Vivaldi uses ping RTTs, and we see that a low packet rate provides poorer estimates than a high packet rate (after 4 minutes). Netvigator is clearly the best, and has a performance very close to the real eccentricity. Further, figure 7.13 visualizes the eccentricity discrepancy in terms of the directed relative error. Vivaldi now uses tcpinfo RTTs, and a similar pattern can be observed. A low packet rate gives poorer estimates than a high packet rate, and Netvigator is best.

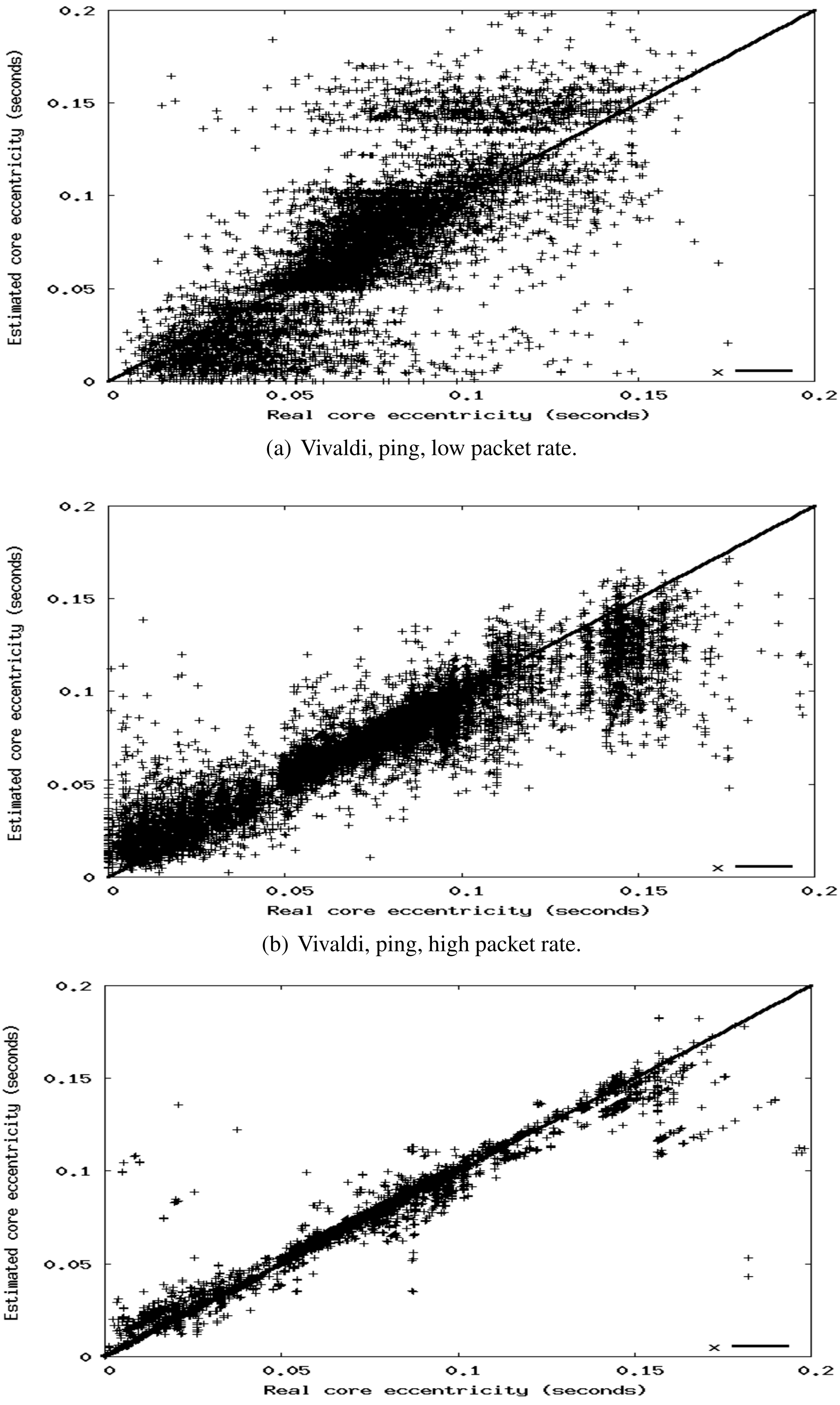
In summary, k-Median is able to find close-to-optimal core-nodes when applied to both Netvigator and Vivaldi latency estimates. However, it is clear that Netvigator estimates enables the core-node selection algorithm to return the more optimal core-nodes.

7.8 Summary of the main points

The need to identify well-placed core-nodes that yield low pair-wise latencies to groups of clients is motivated by the desire to enable distributed interactive applications that support multiple dynamic sub-groups of clients. The chapter has introduced and evaluated several core-node selection algorithms that used latency to search for well-placed core-nodes. These core-nodes were to act as manager-nodes for groups of clients and execute the centralized group management techniques from chapter 5.

The evaluations were based on results from a group communication simulator and also group communication experiments performed on PlanetLab. We found that the k-Center algorithm was the best among the tested algorithms, and is fitting to identify multiple core-nodes in a network. These core-nodes are made available for single core-node selection algorithms that select a manager-node for a group of clients. For single core-node selection, we found that k-Median($k = 1$) was the algorithm that found core-nodes that yielded the lowest maximum one-way latencies to clients in its group.

The core-node selection algorithms that were tested are all centralized algorithms, which is due to the strict latency requirements in distributed interactive applications. One issue with centralized algorithms is that all the neccessary node and link information must be available at the executing node. Therefore, we wanted to measure the penalties involved when latency estimates are used by the core-node selection algorithms to find well-placed core-nodes. The experiments were performed on PlanetLab and the latency estimates were from Vivaldi and Netvigator that are both evaluated in chapter 6. The results showed that when Netvigator's latency estimates are used, the k-Median algorithm finds close-to-optimal core-nodes. Vivaldi's latency estimates did not enable k-Median to find the optimal core-nodes; however, we found the core-nodes it found to be sufficiently well-placed. The main conclusion was that both Vivaldi and Netvigator provide good enough latency estimates to be used for core-node search.

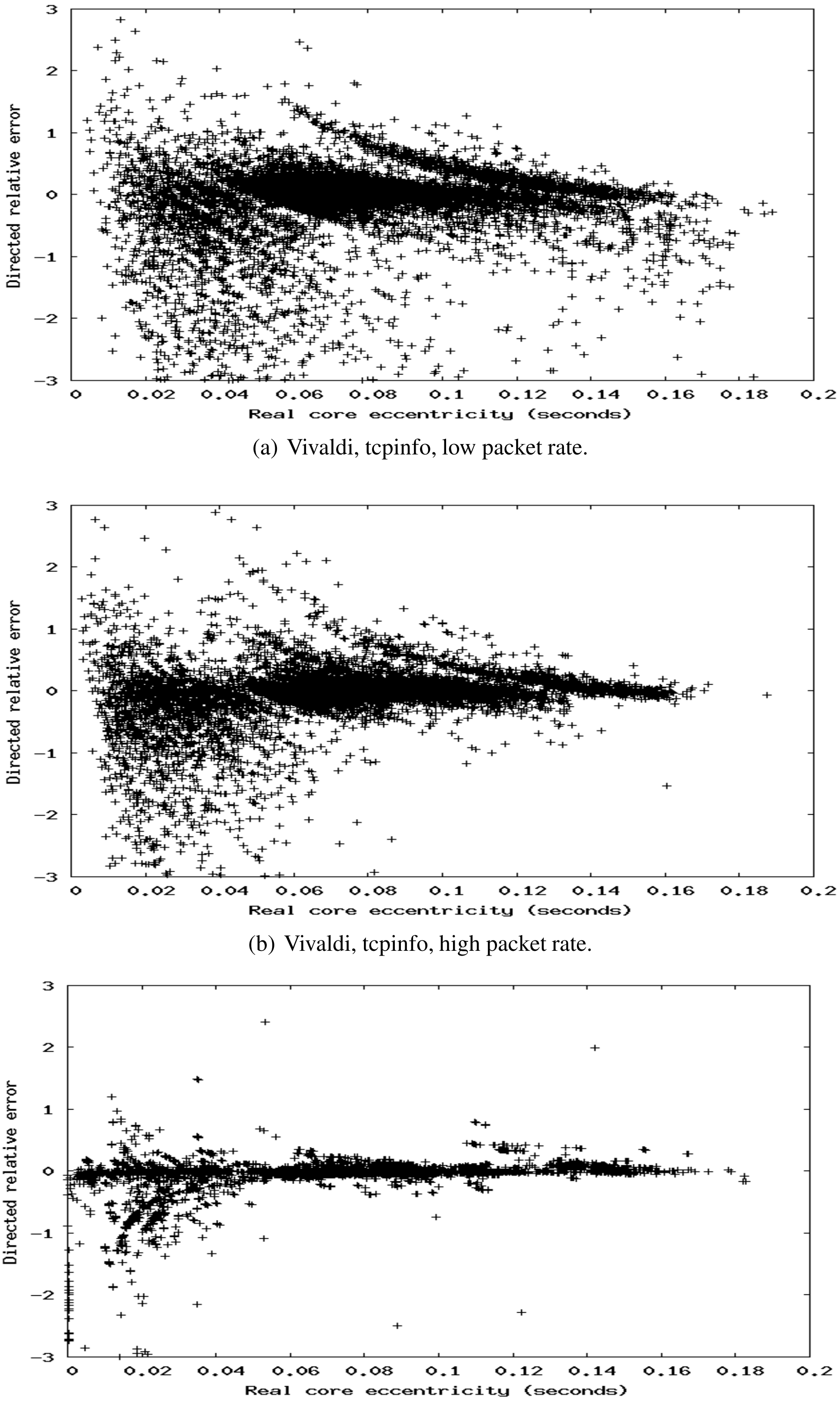


(a) Vivaldi, ping, low packet rate.

(b) Vivaldi, ping, high packet rate.

(c) Netvigator.

Figure 7.12: Reported eccentricity vs. real eccentricity from cores to group members.



(a) Vivaldi, tcpinfo, low packet rate.

(b) Vivaldi, tcpinfo, high packet rate.

(c) Netvigator.

Figure 7.13: Directed relative error of the reported eccentricity to the real eccentricity from cores to group members.

Chapter 8

Group specific enhancements: Graph manipulation algorithms

Chapter 5 introduced a centralized group management approach, which is a focal point of the thesis. This centralized group management includes overlay construction techniques whose main task it is to construct low-latency overlay networks for event distribution. To achieve this goal, the overlay management contains *graph manipulation algorithms* that operate on *complete group graphs*. A complete graph is a graph in which all nodes have a link to all other nodes, and is also commonly referred to as a full mesh. Each complete group graph consists of member-nodes that are the clients in the application who are grouped by the membership management (section 5.2), and graph edges that are overlay links with an associated link latency, which is determined by a latency estimation technique (chapter 6).

In this context, the main task for a graph manipulation algorithm should be to manipulate a group's complete graph such that it enables the overlay construction algorithm to execute fast and build desirable low-latency overlay networks.

One example is to reduce the complete group graphs to only include the better links such that the overlay construction executes faster. Another example is to identify well-placed core-nodes in the application network that may be used as Steiner-points (non-member-nodes) in the group graphs for Steiner-tree or Steiner-subgraph algorithms.

A group graph is used as the input graph to an overlay construction algorithm, which creates an overlay network for event distribution. It is therefore in the overlay construction that we can measure the effect of the graph manipulation algorithms. Hence, the evaluation focuses on how the graph manipulation algorithms alter the behavior of the overlay construction algorithms. The results show how a few selected overlay construction algorithms build overlay networks that exhibit varying properties depending on which graph manipulation algorithm is applied to manipulate the complete group graphs. In that respect, we are focusing on two main metrics, execution time of the overlay construction and the success rate of the overlay construction.

We use two overlay construction algorithms from section 9 and 10 to highlight the importance of the graph manipulation algorithms. The main conclusion from the evaluation is that some graph manipulation algorithms do reduce the execution time of selected overlay construction algorithms, and other graph manipulation algorithms enable the overlay construction algorithm to construct overlays with lower pair-wise latencies.

The rest of the chapter is organized in the following manner. Section 8.1 introduces the goal of the chapter in terms of what type of graph manipulation algorithms are targeted. Section 8.2 presents the background and motivation for reducing the number of edges in a complete graph. Similarly, section 8.3 presents the background and motivation for including well-placed Steiner-points to complete graphs. Section 8.4 introduces two edge-pruning algorithms and one algorithm for including Steiner-points to complete graphs. Section 8.5 evaluates a range of issues related to the execution time of overlay construction and the influence of Steiner-points in complete graphs. Finally, section 8.7 gives a brief summary of the main points.

8.1 Graph manipulation algorithm goals

Graph manipulation algorithms may have very different tasks and properties, many of which are outside the scope of the thesis. In the thesis, our main desire is to:

Identify graph manipulation algorithms that manipulate a group's complete graph such that it enables the overlay construction algorithm to execute fast and build desirable low-latency overlay networks.

One vital consideration of the overlay management is the time it takes to execute the overlay construction. Distributed interactive applications require timely overlay management, such that every membership change of online sessions must be very fast, including the overlay construction algorithms.

Another consideration is the quality of the overlay network that is built by the overlay construction algorithms. Distributed interactive applications require low-latency overlay networks for event distribution. Therefore, an overlay construction algorithm should be given as input a group graph, which is manipulated such that the algorithms's chance of constructing low-latency overlays is increased. These observations result in two main algorithm goals:

- ***Reduced overlay construction time:*** A graph manipulation algorithm should be available, which reduces the number of edges in the otherwise complete group graphs such that the execution time of the overlay construction algorithm is reduced.
- ***Increased overlay construction quality:*** A graph manipulation algorithm should be available, which manipulates the group graph such that overlay construction algorithms produce low-latency overlay networks.

The overlay construction time is directly linked to the time complexity of the overlay construction algorithm. An algorithm's time complexity is often dependent on the size of the input graph, that is, its vertex-set size and edge-set size. The vertex-set size is already bounded to group membership by the membership management. The edge-set size, on the other hand, is very large in a complete graph, which is the case for the group graphs. Overlay application layer networks are always complete graphs (fully meshed), therefore, the edge-set size may severly increase the overlay construction time. As mentioned, a complete graph is a graph in which all nodes have a link to all other nodes.

The overlay construction quality is related to the probability the overlay construction algorithm has to construct overlay networks that yield sufficiently low pair-wise latencies. The pair-wise latencies may be addressed by overlay construction algorithms as a constraint; however, such a constraint makes it hard to solve the overlay construction. Many constrained overlay construction problems are only solvable by exponential time algorithms and belong to graph theoretical problems that are NP-complete.

Chapter 4 introduced a wide range of NP-complete constrained overlay network design problems. The constraints were, for example, degree-limitations, total cost bounds, diameter bounds and radius bounds. Overlay construction algorithms that solve NP-complete problems have non-polynomial time complexities. Therefore, the focus is rather on evaluating overlay construction heuristics that cannot guarantee an optimal solution. The ultimate goal is to enable these non-optimal heuristics to construct low-latency overlay networks.

8.2 Reducing the overlay construction time

The overlay construction time is heavily influenced by the time complexity of the overlay construction algorithm that executes it. Reducing or bounding this execution time is important for time-dependent distributed interactive applications that is to support clients joining and leaving ongoing group sessions (chapter 2). Therefore, we apply three methods to reduce the overlay construction time and still enable the overlay construction algorithm to construct low-latency overlays:

- ***Algorithm time complexity:*** Apply algorithms with time complexities that are dependent on the edge-set size.

- ***Edge-pruning:*** Reduce the number of edges in the otherwise complete input graph, by applying edge-pruning strategies.

- ***Core-node selection:*** Leave the edges in the group graph that connect centrally placed nodes to more deserted nodes to achieve a low-latency reduced graph.

8.2.1 Algorithm time complexity

One way to reduce the overlay construction time, is to reduce the time complexity of the executing algorithm. It is, however, difficult to reduce the time complexity of an algorithm, and at the same time enable it to construct overlays with low pair-wise latencies. However, overlay construction algorithms do all examine graphs, and therefore the execution time is also dependent on the size of this graph. An application layer overlay network is a complete graph (full mesh) of shortest paths. Therefore, the number of edges in a group graph grows exponentially when new clients are connected. Hence, if an overlay construction algorithm operates such that it examines all of the edges in the input graph, its execution time is dependent on the number of edges. Thus, if the number of edges in the graph decreases, the execution time of the algorithm also decreases.

Many of the overlay construction algorithms considered in chapter 9 through 14 have a time complexity, which is dependent on the number of edges the input graph has. A group graph with a reduced number of edges may significantly decrease the execution time of such overlay construction algorithms, but the reduced graph must exhibit qualities that enables the overlay construction algorithm to create overlay networks with low pair-wise latencies.

When core-node selection algorithms are used to identify well-placed group members, and edge-pruning algorithms, are used to identify good links, a reduced graph may be produced that yield qualities enabling the overlay construction algorithm to still construct low-latency overlays.

8.2.2 Edge-pruning algorithms: Search for quality edges

Edge-pruning algorithms are subgraph algorithms, and they typically compute a connected mesh on an input graph by applying, often configurable, edge-selection strategies. A connected mesh is defined here as a connected cyclic subgraph, which connects all the nodes in the input graph. Edge-pruning algorithms include strategies for removing edges from an input graph based on some goal, and also algorithms that pick single edges from an input graph and constructs a mesh. These two approaches are essentially different; however, the algorithms share the same goal. Consequently, we call all of them edge-pruning algorithms.

The goal of applying edge-pruning algorithms on complete group graphs is to reduce the number of edges, such that the overlay construction algorithm finishes faster. However, the reduced group graph must include enough "good" the edges to enable the overlay construction algorithm to construct the desired overlays. There are two basic edge-types to search for:

- ***Low-cost edges:*** Edges that reduce the total-cost of the subgraph.
- ***Low-latency edges:*** Edges that reduce the diameter of the subgraph.

Edges that reduce the total cost of the subgraph are easy to include, because each node's minimum-cost edges are a part of a minimum-cost spanning subgraph. Therefore, each node may trivially include its least cost edges to a subgraph to construct the least cost subgraph.

Edges that reduce the diameter of a subgraph is harder to find, because they rely on path properties (section 4.2). Hence, it is not enough to only consider single edges when creating a subgraph to reduce the diameter. The entire path including edges and nodes must be considered.

The diameter path in a subgraph is likely to include nodes that are centrally located (well-placed) among the group nodes. Therefore, these well-placed nodes are searchable using core-node selection algorithms.

8.2.3 Core-node selection: Search for well-placed nodes

A group graph has nodes that are more centrally located than others, for example, a node with the lowest eccentricity or the lowest pair-wise latency. When a low diameter subgraph is desired, it is these well-placed nodes that should be connected to the nodes with higher eccentricities or pair-wise latencies. This way, the reduced graph is ensured to have a low diameter, and an overlay construction algorithm is therefore able to build low-latency overlays.

The well-placed nodes may be found using, for example, the core-node selection algorithms k-Center and k-Median. These algorithms were introduced and evaluated in chapter 7, where they were found to be the better performing core-node selection algorithms. When the well-placed nodes are identified, one option is to inter-connect them with the more "badly-placed" nodes, and add the edges to the reduced graph.

Section 8.4 introduces two subgraph construction algorithms that apply edge-pruning and core-node selection to reduce the size of the complete group graph's edge-set and still enable the overlay construction algorithm to build low-latency overlays.

8.3 Increased overlay construction quality

Each group graph contains member-nodes that are clients in the application. These clients are, for example, interacting among each other and sharing events. Next, we introduce a particular class of overlay construction algorithms that know the difference between member-nodes and non-member-nodes, namely Steiner algorithms (section 4.6).

A Steiner algorithm typically constructs a connected subgraph, in which each member-node is reachable from every other member-node. The connected subgraph may also include non-member-nodes (Steiner-points) if they help the Steiner algorithm achieve its optimization goal. It has been proven that Steiner algorithms yield (on average) better subgraphs (trees and meshes) than non-Steiner algorithms [69]. However, Steiner algorithms are only usable if the input graph includes non-member-nodes, therefore we apply two steps to take advantage of Steiner-algorithms:

- ***Identify super-nodes***, and make them available as Steiner-points for Steiner algorithms.
- ***Identify Steiner-points***, and include them to a specific group's input graph.

8.3.1 Core-node selection: Search for super-nodes

Distributed interactive applications consists of clients who are currently interacting, and most likely application provided servers and proxies. It is from these network nodes that well-placed Steiner-points should be identified and then made available to a Steiner algorithm. Steiner-points are also found in today's peer-to-peer applications. They are often referred to as super-nodes, core-nodes or relay-nodes, and are used to relay data without reading it. The super-nodes are typically selected because they have some feature capacity, which is desirable for that specific application type. In distributed interactive applications it is low pair-wise latencies that is the most desired super-node metric.

Based on these observations, we apply core-node selection algorithms to identify well-placed super-nodes that yield low pair-wise latencies to the clients in the application network. These super-nodes are then made available as Steiner-points that can be added into a specific group's complete graph.

8.3.2 Core-node selection: Search for Steiner-points

When Steiner algorithms are used, they need Steiner points to be present in the input graph, otherwise they are reduced to spanning algorithms [131]. These Steiner points may be included randomly from the application network, but this is obviously not a good approach. Instead, a sub-set of the previously identified super-nodes can be included as Steiner points in a specific group's graph. Steiner algorithms must be able to use the Steiner-points, therefore it is vital that the identified Steiner-points are located in the close vicinity of the member-nodes. Hence, a group's Steiner points can be identified with a core-node selection algorithm.

In our investigations, we evaluate many Steiner algorithms that build subgraphs while being subject to *degree limits* and a *bounded diameter*. The degree limit is important to limit the stress on overlay nodes, in terms of bandwidth consumption, forwarding stress, etc. The bounded diameter ensures that the constructed subgraph has a maximum pair-wise latency within the application requirements, which is 200 milliseconds for first person shooter games (chapter 2). However, finding a subgraph that yields a diameter below the bound, and node degrees below (or on) the degree limits, is an NP-complete problem. Therefore, due to the time-critical overlay construction requirements, we apply non-optimal polynomial time Steiner heuristics. However, these heuristics cannot guarantee that a solution is found.

Due to the non-optimality of the evaluated Steiner-heuristics, their performance is heavily influenced by which Steiner-points are included to the complete group graph. In general, the Steiner-points should assist the Steiner-heuristics in the overlay construction, and therefore be well-placed to increase the chance of finding a subgraph that obeys the given constraints. Low pair-wise latencies is a target metric, and therefore Steiner points should be identified by core-node selection algorithms considering pair-wise latencies.

Another issue pertains to how many Steiner points that should be included. This is an open question, and we investigate this issue along with the previous mentioned issues later in section 8.5.4.

8.4 Complete group graph manipulation algorithms

We devised three graph manipulation algorithms that address the previous observations regarding reduced overlay construction time, and increased overlay construction quality for Steiner algorithms. All of them take as input a complete group graph.

8.4.1 Edge-pruning algorithms

We propose 2 algorithms that take as input a complete group graph and then constructs a new group graph with a smaller edge-set that still yields low pair-wise latencies. This reduced group graph is then used as input to an overlay construction algorithm. The complete evaluation of these 2 algorithms is later in chapter 9 and 10. There, the edge-pruning algorithms are also applied to complete graphs, and the reduced graphs are used as input graphs to spanning-tree and Steiner-tree algorithms. However, later in this chapter, we also provide examples on how a reduced edge-set influences the execution times of a few selected spanning-tree algorithms.

The goal of the following algorithms is to create a new reduced graph with a smaller edge-set, where a set of core nodes is connected to the remaining nodes. The core nodes are identified using a core-node selection algorithm and may consist of member-nodes and/or Steiner-points, depending on if the overlay construction algorithm is a spanning-tree or a Steiner-tree algorithm.

***add Core Links**(k,O)* (aCL) takes as input a complete graph G and a core-node set $O \subset V$ that was identified by a core-node selection algorithm. First, each non-core-node in V (not in O) includes its k minimum-cost edges to non core-nodes, to the subgraph M. Then, each core-node in O includes edges to every other node in V into M (including core-nodes). The well-placed core-nodes are connected to the remaining nodes, to ensure low pair-wise latencies in M. aCL produces a graph with $|E| = k * |V - O| + |O| * |V|$. After applying aCL, the reduced graph M forms, conceptually, a two-layer graph, where the core nodes are fully meshed and the remaining nodes have a degree that is limited by $k + |O|$. Algorithm 11 illustrates the aCL

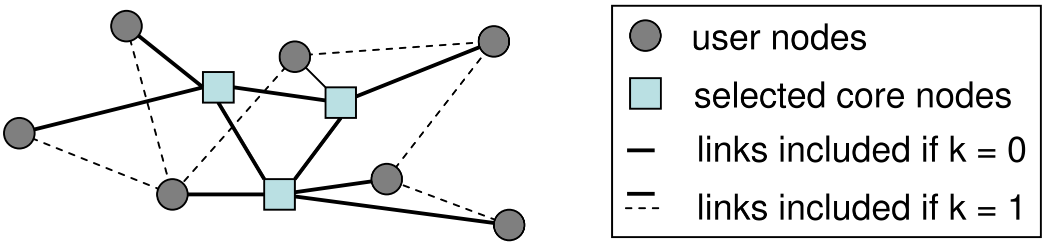


Figure 8.1: Pruned graph using *add* Core Links Optimized.

algorithm when equation 8.1 (section 8.5.4) is used to determine the number of core-nodes, and k-Median identifies these core-nodes among the member-nodes. The algorithm has a time complexity of $O(n^2)$.

Algorithm 11 *add*-CORE-LINKS

```
In: A complete graph $G = (V, E, c)$, and an integer $k \geq 0$.
Out: A connected subgraph $M = (V, E_M)$, where $E_M \subset E$.
$O = k$-Median$(G, l)$
    {find $l$ core-nodes among $V$, $l$ is obtained from equation 8.1}
For each node $m \in (V \setminus O)$, include its $k$ minimum-cost edges to $E_M \subset E$.
For each core node $o \in O$, include an edge to every node $v \in V$.
```

add-Core-Links-Optimized(k, O) (*a*CLO) takes as input a complete graph G, an integer $k \geq 0$, and a set $O \subset V$, which may be identified by a multiple core-node selection algorithm. In *a*CLO, each non-core-node (not in O) includes its k minimum-cost edges to the mesh. Then, *a*CLO builds a full mesh of the nodes in O, and includes to the mesh. Further, *a*CLO lets each core-node o add a number $s = |V - O|/|O|$ of (disjoint) edges to the non-core-nodes. After these steps, the constructed mesh forms, conceptually, a two-layer graph. Figure 8.1 illustrates a mesh generated by *a*CLO. Algorithm 12 illustrates the *a*CLO algorithm when equation 8.1 (section 8.5.4) is used to determine the number of core-nodes, and k-Median identifies these core-nodes among the member-nodes. The algorithm has a time complexity of $O(n^2)$ (algorithm 12).

Algorithm 12 *add*-CORE-LINKS-OPTIMIZED

```
In: A complete graph $G = (V, E, c)$, and an integer $k \geq 0$.
Out: A connected subgraph $M = (V, E_M)$, where $E_M \subset E$.
$O = k$-Median$(G, l)$
    {find $l$ core-nodes among $V$, $l$ is obtained from equation 8.1 in section 8.5.4}
For each node $m \in (V \setminus O)$, include its $k$ minimum-cost edges to $E_M \subset E$.
For each core node $o \in O$, include an edge to every other node $v \in O$.
For each core-node $o \in O$ disjointly connect to $l = |V \setminus O|/|O|$ nodes $v \in (V \setminus O)$ through minimum-cost
    edges.
```

8.4.2 Steiner-point insertion algorithm

The goal of the following algorithm is simply to add a number of Steiner points to a complete group graph, and return a complete graph with added Steiner-points.

add-SteinerPoints(**k, S**) takes as input a global graph $G = (V, E, c)$, a complete group graph $G_g = (V_g, E)$, an integer $k \geq 1$, and a set $S \subset V$ of super-nodes that have been previously identified to be well-placed and made available as Steiner points. First, k Steiner-points are selected from S. The Steiner points are then connected to the group graph, such that it is a complete graph of member-nodes and Steiner-points.

Algorithm 13 shows an add-SteinerPoints algorithm in which k Steiner-points are included to the complete group graph. k-Median identifies the Steiner-points among the set of super-nodes. The algorithm has a time complexity of $O(n^2)$.

Algorithm 13 *add*-STEINER-POINTS

```
In: A global complete graph $G = (V, E, c)$, a complete group graph $G_g = (V_g, E)$, a set $S$ of super-nodes, an integer
    $k \geq 1$, and degree bounds $deg(v) \in \mathbb{N}$ for each $v \in V$
Out: An updated complete group graph $G_g = (V', E')$, where there is a set $O \subset V'$ of Steiner-points.
 1: $O = k$-Median($V$, $S$, $k$)
    {finds $k$ Steiner points among $S$}
 2: $V_g = V_g + O$
 3: For each Steiner point $o \in O$, include edges to to every other node $v \in V_g$
```

8.5 Graph manipulation experiments

Graph manipulation experiments were conducted to highlight the influence of graph manipulation on overlay construction algorithms.

8.5.1 Group communication simulator

We have implemented the core-node selection algorithms, graph manipulation algorithms and the spanning-tree and Steiner-tree algorithms in a simulator that mimics group communication in a distributed interactive application. A preselected central entity is used to execute the group management. In the experiments, we assume that some latency estimation technique from chapter 6 identifies a complete graph (full mesh) graph where all edges have an associated latency.

The network was generated by BRITE [91] topology generator that generate Internet-like router networks. We simulated an application layer overlay network, therefore the network graph was translated into an undirected complete (fully-meshed) shortest-path graph, where each router had one client associated to it. Furthermore, the central entity dynamically divides the users into groups such that each group has a complete group graph. We here present results

Description	*Parameter*
Placement grid	100 × 100 milliseconds
Number of nodes in the network	1000
Super-nodes found by k-Center(k)	$k = 100$

Table 8.1: Experiment configuration.

from simulations using networks with 1000 nodes. The network layout is a square world with sides equal to 100 milli-seconds.

All the nodes join and leave groups throughout the simulation, causing group membership to be dynamic. When a join or leave request is received by the central entity, a graph manipulation algorithm manipulates the complete group graph. The group graph is then used as the input graph to an overlay construction algorithm that builds an overlay network for event distribution. The group popularity is distributed according to a Zipf distribution [20].

For the Steiner-tree simulations, a core-node selection algorithm identifies 100 well-placed super-nodes among the 1000 nodes. These super-nodes are the available Steiner-points in the simulations. Table 8.1 briefly summarizes the experiment configurations.

8.5.2 Target metrics

Overlay construction is time-dependent in distributed interactive applications, therefore, we want to identify graph manipulation algorithms that reduce the *execution time* of overlay construction algorithms.

In a time-dependent scenario it is practical to use fast overlay construction algorithms that approximate close-to-optimal solutions. However, their drawback is that they cannot guarantee a solution if the input constraints are hard. For these reasons, we also want to identify graph manipulation algorithms that increase the chance for the overlay construction algorithm to successfully complete an overlay construction. The rate of successfull completion is referred to as the algorithm's *success rate* (section 4.2.6).

In summary, the graph manipulation algorithms are evaluated towards two metrics: execution time and success rate. The execution time is the time it takes for an overlay construction algorithm to complete, and graph manipulation algorithms should reduce it. The success rate of an overlay construction algorithm, is how often it is able to complete the construction given some constraints. The constraints considered here are degree limits (dl) and a bounded diameter (bd).

8.5.3 Results from edge-pruning

First, we evaluate the possibility of reducing the *execution time* of two spanning-tree algorithms. The first spanning-tree algorithm is the $O(V * E)$ degree-limited shortest path tree (dl-SPT). dl-

SPT is a heuristic of the degree-limited shortest-path spanning-tree problem (definition 40). The second algorithm is the $O(V^3)$ compact tree (CT) spannning tree algorithm, which is a heuristic of the minimum diameter degree limited spanning-tree problem (definition 34). The next execution times are from running the two spanning-tree algorithms on a complete group graph, and then a few reduced group graphs that are produced by aCL and aCLO. All of these algorithms are thoroughly evaluated for tree related metrics in section 9.

Figure 8.2(a) plots different overlay construction times for the spanning-tree algorithm dl-SPT. dl-SPT is given an input graph that has varying number of edges, and we observe that the execution time is influenced by it. The reason is that dl-SPT has a time complexity, $O(V * E)$, which is dependent on the number of edges in the input graph. The reduction in execution time is 80 % when the edge-set is reduced by 60 %.

Figure 8.2(b) plots different overlay construction times for the spanning-tree algorithm CT. CT has a time complexity of $O(V^3)$, and we observe that its execution time is not very influenced by pruning the edge-set. The reason is that the CT algorithm contains routines that is more influenced by the vertex-set size, than the edge-set size.

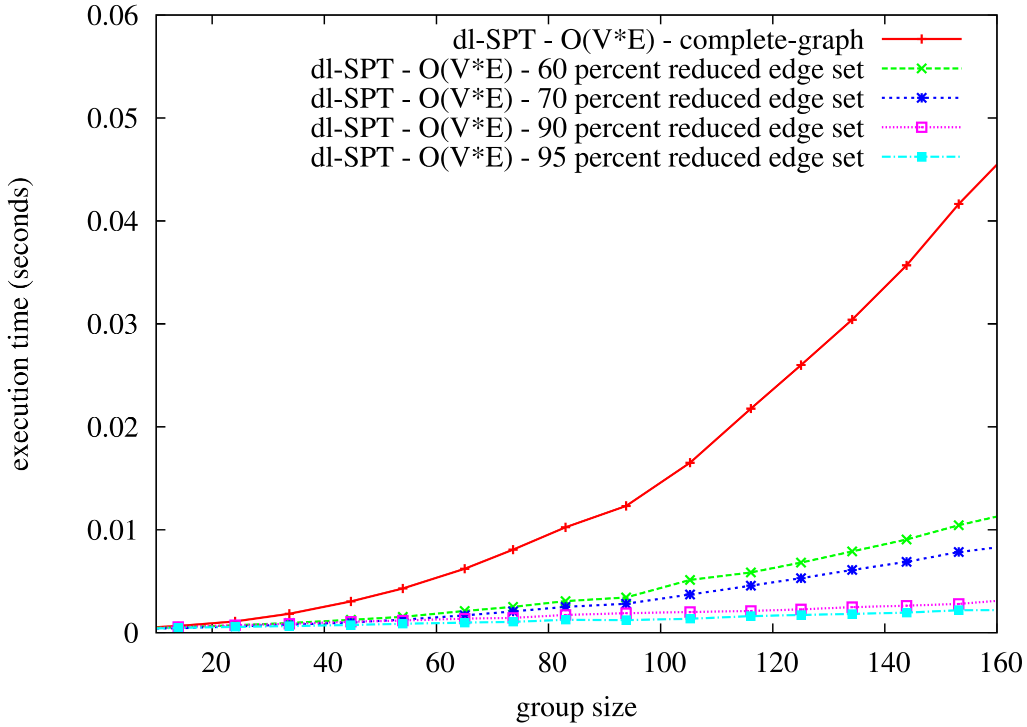
From these observations, we conclude that edge-pruning complete group graphs should, in general, only be applied when the overlay algorithm has a time complexity, which is dependent on the edge-set size. Edge-pruning should not be applied to an algorithm that has a complexity who is dependent on the vertex-set size. Generally, when edge-pruning algorithms are used to manipulate input graphs, the quality of the trees from the tree algorithms suffer somewhat. For example, it typically results in a slightly increased tree-diameter. Chapter 9 further investigates the influence of edge-pruning on the quality of a constructed spanning-tree. In the investigations the subgraph construction algorithms aCL and aCLO are used to reduce a complete group graph, and tree construction algorithms use these as input graphs.

8.5.4 Results from super-node and Steiner-point selections

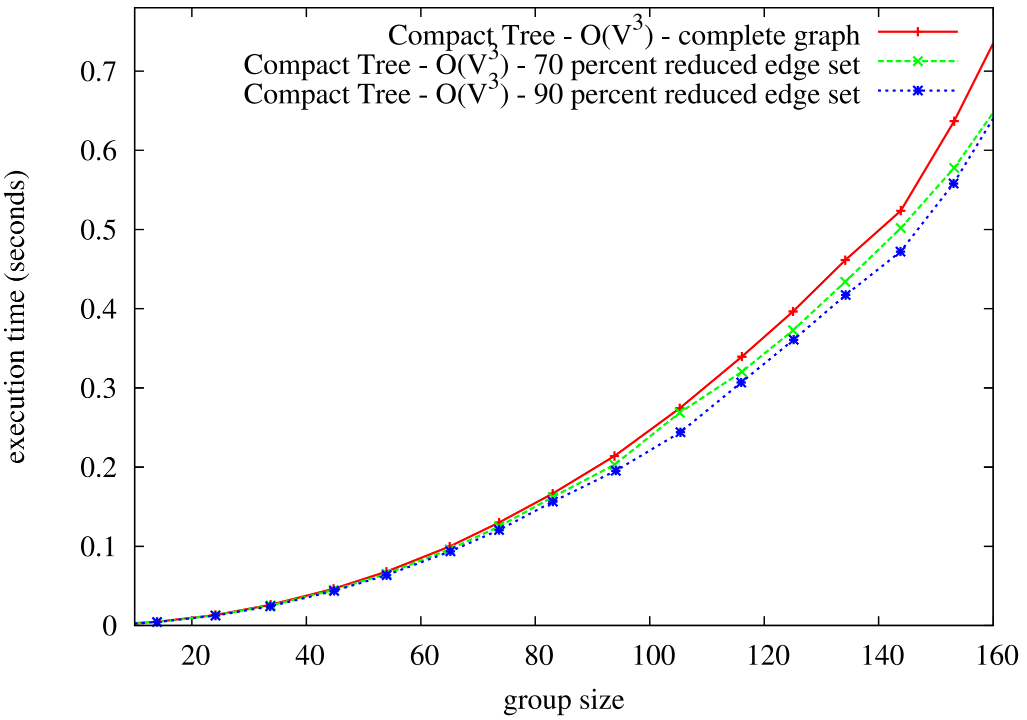
The following results highlight the importance of identifying well-placed super-nodes and adding them as Steiner points to a complete group-graph of member-nodes. We evaluate this by showing the effect the Steiner points have on the overlay construction from the Steiner-tree algorithms sdl-OTTC and smddl-OTTC.

sdl-OTTC is the Steiner degree-limited one-time tree-construction algorithm, which is a Steiner-tree heuristic of the bounded diameter degree limited Steiner minimum-cost tree problem (definition 49). smddl-OTTC is the Steiner minimum-diameter degree-limited one-time tree-construction algorithm, which is a Steiner-tree heuristic of the Steiner minimum diameter degree limited Steiner-tree problem (definition 48).

All of the core-node selection algorithms that are used in the evaluations are thoroughly introduced and evaluated in chapter 7. That investigation revealed several core-node selection



(a) Execution time (seconds) of O(V*E) dl-SPT, is dependent on the size of the edge-set.



(b) Execution time (seconds) of O(V^3) CT, is dependent on the size of the vertex-set.

Figure 8.2: Execution times (ms) of tree construction from the O(V*E) dl-SPT, and (V^3) CT, given an input graph with different sizes of the edge-set.

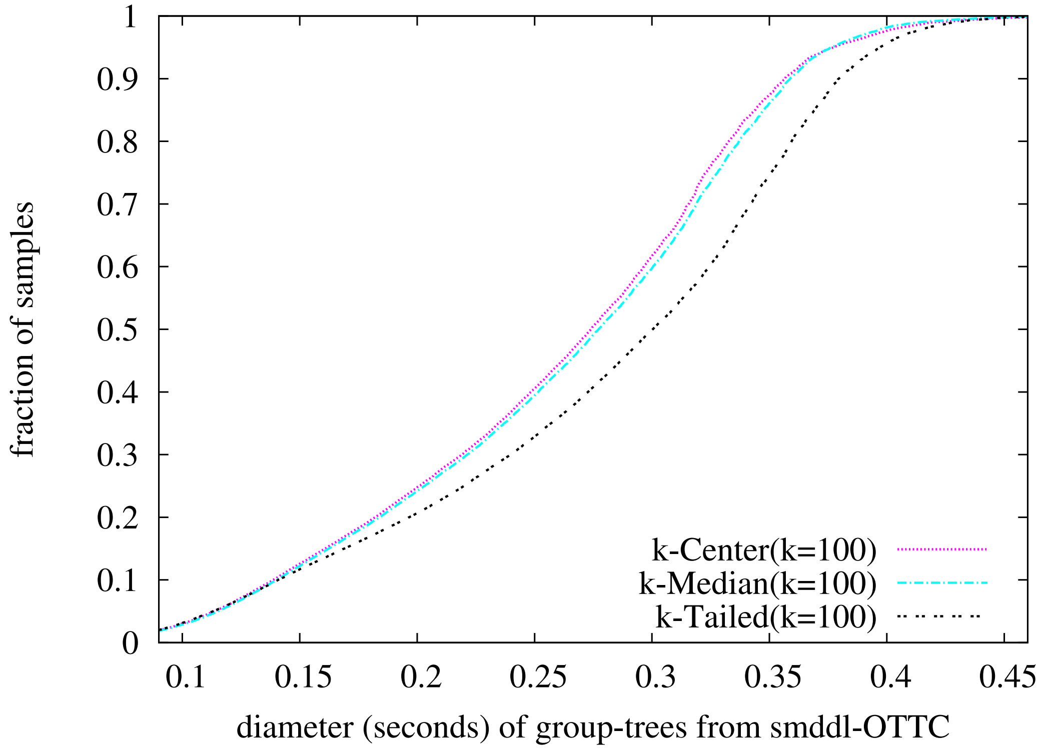


Figure 8.3: Diameter (seconds) of group-trees (CDF)

algorithms to be suitable for finding core-nodes in the application network that yield low pair-wise latencies to groups of clients.

Selecting well-placed super-nodes

Figure 8.3 and 8.4 are cumulative distribution function (CDF) plots of the achieved tree diameters (seconds) as produced by smddl-OTTC when given complete group graphs with Steiner-points.

In figure 8.3, the 100 *super-nodes* are identified by different core-node selection algorithms, and k-Median is used to identify Steiner-points to include to the complete group graph. We observe that when k-Tailed is used to find the super-nodes, smddl-OTTC builds group-trees with a significantly higher diameter than if the super-nodes are found by k-Median or k-Center. smddl-OTTC achieves the lowest group-tree diameter when k-Center is used, but for k-Median it is only slightly higher. The main reasons for this is that k-Tailed chooses super-nodes based on their euclidian coordinates (for example, (x,y) coordinates) and does not take into account the actual latencies between the nodes. k-Median chooses the k super-nodes with the lowest average pair-wise latencies, while k-Center chooses the k super-nodes with the lowest eccentricities.

In figure 8.4 k-Center is used to find the super-nodes ($k = 50, 100, 150, 200, 300$), and k-Median is still used to identify Steiner-points to include to the complete group graph. We

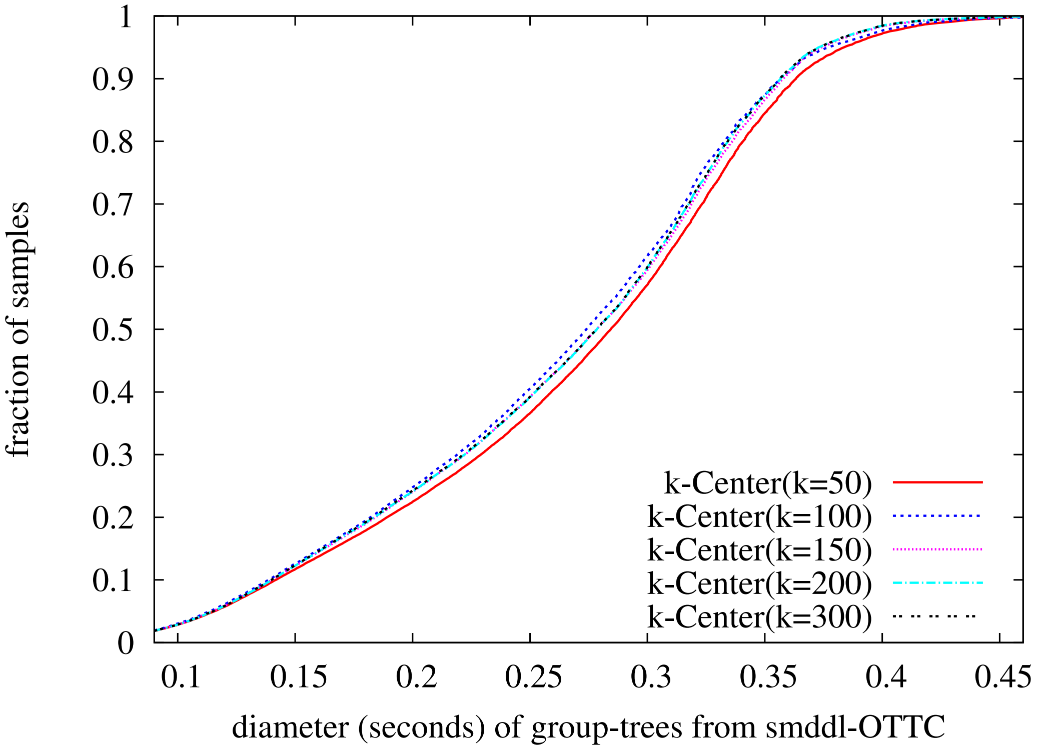


Figure 8.4: Diameter (seconds) of group-trees (CDF)

observe that there is little to gain in the group-tree diameters when there are more than 100 super-nodes in the network.

From these results, we conclude that *k*-Center is the better core-node selection algorithm to find super-nodes in the application network, and make them available as Steiner-points to Steiner-tree algorithms. Therefore, we continue to use *k*-Center to identify super-nodes in the following. We also deduce that 100 super-nodes is a fitting number, and use this number throughout the thesis.

Selecting well-placed Steiner points among the super-nodes

The next results are from running sdl-OTTC on a complete group graph with Steiner-points. The Steiner points are selected by the *k*-Median algorithm from a super-node set of 100 nodes, that are identified by *k*-Center.

Figure 8.5 shows the success rates for the Steiner-tree heuristic sdl-OTTC. sdl-OTTC tries to construct Steiner-trees subject to degree limits (dl) and a bounded diameter (bd); however, sdl-OTTC is a heuristic and cannot guarantee that it finds a solution. A solution is found if sdl-OTTC builds a tree in which each member-node is reachable from all other member-nodes. The figure plots the success rates of sdl-OTTC when it is subject to a fairly loose degree-limit of 10 and a bounded diameter limit of 0.5 seconds. We observe that the success rate of sdl-OTTC gets

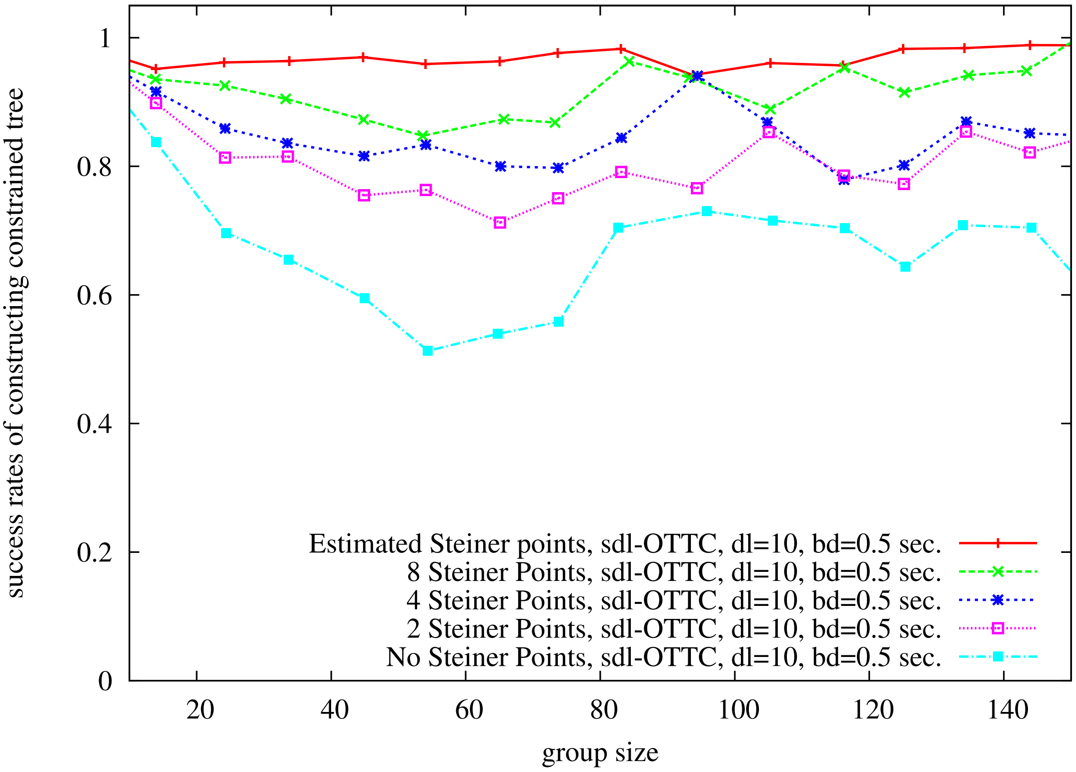


Figure 8.5: Success rates of diameter bounded and degree-limited heuristics. sdl-OTTC is given a complete group graph with Steiner-points identified by k-Median, dl-OTTC is given a complete group graph.

increasingly higher the more Steiner points are added to the input graph. When no Steiner-points are included, the sdl-OTTC reduces to a spanning-tree heuristic, and the success rate drops to around 60 %. 2 Steiner points around 80 %, 4 Steiner points between 80-85 %, and 8 Steiner points varies between 85-90 %. These fixed numbers of Steiner points increase the success rate, but the added number of Steiner points should be made dynamic. More specifically, we want it to vary depending on the current number of group members and the current degree-limitations. Therefore, we propose the following function to find a "good" number of Steiner-points to include to a complete group graph $G = (V, E, c)$:

$$f(x) = \frac{|V|}{\left(\frac{\sum_{v \in V} d(v)}{|V|}\right)} \tag{8.1}$$

$d(v)$ is the degree limit for a vertex $v \in V$, and $f(x)$ is the number of Steiner-points to be included to graph G. From figure 8.5 we see results from using equation 8.1 to find the number of Steiner points to include. The results show a very consistent success rate of around 98 %. Equation 8.1 is further used in section 10.3 and 11.3 to find the number of Steiner-points to be included to group-graphs such that a Steiner tree or subgraph algorithm increases its success

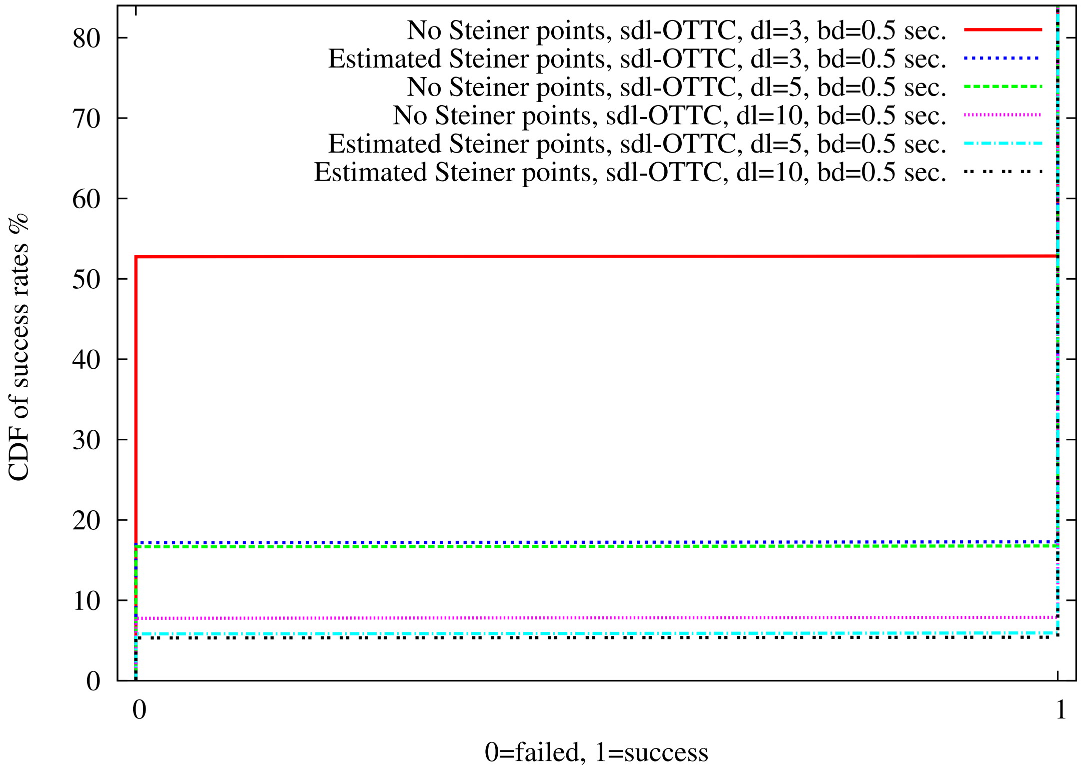


Figure 8.6: Success rates of diameter bounded and degree-limited heuristics. sdl-OTTC is given a complete group graph with Steiner-points identified by k-Median, dl-OTTC is given a complete group graph.

rate. The same equation is also used to find the number of core-nodes to be included to aCL and aCLO subgraph construction algorithms (section 8.4), such that a tree-construction algorithm increases its success rate.

Figure 8.6 plots the CDF of the success rate for the tree-construction algorithm sdl-OTTC, given varying degree-limits and a bounded diameter of 0.5 seconds. The sdl-OTTC algorithm constructs the tree on a complete group graph enhanced with a number s of Steiner points, found with equation 8.1, or no Steiner points. For a degree limit of 3, the success rate of sdl-OTTC given no Steiner points is only 47 %, while it is 82 % with Steiner points. A degree limit of 5 yields 82 % success rate for no Steiner point, and 95 % with Steiner points. A degree limit of 10 only shows a minor difference. These results show that low degree limits and a feasible diameter bound, increases the importance of including well-placed Steiner points to the input graph.

The diameter bound must be feasible, otherwise an overlay construction algorithm cannot find a solution. Figure 8.7 plots the average success rates of sdl-OTTC for group tree construction given varying degree limits and diameter bounds. We observe that a strict diameter bound of 0.25 seconds drops the success rate for sdl-OTTC pretty much to nil, even though the degree limit is high (10). A diameter bound of 0.75 seconds yields continuous overlay construction

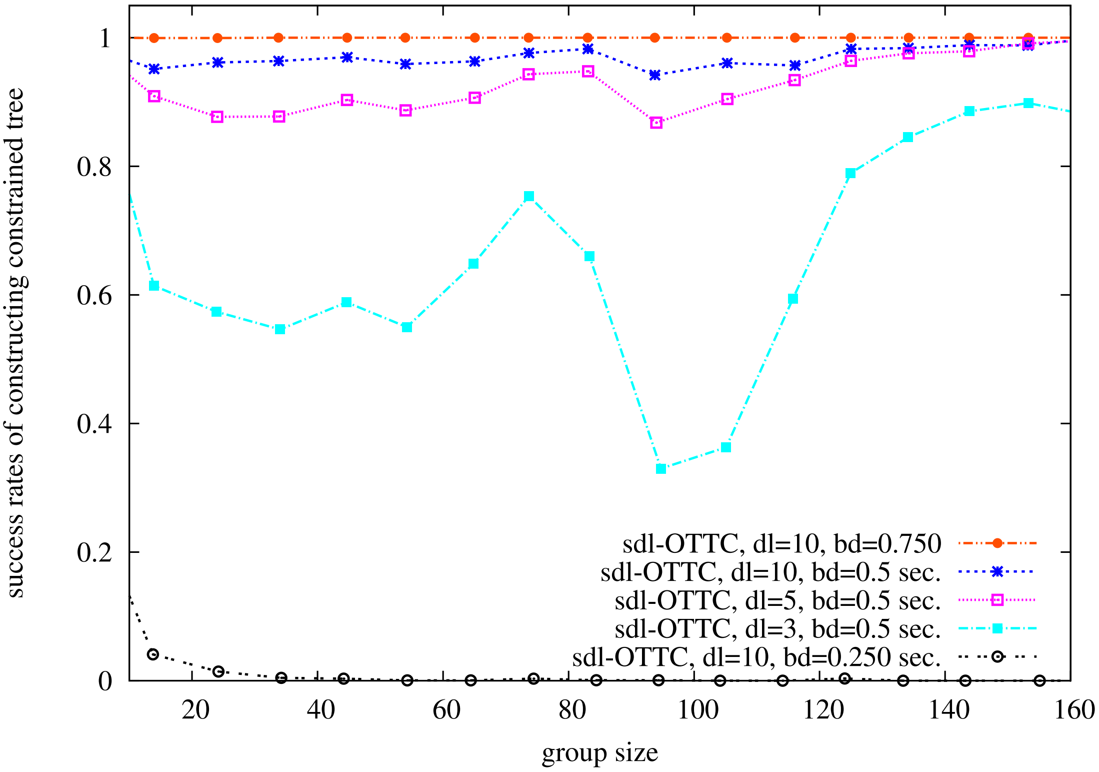


Figure 8.7: Success rates of diameter bounded and degree-limited heuristics. sdl-OTTC is given a complete group graph with Steiner-points identified by k-Median, dl-OTTC is given a complete group graph.

success for sdl-OTTC.

The reason for why the success rates vary in the group size range $(10-160)$ is that in the simulations, nodes are changing groups all the time. Therefore, there is a great chance that some of the group tree-diameters suffer because the member-nodes are very spread out. This makes it hard for an algorithm to find a valid tree, and the success rate drops.

In summary, we deduce that including well-placed Steiner points to a complete group graph, increases the success rate of degree limited diameter bounded Steiner tree heuristics. However, it also suggests that degree limited Steiner tree heuristics aiming for a minimum diameter are able to construct Steiner-trees of a smaller diameter because of the added Steiner points. Chapter 10.3 further investigates and confirms these observations in terms of several tree-related metrics.

8.5.5 Steiner-points and group graph churn

In our group communication scenario the group membership is dynamic; there are nodes that join and leave groups continuously. An important question then relates to how a complete group graph is reconfigured due to membership churn:

If Steiner-points were found and included to a group's overlay network, should these Steiner-points be kept in the complete group graph across reconfigurations?

When Steiner-points are kept across reconfigurations, the following happens. Equation 8.1 is still used to return the number s of Steiner-points to add to the complete group graph. The Steiner-points already present in the group's overlay network are substracted from s, and only the remainder is used to find new Steiner points. The following investigates how keeping Steiner-points across reconfigurations affects the tree that smddl-OTTC builds.

Figure 8.8(a) is a CDF of the achieved group-tree diameter (seconds). It compares keeping Steiner-points across reconfigurations, and selecting all new Steiner-points for each reconfiguration, using different core-node selection algorithms to find the super-nodes. We see that there is no real difference in the group-tree diameters between keeping Steiner-points and finding all new Steiner-points.

Another consideration is the *stability* of the overlay networks when there is client churn. This may be measured by the number of edges that change in the overlay networks across reconfigurations. Figure 8.8(b) is a CDF of the number of edge-changes that occur across reconfigurations. We see that keeping Steiner-points across reconfigurations has a positive effect on the stability, and reduces the number of edge-changes.

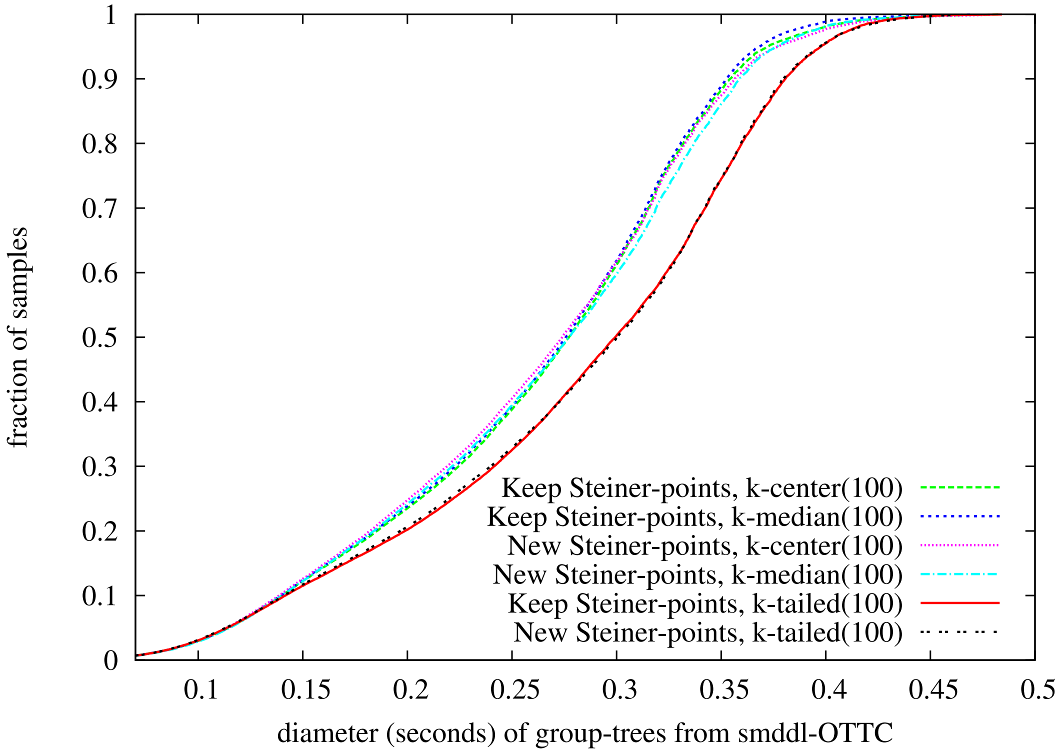
From these results we decide to keep Steiner-points across reconfigurations because it has no noticable negative effect, and increases the stability of the overlay networks when there is client churn.

8.5.6 Influence of source selection on the tree diameter

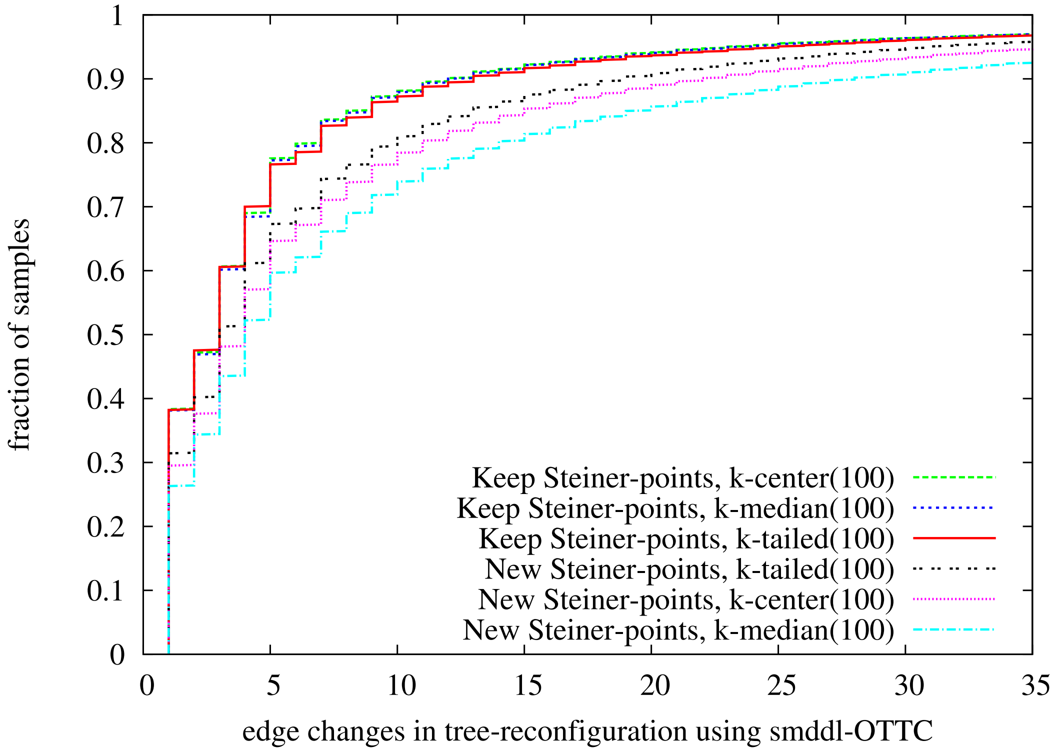
Many overlay construction algorithms start building overlays from a given source, which is oftencase input to the algorithm. Choosing a source that enables the tree algorithm to construct low-latency overlays is important. In this respect, figure 8.9 plots the diameter (seconds) achieved by a few selected tree algorithms.

The degree-limited minimum-cost spanning-tree (dl-MST) algorithm is a $O(V * E)$ heuristic of the degree-limited minimum-cost spanning tree problem (definition 31). The minimum-diameter degree-limited one-time tree-construction (mddl-OTTC) algorithm is a $O(n^3)$ heuristic of the minimum-diameter degree-limited spanning-tree problem (definition 34). And, as mentioned previously, the dl-SPT is a heuristic of degree-limited shortest-path tree problem (definition 40). These are all thoroughly introduced in chapter 9.

We observe from the plot that both dl-SPT and mddl-OTTC produce group-trees with a much lower diameter when the source is selected using k-Median($k = 1$) from the input graph, compared to the source with the worst average pair-wise distance. Between the two, it is the tree-diameter from dl-SPT that increases the most when the source is not well-placed. This is mainly because dl-SPT is a source-specific tree algorithm and aims to build trees that minimize



(a) Diameter (seconds) of group-trees (CDF).



(b) Edge changes in reconfiguration of group-trees (CDF).

Figure 8.8: CDFs of the effect of keeping Steiner-points across tree-reconfigurations that occur when the group membership changes.

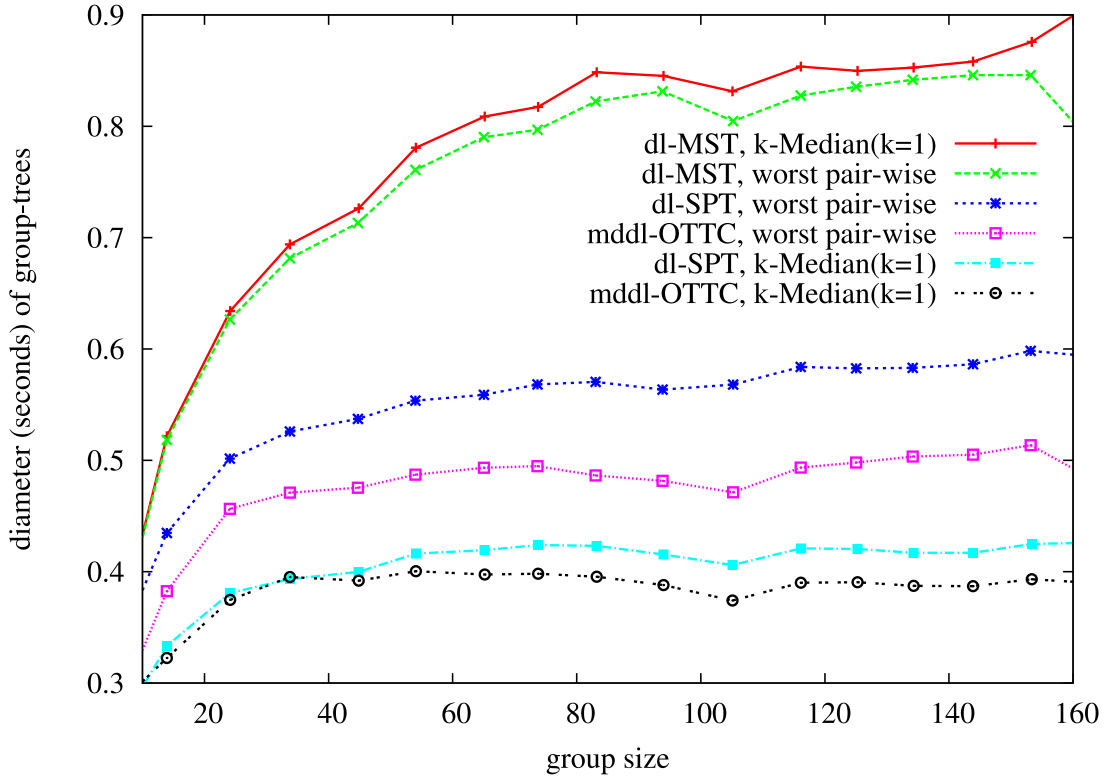


Figure 8.9: Influence of source selection on the tree diameter, for a few selected tree algorithms.

the distance to the given source. mddl-OTTC, on the other hand, is a minimum-diameter degree-limited heuristic (definition 34), and aims to reduce the tree-diameter. However, we do see that mddl-OTTC also produces trees with a significantly higher diameter when the source is not well-placed. dl-MST is a minimum-cost algorithm, and does not care about the tree-diameter, and therefore constructs similar trees regardless of the source.

We deduce that non-optimal diameter-reducing tree-heuristics that start constructing from one source, construct trees of a lower diameter when the given source is well-placed. Therefore, we continue to choose the source with k-Median($k = 1$) from the input graph, throughout the thesis.

8.6 Graph manipulation process

The graph manipulation algorithms are themselves algorithms that take time to execute. It is obvious that to speed up the overlay construction process, the graph manipulation algorithms and the construction algorithms should not be run sequentially. The graph manipulation should be performed in a seperate process that is run in parallel with the overlay construction. The graph manipulation algorithms should then continuously operate on the complete group graphs, and the overlay construction algorithm uses the latest available group graph from the graph manipulation process. The only requirement for the graph manipulation process, is that the group graph includes all the current members of the group.

8.7 Summary of the main points

The graph manipulation techniques take part in the overlay network management that was introduced in section 5.4. The main task of the graph manipulation algorithms is to speed up the overlay construction and increase the quality of the constructed overlay.

The evaluations of the graph manipulation algorithms highlighted the influence of edge-pruning. Edge-pruning significantly reduced the execution time of the tree-heuristic dl-SPT, because its time complexity $O(V * E)$ is dependent on the edge-set size (figure 8.2(a)). The tree-heuristic CT did not reduce its execution time, mainly, because its time complexity $O(V^3)$ is more dependent on vertex-set size (figure 8.2(b)). The evaluations also showed the major influence that well-placed Steiner points have when they are included to complete group graphs. The degree limited and diameter bounded Steiner tree heuristic sdl-OTTTC increased its success rate quite significantly when Steiner points were included. Equation 8.1 was used to find a "good" amount of Steiner points to include to group graphs, based on the number of group members and the current degree limitations. The results showed that the equation increased the success rate substantially (figure 8.5), especially when the degree-limits and diameter bounds were made stricter.

The subgraph construction algorithms aCL and aCLO combine the advantages of edge-pruning and vertex search. Figure 8.10 is a preview of aCL and aCLO's influence on the tree-diameter as constructed by dl-SPT. We observe that the tree-diameter does not increase by more then 10 milliseconds, while the execution time of dl-SPT (figure 8.2) is down by more than 80 $\%$. A thorough evaluation of aCL and aCLO, and their influence on overlay constrution is given in section 9.3 and 10.3. In addition, the subgraph construction algorithm aCLO is considered as an alternative for constructing cyclic subgraphs for multicasting events. Evaluations are given in section 11.2 and 11.3.

In summary, the main results we take away from this chapter and that we are going to use in later chapters are:

- **Super-nodes:** We deduced that it is advantageous to search for super-nodes and make them available as Steiner-points for Steiner-tree and -subgraph algorithms. Furthermore, we found that 100 super-nodes among 1000 nodes was a fitting number using a 100 × 100 milliseconds BRITE graph layout.

- **Group-graph churn:** We found that it is advantageous to keep Steiner-points across overlay network reconfigurations when there are membership changes, because it has no

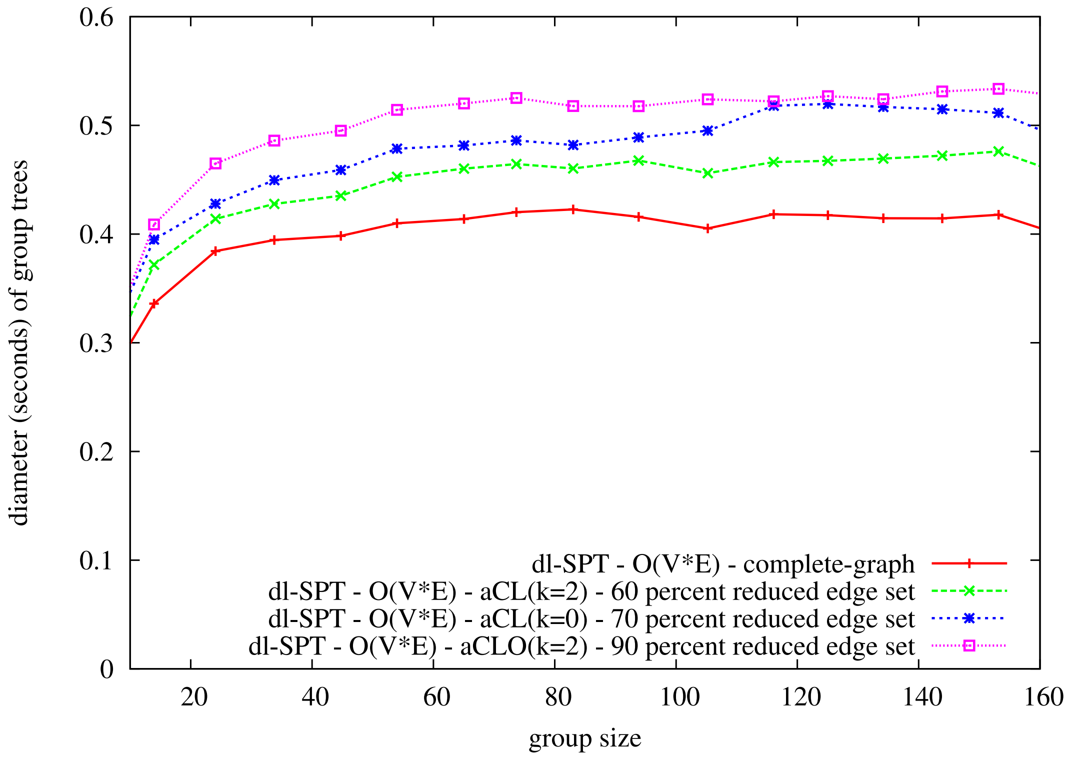


Figure 8.10: Preview of section 9.3: Tree diameter (seconds) of trees constructed by dl-SPT, using different input graphs.

noticable negative effect. It does have a positive effect on the stability of the overlay networks, where the stability is measured in terms of the number of edges that change in each reconfiguration.

- **Source-node for overlay network construction:** We found that for non-optimal diameter-reducing tree-heuristics that start constructing from one source, it is advantageous to construct trees from a source with low pair-wise latencies. These tree-heuristics construct trees of a lower diameter when the given source is found using k-Median($k = 1$).

- **Overlay construction time:** We found that edge-pruning algorithms generally should be applied to algorithms that have time complexities dependent on the edge-set. In such cases, the edge-pruning did reduce the overlay construction time. However, further studies are needed to see the effect of edge-pruning on the pair-wise latencies. Such studies are performed in the following chapter, where we observe this and other effects on various spanning-tree algorithms.

Chapter 9

Overlay construction techniques: Spanning-tree algorithms

The overlay network management introduced in section 5.4 includes overlay construction techniques whose task is to construct low-latency overlay networks for distribution of time-dependent events. In that respect, we are evaluating a class of overlay construction techniques that are called spanning-tree algorithms.

Spanning-tree algorithms build a spanning-tree on an input graph, where a spanning-tree is a connected acyclic subgraph (tree) that connects all the vertices [144]. A spanning-tree may be useful in many situations, for example, to connect a set of terminals in a cheap and efficient manner. The terminals may be just about anything: computers, phones, cities, train-stations, factories, etc. In this chapter, we evaluate a number of spanning-tree algorithms in terms of their applicability to distributed interactive applications. By doing this we address a goal of the thesis, which was to:

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

The investigation includes many spanning-tree algorithms with the particular focus on reducing the diameter of a spanning-tree, that is, the maximum pair-wise latency between users. In addition, we focus on reducing the time it takes to execute membership changes. In that context, we use the core-node selection algorithm k-Median to find well-placed group nodes, and the edge-pruning algorithms aCL and aCLO to reduce the number of edges in an otherwise fully meshed overlay. Our edge-pruning algorithms strongly connect well-placed group nodes to the remaining group members, to create new and pruned group graphs, such that, when a tree algorithm is applied to a pruned group graph, it is manipulated into creating trees with a small diameter.

We implemented and analyzed a wide range of spanning-tree algorithms, core-node selection algorithms and edge-pruning algorithms. The evaluations were conducted using a group communication simulator we implemented (section 2.6.4). In our investigation, we were able to identify 4 spanning-tree algorithms that meet our goals of constructing low-latency overlays sufficiently fast. One general observation was that faster heuristics that do not explicitly optimize the diameter are able to compete with slower heuristics that do optimize it.

The rest of the chapter is organized in the following manner. Section 9.1 introduces some basic spanning-tree algorithm types. Section 9.2 introduces the evaluated and proposed spanning-tree algorithms. Section 9.3 provides group communication simulation results and evaluates the spanning-tree algorithms. Finally, a summary of the main points is given in section 9.4.

9.1 Spanning-tree algorithm types

Chapter 4.5 introduced a wide range of spanning-tree problems, related to total tree-cost, diameter and radius. These graph theoretical problems are addressed by spanning-tree algorithms that try to solve them. Many spanning-tree algorithms have been developed over the years, and the following introduces some of the most basic spanning-tree algorithms.

9.1.1 Minimum-cost spanning-tree algorithms

The minimum-cost spanning-tree on a graph is the acyclic spanning-graph that has least cost (definition 30). An MST algorithm constructs a tree of minimum total cost, where the total cost is the sum of all the link weights in the tree. MST algorithms were first proposed semi-independently by Boruvka, Jarnik and Kruskal [92], then Jarnik's algorithm was rediscovered by Prim and Dijkstra [58], and Boruvka's algorithm rediscovered by Sollin. Essentially, there are three MST algorithms that we shall refer to as Boruvka's MST, Prim's (Jarnik's) MST and Kruskal's MST. The MST algorithms are all considered to have a time complexity of $O(E \log V)$ ($G = (E, V)$). Boruvka's MST algorithm was discovered as early as 1926, and is considered to be the first MST algorithm. It starts by electing leaders among the components in a graph, each component then adds a safe-edge (lowest-cost edge) to the MST. The leaders are found using a depth-first search (chapter 4). Components are merged until there is only one left; the MST. Prim's MST builds the tree starting from a given source, and for each iteration, it connects a vertex through the minimum cost path to the tree, until all nodes are spanned. Kruskal's MST algorithm starts with a forest of trees that are merged through minimum-cost edges until there is only one left. MSTs have many applications related to computer networks, for example, minimizing the consumed bandwidth in a given network.

9.1.2 Shortest path spanning-tree algorithms

The shortest path spanning-tree on a graph is the acyclic graph that have shortest paths from a given source to all target nodes (definition 37). A *shortest path tree* (SPT) algorithm constructs a tree that has p shortest paths from a given source node to the destinations, where p is the number of destinations. The most famous SPT algorithm is the $O(E + V \log V)$ Dijkstra's SPT algorithm [58]. Dijkstra's SPT builds an SPT from a given source, and adds the next vertex through the edge that results in the shortest path to the source. Another SPT algorithm is Shimbel's SPT algorithm [136], most commonly referred to as the Bellman-Ford's SPT algorithm [136]. It has a worst case running time of $O(V \times E)$, and has the advantage (opposed to Dijkstra's SPT) that it can discover negative cycles in graphs. SPT algorithms are often employed in the Internet to find the shortest-paths between communicating computers.

9.1.3 Minimum-diameter spanning-tree algorithms

Another spanning-tree variant is a minimum-diameter spanning-tree. A minimum-diameter spanning-tree of a graph is the tree that has the shortest diameter of all possible spanning-trees, where the diameter is the longest shortest-path (definition 32). Ho, Lee, Chang and Wong [68] proved that an optimal minimum diameter spanning-tree has one or two nodes that are connected through shortest paths to the remaining nodes. For a complete graph (definition 23), the resulting tree is actually a star topology.

A recent algorithm that solves the minimum diameter spanning-tree problem is the $O(n^3)$ algorithm [22] proposed by Bui et al. Their algorithm is a distributed implementation, but it is easily transformed to a centralized algorithm. The algorithm finds the absolute 1-center of a graph (definition 12), and builds a shortest-path tree from that point. Minimum-diameter spanning-trees are applicable to event-distribution for time-sensitive distributed interactive applications.

9.1.4 Constrained spanning-tree algorithms

Many spanning-tree algorithms add constraints to the tree construction [144], where the most common constraints are variations of delay bounds and degree limitations [80, 64]. The delay bounds are typically added in algorithms that optimize for the total cost [110,78,77]. The delay bounds may be from a given source, in which case, the tree is considered a source-tree [80, 64]. The delay bound may also express the maximum allowed diameter in the tree [116]. Degree limitations are added to bound the (forwarding) stress on each node in the tree [95], where the stress is mainly linked to bandwidth consumption.

The constraints are important to control and tune the tree construction. For example, a delay bound ensures that the latencies are controlled while the total cost is minimized, and degree limitations ensure a controlled stress level on each node. However, generally the constraints make the tree construction harder to solve, such that the tree construction takes longer. Many constrained tree construction problems are NP-complete [131], such that polynomial time heuristics are required to approximate solutions.

9.2 Evaluated spanning-tree algorithms

The following sections introduce the spanning-tree algorithms that are evaluated in the thesis. Among these spanning-tree algorithms there are algorithms that consider the minimum-cost, the diameter and the radius. In addition, the spanning-tree algorithms are subject to constraints, for example, degree-limits, bounded diameter, bounded radius, and bounded total cost. Some of the spanning-tree algorithms exist in the literature, however, most of the presented algorithms are variations of existing spanning-tree algorithms, that are tuned to target the timely requirements of distributed interactive applications. Table 9.1 provides an overview of all the spanning-tree algorithms in the thesis.

Formally, a spanning-tree algorithm $\mathscr{A}_{\mathscr{T}}$ takes as input a connected undirected weighted graph $G = (V, E, c)$, where V is the set of vertices, E is the set of edges, and $c : E \to R$ is the edge cost function. $n = |V|$ is the number of nodes in the graph G. The spanning-tree algorithm $\mathscr{A}_{\mathscr{T}}$ then constructs a connected acyclic graph (tree) $T = (V_T, E_T)$ on G, where $V_T = V$ (definition 29).

9.2.1 Tree heuristics considering the minimum-cost

Among the evaluated spanning-tree algorithms that consider minimum-cost, there are two minimum-cost spanning-tree (MST) algorithms. One is Prim's MST and the other is a degree-limited MST (dl-MST) algorithm based on ideas from Narula and Ho's algorithms [95]. The proposed dl-MST algorithm addresses the drawback of the Narula and Ho's dl-MST and suggests an improvement.

Minimum-spanning-tree (MST) [58] is Prim's minimum-spanning-tree algorithm. Prim's MST has been empirically shown to be the fastest MST algorithm for large dense-graphs [93]. Prim's MST builds the tree starting from a given source, and for each iteration, it connects a vertex through the minimum cost path to the tree. Prim's MST has a best-case time complexity of $O(E \log V)$, but is interchangeably also referred to as a $O(n^2)$ algorithm in the thesis.

Degree-limited minimum-spanning-tree (dl-MST) is a $O(V \times E)$ heuristic of the d-MST problem (definition 31), and is a modification of Prim's MST algorithm. dl-MST greedily adds a vertex to the tree through the minimum weight edge, while obeying the degree constraints.

Algorithm 14 presents the algorithm details of dl-MST. The data structures particular for the dl-MST implementation are:

- edges(u): contains all the out edges u has, sorted by increasing edge weight.
- undiscovered(u): is the number of undiscovered edges from u.
- outdegree$_T(u)$: is the current out-degree in the partially made tree.

The proposed dl-MST algorithm starts by sorting the out edges for each node $u \in V$ by increasing edge weight and stores the order in the edges(u) structure. Moreover, the structure undiscovered(u), is initialized to the current undiscovered neighbors of u, which is the number of edges(u). The outdegree$_T(u)$ is the current out-degree in the partially discovered tree, and is initialized to zero. dl-MST then continues like Prim's MST and inserts the given source node to a minimum-heap.

For each iteration, a tree-node u, attached to the tree through the least-cost edge, is popped from the minimum-heap and iterates through its edges(u). If u discovers a node $v \in V$ that u has a lower cost edge to, and v is currently not in the tree, then the algorithm changes parent(v) to u, but only if u has an available degree. The "old" parent(v) $p \in V$ is reinserted to the minimum-heap if p has undiscovered(p) neighbors (> 0). The reason is that p now has an available degree, and the greediness of the algorithm combined with degree-limitations makes it possible for "old" parents to still have least-cost edges to undiscovered neighbors not yet in the tree. The algorithms terminates when all neighbors of all nodes are discovered.

The complexity of dl-MST is higher than Prim's MST, because as long as a tree-node u has undiscovered edges it is re-inserted into the minimum-heap. This is an attempt to avoid that the greediness of the original Prim's MST puts the partially made tree in a bad situation. Bad situations may happen if nodes have filled their degree through relatively high-cost edges because of the greedy first-come first-connect principle of Prim's MST. Then, later on, parents change and the available degree on the "old" parent is not discovered and taken advantage of unless it is re-inserted to the minimum-heap. The Narula and Ho's [95] dl-MST suffered from this problem.

If the dl-MST algorithm finishes the main loop (line 15) without spanning all the nodes, then it is due to the degree-limitations (if the input graph is connected). In this case, the algorithm was not able to finish a tree that spans all the vertices, therefore, it is necessary to relax the degree limitations on the nodes that are adjacent to the nodes that are not spanned yet. This routine is only necessary if the input graph G is not a complete graph, because the dl-MST algorithm never fails on complete graphs. In a complete graph all nodes are directly connected to each other, hence, the leaf nodes in the tree are always connected to uncovered nodes (section 10.3).

Algorithm 14 DL-MINIMUM-SPANNING-TREE

```
In: $G = (V, E, c)$, a source node $s \in V$, degree bound $deg(v) \in \mathbb{N}$ for each $v \in V$
Out: Spanning Tree $T = (V_T, E_T)$
SortedNodeWeightMap edges
Vector undiscovered
FibonacciHeap heap
color = enum Color {WHITE, GRAY, BLACK}
for all nodes $v \in V$ do
    parent(v) = v; distance(v) = infinity; outdegree$_T$(v) = 0; color(v) = WHITE
    undiscovered(v) = sizeof(out-edges(v))
    for all out-edges(v) $e \in E$ do
        edges(v).insert(weight(e), target(e)) {sorted according to (min) weight}
    end for
end for
distance(s) = 0; color(s) = GRAY
heap.insert(s, 0)
BuildTree:
while heap is not empty do
    u = heap.getMinimum()
    for nodes v in edges(u) do
        if weight(u,v) < distance(v) and outdegree$_T$(u) < deg(u) and color(v) != BLACK then
            p = parent(v)
            outdegree$_T$(p)- -; outdegree$_T$(v) = max(0, outdegree$_T$(v)- -)
            {"old" parent p has available degree, reinsert it to heap if undiscovered neighbors}
            if undiscovered(p) > 0 then
                heap.insert(p, infinity)
            end if
            undiscovered(u)- -; undiscovered(v)- -
            outdegree$_T$(u)+ +; outdegree$_T$(v)+ +
            distance(v) = weight(u,v); parent(v) = u
            if color(v) == WHITE then
                color(v) = GRAY; heap.insert(v, distance[v])
            else if color(v) == GRAY then
                heap.decreaseKey(v, distance[v])
            end if
        else if weight(u,v) < distance(v) and color(v) != BLACK then
            undiscovered(u)- -; undiscovered(v)- -
        end if
        color(u) = BLACK
    end for
end while
{relax degree bounds if all nodes are not included}
if $V \neq V_T$ then
    for all nodes v not in $V_T$ do
        for all out-edges(v) $e \in E$ do
            if u = target(v) not yet seen then
                RelaxDegree(u)
                heap.insert(u, distance(u))
            end if
        end for
    end for
    goto BuildTree
end if
```

9.2.2 Tree heuristics considering the diameter

The diameter of a tree T is defined as the longest of the paths in T among all the pairs of nodes in V (section 4.1.1). The thesis evaluates a number of spanning-tree heuristics that consider the diameter of trees. Detailed descriptions of these spanning-tree heuristics are given in the following and in table 9.1).

One-time tree construction (OTTC) [1, 38] is a $O(n^3)$ heuristic of the bounded diameter minimum spanning-tree (BDMST) problem [131] (definition 33). The NP-complete BDMST-problem optimizes for the *total cost* while obeying an upper bound diameter constraint. OTTC is a modification of Prim's minimum spanning-tree (MST) algorithm to accommodate the diameter bound. It maintains the node eccentricities as the tree is built and (if possible) adds the minimum weight edges that result in node eccentricities below the diameter bound.

Degree-limited one-time tree construction (dl-OTTC) [131] is a $O(n^3)$ heuristic of the bounded diameter degree limited (BDDLMST) problem [68] (definition 35). The NP-complete BDDLMST-problem is identical to the BDMST-problem, except it adds degree limits on each node. The dl-OTTC heuristic is proposed by the thesis and builds the tree in the same way as OTTC, while obeying given degree limits. The dl-OTTC is described in algorithm 15, in which each node $u \in V$ maintains the following information [38]:

- near(u) is the node in the tree nearest to the non-tree node u.
- wnear(u) is the weight of edge (u, near(u)).
- dist(u, $1 \dots n$) is the distance (unweighted path length) from u to every other node in the tree if u is in the tree, and is set to -1 if u is not yet in the tree.
- ecc(u) is the eccentricity of node u, (the distance in the tree from u to the farthest node) if u is in the tree, and is set to -1 if u is not yet in the tree.

To update near(u) and wnear(u), dl-OTTC determines the edges that connect u to the partially-formed tree T without increasing the diameter (as the first criterion) and among all such edges dl-OTTC chooses the one with minimum weight. This is done efficiently, without having to recompute the tree diameter for each edge addition. Code segment 1 of the dl-OTTC algorithm, sets the dist(v) and ecc(v) values for node v by copying from its parent node near(v). Code segment 2, updates the values of dist and ecc for the parent node in n steps. Code segment 3, updates the values of dist and ecc for other nodes. Code segment 4 updates the near and wnear values for nodes not currently in the partial tree T. Code segment 5 finds the next node v to be added to the partial tree T. It includes a dynamic relaxation method that relaxes the diameter-bound and degree-limts if a next node cannot be found.

Algorithm 15 DL-ONE-TIME-TREE-CONSTRUCTION

In: $G = (V, E, c)$, diameter-bound ≥ 0, degree bound $deg(v) \in \mathbb{N}$ for each $v \in V$

Out: Spanning Tree $T = (V_T, E_T)$

```
select a root node $v_0$ to be included in $V_T$
initialize near(u) := $v_0$ and wnear(u) := weight(u,$v_0$), for every $u \in V$
compute a next-nearest-node $v \in V$ such that: wnear($v$) = MIN$_u${wnear(u)}
select the node $v$ with the smallest value of wnear($v$)
while $|E_T|$ < (n - 1) do
    set $V_T := V_T \cup \{v\}$ and $E_T := E_T \cup \{(v,$near$(v))\}$
    {1. set dist(v,u) and ecc(v)}
    for u = 1 to n do
        if dist(near(v),u) > âĹŠ1 then
            dist(v,u) := weight(v, near(v)) + dist(near(v),u)
        end if
    end for
    dist(v, v) := 0
    ecc(v) := weight(v, near(v)) + ecc(near(v))
    {2. update dist(near(v),u) and ecc(near(v))}
    dist(near(v),v) = weight(v, near(v))
    if ecc(near(v)) < 1 then
        ecc(near(v)) = weight(v, near(v))
    end if
    {3. update other nodes' values of dist and ecc}
    for each tree node u other than v or near(v) do
        dist(u,v) = weight(v, near(v)) + dist(u,near(v))
        ecc(u) = MAX{ecc(u),dist(u,v)}
    end for
    {4. update the near and wnear values for other nodes in G}
    for each node u not in the tree do
        if weight(u, near(u)) + ecc(near(u)) > diameter-bound or $deg_T(v) \geq deg(v)$ then
            for each node w in $T$ do
                if weight(u, near(u)) + ecc(near(u)) > ecc(w) + weight(u, w) and $deg_T(w) < deg(w)a$ then
                    near[u] = w, wnear[u] = weight(u,w) + ecc(w)
                end if
            end for
        else
            if weight(u,v) < weight(u, near[u]) then
                near[u] = v, wnear[u] = weight(u,v) + ecc(v)
            end if
        end if
    end for
    {5. find next node v, relax constraints if necessary}
    while not found next node $v$ do
        try to find the node $v$ with the smallest value of wnear($v$)
        if failed to find $v$ due to diameter-bound violation then
            Relax(diameter-bound)
        end if
        if failed to find $v$ due to degree bound violation then
            Relax($deg_T(v)$) for each node $v \in V_T$
        end if
    end while
end while
```

Minimum diameter one-time tree construction (md-OTTC) [131] is a $O(n^3)$ heuristic of the minimum diameter spanning-tree (MDST) problem [68] (definition 32). The MDST-problem is to find a spanning-tree of a graph such that the diameter is minimized. MDST-algorithms that operate on complete graphs often construct a star-shaped tree, where the work-load (stress) of the central nodes is high. The algorithm md-OTTC is another alteration of the BDMST-heuristic OTTC [1] proposed by the thesis. Instead of minimizing the total cost within a diameter bound, md-OTTC always adds the vertex that minimizes the diameter of the partially made tree.

Minimum diameter degree-limited one-time tree construction (mddl-OTTC) [131] is a proposed $O(n^3)$ heuristic of the minimum diameter spanning-tree (MDDL) problem [114] (definition 34). The NP-complete MDDL-problem introduces degree limits to solve the stress issues of the MDST. mddl-OTTC works as md-OTTC while obeying the degree limits. The algorithm of mddl-OTTC is identical to dl-OTTC if the algorithmic details inside the for loop on line 22 is changed to pick the edge that minimizes the diameter of the partially built tree.

Randomized greedy heuristic (RGH) [104] is a fast $O(n^2)$ heuristic of the BDMST problem (definition 33) and was originally designed to be used on complete graphs. However, with slight adjustments RGH works for general graphs as well. RGH is given as input a graph G, and starts off by choosing and adding to the spanning-tree one or two center nodes from V, depending on whether $|V|$ is even or odd. When extending the spanning-tree, it chooses the next vertex to add to the tree at random and connects it via the lowest weight edge that maintains the diameter constraint. The diameter constraint is only maintained towards the source.

The original RGH did not specify how the center nodes are chosen. If they are randomly chosen, the resulting spanning-tree will suffer from this. In our algorithm, the center nodes are chosen using a core-node selection algorithm, for example, the algorithm k-Median (chapter 7). The core-node selection algorithm ensures that center nodes are chosen that are well-placed. For example, k-Median ensures that the source is the node with the minimum average pair-wise latency (section 4.2). The added time-complexity is minor compared to the benefit.

Degree-limited randomized greedy heuristic (dl-RGH) [131] is a $O(n^2)$ heuristic of the BDDLMST problem. Algorithm 16 presents the pseudo-code of dl-RGH as implemented by the author of the thesis. The dl-RGH heuristic builds the spanning-tree much the same way as RGH, the major difference is that dl-RGH obeys given degree-limits for each node. Both for RGH and dl-RGH there is a chance that a constrained tree cannot be found. Therefore, a relaxation procedure is added that relaxes the diameter-bound (for RGH) and degree limits if necessary.

Compact-tree heuristic (CT) [115] is a $O(n^3)$ heuristic of the MDDL problem, and is based on Prim's MST algorithm. It builds a tree with close to minimum diameter while obeying the degree limits. It works similarly to OTTC, and adds the next node to the tree that results in the close-to-minimum diameter of the partially made tree. Algorithm 17 presents the pseudo-

Algorithm 16 DL-RANDOMIZED-GREEDY-HEURISTIC

```
In: $G = (V, E, c)$, diameter-bound $\geq 0$, degree bound $deg(v) \in \mathbb{N}$ for each $v \in V$
Out: Spanning Tree $T = (V_T, E_T)$
k = 1
if $|V|$ is odd then
    $k = 2$
end if
$C$ = $k$-MEDIAN-SELECTION($G$, $k$) {select core node set}
connect $C$ to tree $T$
initialize depth(u) = 0, for every $u \in V$
while $|E_T|$ < ($|V|$ - 1) do
    choose first node $v$ with edge($v$,$c$) $c \in C$, where $deg_T(c) < deg(c)$
    {success, then connect v to core node set}
    if found node $v$ then
        add edge($v$,$c$) to $T$
        depth($v$) = depth($c$) + weight(edge($v$,$c$))
        if depth($v$) $\leq$ diameter-bound/2 then
            C $\cup$ $v$
        end if
        {failed, then relax constraints}
    else
        if failed to find $v$ due to degree bound violation then
            Relax($deg_T(c)$) for each node $c \in C$
        end if
        Relax(diameter-bound)
        for nodes $v \in V_T$ not in $C$ do
            if depth($v$) $\leq$ diameter-bound/2 then
                C $\cup$ $v$
            end if
        end for
    end if
end while
```

code from the implementation of CT done by the author.It adds the functionality of relaxing the degree limits whenever the tree construction cannot continue. In the evaluations of section 9.3, the results reveal that CT is quite time-consuming in its tree construction.

Bounded compact-tree (BCT) [115] is a generalization of the CT algorithm. It is a $O(n^3)$ heuristic of the bounded diameter residual balanced tree problem (definition 36). It uses a balancing factor M, when searching for the next node to be included in the tree. It selects the node with the largest available degree among the M nodes with lowest diameter. The authors of [115] found that $M = 4$ fits best.

9.2.3 Tree heuristics considering the radius (shortest-path)

The radius of a tree T is defined to be the smallest eccentricity among the nodes $v \in V$ (section 4.1.1). A spanning-tree algorithm that considers the shortest-paths from a given source is also a spanning-tree algorithm that minimizes the radius if the source node is located close to the center of the input graph. From section 4.5 it was identified that in a complete graph,

Algorithm 17 COMPACT-TREE-HEURISTIC

```
In: $G = (V, E, c)$, degree bound $deg(v) \in \mathbb{N}$ for each $v \in V$
Out: Spanning Tree $T = (V_T, E_T)$
choose source $s$ and include to $T$
for all nodes $v \in V$ do
    if edge(v,s) exist then parent($v$) = $s$ and distance(v) = weight(s, v)
end for
let $u \in V - V_T$ be the vertex with smallest distance(u)
while $V_T \neq V$ do
    include edge(u,parent(u)) to $T$
    for $v \in V - u$ do
        distance(v) = MAX{distance(v),dist$_T$(u,v)}
    end for
    for $v \in V - V_T$ do
        distance(v) = infinity
        for $q \in V_T$ do
            if $deg_T(q) < deg(q)$ and weight(v,q) + distance(q) < distance(v) then
                distance(v) = weight(v,q) + distance(q)
                parent(v) = q
            end if
        end for
    end for
    {Find next node to add, if failed then relax degree bounds}
    FIND-NEXT:
    let $u \in V - V_T$ be the vertex with smallest distance(u)
    if distance(u) == infinity then
        Relax($deg_T(v)$) for each node $v \in V_T$
        update distance and parent for each node $v \in V_T$
        goto FIND-NEXT
    end if
end while
```

the topology of a close-to-optimal minimum-diameter spanning-tree is a star. The shortest-path spanning-tree algorithms do not explicitly consider the diameter, but are cheaper in terms of the execution time. For example, a shortest-path tree (SPT) is a source specific tree in which all nodes have a shortest-path to the source. It was solved by Dijkstra [58] and has a worst-case time complexity of $O(n^2)$.

Dijkstra's shortest path tree (SPT) algorithm constructs a tree that has p shortest paths from a given source node to the destinations (definition 37), where p is the number of destinations [58]. Dijkstra's SPT is the most famous SPT algorithm and is solved in $O(E + V \log V)$. Dijkstra's SPT builds an SPT from a given source, and adds the next vertex that has the shortest path to the source. A shortest-path tree is actually a simple MDST heuristic (definition 32) if the source vertex is centrally located in the input graph. The source node may, for example, be selected by the single core-node selection algorithm k-Median($k = 1$) (chapter 7).

Degree-limited shortest-path tree (dl-SPT) is a proposed $O(V \times E)$ heuristic of the degree-limited shortest-path tree (d-SPT) problem (definition 40). The NP-complete d-SPT problem is to find a spanning-tree from a given source such that the source destination distance is minimized. The dl-SPT algorithm is designed exactly like dl-MST (introduce previously), except, dl-SPT minimizes the path-length from a predefined source to each destination, while obeying the degree-limitations on each node. Therefore, the implementation of dl-SPT follows the pseudo-code of dl-MST, with the exception of line numbers 18 and 25, that needs to be changed to address the shortest-path length, rather than edge-cost.

Bounded radius minimum spanning-tree (br-MST) is a $O(n^2)$ heuristic of the BRMST problem (definition 39). It adds a radius bound and dynamic relaxation of the radius bound to the dl-MST algorithm, but otherwise works in the same fashion.

Bounded radius degree-limited minimum spanning-tree (brdl-MST) is a $O(n^2)$ heuristic of the BRDLMST problem (definition 42). It adds a radius bound and degree limits, and then a dynamic relaxation of the radius bound to the dl-MST algorithm, but otherwise they work in the same fashion.

9.3 Group communication simulations of spanning tree algorithms

We evaluate the previously introduced spanning-tree algorithms through group communication simulations. The simulator is thoroughly introduced in chapter 2.6.4. Table 9.1 provides an overview of all the evaluated spanning-tree algorithms.

9.3.1 Experiment configurations

In the experiments, we use different input graphs to the tree algorithms: complete graphs and reduced graphs. To reduce the complete group graphs, we use the edge-pruning algorithms aCL and aCLO that both were introduced in section 8.4. The reason we apply these edge-pruning algorithms is to reduce the execution time of the spanning-tree algorithms.

For the tests that apply aCL and aCLO , the core node set (O) is found among the group's member-nodes using the k-Median(k) core-node selection algorithm (chapter 7). The core-node set size k is found applying equation 8.1. More specifically, the size of the core node set O is found using the degree limit d in the current experiment and the current group size $|V|$: $k = (|V|/d)$. The function approximates the number of core nodes that is needed to ensure that the constrained tree algorithms are still able to build a tree (section 8.5).

Chapter 8 introduced and evaluated the problems that constrained overlay construction heuristics have in finding a solution, especially when the constraints are strict. That is also the case with the constrained spanning-tree heuristics evaluated here. The success rate of the algorithms depend on the constraint and the input graph. We use dynamic relaxation on the

Algorithm	*Meaning*	*Optimization*	*Constraints*	*Complexity*	*Problem*	*Reference*
MST	Prim's minimum-cost spanning tree	total cost	-	$O(n^2)$	30) MST	[58]
SPT	Dijkstra's shortest-path tree	core/destination cost	-	$O(n^2)$	37) SPT	[58]
br-MST	Bounded radius minimum-cost spanning tree	total cost	-	$O(n^2)$	39) BRMST	-
brdl-MST	Bounded radius degree-limited MST	total cost	-	$O(n^2)$	42) BRDLMST	-
md-OTTC	Minimum diameter one-time tree construction	diameter	-	$O(n^3)$	32) MDST	-
OTTC	One-time tree construction	total cost	diameter	$O(n^3)$	33) BDMST	[1]
RGH	Randomized greedy heuristic	total cost	diameter	$O(n^2)$	33) BDMST	[104]
CT	Compact-tree	diameter	degree	$O(n^3)$	34) MDDL	[115]
mddl-OTTC	Minimum diameter degree-limited one-time tree construction	diameter	degree	$O(n^3)$	34) MDDL	-
dl-OTTC	Degree-limited one-time tree construction	total cost	diameter and degree	$O(n^3)$	35) BDDLMST	-
dl-RGH	Degree-limited randomized greedy heuristic	total cost	diameter and degree	$O(n^2)$	35) BDDLMST	-
BCT	Bounded compact-tree	diameter	diameter and degree	$O(n^3)$	36) BDRB	[115]
dl-SPT	Degree-limited Dijkstra's shortest-path tree	core/destination cost	degree	$O(n^2)$	40) d-SPT	[95]
dl-MST	Degree-limited Prim's minimum-cost spanning tree	total cost	degree	$O(n^2)$	31) d-MST	[95]

Table 9.1: Tree algorithms.

Description	*Parameter*
Placement grid	100 × 100 milliseconds
Number of nodes in the network	1000
Degree limits	3,5 and 10
Diameter bound	250 milliseconds
Core node set size	(group-size/degree-limit)

Table 9.2: Experiment configuration.

degree-limits and the diameter bounds whenever a tree heuristic cannot continue the tree construction.

A low diameter is a target metric, such that in our simulations we use a strict diameter bound of 0.250 to the diameter bounded algorithms. The heuristics are then frequently forced to relax the bound. The experiment configurations are summarized in table 9.2.

9.3.2 Target metrics

A spanning tree algorithm is considered good if it can produce overlays with a low diameter, within a reasonable time. In the evaluations, we therefore consider two metrics to address the application requirements: *diameter* and *execution time*. The diameter expresses the worst-case latency between any pair of group members. The execution time or *reconfiguration time* of an algorithm is the time that is required to execute a group membership change. In addition, a degree-unlimited algorithm is not desirable if the constructed tree has a very high maximum degree. Many of the tree algorithms in our investigation use a constraint on the degree-limit.

In the following, we evaluate the results from our group communication simulations. In the evaluation, we focus on the target metrics and also evaluate the different spanning tree algorithms against each other.

9.3.3 Fully meshed results

We here present results from using a fully meshed input graph to every tree algorithm. The *diameter* achieved, with degree limit 10, is plotted in figure 9.1. As expected, MST constructs trees with high diameter, because it optimizes for the *total cost*. dl-MST performs similarly to MST but is not plotted. Hence, we can safely disregard MST and dl-MST. SPT performs best, and constructs trees with a diameter close to 0.3 regardless of group size. The source of SPT is chosen by the k-Median($k = 1$) algorithm to be in the group center, hence, the combination results in an MDST-heuristic (problem 32). We do not plot all the algorithms because many of them construct trees with very similar diameter. In fact, all remaining algorithms construct trees with a diameter between 0.3 and 0.4 seconds.

In figure 9.2, the diameters with degree limits 3, 5 and 10 are plotted. When the degree limit is 3, the degree-limited algorithms struggle to find trees with a diameter below 0.6 seconds. A

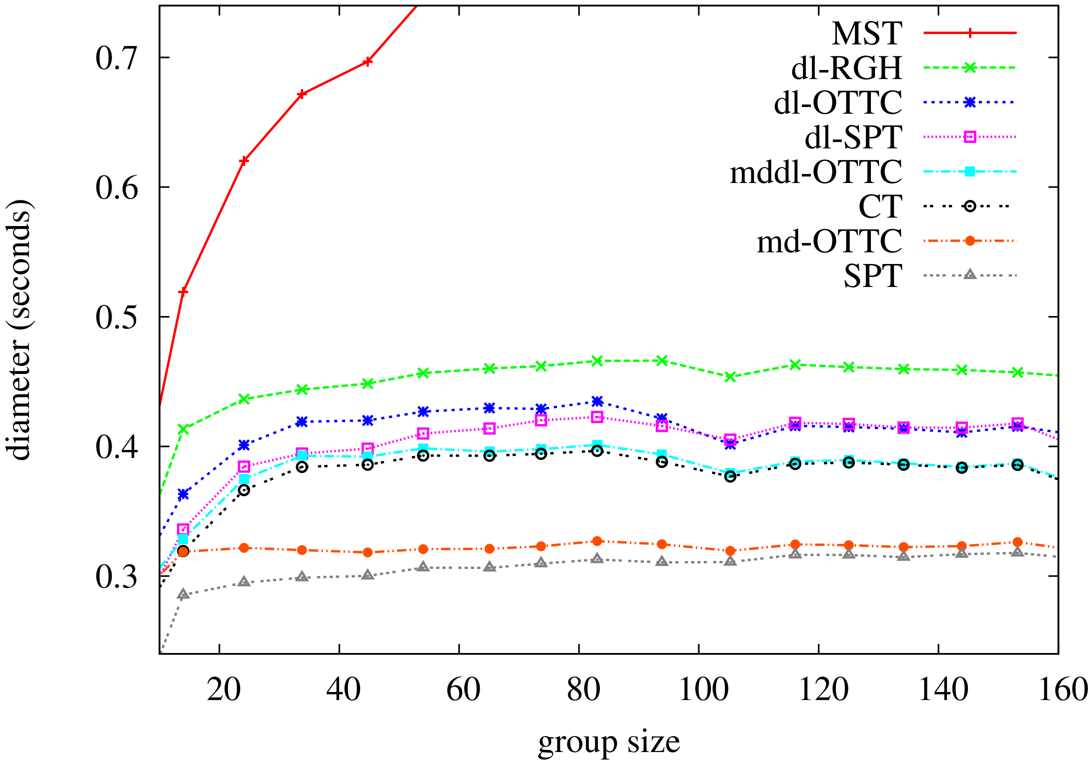


Figure 9.1: Diameter of fully meshed graph (degree limit=10).

degree limit of 5 reduces the diameter to 0.5, while a degree limit of 10 further lowers it to 0.4. We deduce that the degree limit can not be stringent if a low diameter is the desired goal. Thus, henceforth, we use a degree limit of 10 in all our plots.

The *execution time* of selected algorithms is plotted in figure 9.3. CT and BCT clearly perform worst (only CT is plotted). In fact, during frequent group tree updates they are almost useless for larger group sizes. The remaining algorithms are considerably faster, with RGH/dl-RGH being the fastest. An SPT algorithm on a fully meshed application layer graph reduces the complexity to $O(1)$, because the input mesh contains all the shortest paths from the source to any destination.

Figure 9.4 plots the *maximum degree* in the group trees. From the results we see that SPT, md-OTTC, OTTC and RGH all have maximum degrees that would not be tolerated by average users of distributed interactive applications. Especially, since the trees are considered to be shared-trees, it adds a significant load on the nodes with high degrees.

To summarize, MST and dl-MST produce trees with too high diameter. CT and BCT are too slow to handle frequent tree updates. SPT, md-OTTC, OTTC and RGH all have a maximum degree above acceptable. Hence, these are all *poor* alternatives in the tree construction. The *better* alternatives are dl-SPT, mddl-OTTC, dl-OTTC and dl-RGH. Table 9.3 gives an overview of some pros and cons of the algorithms. In the following, we consider only the diameter, reconfiguration time and maximum degree.

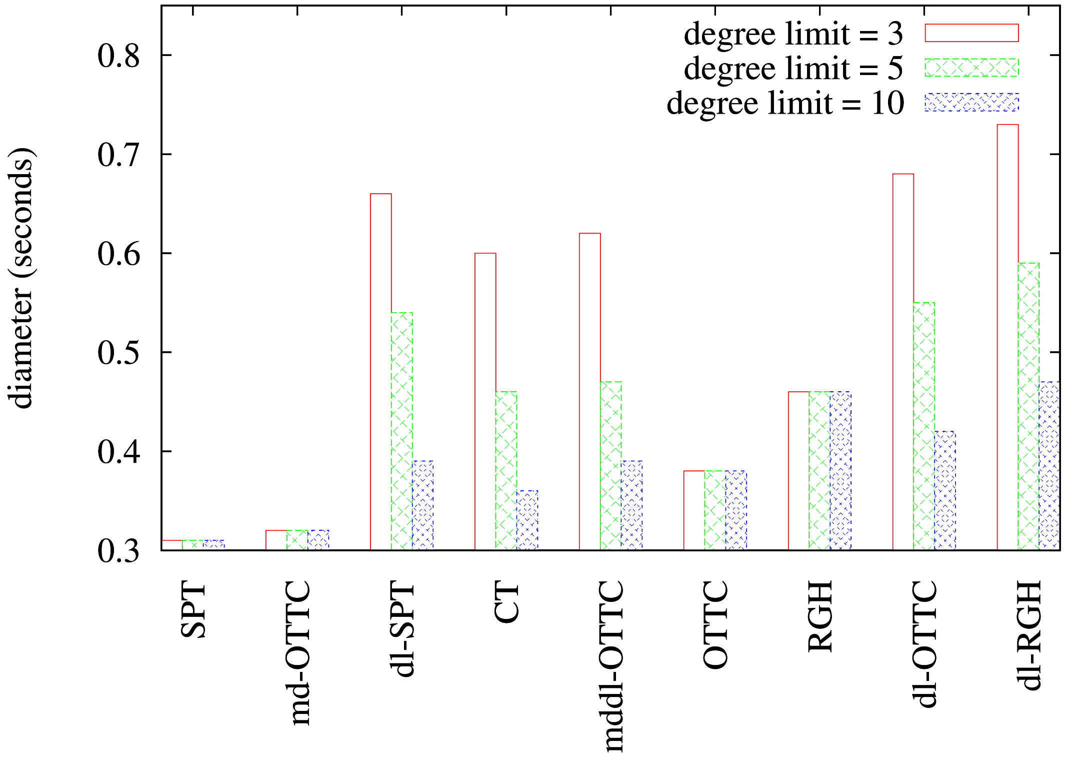


Figure 9.2: Diameter of trees with 100 nodes (degree limits $3,5$ and 10).

9.3.4 Reduced Graphs

Here, we present results from combining a core selection heuristic and edge-pruning with tree algorithms. We use aCL and aCLO ($k = 2,1,0$) to reduce the input graph size, and the k-Median core-node selection algorithm to find the core nodes, that is, well located group nodes. All the remaining plots are of 100 nodes with a degree limit of 10.

Figure 9.5 plots the *diameter* when using a fully meshed input graph and *aCL*. Overall, the algorithms produce the lowest diameter when using a fully meshed input graph. However, when aCL is used, the diameter suffers on average only 15 % even when $k = 0$, and the edge set is reduced with 80 %, compared to the fully meshed graph.

Figure 9.6 plots the diameter for *aCLO* as well. We observe that the diameter suffers on average just below 20 % when aCLO is used, instead of the full mesh. aCLO with $k = 0$ reduces the edge set by 95 %, and the construction results are still competitive.

The reconfiguration times of the construction algorithms applied to the full mesh and aCL/aCLO graphs are plotted in figure 9.7. The tendency is very clear. When the edge set is pruned the reconfiguration time is substantially reduced. However, CT (and BCT) continue to be very slow regardless of edge-pruning, because its execution time is dependant on the node set size.

We expect the *maximum degree* to decrease for the algorithms without degree limits when applying aCL and aCLO, see figure 9.8. We observe that the maximum degree is reduced to

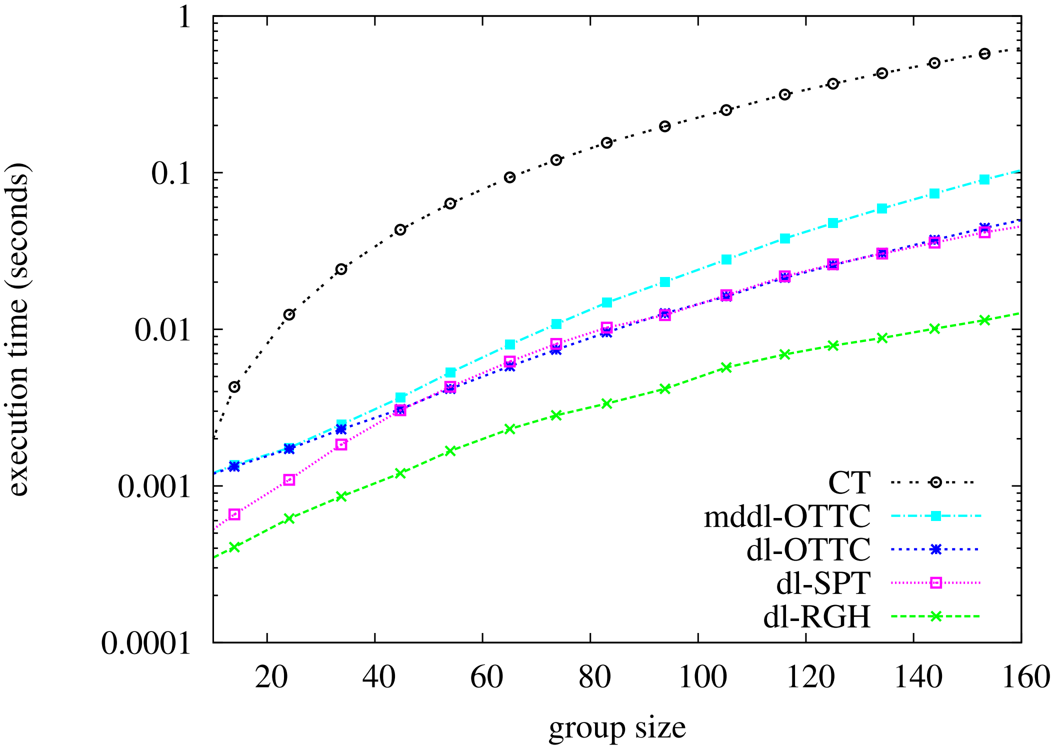


Figure 9.3: Reconfiguration time using full mesh.

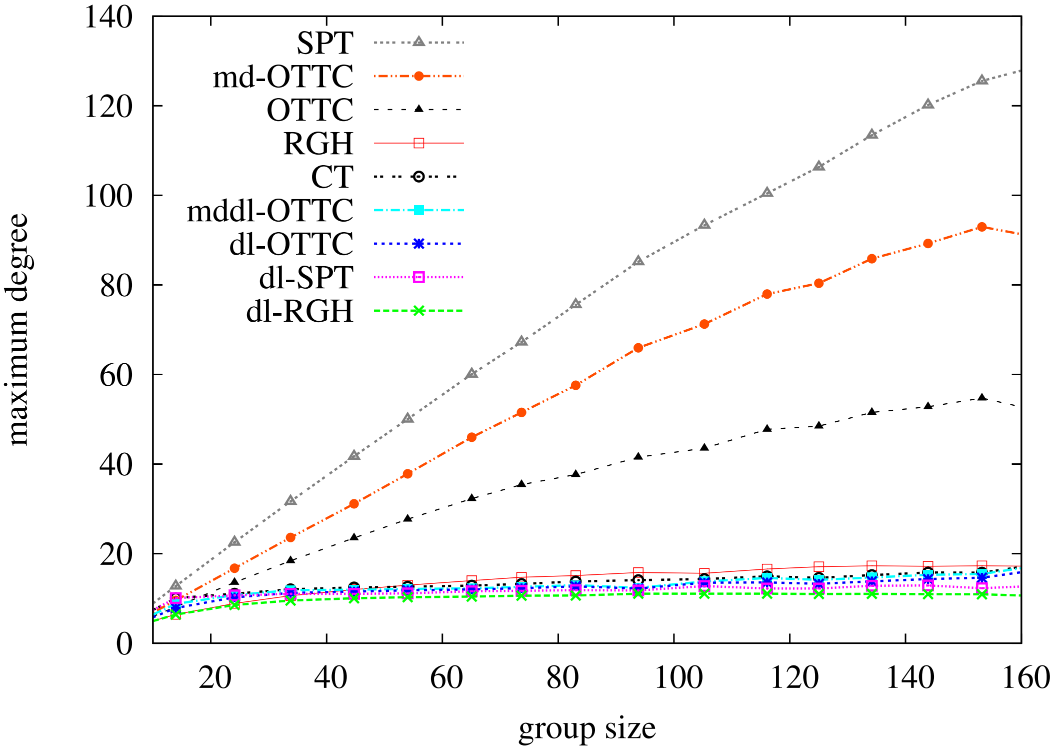


Figure 9.4: Maximum degree using full mesh (degree limit=10).

Algorithm	*Diameter*	*Time*	*Degree*	*Rank*
MST	–	+	+	–
SPT	+	+	–	–
md-OTTC	+	+	–	–
OTTC	+	+	–	–
RGH	+	+	–	–
CT	+	–	+	–
mddl-OTTC	+	+	+	+
dl-OTTC	+	+	+	+
dl-RGH	+	+	+	+
BCT	+	–	+	–
dl-SPT	+	+	+	+
dl-MST	–	+	+	–

Table 9.3: Tree algorithm characteristics using full mesh.

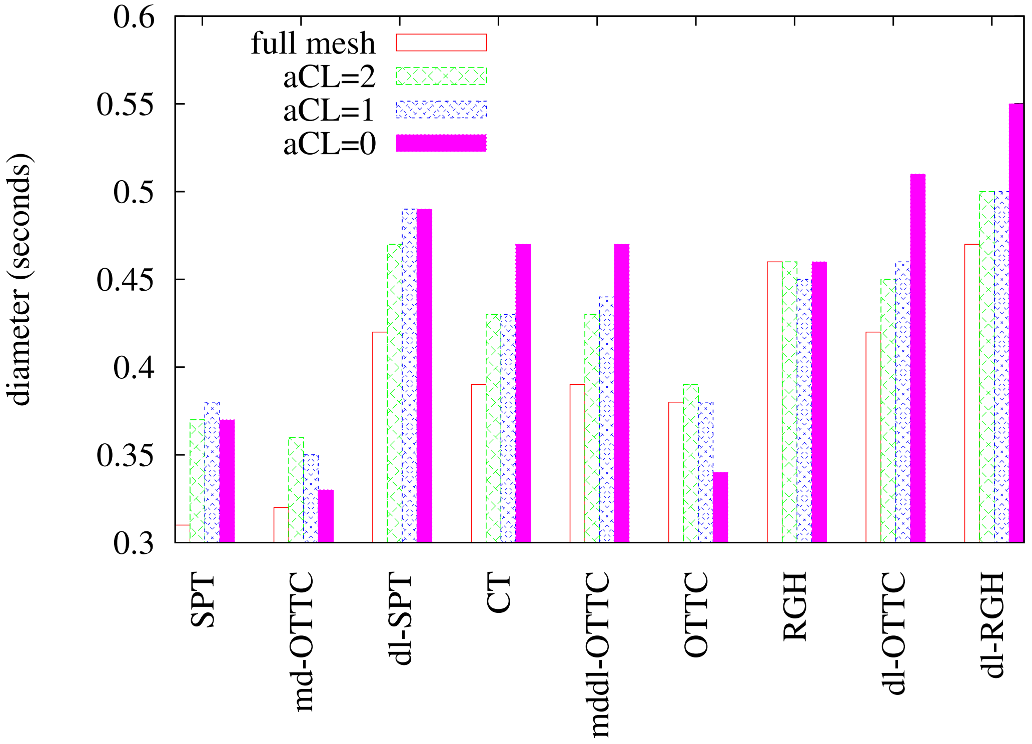


Figure 9.5: Diameter for full mesh and *a*CL.

about 20 when *a*CLO is used. Hence, degree unlimited algorithms are an option for very low bandwidth streams, but, only if *a*CLO and a core-node selection algorithm are used to manipulate the input graph, such that the the node-degree is bounded. However, the degree-unlimited algorithms all have (almost) equally fast algorithm versions with degree limits. Table 9.4 gives an overview.

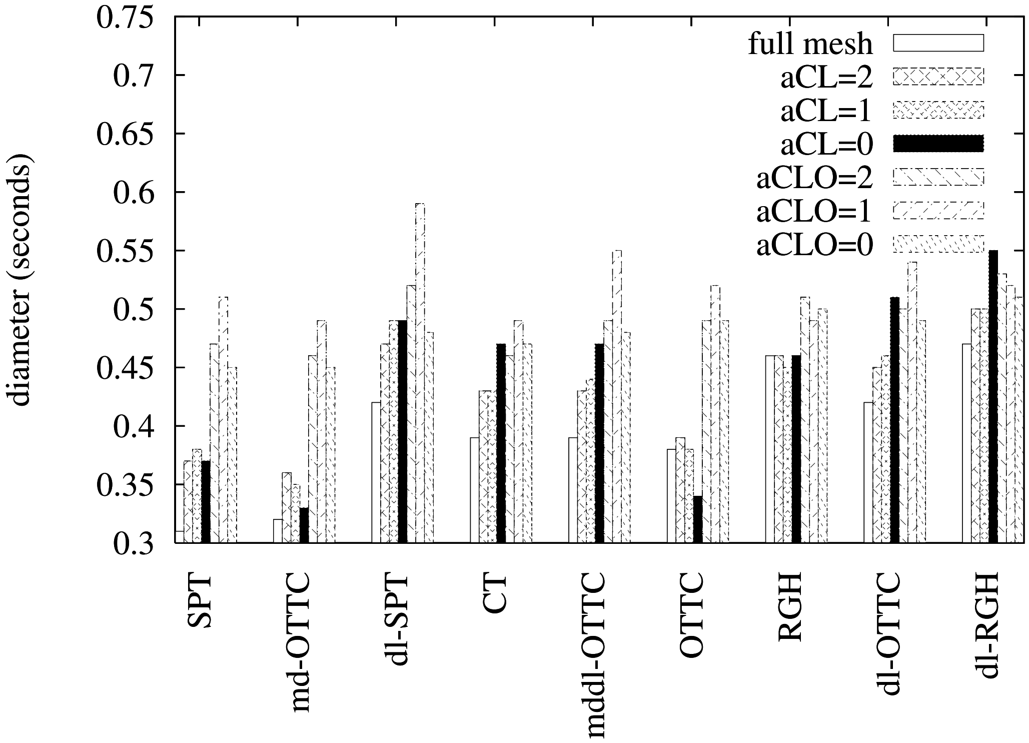


Figure 9.6: Diameter for full mesh, *a*CL and *a*CLO.

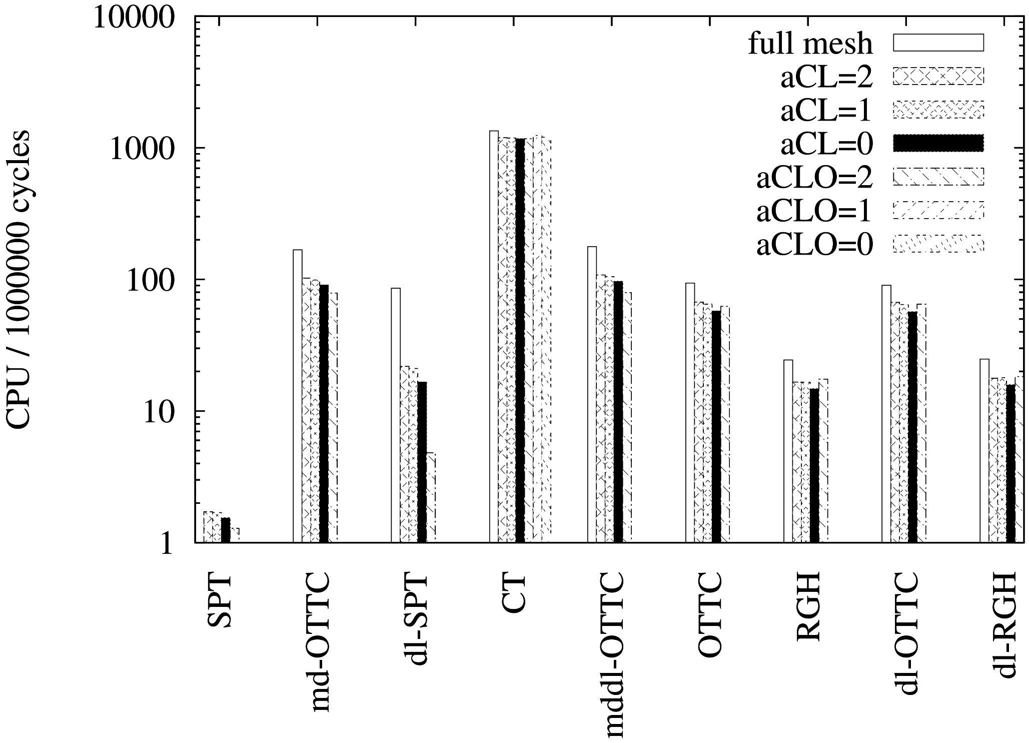


Figure 9.7: Reconfiguration time for full mesh, *a*CL and *a*CLO.

Algorithm	*Diameter*	*Time*	*Degree*	*Rank*
SPT	+	+	+	+
md-OTTC	+	+	+	+
OTTC	+	+	+	+
RGH	+	+	+	+
dl-SPT	+	+	+	+
mddl-OTTC	+	+	+	+
dl-OTTC	+	+	+	+
dl-RGH	+	+	+	+

Table 9.4: Tree algorithm characteristics using *a*CLO.

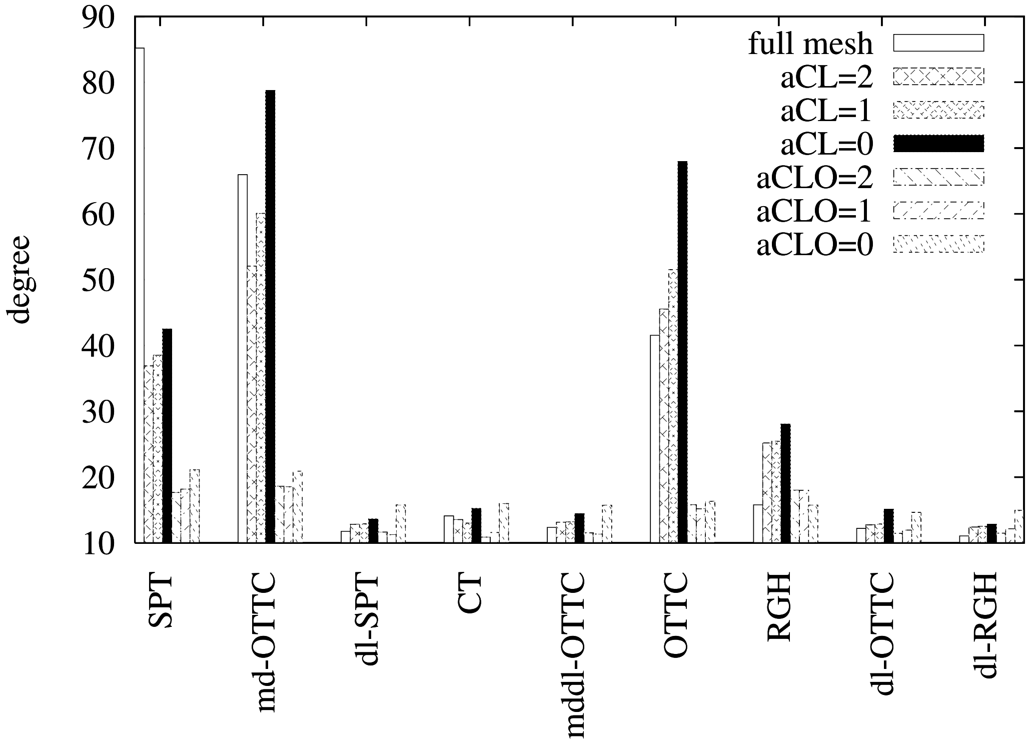


Figure 9.8: Maximum degree for full mesh, *a*CL and *a*CLO.

9.3.5 Discussions on the results

A tree algorithm for our construction process should produce trees with a low diameter, keep the reconfiguration time fast and be able to obey degree limits. We have seen that CT is the best algorithm when only the diameter and maximum degree are considered. However, the reconfiguration time when using CT is very high, even with a pruned edge set. Remember, low reconfiguration time is particularly desirable during frequent tree updates, which is often the case for our target applications.

The algorithms that have all the properties that our target applications want are listed in table 9.5. dl-RGH is the fastest algorithm, and still manages to produce low diameter trees within the degree limits. mddl-OTTC and dl-OTTC are similar to each other, but mddl-OTTC

Algorithm	*Diameter*	*Time*	*Degree*	*Rank*
dl-OTTC	+++	++	+	++++
mddl-OTTC	++++	+	+	+++
dl-SPT	++	+++	+	++
dl-RGH	+	++++	+	+

Table 9.5: Final tree algorithm ranking.

is slightly slower and does not have the flexibility of a bounded diameter algorithm. dl-SPT was a surprisingly good alternative, and is a good algorithm for a source-based tree.

Our ranking is subjective and not related to specific application needs. All the algorithms fit different needs, because they vary in performance between diameter and reconfiguration time, see figure 9.9. dl-RGH is a fast $O(n^2)$-heuristic. When extending the tree, it chooses the next vertex at random and connects it via the lowest weight edge that maintains the diameter constraint. The diameter constraint is only maintained towards the source, and is actually the radius. The algorithm works surprisingly well to produce trees with a small diameter. dl-OTTC extends the tree through the minimum weight edge that obeys the diameter bound. It is slower because it has a more time consuming maintenance of the diameter, but it produces trees with smaller diameter than dl-RGH. mddl-OTTC always minimizes the maximum diameter, and is slightly slower because of that. However, mddl-OTTC is much faster than CT, and constructs trees with almost the same small diameter. dl-SPT avoids diameter bounds, that may not be available, and minimum diameter goals, that may not be desirable in many applications. It rather optimizes for source destination cost, which is often desired by streaming applications.

9.4 Summary of the main points

We have investigated a wide range of spanning-tree algorithms, where the particular goal was to identify those that quickly produce spanning-trees of a low diameter. In addition, the spanning-tree should not have nodes with an unreasonable stress level, which is measured in terms of the maximum degree. All the spanning-tree algorithms were evaluated through group communication simulations.

From the results, we found that the fairly simple degree-limited heuristics dl-RGH, dl-OTTC, mddl-OTTC and dl-SPT all produce trees with small diameters. These 4 heuristics are also fast, which is important in highly dynamic distributed applications. The heuristics CT and BCT were found to be too slow, although they otherwise met our requirements. However, further verification of our algorithm implementation is needed to be absolutely certain of this result.

We also investigated algorithms for reducing the time it takes to execute membership changes. We found that the edge-pruning algorithms aCL and aCLO are powerful means that generally

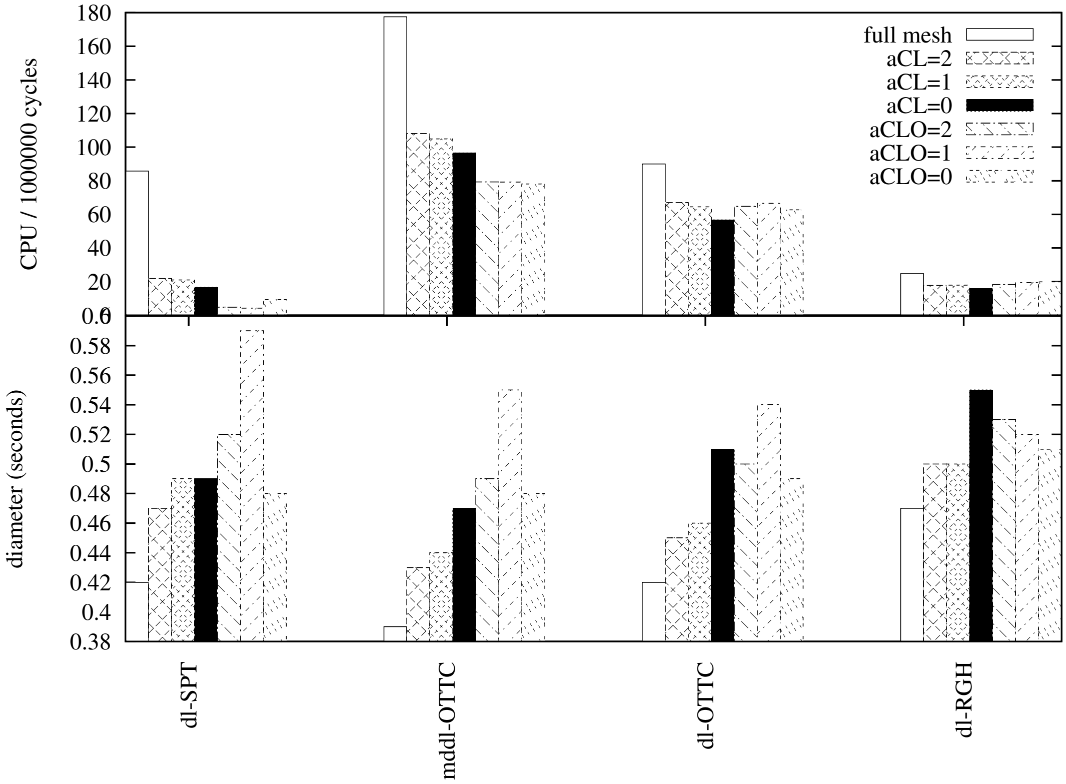


Figure 9.9: Diameter and reconfiguration time for complete, *a*CL and *a*CLO.

reduce the time a tree algorithm needs to construct a spanning-tree. The penalty lies in the diameter of the spanning-trees, which are slightly higher. Nevertheless, we found that most spanning-tree algorithms still produce competitive results.

Chapter 10

Overlay construction techniques: Steiner-tree algorithms

The overlay network management introduced in section 5.4 includes overlay construction techniques whose task is to construct low-latency overlay networks for distribution of time-dependent events. In that respect, we continue our evaluation of overlay construction techniques and evaluate Steiner-tree algorithms. By doing this we address a goal of the thesis, which was to:

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

Steiner-tree algorithms have the important property that they can distinguish between member-nodes and non-member-nodes (definition 43). Therefore, the main difference between a Steiner-tree and a spanning-tree is that a Steiner-tree is only required to span the member-nodes. Selected non-member-nodes (Steiner points) may be used in the Steiner-tree as relay nodes to reach the member-nodes. A Steiner-tree algorithm, typically, only includes Steiner points to the Steiner-tree if they help the algorithm optimize the tree. The Steiner-points in real networks are often nodes that forward data without reading it or contributing with new data. They are there to help the Steiner-tree algorithm achieve its optimization goal.

In our studies, we develop a range of Steiner-tree heursitics and evaluate them in terms of how well they reduce the diameter. We used algorithm ideas from the well-known minimum-cost Steiner-tree heuristics shortest-path heuristic (SPH) [120], distance-network heuristic (DNH) [82] and average-distance heuristic (ADH) [106]. From these algorithms we devised Steiner-tree heuristics that reduce the diameter. The algorithms SPH and DNH are themselves based on ideas from Prim's minimum spanning tree (MST) and Dijkstra's shortest path tree (SPT), while ADH is based on Kruskal's MST [58]. We also used ideas from existing spanning tree algorithms on how to keep track of the pair-wise latencies in a graph [1].

All the algorithms are implemented, and the performance is analyzed using experiments. We found that a full-mesh of shortest paths makes it difficult for Steiner-tree heuristics to find better trees than spanning tree algorithms. In this case, many of the Steiner-tree heuristics only has a best-case (and worst-case) performance equal to an MST. The network seen from the application layer is in fact a full mesh of shortest paths. Therefore, these Steiner-tree heuristics should not be used when a full mesh of shortest paths is the input graph. We also reduced the full mesh using pruning algorithms and used the pruned graph as input to the Steiner-tree heuristics. However, we still found that faster Steiner-tree heuristics that do not use shortest path information and do not explicitly optimize the diameter are able to compete with slower heuristics that do.

The rest of the chapter is organized in the following manner. Section 10.1 introduces a few of the most common Steiner-tree algorithm types. Section 10.2 presents the Steiner-tree algorithms that we evaluate. Many of these algorithms are contributions by the thesis. Section 10.3 evaluates every Steiner-tree algorithm in a group communication simulator. Finally, section 10.4 provides a brief summary of the main points.

10.1 Steiner-tree algorithm types

Chapter 4.6 introduced a wide range of Steiner-tree problems found in graph theory. The following sections introduce Steiner-tree algorithms for finding Steiner-trees of minimum or bounded cost, diameter and radius.

10.1.1 Minimum-cost Steiner-tree algorithms

A *minimum-cost Steiner-tree* is the least cost tree that spans all the member-nodes of a graph. The problem of finding a minimum-cost Steiner-tree in networks (SMT) is an NP-complete problem (definition 44) that was originally formulated independently by Hakimi and Levin [142] in 1971. Several exact algorithms and heuristic have been suggested, implemented and compared [134,69]. In the following we introduce four Steiner-tree heuristic classes called *spanning-heuristics*, *path-heuristics*, *tree-heuristics* and *vertex-heuristics* [142]. From the four heuristic classes we, respectively, give the algorithm details of the four SMT-heuristics *pruned minimum spanning tree*, *shortest path heuristic* (SPH), *distance network heuristic* (DNH) and *average distance heuristic* (ADH). From these SMT-heuristics, we have proposed new Steiner-tree heuristic variations (section 10.2) that address the diameter and reconfiguration time issues faced by distributed interactive applications.

Spanning-heuristic: Pruned minimum spanning tree heuristic

The simplest SMT-heuristics are the spanning-heuristics [36]. They apply an MST algorithm on a network, and then remove (prune) Steiner-points with degree one (leaves). The MST algorithm generates a minimum-cost spanning tree of a network graph, spanning all the nodes. An approximate Steiner-tree is then obtained by removing, from the MST, subtrees containing only Steiner-points. A further enhancement, is to remove Steiner-points with degree two, as they are only forwarding nodes. Prim's and Kruskal's MST algorithms [58] are two types of MST algorithms that are commonly used (chapter 9). The main drawback of a spanning-heuristic is that, potentially, all the nodes in the network are involved in the execution of the MST algorithms, hence, some membership management should be applied that limit the number of Steiner-points. A spanning-heuristic is a fairly naive approach to solve the SMT problem and algorithm 18 describes a generic algorithm for a spanning-heuristic. The generic spanning-heuristic algorithm takes as input a spanning tree algorithm $\mathscr{A}_{\mathscr{T}}$, which is executed to construct a tree that is pruned for leaf Steiner points.

Algorithm 18 GENERIC-SPANNING-HEURISTIC

```
In:  $G = (V, E, c)$, a set $Z \subset V$, and a spanning tree algorithm $\mathscr{A}_{\mathscr{T}}$.
Out: Steiner Tree $T_Z = (V_T, E_T)$
1: $T_Z = \mathscr{A}_{\mathscr{T}}(G)$
2: Delete all leaf Steiner points from $T_Z$.
```

Path-heuristic: Shortest path heuristic

A Steiner-tree path-heuristic starts from a pre-chosen source and includes the member-nodes one by one, until the tree spans all member-nodes. Typically, the tree grows based on the addition of (shortest) paths between member-nodes in the tree and member-nodes not yet in the tree. SPH is an SMT path-heuristic and was suggested by Takahashi and Matsuyama [120]. SPH runs an SPT algorithm $|Z| = p$ times (once for each member-node), and stores the shortest path information. Next, it builds the tree starting from a given source, and for each iteration, it connects a member-node through the minimum cost path to the tree. SPH is implemented by a straight forward modification of Prim's MST algorithm [58]. We can see that the bottleneck of SPH is the determination of the shortest paths. Consequently, the SPH has a time complexity of $O(pn^2)$ ($|V| = n$) since shortest paths from each member-node can be determined by for example Dijkstra's SPT algorithm [58] in $O(n^2)$. SPH is sensitive to the choice of the source node from which the tree is constructed. Variations include running SPH more than once, each time from a different source. These repetetive SPH variations yield better solutions, but the execution time is increased. For the simulations in section 10.3, the source is selected using the core-node selection algorithm k-Median($k = 1$), and every SPH variation is run only once. Algorithm 19 describes a generic path-heuristic algorithm, which is also how SPH works. Section 10.3 discusses SPH's limitations, especially when applied to complete graphs, for example, application layer overlay networks.

Algorithm 19 GENERIC-PATH-HEURISTIC

```
In: $G = (V, E, c)$, a set $Z \subset V$
Out: Steiner Tree $T_Z = (V_Z, E_Z)$
for each $v \in Z$ run SHORTEST-PATH-TREE($G$, $v$) and store shortest paths
choose a source node $s \in Z$ and include to $T_Z$
while all nodes in $Z$ not in $V_Z$ do
    find the node $u$ optimizing a cost function together with a node $v$ in the current $T_Z$
    add edge $(u, v)$ to $T_Z$
end while
```

Tree-heuristic: Distance network heuristic

The Steiner-tree tree-heuristics are based on the idea of constructing an initial tree that spans all member-nodes, and then optimize it towards a close-to-optimal SMT. Commonly, a minimum spanning tree variant is used to obtain the initial tree. DNH is an SMT tree-heuristic and was suggested, among others, by Kou, Markowsky and Berman [82]. DNH (like SPH) runs an SPT algorithm for each member node, and stores the shortest path information. Then, it builds a distance network graph using only the member nodes, that is, a complete graph with $|Z| = p$ vertices and edge weights equal to the shortest path lengths. It then replaces the individual edges in the distance network graph with the original paths in the input graph. Finally, an MST is run from a given source to find the SMT. As in SPH, the bottleneck of DNH is the determination of the shortest paths. Consequently, the time complexity of DNH is also $O(pn^2)$. The error bound for DNH is the same as SPH, but, SPH often produces better solutions [140]. Moreover, it has been shown that the time complexity of DNH can be reduced by growing shortest path trees from each member-node simultaneously. However, the cost is more complex data structures. For the simulations in section 10.3, an optimized DNH is not used. Algorithm 20 describes the approach of a generic tree-heuristic. The algorithm is identical to DNH if the input spanning tree algorithm $\mathscr{A}_{\mathscr{T}}$ is an MST algorithm. Although DNH is considered an efficient Steiner-tree algorithm, it does not perform well on complete graphs made out of shortest pahts, as discussed in section 10.3

Algorithm 20 GENERIC-TREE-HEURISTIC

```
In: $G = (V, E, c)$, a set $Z \subset V$, and a spanning tree algorithm $\mathscr{A}_{\mathscr{T}}$.
Out: Steiner Tree $T_Z = (V_Z, E_Z)$
Construct distance network graph $G_{dn}$ solving $|Z|$ shortest paths
$T_{dn} = \mathscr{A}_{\mathscr{T}}(G_{dn})$
Construct $T_{sn}$ of $T_{dn}$ where each edge in $T_{dn}$ is replaced by the corresponding shortest path in $G$
Construct a distance network graph $G_{sn}$ of $T_{sn}$, excluding unused Steiner points
$T_Z = \mathscr{A}_{\mathscr{T}}(G_{sn})$
Delete all leaf Steiner points from $T_Z$
```

Vertex-heuristic: Average distance heuristic

The general idea behind Steiner-tree vertex-heuristics is to identify "good" non-member-nodes (Steiner points). It has been shown [142] that one big difficulty of the SMT problem is to identify non-member-nodes that belong to an SMT. Once the Steiner points are given, the SMT is an MST for the subnetwork induced by the member-nodes and selected Steiner points [142]. ADH is an SMT vertex-heuristic and was suggested by Rayward-Smith [106]. It is based on Kruskal's MST algorithm [58]. The idea is to connect already constructed subtrees of a solution by shortest paths through some centrally located vertex. ADH computes the shortest paths between all pairs of vertices, which is $O(n^3)$ (for example, Floyd Warshall's algorithm [58]), and stores the shortest path information. ADH starts with a forest of trees, where each tree contains only a single member-node. A value $f(v)$ is determined for each $v \in V$, and the vertex m with the smallest f-value is chosen. The m vertex connects a number r of trees that are closest to m, where r is decided when computing the f-value. The worst-case time complexity of ADH is dominated by the computation of all-pairs shortest paths and is $O(n^3)$. On average, ADH does perform better than SPH and DNH in terms of total cost, but at the expence of increased worst-case time complexity. Algorithm 21 describes a generic ADH implementation, for which the function $f(v)$ must be determined by the programmer. Chapter 10.3 evaluates ADH and its ability to include Steiner-points in complete graphs built out of shortest paths. However, it is also found that adapting ADH to reduce the diameter in trees is harder, and left for future work.

Algorithm 21 GENERIC-AVERAGE-DISTANCE-HEURISTIC

```
In: $G = (V, E, c)$, a set $Z \subset V$
Out: Steiner Tree $T_Z = (V_T, E_T)$
1: For each iteration examine a list $L = \{T_1, T_2, \cdots, T_k\}$ of trees which will be subtrees of the final tree, initially
   $L$ consists of isolated Z-vertices.
2: Use a heuristic function,$f : V \rightarrow R$, to choose which two subtrees are selected and joined by a minimum cost
   path in $G$, such that for each iteration the tree is expanded with one $\mathrm{T}_i$. The function $f(i)$ gives a measure of
   the least average distance (via $i$) from a tree closest to $i$ (denoted $\mathrm{T}_1$) to subsets $\{\mathrm{T}_1\}$, $\{\mathrm{T}_1, \mathrm{T}_2\}$ ,$\cdots$,$\{\mathrm{T}_1, \mathrm{T}_2,$
   $\mathrm{T}_3, \cdots, \mathrm{T}_k\}$.
3: After $p - 1$ iterations, $L$ is one tree spanning all $Z$-vertices.
```

10.1.2 Shortest path Steiner-tree algorithms

The shortest path Steiner-tree on a graph is the acyclic graph (tree) that has shortest paths from a given source to all member nodes (definition 50). Chapter 4.6.3 introduces many shortest path Steiner-tree problems. There, they are introduced as problems that minimize the radius, where the radius is defined as the longest shortest path from a member-node to the source-node. A shortest path Steiner-tree algorithm constructs a tree that has p shortest paths from a given source node to the destinations, where p is the number of destinations. A shortest path

Steiner-tree may be found by the $O(n^2)$ algorithm Dijkstra's SPT (section 9.1.2). Many shortest path Steiner-tree heuristic variations exist that obey a latency bound from a given source, while optimizing for the total cost [45, 72, 103, 100, 145, 7, 10, 16, 102]. These variations often base the edge selection on a mix between path latency from the source and minimum cost edges.

10.1.3 Minimum diameter Steiner-tree algorithms

A minimum-diameter Steiner-tree on a graph is the acyclic graph (tree) that exhibits the lowest diameter among the member-nodes in the tree (definition 46). The diameter is the maximum length shortest path in a tree (section 4.1.1). A minimum-diameter Steiner-tree may be found in polynomial time, as shown by Ho, Lee, Chang and Wong [68]. They prove that for a complete graph, there is an optimal tree in which either one or two vertices in V are connected to the remaining vertices. A range of graph theory problems related to constructing Steiner-trees of minimum and bounded diameter are introduced in section 4.6.2.

For general graphs, one minimum diameter Steiner-tree algorithm is to find the absolute 1-center of a graph (section 4.1.1) and connect the member-nodes through shortest-paths to that center-point. The absolute 1-center is the one point that may be located at any point in the graph (including edges), that has lowest radius. For a complete graph, a similar but simpler heuristic for building a close-to-optimal MDST is to find a single node located close to the center of the graph that connects to the remaining nodes through shortest paths (direct links). The topology of the resulting tree T is that of a star.

There are some heuristics that have attempted to reduce the diameter in trees [65, 3, 150, 21]. These proposals typically pre-compute the shortest paths between the nodes, and use them when selecting paths to add to the minimum diameter Steiner-tree. However, these heuristics do not take into account the inherent fully meshed application layer network made of shortest paths.

10.2 Evaluated Steiner-tree algorithms

The Steiner-tree algorithms evaluated by the thesis include algorithms that optimize for minimum cost, minimum diameter and minimum radius. In addition, they include algorithms that use cost as a constraint, while trying to minimize the diameter and radius, and many of them also obeying degree limitations. Some of the evaluated Steiner-tree algorithms exist in the literature; however, the thesis proposes many variations that are believed to be evaluated for the first time. Every Steiner-tree algorithm in the thesis are listed in table 10.1.

Formally, a Steiner-tree algorithm $\mathscr{A}_{\mathscr{S}}$ takes as input a connected undirected weighted graph $G = (V, E, c)$, where V is the set of vertices, E is the set of edges, and $c : E \rightarrow R$ is the edge cost function. In addition, there is a set of member nodes $Z \subset V$. The Steiner-tree algorithm $\mathscr{A}_{\mathscr{S}}$

constructs a connected acyclic graph (tree) $T_Z = (V_T, E_T)$ on G, where $V_T \supseteq Z$ (definition 43). In other words, the Steiner-tree T_Z is required to span every member-node $z \in Z$.

10.2.1 Variations of the shortest-path heuristic

Steiner-tree path-heuristics were introduced previously in section 10.1.1 with an example of a generic path-heuristic algorithm (algorithm 19). This generic path-heuristic algorithm is the basis for the proposed path-heuristic variations in the thesis. Every proposed path-heuristic builds the tree incrementally, and is based on the SPH path-heuristic with added functionality such that the diameter is reduced. The added functionality is based on ideas from the bounded diameter spanning tree algorithms OTTC and RGH (section 9). In particular, OTTC's approach of keeping track of the node eccentricities, while constructing trees, and RGH's randomized approach choosing a random next vertex and connect it to well-placed nodes. The path-heuristic variations we have implemented and tested are described below and listed in table 10.1.

Bounded diameter degree-limited optimized shortest-path heurisitic (bddlo-SPH) is a heuristic of the Steiner-BDDLSMT-problem (definition 49). Each round adds the node that minimizes the total cost of the tree, within a given diameter bound, and given degree limits. bddlo-SPH keeps track of the eccentricities like OTTC [1]. A somewhat similar heuristic was proposed by Aggarwal et al. [3].

Algorithm 22 describes the general implementation for bddlo-SPH. It is worth noticing that the algorithm is basically the generic path-heuristic (algorithm 19) including OTTC's approach of keeping track of the eccentricities while building the tree (section 9.2). The time complexity of bddlo-SPH is $O(n^3)$, and not $O(pn^2)$ like SPH ($p = |Z|$). This is mainly because updating the eccentricities and connecting new nodes through degree-limited paths, require all-to-all shortest path information, which is determined by n shortest path computations $O(n^2)$. bddlo-SPH also has a dynamic relaxation step, in case the diameter bounds or degree limits cannot be met.

Algorithm 22 BDDLO-SHORTEST-PATH-HEURISTIC

```
In: $G = (V, E, c)$, diameter-bound ≥ 0, degree bound $deg(v) \in \mathbb{N}$ for each $v \in V$
Out: Steiner Tree $T = (V_Z, E_Z)$
for each $v \in Z$ run SHORTEST-PATH-TREE($G$, $v$) and store shortest paths
choose a source node $s \in Z$ and include to $T_Z$
initialize distance structures {same as DL-ONE-TIME-TREE-CONSTRUCTION}
while all nodes in $Z$ not in $V_Z$ do
    if find a node $u$ with minimum-cost distance to $v \in V_Z$, obey diameter-bound and degree limits then
        add edge $(u, v)$ to $T_Z$
    else
        relax diameter-bound and degree-limits
    end if
    update distance structures {DL-ONE-TIME-TREE-CONSTRUCTION with shortest paths}
end while
```

The other variations of SPH and OTTC are briefly described below. The implementation of these algorithms follow the algorithmic details of bddlo-SPH, but with different constraints. The adaptation from bddlo-SPH to these algorithms is straight forward.

Minimum diameter shortest-path heuristic (md-SPH) is a heuristic of the Steiner-MDST-problem (definition 46). In each round, a node is added to the tree that minimizes its eccentricity. md-SPH keeps track of the eccentricities like one-time tree-construction (OTTC) spanning tree algorithm [1]. A similar heuristic was proposed by Brosh et al. [21].

Bounded diameter optimized shortest-path heuristic (bdo-SPH) is a heuristic of the Steiner-BDSMT-problem (definition 47). Each round adds the node that minimizes the total cost of the tree, within a given diameter bound. bdo-SPH keeps track of the eccentricities like OTTC [1]. A similar heuristic was proposed by Aggarwal et al. [3].

Minimum diameter degree limited shortest-path heuristic (mddl-SPH) is a heuristic of the Steiner-MDDL-problem (definition 48). mddl-SPH is identical to md-SPH but obeys the degree limits on each node much like mddl-OTTC [131].

The next path-heuristic variations combine ideas between SPH and the randomized bounded diameter spanning-tree algorithm RGH. RGH was introduced in detail in section 9.2.

Bounded diameter degree-limited randomized shortest-path heuristic (bddlr-SPH) is a heuristic of the Steiner-BDDLSMT-problem (definition 49). In each round, a member-node is randomly selected and added to the tree through the minimum cost edge within a given diameter bound. bddlr-SPH is an adaptation of randomized greedy heuristic (RGH) [1] to the Steiner-BDDLSMT-problem.

Algorithm 23 describes the general implementation for bddlr-SPH. The algorithm is in short, the dl-RGH algorithm (algorithm 16 in section 9.2) enhanced to use shortest path information when selecting new random vertices to add to the tree. The time complexity of bddlr-SPH is increased to $O(n^3)$ (from $O(n^2)$) because of n shortest path computations in $O(n^2)$. However, the randomized features of bddlr-SPH makes it significantly faster than bddlo-SPH, but also more unpredictable in its behavior.

The thesis also evaluates one SPH/RGH variation that does not consider the degree-limits. It is directly implementable by using bddlr-SPH's algorithm, and then exclude the degree-limitation checks.

Bounded diameter randomized shortest-path heuristic (bdr-SPH) is a heuristic of the Steiner-BDSMT-problem (definition 47). In each round, a member-node is randomly selected and added to the tree through the minimum cost edge within a given diameter bound. bdr-SPH is an adaptation of randomized greedy heuristic (RGH) [1] to the Steiner-BDSMT-problem.

Algorithm 23 BDDLR-SHORTEST-PATH-HEURISTIC

```
In: $G = (V, E, c)$, diameter-bound $\geq 0$, degree bound $deg(v) \in \mathbb{N}$ for each $v \in V$
Out: Steiner Tree $T = (V_Z, E_Z)$
for each $v \in Z$ run SHORTEST-PATH-TREE($G$, $v$) and store shortest paths
{refer to DL-RANDOMIZED-GREEDY-HEURISTIC}
choose a core node set $C \subset Z$ and include to $T_Z$
initialize distance structures
while all nodes in $Z$ not in $V_Z$ do
    choose first node $v$ with path($v$,$c$) $c \in C$, where $deg_T(c) < deg(c)$
    {success, then connect v to core node set}
    if found node $v$ then
        add weight-path($v$,$c$) to $T$
        update distance structures
        {failed, then relax constraints}
    else
        if diameter-bound or degree bound violation then
            Relax diameter-bound and degree bound
        end if
        for nodes $v \in V_T$ not in $C$ do
            if depth($v$) $\leq$ diameter-bound$/2$ then
                C $\cup$ $v$
            end if
        end for
    end if
end while
```

10.2.2 Variations of the distance network heuristic

Steiner-tree tree-heuristics were introduced in section 10.1.1 with a generic tree-heuristic (algorithm 20) algorithm and an example heuristic; DNH. This generic tree-heuristic algorithm is the basis for the proposed tree-heuristic variations in the thesis. Each of the proposed tree-heuristic variations use the generic tree-heuristic variation but with different spanning tree algorithm $\mathscr{A}_{\mathscr{T}}$ as input.

The tree-heuristics we implemented and tested are described below and listed in Table 10.1. Every heuristic is based on the idea of DNH, and the only difference is the spanning tree algorithm $\mathscr{A}_{\mathscr{T}}$ used to create the final tree on the distance network graph. DNH uses MST, but we test many different spanning tree algorithms found in the literature.

One example is algorithm 24, which shows the algorithm of bounded diameter degree-limited DNH (bddl-DNH). It uses the dl-OTTC algorithm (chapter 9) to construct spanning trees at lines 2 and 5.

The remaining tree-heuristics use variations of OTTC, RGH and MST as the input spanning tree algorithm $\mathscr{A}_{\mathscr{T}}$. Chapter 10.3 evaluates the performance of these variations:

- *Minimum diameter distance network heuristic* (md-DNH) uses md-OTTC [131] to create the final tree on the distance network graph.
- *Bounded diameter optimized distance network heuristic* (bdo-DNH) uses OTTC [1] to

Algorithm 24 BDDLO-DISTANCE-NETWORK-HEURISTIC

```
In: $G = (V, E, c)$, a set $Z \subset V$, diameter-bound $\geq 0$, degree bound $deg(v) \in \mathbb{N}$ for each $v \in V$
Out: Steiner Tree $T_Z = (V_Z, E_Z)$
Construct distance network graph $G_{dn}$ solving $|Z|$ shortest paths
$T_{dn}$ = DL-ONE-TIME-TREE-CONSTRUCTION($G_{dn}$, diameter-bound)
Construct $T_{sn}$ of $T_{dn}$ where each edge in $T_{dn}$ is replaced by the corresponding shortest path in $G$
Construct a distance network graph $G_{sn}$ of $T_{sn}$, excluding unused Steiner points
$T_Z$ = DL-ONE-TIME-TREE-CONSTRUCTION($G_{sn}$, diameter-bound)
Delete all leaf Steiner points from $T_Z$
```

create the final tree on the distance network graph.

- *Bounded diameter randomized distance network heuristic* (bdr-DNH) uses RGH [1] to create the final tree on the distance network graph.
- *Minimum diameter degree limited distance network heuristic* (mddl-DNH) uses mddl-OTTC [131] to create the final tree on the distance network graph.
- *Bounded diameter degree limited optimized distance network heuristic* (bddlo-DNH) uses dl-OTTC [131] to create the final tree on the distance network graph.
- *Bounded diameter degree limited randomized distance network heuristic* (bddlr-DNH) uses dl-RGH [1] to create the final tree on the distance network graph.
- *Degree-limited distance network heuristic* (dl-DNH) uses dl-MST to create the final tree on the distance network graph.
- *Bounded radius distance network heuristic* (br-DNH) is a heuristic of the Steiner-BRSMT-problem, and uses br-MST to create the final tree on the distance network graph.
- *Bounded radius degree limited distance network heuristic* (brdl-DNH) is a heuristic of the Steiner-BRDLSMT-problem, and uses br-MST to create the final tree on the distance network graph.

10.2.3 Variations of spanning-heuristic algorithms

Steiner-tree spanning-heuristics were introduced in section 10.1.1 along with a generic spanning-heuristic algorithm (algorithm 18). It is this generic spanning-heuristic algorithm that is the foundation of the spanning-heuristic algorithms in the thesis. The proposed spanning-heuristic algorithms use the generic spanning-heuristic with different input spanning tree algorithm $\mathscr{A}_{\mathscr{T}}$.

Every spanning tree algorithm that are used as input are evaluated in section 9.3, where they are considered to be the better of the algorithms. We apply these degree limited spanning tree algorithms to see the effect of adding Steiner points to the input graph.

Algorithm	*Meaning*	*Optimize*	*Algorithm basics*	*MN*[1]-*aware*	*Constraint*	*Complex.*	*Problem*	*Ref.*
SPH	Shortest Path heuristic	total cost	Prim's MST	✓	-	$O(pn^2)$	44) SMT	[120]
DNH	Distance network heuristic	total cost	Prim's MST	✓	-	$O(pn^2)$	44) SMT	[82]
ADH	Average distance heuristic	total cost	Kruskal's MST	✓	-	$O(n^3)$	44) SMT	[106]
dl-SPH	Degree limited SPH	total cost	Prim's MST	✓	degree	$O(pn^2)$	45) d-SMT	[130]
dl-DNH	Degree limited DNH	total cost	Prim's MST	✓	degree	$O(pn^2)$	45) d-SMT	[130]
md-SPH	Minimum diameter SPH	diameter	md-OTTC	✓	-	$O(n^3)$	46) Steiner-MDST	[130]
md-DNH	Minimum diameter DNH	diameter	md-OTTC	✓	-	$O(n^3)$	46) Steiner-MDST	[130]
bdo-SPH	Bounded diameter optimized SPH	total cost	OTTC	✓	diameter	$O(n^3)$	47) BDSMT	[130]
bdo-DNH	Bounded diameter optimized DNH	total cost	OTTC	✓	diameter	$O(n^3)$	47) BDSMT	[130]
bdr-SPH	Bounded diameter randomized SPH	total cost	RGH	✓	diameter	$O(pn^2)$	47) BDSMT	[130]
bdr-DNH	Bounded diameter randomized DNH	total cost	RGH	✓	diameter	$O(pn^2)$	47) BDSMT	[130]
mddl-SPH	Minimum diameter degree-limited SPH	diameter	mddl-OTTC	✓	degree	$O(n^3)$	48) Steiner-MDDL	[130]
mddl-DNH	Minimum diameter degree-limited DNH	diameter	mddl-OTTC	✓	degree	$O(n^3)$	48) Steiner-MDDL	[130]
smddl-OTTC	Steiner minimum diameter degree-limited OTTC	diameter	Prim's MST	×	degree	$O(n^3)$	48) Steiner-MDDL	[131]
bddlo-SPH	Bounded diameter degree-limited optimized SPH	total cost	dl-OTTC	✓	diam./degree	$O(n^3)$	49) BDDLSMT	[130]
bddlo-DNH	Bounded diameter degree-limited optimized DNH	total cost	dl-OTTC	✓	diam./degree	$O(n^3)$	49) BDDLSMT	[130]
bddlr-SPH	Bounded diameter degree-limited randomized SPH	total cost	dl-RGH	✓	diam./degree	$O(pn^2)$	49) BDDLSMT	[130]
bddlr-DNH	Bounded diameter degree-limited randomized DNH	total cost	dl-RGH	✓	diam./degree	$O(n^3)$	49) BDDLSMT	[130]
sdl-OTTC	Steiner degree-limited OTTC	total cost	Prim's MST	×	diam./degree	$O(n^3)$	49) BDDLSMT	[131]
sdl-RGH	Steiner degree-limited RGH	total cost	Prim's MST	×	diam./degree	$O(n^2)$	49) BDDLSMT	[131]
s-SPT	Steiner Dijkstra's SPT	src eccentr.	Dijkstra's SPT	×	-	$O(n^2)$	50) Steiner-MRST	[58]
br-SPH	Bounded radius SPH	total cost	Prim's MST	✓	radius	$O(n^3)$	51) BRSMT	[130]
br-DNH	Bounded radius DNH	total cost	Prim's MST	✓	radius	$O(n^3)$	51) BRSMT	[130]
sdl-SPT	Steiner degree-limited Dijkstra's SPT	src eccentr.	Dijkstra's SPT	×	degree	$O(n^2)$	52) Steiner-MRDLST	[95]
brdl-SPH	Bounded radius degree-limited SPH	total cost	Prim's MST	✓	radius/degree	$O(n^3)$	53) BRDLSMT	[130]
brdl-DNH	Bounded radius degree-limited DNH	total cost	Prim's MST	✓	radius/degree	$O(n^3)$	53) BRDLSMT	[130]

[1] Steiner tree heuristics are member node (MN) aware.

Table 10.1: Steiner tree heuristics ($\mathscr{A}_{\mathscr{S}}$) evaluated in the thesis (section 10.3).

One example is algorithm 25, which is the Steiner dl-OTTC algorithm. It uses dl-OTTC to construct a spanning-tree. A spanning tree algorithm spans every node in the input graph in its tree, therefore, the constructed spanning-tree is pruned for leaf Steiner points.

Algorithm 25 SDL-ONE-TIME-TREE-CONSTRUCTION

```
In: $G = (V, E, c)$, a set $Z \subset V$, diameter-bound $\geq 0$, degree bound $deg(v) \in \mathbb{N}$ for each $v \in V$
Out: Steiner Tree $T_Z = (V_Z, E_Z)$
  $T_Z$ = DL-ONE-TIME-TREE-CONSTRUCTION($G$, diameter-bound)
  Delete all leaf Steiner points from $T_Z$.
```

The spanning-heuristic variations in the thesis use dl-OTTC, dl-RGH, mddl-OTTC and dl-SPT as the input spanning tree algorithm $\mathscr{A}_{\mathcal{T}}$. Chapter 9 describes all of these spanning tree algorithms in detail. The spanning-heuristic variations are briefly described next.

- *Steiner degree limited one-time tree construction* (sdl-OTTC) is a spanning tree heuristic that addresses the BDDLSMT problem in this paper. sdl-OTTC is exactly dl-OTTC presented in [131] which is a modified OTTC [1].
- *Steiner degree limited randomized greedy heuristic* (sdl-RGH) is a spanning tree heuristic that addresses the BDDLSMT problem in this paper. sdl-RGH is exactly dl-RGH prestended in [131] which is a modified RGH [104].
- *Steiner minimum diameter degree-limited one-time tree construction* (smddl-OTTC) is a spanning tree heuristic that addresses the Steiner-MDDL problem in this paper. smddl-OTTC is exactly mddl-OTTC presented in [131] which is a modified OTTC [1].
- *Steiner degree-limited shortest-path tree* (sdl-SPT) [95] is a spanning tree heuristic that addresses the Steiner-MRDL problem in this paper.

10.3 Group communication simulations of Steiner tree algorithms

From the spanning-tree algorithms in chapter 9.3 we now investigate the Steiner-tree algorithms that were rigorously introduced in the previous sections. Table 10.1 provides brief and tabulated information regarding the Steiner-tree heuristic that are evaluated.

10.3.1 Target metrics

A Steiner tree algorithm is considered good if it can produce overlays with a low diameter, within a reasonable time. In the evaluations, we therefore investigate the *diameter*, which expresses the worst-case latency between any pair of group members. In addition, we investigate

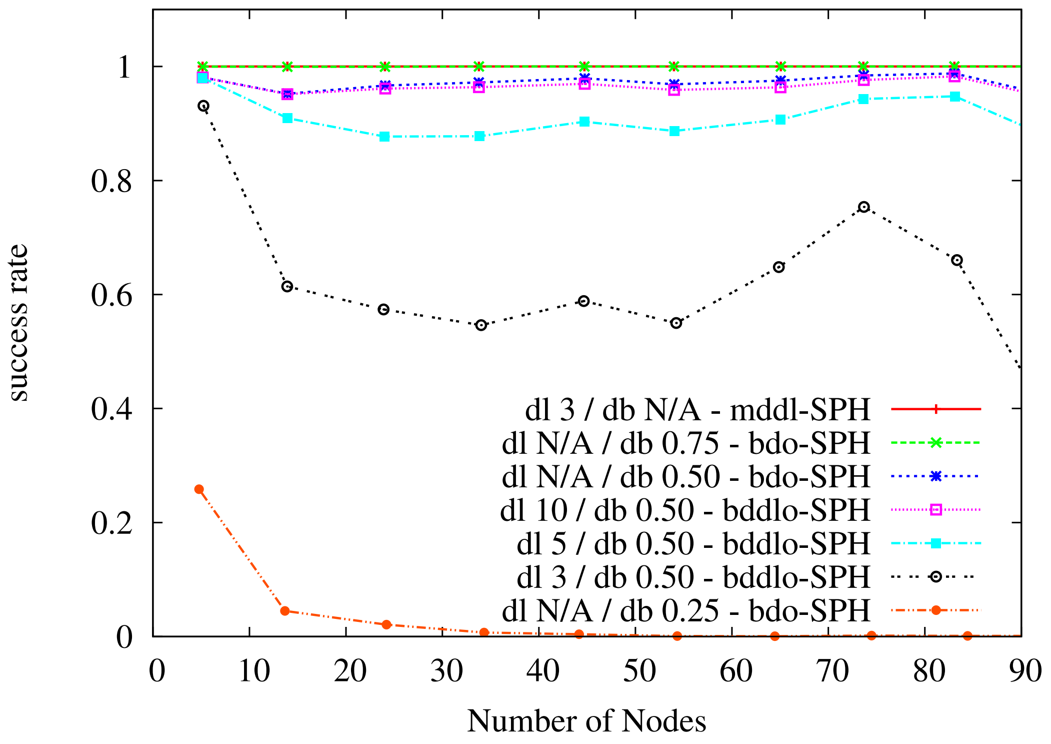


Figure 10.1: Success rate of constrained algorithms subject to varying degree limits and diameter bounds.

the *execution time* of an algorithm, which is the time that is required to execute a group membership change. It must be low such that a centralized group manager can handle group dynamics and reconfigure a group tree quickly. In addition, we address client side stress issues, which in our graph theory approach is the *degree* in a constructed tree. We investigate graph algorithms that can limit the degree of clients in a tree structure.

10.3.2 Algorithm constraints

Algorithms that take several target metrics into account often do this by choosing one metric as its optimization goal, and then address the remaining metrics by adding constraints. In general, adding constraints to an algorithm increases the algorithm complexity if an optimal solution is targeted. Many constrained tree heuristics cannot guarantee that a constrained tree is found. That is also the case with the constrained tree heuristics in this paper. The *success rate* of an algorithm depends on the constraint and the input graph. For example, it is more difficult to find a degree limited tree in a *sparse* graph than in a *dense* one.

Figure 10.1 plots the *success rates* of selected Steiner tree heuristics given a fully meshed graph. The heuristics are subject to varying degree limits (dl) and diameter bounds (db). Heuristics with degree limits as only constraint, always find a tree when given a fully meshed graph and a degree limit > 1. A tree has a minimum of two leaf nodes at all times, in case of the tree being a snake. Therefore, in a fully meshed graph, the leaf node always has an available degree and a link to all other member-nodes, such that it can continue building the tree. As expected, we see that the success rate for the Steiner-MDDL heuristic mddl-SPH is 100 % for degree limit 3.

Heuristics with diameter bounds do face situations in which it is impossible to find a tree within the diameter bound, even when given a fully meshed graph. The worst-case edge in a fully meshed graph is an approximation to the lowest diameter bound possible for any diameter bounded heuristic. We can see from figure 10.1 that when the diameter bound is 0.250 second the success rate of the BDSMT heuristic bdo-SPH is pretty much 0 % for group sizes above 40. While, a diameter bound of 0.750 gives a success rate of 100 % for the same heuristic. When degree limits are added to the diameter bounds, we expect the success rate to drop. We see that for the BDDLSMT heuristic bddlo-SPH, a degree limit of 10 and a diameter bound of 0.750 still gives a success rate of 100 %. However, when the diameter bound is dropped to 0.500 second failures do occur, and the success rate is on average 95 %. When degree limits are added the success rate continues to drop.

To summarize, the Steiner-MDDL heuristics always find a tree when given a full mesh. But, the diameter bounded heuristics fail when given a diameter bound that is less than the worst-case edge in the input graph. For diameter bounded and degree limited heuristics it is even more difficult to find a constrained tree. When a heuristic fails to find a constrained tree, the options are to i) rebuild the tree from scratch with relaxed constraints, ii) abandon the constraints and add the remaining member-nodes through some shortest paths, or iii) relax the constraints dynamically while building the tree.

In our application scenario it is not an option to rebuild the tree from scratch, as it may potentially take a very long time. Furthermore, we do not want to completely abandon the constraints, because they are among our target metrics. Rather, we relax the constraints dynamically whenever a heuristic cannot continue the tree construction process. A low diameter is a target metric, such that in our simulations we use a strict diameter bound of 0.250 to the diameter bounded algorithms. The heuristics are then frequently forced to relax the bound.

When a degree unlimited Steiner-MDST algorithm is applied to a full mesh made of shortest paths, the algorithm includes at most one Steiner point in the Steiner-tree, i.e., the Steiner point that is closest to the center. However, for a Steiner-MDDL algorithm the number of Steiner points added to the tree depends on the degree limits. Hence, given a full mesh of shortest paths, it is enough for a Steiner-MDST algorithm to find the one node that is closest to the center and connect it to the remaining member-nodes. While, for a Steiner-MDDL algorithm the number of Steiner points that is included in the input graph is a function of the degree limits.

Description	*Parameter*
Placement grid	100×100 milliseconds
Number of nodes in the network	1000
Degree limits	3,5 and 10
Super-nodes found by k-Center(k)	$k = 100$
Diameter bound	250 milliseconds
Core node set size	(group-size/degree-limit)

Table 10.2: Experiment configuration.

10.3.3 Experiment configurations

In the experiments, the complete group-graphs and the pruned group graphs include k Steiner-points. The aCL and aCLO algorithms are given a core node set of k Steiner points to create pruned group graphs. The Steiner-points are added to the input graph to enable the Steiner tree algorithms to reduce the diameter and increase their success rate. We use equation 8.1 to calculate the number of Steiner-points to include. More specifically, we use the degree limit d in the current experiment and the current group size $|V|$: $|O| = |V|/d$. The function approximates the number of Steiner-points that are needed to ensure that the degree-limited tree algorithms are able to build a tree. The k Steiner points are chosen by k-Median(k) from 100 Steiner-points (super-nodes) that are identified at the beginning among the 1000 nodes in the overlay network. These well-placed super-nodes (chapter 8) are found by the multiple core-node selection algorithm k-Center($k = 100$). If not otherwise noted, we use k-Median($k = 1$) to select a source node for every Steiner-tree heuristic (see chapter 7.6.1). The experiment configurations are summarized in table 10.2.

10.3.4 Steiner tree heuristic limitations on a full mesh

In a full mesh that is built as a shortest path graph, the shortest paths in the graph are all direct links. Running SPH and DNH on a full mesh of shortest paths has a major impact on their performance. Both heuristics run Dijkstra's SPT for each member node ($z \in Z$) and use the shortest path information when they build a tree. The only way of including a Steiner point is if the shortest path information contains one. In a full mesh built of shortest paths it is impossible for the shortest path information to contain anything other than a single hop, as long as ties are broken to the smaller number of hops. Therefore, SPH and DNH *fail* as Steiner tree heuristics when the input graph is a full mesh built of shortest paths. The shortest path information is available in $O(1)$, and removes the p SPT computations in SPH and DNH. Further, the final trees in both SPH and DNH are built exactly like Prim's MST, which is $O(n^2)$. Hence, in a full mesh SPH and DNH are reduced to an MST algorithm. In fact, any Steiner tree path-heuristic and tree-heuristic that solely uses shortest path information fails to be Steiner point aware when the input graph is a full mesh built of shortest paths.

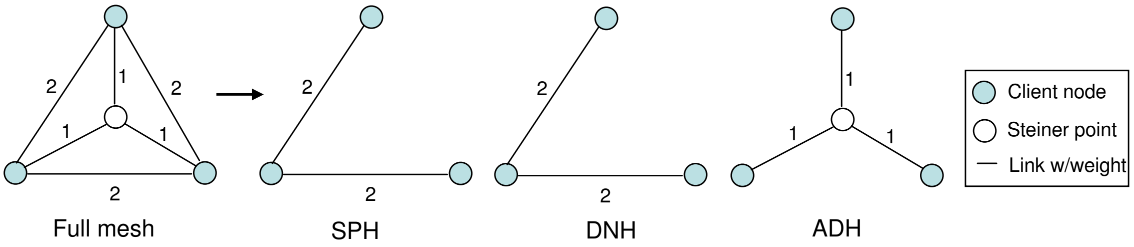


Figure 10.2: Full mesh as input to SPH, DNH and ADH.

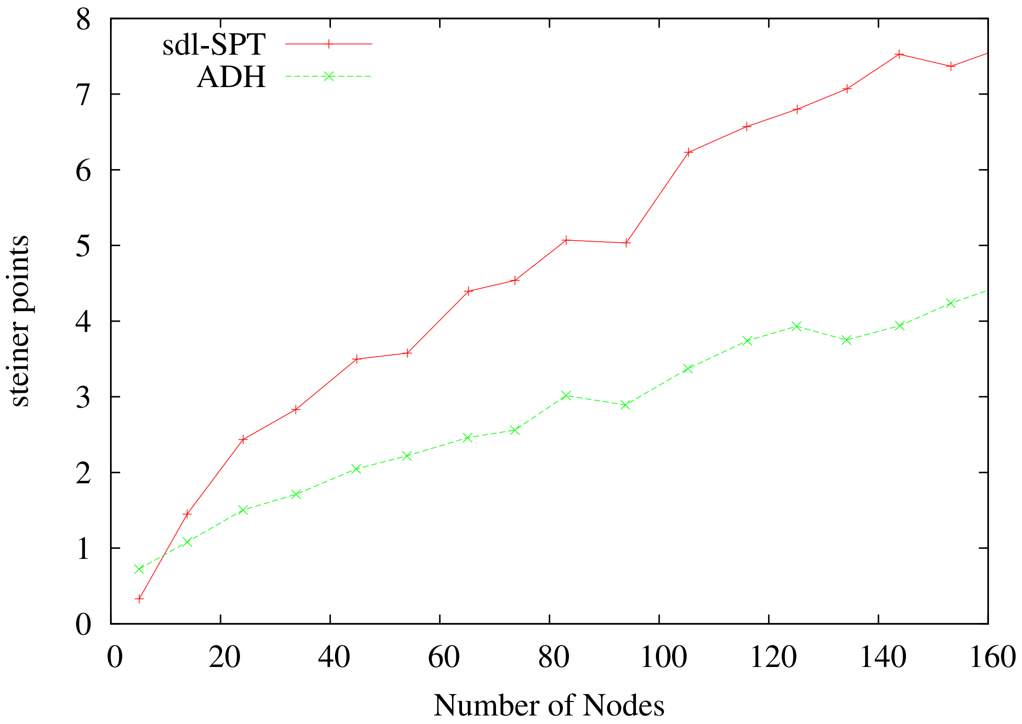


Figure 10.3: Average number of Steiner points in the group trees when a fully meshed graph is input (degree limit=10).

From these observation we can deduce that it is *incomplete* to only consider shortest paths in a fully meshed shortest path graph. Rather, a Steiner tree heuristic must consider triangulation properties when building the Steiner tree, such that it is possible to add a Steiner point. The vertex-heuristics, for example ADH, typically consider the location of each node in the graph, including the Steiner points, in relation to other member-nodes. That way, centrally located Steiner points may be included in a tree. When ADH is run on a full mesh built of shortest paths the computation of all-pairs shortest paths ($O(n^3)$) is avoided. Therefore, the worst-case time complexity of ADH is reduced to $O(n^2 * log(n))$. Figure 10.2 illustrates an example in which SPH and DNH find the MST and ADH the SMT.

Figure 10.3 plots the average number of Steiner points in the group trees when a fully meshed graph is input. We found that no Steiner tree heuristic derived from SPH and DNH

	Full mesh		*Reduced mesh*	
Algorithm	*Complexity*	*Performance*	*Complexity*	*Performance*
SPH	$O(n^2)$	Ω(MST)	$O(pn^2)$	Ω(SMT)
DNH	$O(n^2)$	Ω(MST)	$O(pn^2)$	Ω(SMT)
ADH	$O(n^2 * log(n))$	Ω(SMT)	$O(n^3)$	Ω(SMT)

Table 10.3: Algorithm performance.

finds a Steiner point to include in the tree. We can conclude that our previous observations are correct, and that it is useless to apply Steiner tree heuristics derived from SPH and DNH to a fully meshed graph built from shortest paths. The only Steiner tree heuristic that finds Steiner points is ADH. As previously described, ADH optimizes for the total cost and is based on Kruskal's MST algorithm. Tree algorithms that are based on Kruskal's MST start with a forest of trees and connect them until there is only one left. For algorithms that aim at a low or bounded diameter it is not possible to use algorithms based on Kruskal's MST unless there is some global knowledge, or some reference points between the subtrees while they are built. Every diameter reducing tree algorithm that the authors of this paper are aware of, starts from a given source and builds one tree sequentially using data structures that update the current diameter, radius and/or eccentricities. Based on these observations we conclude that ADH can not be adapted to reduce the diameter in its current form. The other algorithm that includes Steiner points to the tree is sdl-SPT, which is merely a dl-SPT algorithm that uses an input graph that includes Steiner points. Therefore, one approach may be to use conventional spanning tree algorithms, add Steiner points to the input graph and remove the Steiner points with degree one (leaves) from the tree. Table 10.3 summarizes our performance observations regarding SPH, DNH and ADH on a full mesh built of shortest paths compared to a reduced mesh.

10.3.5 Fully meshed results

We have seen that the diameter heuristics derived from SPH and DNH cannot include Steiner points, hence they cannot perform better than the spanning tree algorithms they use as algorithm base. In the following, we present results from some selected spanning tree algorithms that are given a fully meshed input graph. Figure 10.4 plots the diameter of trees using sdl-SPT and dl-SPT with varying degree limits. The only difference is the input graphs, where sdl-SPT was given a number of Steiner points. We can see that sdl-SPT performs better than dl-SPT. The reasons are that, first of all, in our experiments the Steiner points that are added to the input graph are selected using the k-Median(k) algorithm from the group-center (see chapter 7). Hence, if a Steiner point is added to the tree it is most likely located somewhere in the group center. Secondly, the added Steiner points increases the degree capacity centrally. Finally, the combination of Steiner point location and capacity helps sdl-SPT to create trees with lower diameter than the dl-SPT with no Steiner points. The remaining spanning tree algorithms are

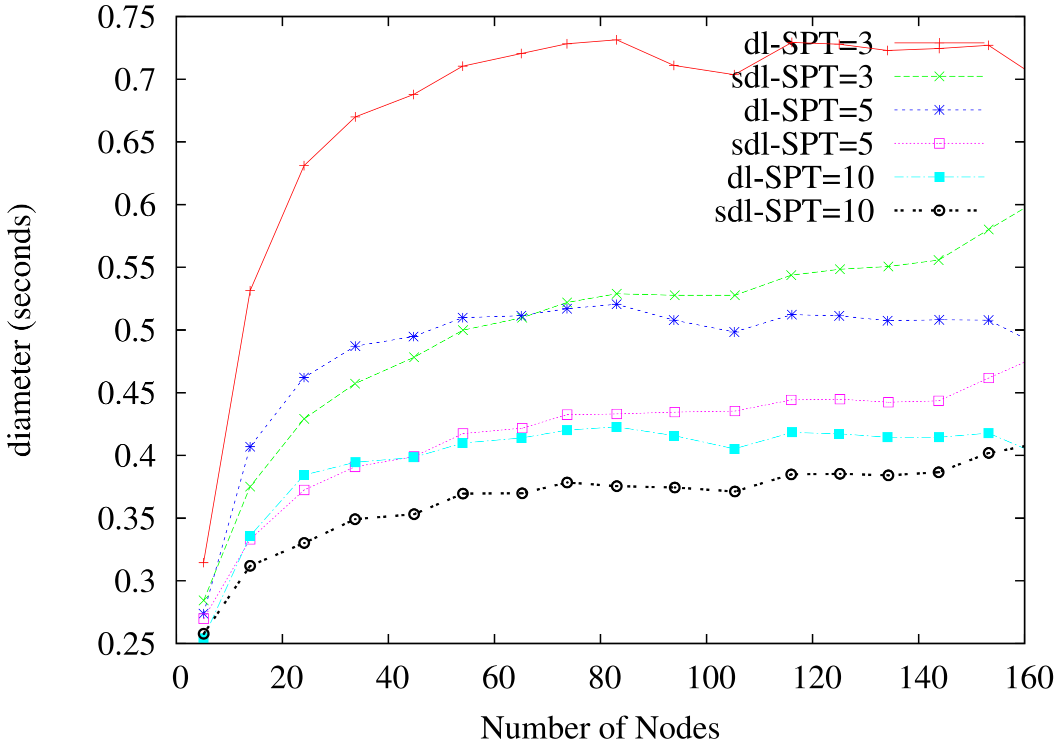


Figure 10.4: Diameter (seconds) of sdl-spt and dl-spt.

plotted in figure 10.5. We observe the same tendency for all of the spanning tree algorithms. Figure 10.6 plots the hop-diameter, and we can see that the number of hops does increase slightly as a result of adding Steiner points to the input graph. This is to be expected when more nodes are included in a tree. Figure 10.7 plots the average number of Steiner points in the trees for sdl-SPT and smddl-OTTC and varying degree limits. The number of Steiner points is quite high when the degree limit is 3, but, as we saw above, is actually reducing the diameter.

The main observation is that the spanning tree algorithms produce trees with lower diameter (seconds) when given a set of Steiner points in the input graph, while the hop-diameter increases slightly. It follows from the geometric version of the Steiner-MDDL and Steiner-MDST, that if we had an infinite amount of Steiner points located close to the group center the degree-limited heuristics would perform as well as their degree-unlimited versions. In fact, as long as the diameter among the Steiner points is less than the diameter among the member-nodes, new Steiner points can be added without increasing the member-node diameter. Of course, we can not assume that we have that many Steiner points available. It is inevitable that new Steiner tree heuristics must be designed for the degree-limited diameter- and radius-related Steiner tree problems listed in section 4.6.

Figure 10.9 plots the *execution time* of the selected algorithms. The execution times are lowest for sdl-SPT and highest for smddl-OTTC. However, they are all faster than 10 milliseconds for group sizes up to 120 member-nodes (see figure 10.8). The added Steiner points

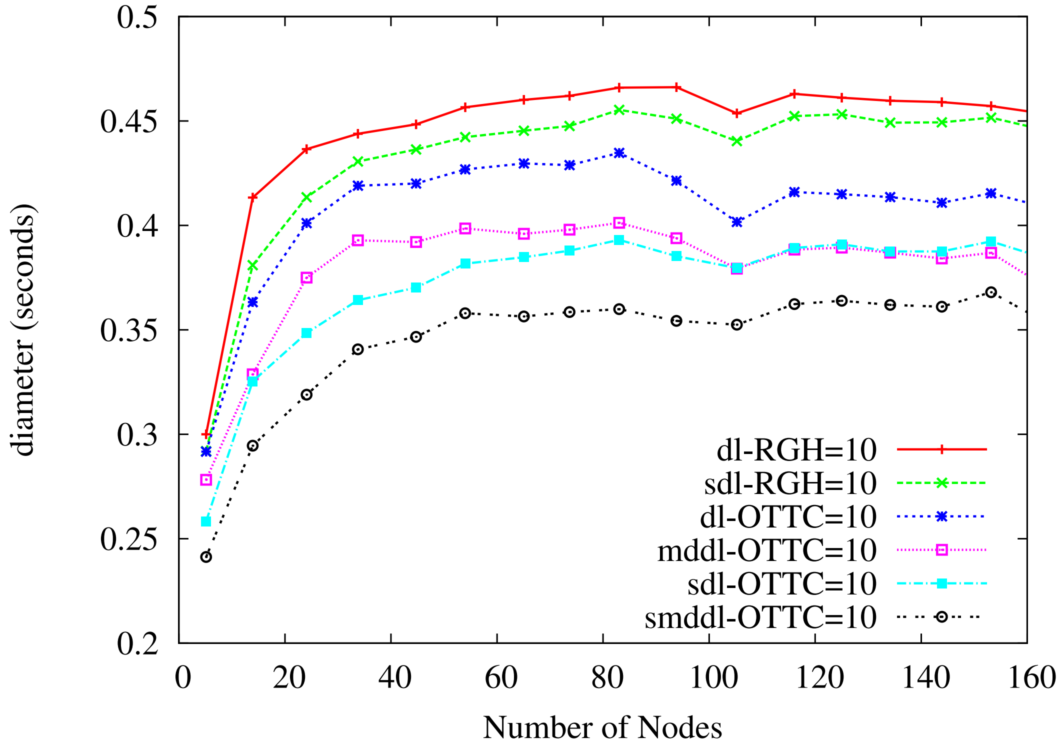


Figure 10.5: Diameter (seconds) of spanning tree algorithms with and without Steiner points (degree limit 10).

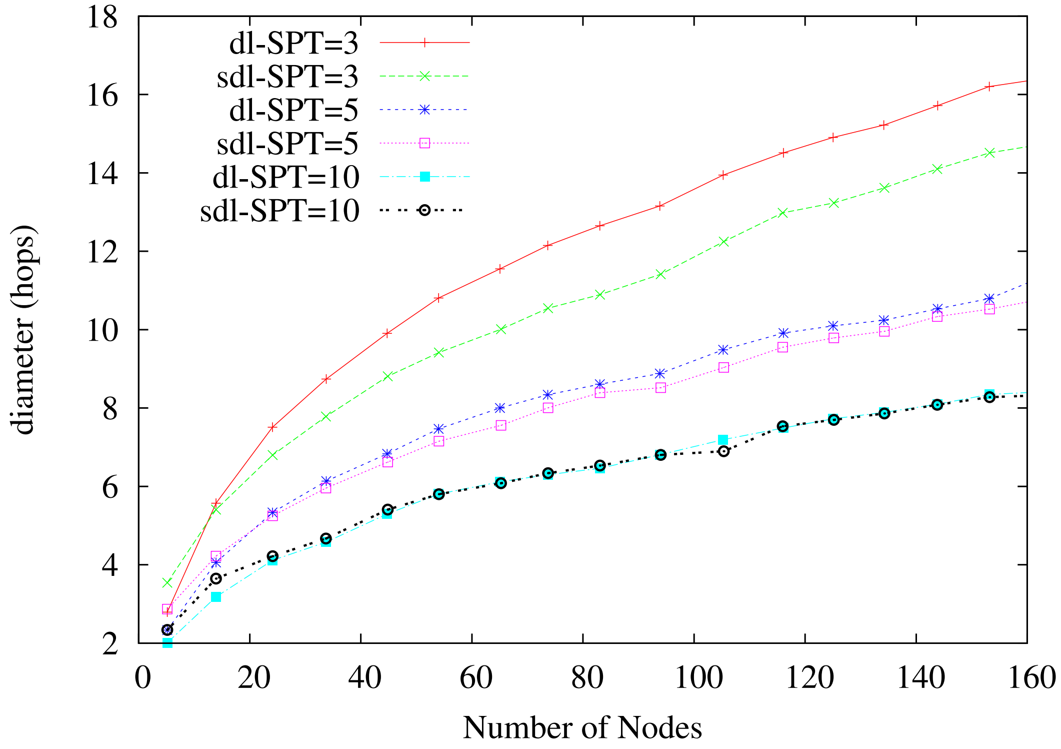


Figure 10.6: Diameter (hops) of sdl-spt and dl-spt.

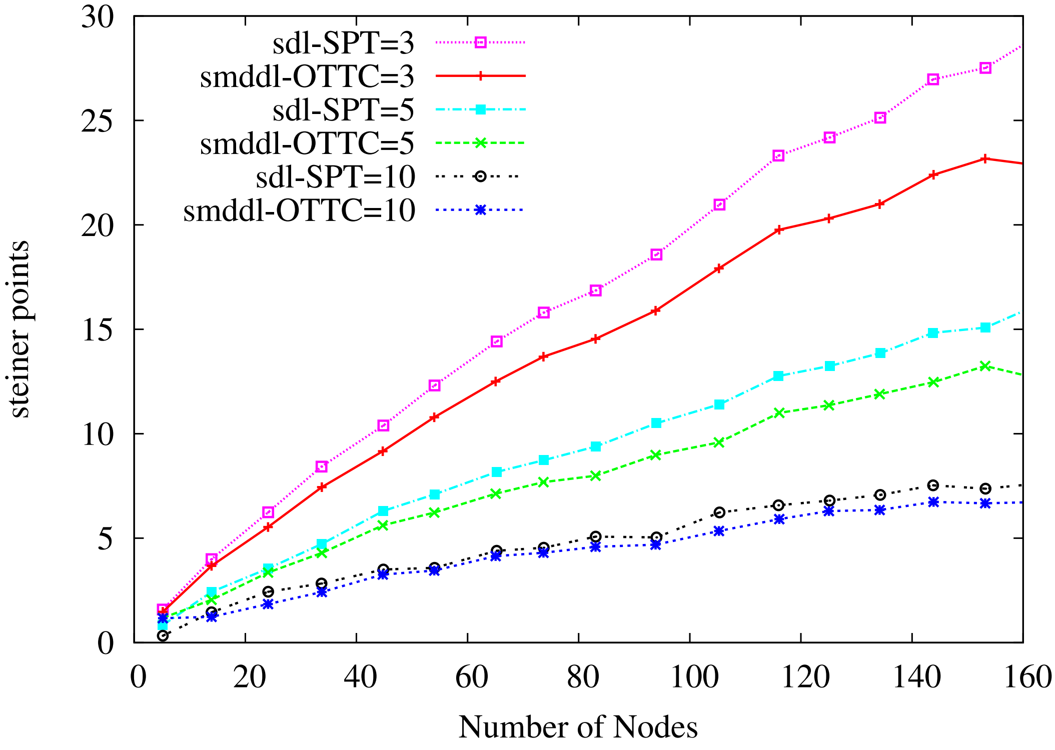


Figure 10.7: Average number of Steiner points in trees (degree limits 3, 5 and 10).

do not significantly increase the execution times of the algorithms. Hence, we can add Steiner points without noticable penalties to our target metrics.

To summarize, the Steiner tree heuristics derived from SPH and DNH should not be used on a full mesh of shortest paths because they do not perform better than any given spanning tree algorithm. The spanning tree algorithms we tested instead achieve smaller diameters with slightly increased hop-diameters when Steiner points are added to the input graph. The main reasons for this improvement are that Steiner points are found using the k-Median(k) core-node selection algorithm and that they increase the degree capacity centrally. We also observed that the *execution time* does not suffer noticably when the number of edges and nodes in the input graph is increased.

10.3.6 Discussions for fully meshed results

The spanning tree algorithms that we tested are evaluated with respect to our target metrics in table 10.4. sdl-RGH is the fastest algorithm, but produces trees with the highest diameter. smddl-OTTC and sdl-OTTC are similar to each other, but smddl-OTTC is slightly slower and does not have the flexibility of a bounded-diameter algorithm. sdl-SPT was a surprisingly good alternative. It is a good algorithm for a source-based tree, and when the source is located in the group center sdl-SPT builds good shared trees as well.

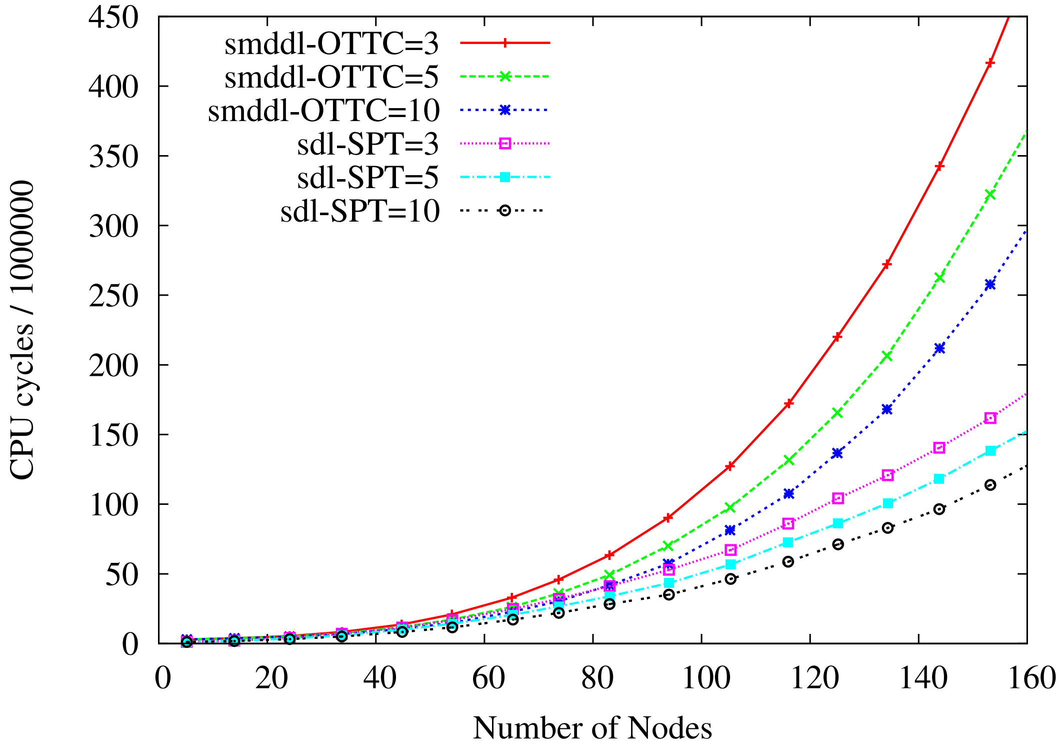


Figure 10.8: Execution time of sdl-spt, dl-spt and mddl-OTTC (group size 120).

Algorithm	*Diameter*	*Time*	*Degree*	*Rank*
sdl-OTTC	+++	++	+	++++
sdl-SPT	++	+++	+	+++
smddl-OTTC	++++	+	+	++
sdl-RGH	+	++++	+	+

Table 10.4: Tree algorithm characteristics using full mesh.

Our ranking is subjective and not related to specific application needs. All the algorithms fit different needs, and figure 10.9 shows that they vary in performance between diameter and reconfiguration time. sdl-RGH is a fast $O(n^2)$-heuristic. When extending the tree, it chooses the next vertex at random and connects it via the lowest-weight edge that maintains the diameter constraint. The diameter constraint is only maintained towards the source, and is actually the radius. The algorithm works surprisingly well to produce trees with a relatively small diameter. sdl-OTTC extends the tree through the minimum-weight edge that obeys the diameter bound. It is slower than sdl-RGH because it performs a more time consuming maintenance of the diameter, but it produces trees with smaller diameter. smddl-OTTC always minimizes the maximum diameter, and is therefore even slower. sdl-SPT avoids diameter bounds and doesn't minimize the diameter, either. For many applications a bound may not be known, and minimization may not be necessary. sdl-SPT rather searches for source destination shortest paths, which is often desired by streaming applications.

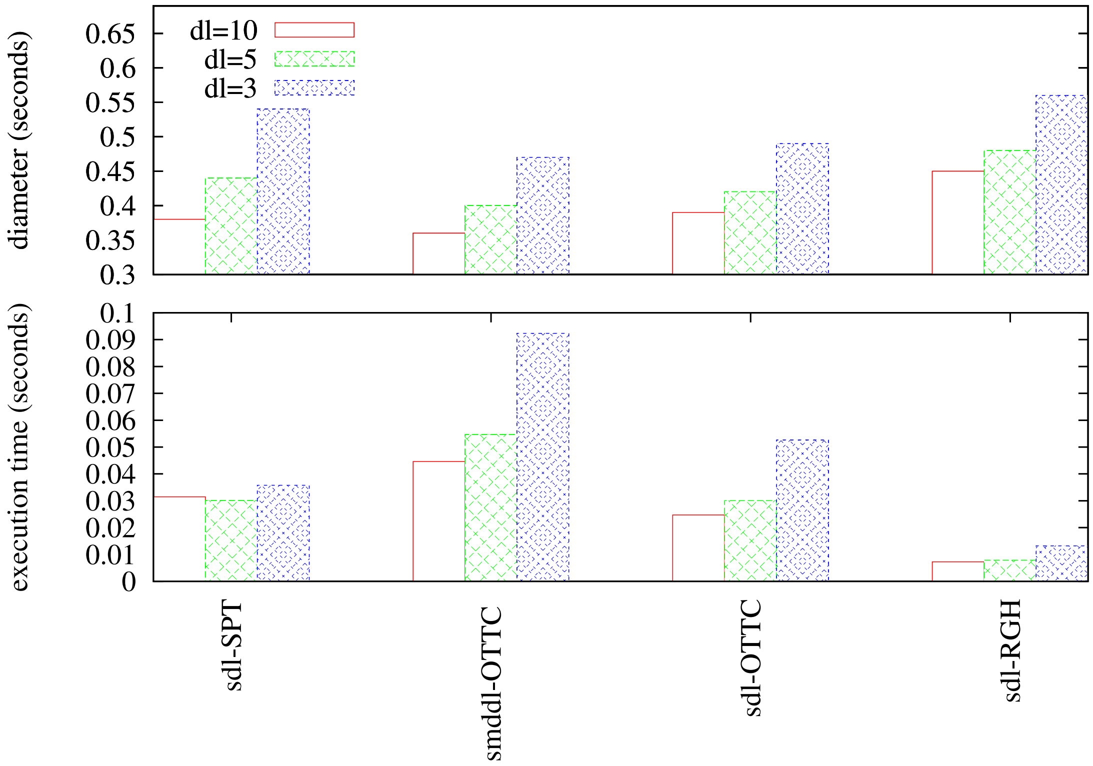


Figure 10.9: Diameter and execution time with degree limits 3, 5 and 10.

10.3.7 Reduced graphs results

We pointed out in chapter 8 that graph manipulation can save time in the construction. In the following, we present results from combining the k-Median core-node selection algorithm and edge-pruning with tree algorithms. In this section, we use aCL and aCLO ($k = 2, 1, 0$) to reduce the input graph size, and the k-Median core-node selection algorithm (see chapter 7.6.1) to find the Steiner points. All the plots use a group size of 120 nodes with a degree limit of 10. We have included results from using a full mesh as a reference point. s-SPT and sdl-SPT are used as the representative spanning tree algorithms.

Figure 10.10 compares the *diameter* achieved, as applied to a fully meshed graph and *aCL*. As expected, SPH, DNH, dl-SPH and dl-DNH all produce trees with high diameter because they optimize the total cost. However, we see that for aCL with $k = 0$ all the algorithms produce trees with a lower diameter, and the total cost algorithms are down to a diameter of around 0.500 seconds. The algorithms that produce the lowest diameter are the degree-unlimited algorithms bdo-DNH, md-DNH and s-SPT. Among the degree-limited algorithms bddlo-DNH, bddlr-SPH, mddl-DNH and sdl-SPT produce the lowest diameter trees. Overall, the diameter related heuristics produce the lowest diameter when using a fully meshed input graph. However, when aCL is used, the diameter suffers on average only 15 % even when $k = 0$, and the

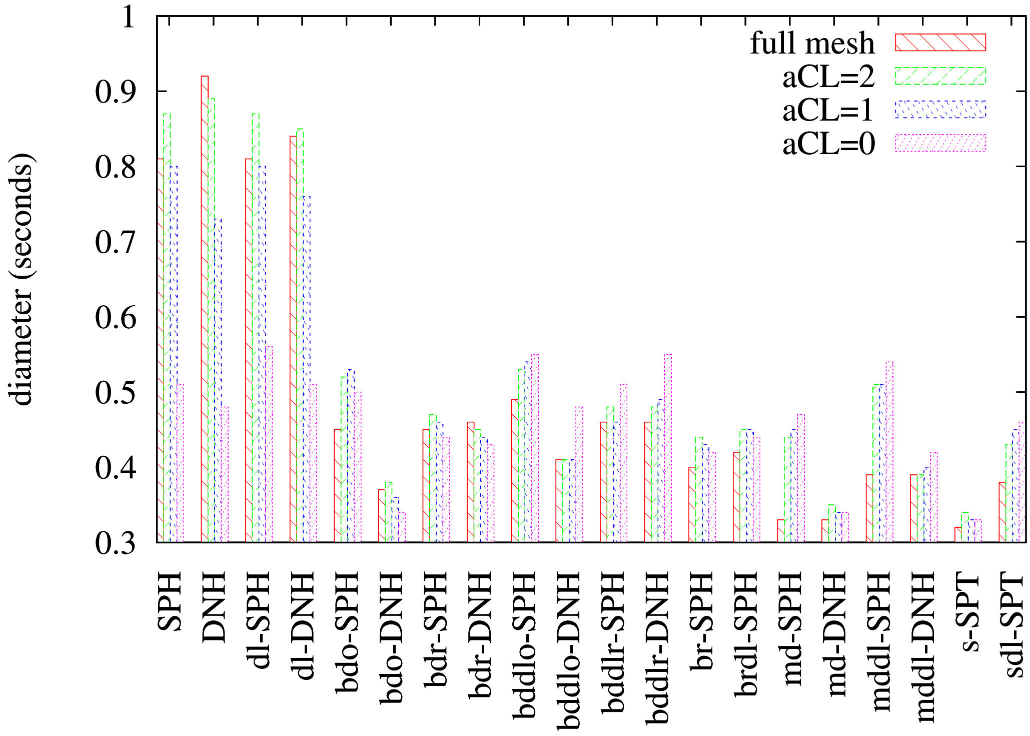


Figure 10.10: Diameter (seconds) for *a*CL.

edge set is reduced with 80 %, compared to the fully meshed graph.

Figure 10.11 plots the *maximum degree* in the trees when using *a*CL to reduce the graph. We observe that the algorithms with a high maximum degree do all produce trees with very low diameter. s-SPT constructs the trees with the lowest diameter. We observe that it consistently constructs trees that resemble a star because the maximum degree is approximately the group size. Furthermore, the degree-limited algorithms do all construct trees within the maximum degree limit of the core nodes (Steiner points). From these observations, we deduce that a Steiner tree heuristic that optimizes the diameter must be able to exploit the degree capacity of centrally located nodes. However, in our application scenario the degree capacity is often limited, such that the degree-unlimited algorithms with low diameter and high maximum degree are not really an alternative. From figure 10.12 we see the average number of leaf nodes when using *a*CL. We observe that that when $k = 0$ every member-node is forced to be a leaf node, regardless of location. Hence, the degree of the member nodes is one, and the stronger core nodes (Steiner points) is higher.

Figure 10.13 plots the diameter for *aCLO* as well. We observe that the diameter suffers on average just below 20 % when *a*CLO is used, instead of the full mesh. *a*CLO with $k = 0$ reduces the edge set by 95 %, and the construction results are still competitive. Furthermore, figure 10.14 shows the hop-diameter. It is interesting that the hop-diameter is, on average, reduced when *a*CLO is applied. A low hop-diameter is often desirable in peer-to-peer file sharing applications.

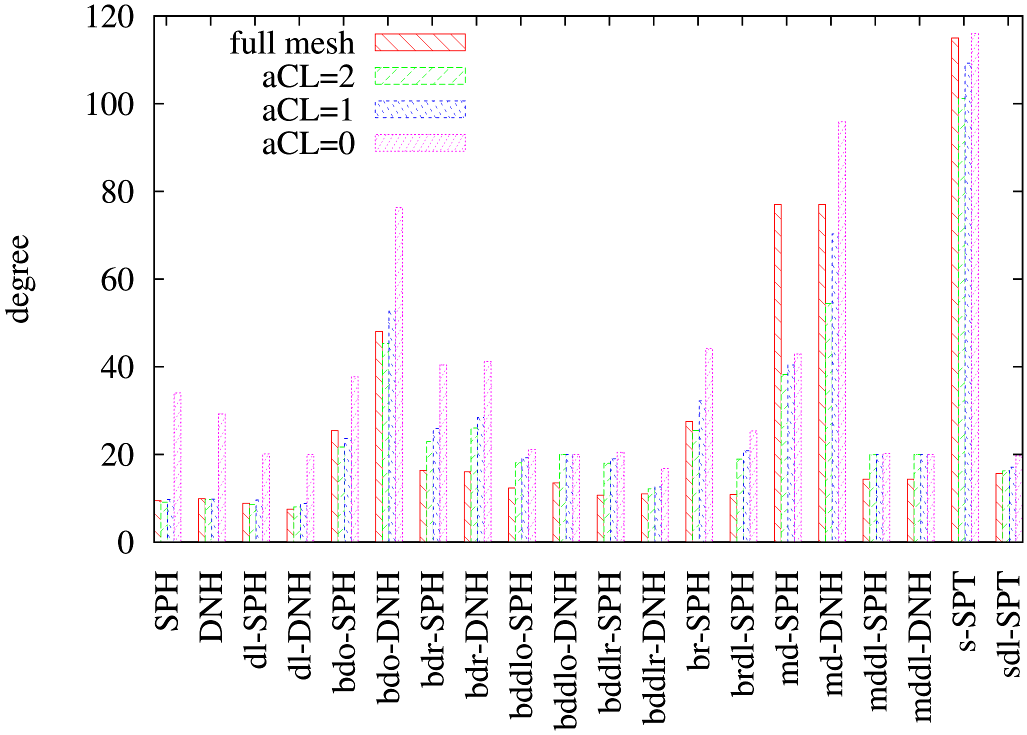


Figure 10.11: Maximum degree when using full mesh and *a*CL.

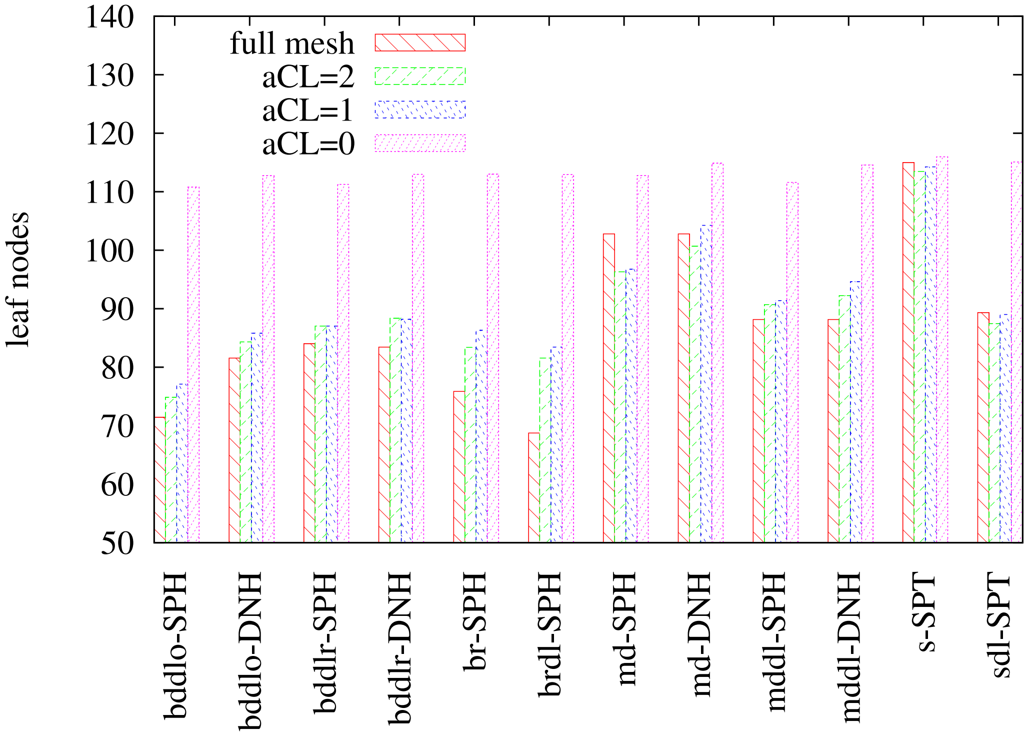


Figure 10.12: Average number of leaves (member-nodes) when using *a*CL.

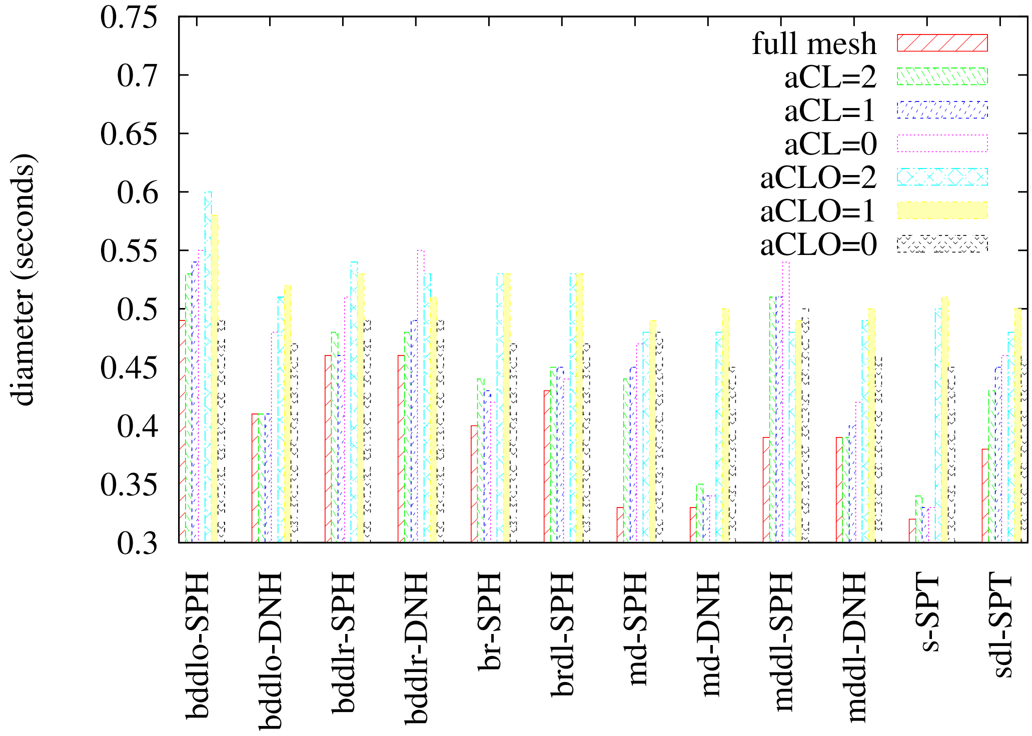


Figure 10.13: Diameter (seconds) for *a*CL and *a*CLO.

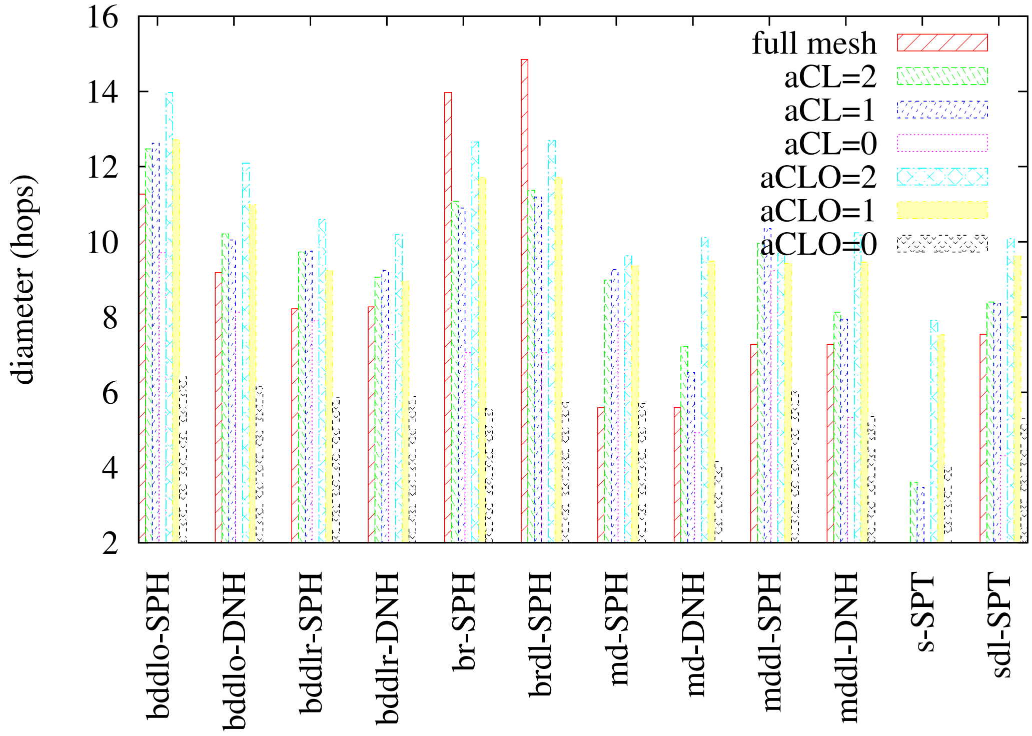


Figure 10.14: Diameter (hops) for *a*CL and *a*CLO.

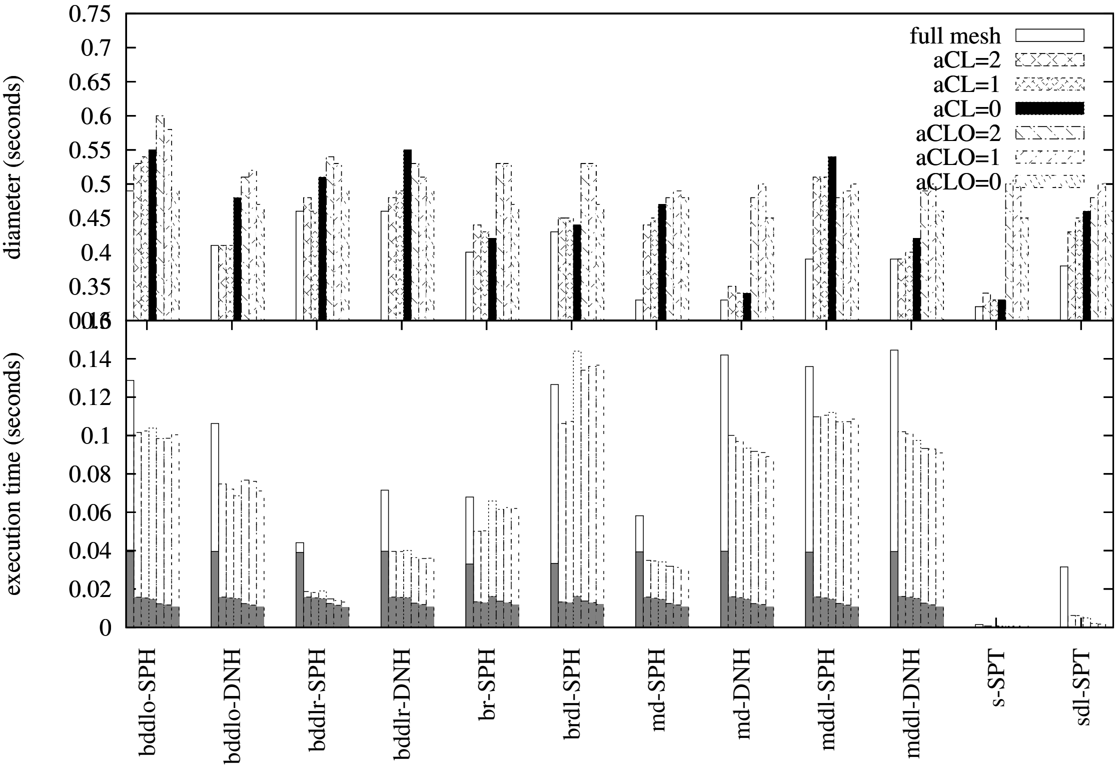


Figure 10.15: Diameter and execution time with the overhead of computing the shortest paths (grey scale), for full mesh, *a*CL and *a*CLO.

The diameter and the reconfiguration times of the construction algorithms applied to *a*CL and *a*CLO graphs are plotted in figure 10.15. The time that is consumed by the p SPT computations are highlighted in grey scale for each execution time bar. Generally, the pruning algorithms have a marginally positive effect on the reconfiguration time. As expected, the fastest algorithms are the spanning tree algorithms, here represented by s-SPT and sdl-SPT, both of which execute in $O(n^2)$. Among the Steiner heuristics, the fastest are the randomized heuristics bddlr-SPH and bddlr-DNH that both use dl-RGH as their algorithm base. The reconfiguration times for these randomized SPH and DNH variations are overshadowed by the computation of the p SPTs. The remaining algorithms are comparatively quite slow and all have a complexity of $O(n^3)$. It is intersting that there is no clear correlation between high reconfiguration time and low diameter. However, mddl-DNH does produce the lowest diameter at the price of high execution time.

We expect the *maximum degree* to decrease for the algorithms without degree limits when applying *a*CL and *a*CLO. In figure 10.16 we observe that the maximum degree is reduced to about 20 when *a*CLO is used. Hence, degree-unlimited algorithms are an option for very low bandwidth streams, but only if *a*CLO and the group-center heuristic are used to manipulate the input graph. However, for all of the degree-unlimited algorithms there are (almost) equally fast algorithm versions with degree limits.

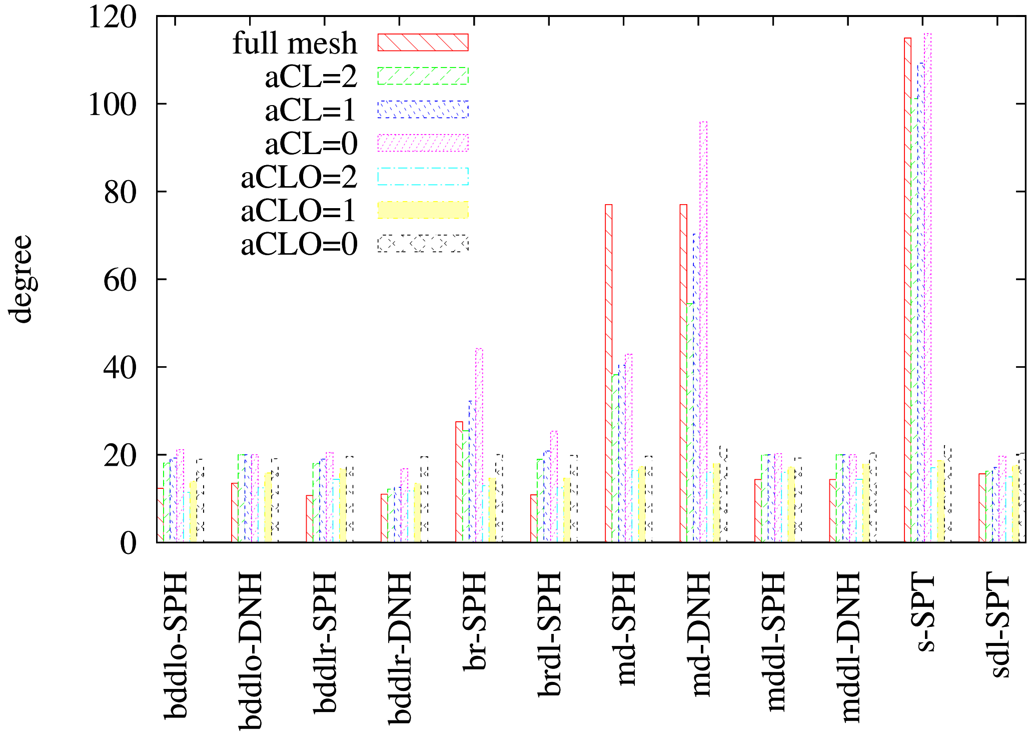


Figure 10.16: Maximum degree for full mesh, aCL and aCLO.

10.3.8 Discussions for reduced graphs results

The Steiner tree heuristics are applied to reduced graphs because we want to see the effect on the diameter and reconfiguration time. The reduced graphs are built by combining the k-Median algorithm, which found the Steiner points, and the pruning algorithms aCL and aCLO , which created a reduced group graph. The results from the reduced graphs showed that the SPH and DNH based Steiner tree heuristics cannot outperform the naive approach of running a spanning tree algorithm on the same input graphs (with Steiner points). Among the degree-unlimited algorithms, s-SPT outperformed the minimum-diameter algorithms (md-DNH and md-SPH) both in terms of diameter achieved and reconfiguration time. Furthermore, among the degree-limited algorithms, the sdl-SPT algorithm performed similarly to the minimum-diameter degree-limited algorithm (mddl-DNH and mddl-SPH) but sdl-SPT is much faster.

We expected the reconfiguration times to decrease when aCL and aCLO were used, but the reduction was limited to about 30 %. However, if the shortest paths are precomputed the randomized algorithms based on RGH proved to be very fast. Table 10.5 gives an overview of the Steiner tree heuristics performances when the shortest path information is available.

aCLO reduced the maximum degree and allowed thereby the feasible use of the degree-unlimited algorithms. Table 10.6 gives an overview. The pruning algorithms bound the stress on the member-nodes. For example, when $k = 0$ for aCL and aCLO, every group member is

Algorithm	*Diameter*	*Time*	*Degree*	*Rank*
md-SPH	+	−	+	−
bdo-SPH	+	−	+	−
bdr-SPH	+	+	+	+
br-SPH	+	−	+	−
mddl-SPH	+	−	+	−
bddlo-SPH	+	−	+	−
bddlr-SPH	+	+	+	+
brdl-SPH	+	−	+	−
md-DNH	+	−	+	−
bdo-DNH	+	−	+	−
bdr-DNH	+	+	+	+
br-DNH	+	−	+	−
mddl-DNH	+	−	+	−
bddlo-DNH	+	−	+	−
bddlr-DNH	+	+	+	+

Table 10.5: Steiner tree heuristic characteristics assuming shortest path information.

a leaf node (degree one), and all the stress is put on the core nodes (Steiner points) that are assumed to have a higher capacity.

10.3.9 Discussions for all results

A tree algorithm for our construction process should produce trees with low diameter, keep the reconfiguration time fast and be able to obey degree limits. We have seen that the mddl-DNH algorithm produces trees with low diameter within the degree limits. However, the reconfiguration time is very high compared to the simple but efficient sdl-SPT algorithm. Remember, low reconfiguration time is particularly desirable during frequent tree updates, which is often the case for our target applications.

The common denominator for every SPH and DNH based Steiner tree heuristic we tested is the high reconfiguration time, which is largely due to their shortest path computations and the increased complexity of the data structures. For the full mesh, the shortest paths are given but the Steiner tree heuristics failed to include any Steiner points. The Steiner tree heuristics did work when applied to reduced graphs, but then the shortest paths must be computed. In a centralized approach to group membership management the central entity has access to the global graph and every group graph. It is therefore possible for the central entity to pre-compute the shortest paths, which would reduce the reconfiguration time for the Steiner tree heuristics quite significantly.

The Steiner tree heuristics in this paper have been derived from the SPH path-heuristic or the DNH tree-heuristic. Similar algorithms addressing the same Steiner tree problems have been derived from both heuristics. Overall, we saw a tendency favoring DNH as the most suitable algorithm base of the two. The DNH-based heuristics produced trees with a lower diameter, whereas the degree and reconfiguration time are all similar to the SPH-based heuristics.

Algorithm	*Diameter*	*Time*	*Degree*	*Rank*
md-SPH	+	−	+	−
bdo-SPH	+	−	+	−
bdr-SPH	+	+	+	+
br-SPH	+	−	+	−
mddl-SPH	+	−	+	−
bddlo-SPH	+	−	+	−
bddlr-SPH	+	+	+	+
brdl-SPH	+	−	+	−
md-DNH	+	−	+	−
bdo-DNH	+	−	+	−
bdr-DNH	+	+	+	+
mddl-DNH	+	−	+	−
bddlo-DNH	+	−	+	−
bddlr-DNH	+	+	+	+
s-SPT	+	+	+	+
sdl-SPT	+	+	+	+
sdl-OTTC	+	+	+	+
sdl-RGH	+	+	+	+
smddl-OTTC	+	+	+	+

Table 10.6: Tree algorithm characteristics assuming shortest path information and using *a*CLO.

To summarize our observations, none of the SPH and DNH based Steiner tree heuristics in this paper work as Steiner tree heuristics on fully meshed shortest path graphs. We saw that the spanning tree algorithms sdl-SPT and smddl-OTTC are very good alternatives when applied to a full mesh. When the input graphs are reduced using *a*CL and *a*CLO, the DNH-based heuristics performed better than the SPH-based heuristics. Overall, we deduce that DNH is better suited for adaptation to reduce the diameter than SPH. The main drawback of every SPH and DNH based Steiner tree heuristic on a reduced mesh is the necessity of computing the shortest paths and the increased complexity of the data structures that this implies. The shortest path computation is time-consuming and increases the reconfiguration time. Figure 10.17 shows the diameter (seconds) and execution time (seconds) for selected Steiner tree heuristics and spanning tree algorithms. We observe that their diameter is similar; however, the execution times of the Steiner tree heuristics are clearly higher. Our main conclusion is that the Steiner tree heuristics based on SPH and DNH are only suited for our application scenario if the membership dynamics is medium to low, and if the shortest path information is pre-computed and available on the server.

10.4 Summary of the main points

We have investigated group communication in relation to distributed interactive applications. Our investigation involved experiments with many Steiner tree heuristics, where our main target metric was a low tree diameter. We applied the heuristics to an application layer overlay

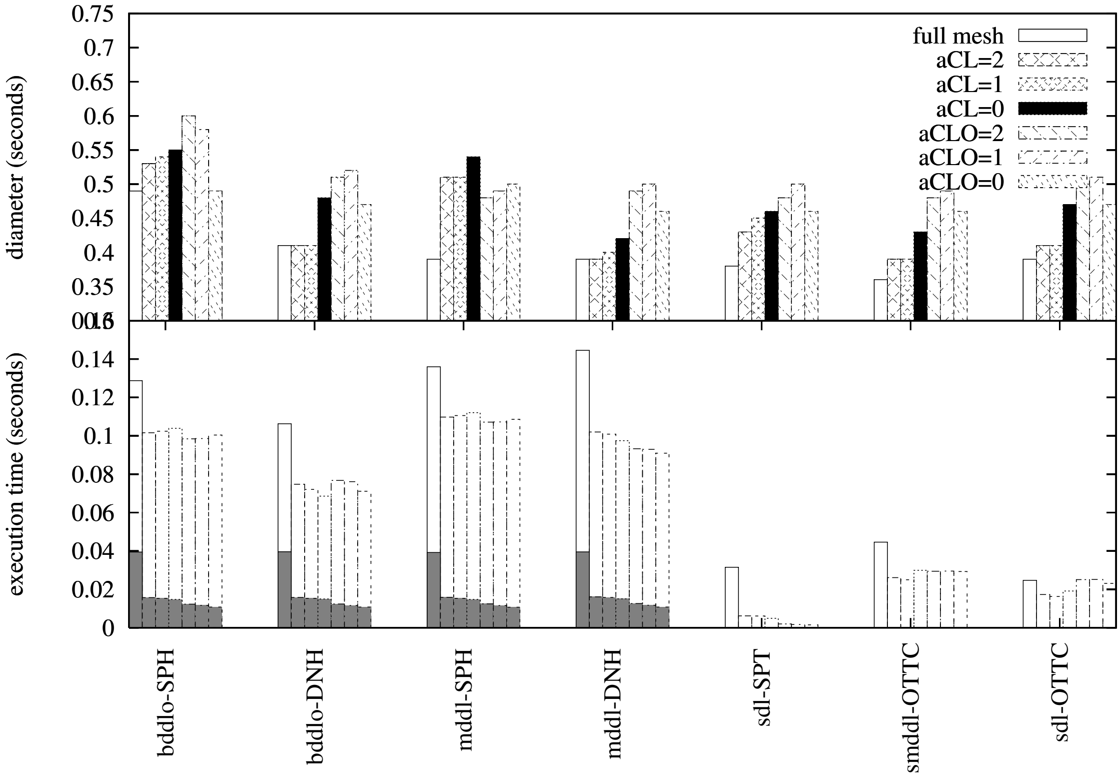


Figure 10.17: Diameter and execution time with the overhead of computing the shortest paths (grey scale), for full mesh, *a*CL and *a*CLO.

network where the network is always a full mesh. This should give a tree algorithm the optimal conditions for finding the best tree. However, we observed that the Steiner tree heuristics that should be aware of Steiner points and add the ones that help optimize the tree, failed to do so on a full mesh made of shortest paths. We then tested the spanning tree algorithms sdl-RGH, sdl-OTTC, smddl-OTTC and sdl-SPT using input graphs that included Steiner points. The results showed that these algorithms produced trees with a smaller diameter when Steiner points were included. Moreover, the heuristics are fast, which is important in highly dynamic distributed applications. However, a Steiner-heuristic that uses a spanning tree algorithm and then prunes leaf Steiner points is a naive heuristic and a simplistic approach to addressing the diameter related Steiner tree problems. Better Steiner tree heuristics should be designed that work on fully meshed graphs [62].

In addition, we investigated algorithms for reducing the time it takes to execute membership changes. We found that the k-Median core node selection algorithm, and the edge-pruning algorithms *a*CL and *a*CLO are powerful means to manipulate the input graph. However, the Steiner tree heuristics only reduced their execution time slightly even though the edge set was reduced by 95 % in the most extreme case. But, every Steiner tree heuristic and spanning tree algorithm still performed comparatively well in terms of the diameter.

In our simulations, the node layout was a square world with sides equal to 100 milli-seconds (approximately Europe). The diameter that the best Steiner-tree heuristics yielded was below 400 milli-seconds for tree sizes up to 160 and a degree-limit of 10. This is outside the requirements for first-person shooter games, but is just inside for most distributed interactive applications (see chapter 2). When the degree limit is less than or equal to 5 the latency requirements are not met. A degree limit above 5 is not a problem for multi-player online games, because the data streams are very thin [59]. However, for video/audio conferences it may be an issue due to somewhat limited bandwidth capacity on average clients in the Internet.

Chapter 11

Overlay construction techniques: Connected subgraph algorithms

The overlay network management introduced in section 5.4 includes overlay construction techniques whose task is to construct low-latency overlay networks for distribution of time-dependent events. In that respect, we continue our evaluation of overlay construction techniques and evaluate connected subgraph algorithms. By doing this we address a goal of the thesis:

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

A connected subgraph is a subgraph of a given graph, in which each vertex is connected with at least one path (definition 22). A subgraph may therefore be either acyclic (tree) or cyclic (mesh). The advantage of subgraph algorithms over tree algorithms is that they are not bound to create an acyclic subgraph, but are rather more "free" in the subgraph construction. However, in cases of highly connected subgraphs, the main drawback is the added cost to the network, which typically results in an increased bandwidth consumption.

We have implemented and experimentally analyzed tree heuristics and mesh construction heuristics, and compared their performance and applicability to distributed interactive applications [127]. Due to this time-dependent application scenario, we concentrate particularly on heuristics that minimize the pair-wise latency in overlays. The maximum pair-wise latency in an overlay is known as the diameter.

Our results show that trees can compete with meshes when it comes to computing low diameter overlays; however, the average pair-wise latency in meshes is lower. Generally, we found that trees are faster to construct and save considerable amounts of resources in the network. Meshes, on the other hand, increase the fault tolerance, but at the expense of increased resource consumption. Furthermore, we show that both mesh and tree heuristics yield vital properties for use in distributed interactive applications.

A connected subgraph algorithm belongs to the graph theoretical problems of connected subgraphs, which are thoroughly introduced in section 4.9 and 4.10. In this chapter, we introduce connected subgraph algorithms that address these spanning subgraph and Steiner subgraph problems. However, our focus is not on solving them exactly, since many of the problems are NP-complete. Rather, we focus on subgraph heuristics that approximate a solution which is good enough to be used by distributed interactive applications.

The rest of the chapter is organized in the following manner. Section 11.1 introduces the evaluated spanning-subgraph and Steiner-subgraph algorithms. Section 11.2 evaluates the spanning-subgraph algorithms through group communication simulations. Section 11.3 evaluates the Steiner-subgraph algorithms also through group communication simulations. Finally, section 11.4 gives a summary of the main points.

11.1 Spanning subgraph and Steiner subgraph algorithms

The main difference between a spanning and a Steiner subgraph is that a spanning subgraph of an input graph spans all the vertices of the input graph [144], while a Steiner subgraph of an input graph spans all the member-nodes of the input graph [90]. A Steiner subgraph may also include a number of Steiner points (see chapter 10 for Steiner-tree algorithms). In a group communication scenario, Steiner points are non-member-nodes that are not actively participating in the group defined by the membership management (section 5.2).

The following sections refer interchangeably to a connected subgraph as a subgraph or a *mesh*. As described in section 3.1, a cyclic mesh increases the node failure tolerance compared to a tree, because multiple paths to a node exist. However, unless some path routing is applied to a mesh it also introduces data redundancy because some nodes receive two copies of the same data. Data redundancy due to multiple paths may be valuable in cases of fluctuating link costs and to reduce the pair-wise latencies among the nodes.

Formally, a mesh algorithm $\mathscr{A}_{\mathscr{M}}$ takes as input a connected undirected weighted graph G, and constructs a connected undirected graph (mesh) $M = (V_M, E_M)$ on G, where $V_M = V$. Observe that a mesh can be constructed $M = G$ directly from the input graph. However, an application layer overlay network graph G is a complete graph (fully meshed), therefore, this is not a viable solution because the *total cost* of the mesh would be tremendous. The mesh algorithms should rather construct the mesh using the application's requirements, without being bound to the requirement of constructing a tree. The requirements of a distributed interactive application are especially linked to low pair-wise latencies (chapter 2). The following investigated mesh algorithms can be divided into the three categories:

- ***Interleaved-trees algorithms*** are introduced in section 11.1.1
- ***Enhanced tree algorithms*** are introduced in section 11.1.2
- ***Edge pruning (removal and addition) algorithms*** are introduced in section 11.1.3

For each category, we present algorithms found in the literature, but also propose new mesh algorithm variations based on ideas from these (see table 11.1). Many of the following mesh algorithms are applicable both as spanning and Steiner subgraph algorithms. The reason is that those algorithms take as input a tree algorithm $\mathscr{A}_{\mathscr{T}}$ and a configurable integer $k > 0$. The tree algorithms are, in this investigation, selected among the spanning-tree and Steiner-tree algorithms presented in chapter 9 and 10.

11.1.1 Interleaved-trees subgraph algorithms

Interleaved-trees algorithms are subgraph algorithms that typically compute multiple connected trees on an input graph, and then merge the trees to one mesh (subgraph). The mesh is the sum of all the computed trees. One advantage of interleaved-trees algorithms is their simplicity, however, a drawback is that their execution time is dependent on the time complexity of the tree algorithms that compute the trees (see table 11.1).

Interleaved-trees algorithms may compute trees *sequentially* or in *parallel*. The sequential approach is round-based, where one tree is computed and merged with the previous trees in each round, and the previously chosen tree edges are excluded from the input graph. In the parallel approach, the trees are computed concurrently and possibly independantly of each other and then merged at the end.

The interleaved-trees algorithms may also be referred to as k-trees algorithms. Formally, a sequential k-trees algorithm comprises a tree algorithm $\mathscr{A}_{\mathscr{T}}$ that constructs k trees, where T_i is the ith tree. For each round $i \geq 1$ an input graph $G_i = G - M_{i-1}$ is created, where $M_i = T_1 \cup \ldots \cup T_i$, and $i \leq k$. The graph G_i is then input to the tree algorithm $\mathscr{A}_{\mathscr{T}}$ to produce T_{i+1}. Young et al. [147] described such an algorithm called k-MST, which computes k minimum spanning trees that are merged into one mesh.

The k-trees algorithms construct meshes that include the "best" edges given the optimization goal of the tree algorithm. For example, the k-MST algorithm ensures that the k minimum weight links of every node are included in the graph [147]. A k-trees algorithm does also produce an approximate k-connected graph (section 4.9). A k-connected graph is a graph in which the removal of any $k-1$ nodes does not disconnect the graph (a 1-connected graph is a tree). Informally, there are at least k independent paths from any vertex to any other vertex. A related graph-theoretic problem is a k-connected minimum weight subgraph (definition 72), for which several approximation algorithms have been proposed [78]. However, these heuristics do not take degree constraints into account, and do not opt for a reduced diameter.

The following interleaved-trees algorithms take as input a tree algorithm $\mathscr{A}_{\mathscr{T}}$ and a configurable integer $k > 0$. The tree algorithms may, for example, be selected among the spanning-tree and Steiner-tree algorithms presented in chapter 9 and 10.

k-Iterative Tree-construction($\mathscr{A}_{\mathscr{T}}$,k) (kIT) takes as input a tree algorithm $\mathscr{A}_{\mathscr{T}}$ and an integer $k > 0$. kIT then sequentially executes the tree algorithm $\mathscr{A}_{\mathscr{T}}$ k times on an input graph, which produces k trees that are merged to a mesh (see algorithm 26). In each round, the current input graph is updated such that the previously chosen tree-edges are excluded. For the degree-limited algorithms the current available degree on each node is reduced according to the previously chosen tree-edges.

Algorithm 26 k-ITERATIVE-TREE-CONSTRUCTION

```
In: $G = (V,E,c)$, a $k > 0$, a tree algorithm $\mathscr{A}_{\mathscr{T}}$
Out: Connected subgraph $M = (V_M, E_M)$
for i = 0; i < k; i++ do
    call algorithm $\mathscr{A}_{\mathscr{T}}(G,M_i)$
    $M \cup M_i$ {update the mesh}
    $E = E - E_M$ {update the input graph}
    updateDegreeLimits($M$, $G$)
end for
```

k-Iterative Combined-tree-construction($\mathscr{A}_{\mathscr{T}}$,k) (kICT) takes as input a set of tree algorithms $\mathscr{A} = \{ \mathscr{A}_{\mathscr{T}1} \ldots \mathscr{A}_{\mathscr{T}k} \}$ and an integer $k > 0$. kICT sequentially executes each $\mathscr{A}_{\mathscr{T}i}$, $i \leq k$, which computes a tree on an input graph. These trees are merged to a mesh (see algorithm 27). In each round, the current input graph is updated such that the previously chosen tree-edges are excluded. For the degree-limited algorithms the current available degree on each node is reduced according to the previously chosen tree-edges.

Algorithm 27 k-ITERATIVE-COMBINED-TREE-CONSTRUCTION

```
In: $G = (V,E,c)$, a $k = |\mathscr{A}|$, a set of tree algorithms $\mathscr{A} = \{ \mathscr{A}_{\mathscr{T}1} \ldots \mathscr{A}_{\mathscr{T}k} \}$.
Out: Connected subgraph $M = (V_M, E_M)$
for all $\mathscr{A}_{\mathscr{T}i} \in \mathscr{A}$ do
    call algorithm $\mathscr{A}_{\mathscr{T}i}(G,M_i)$
    $M \cup M_i$ {update the mesh}
    $E = E - E_M$ {update the input graph}
    updateDegreeLimits($M$, $G$)
end for
```

k-Parallel Tree-construction($\mathscr{A}_{\mathscr{T}}$,k) (kPT) takes as input a tree algorithm $\mathscr{A}_{\mathscr{T}}$ and an integer $k > 0$. kPT then concurrently forks k executions of the tree algorithm $\mathscr{A}_{\mathscr{T}}$ on an input graph. kPT waits until all k executions are done and merges the trees to one mesh (see algorithm 28). It is optional if the previously chosen tree-edges are excluded among the k executing tree algorithms. For degree-limited algorithms the current available degree may also be reduced according to the previously chosen tree-edges. However, these optional features increases the complexity of the kPT implementation.

Algorithm 28 k-PARALLEL-TREE-CONSTRUCTION

```
In: $G = (V, E, c)$, a $k > 0$, a tree algorithm $\mathscr{A}_{\mathscr{T}}$
Out: Connected subgraph $M = (V_M, E_M)$
for i = 0; i < k; i++ do
    fork algorithm $\mathscr{A}_{\mathscr{T}}(G,M_i)$ and continue
end for
wait until $k$ algorithms have completed
$M \cup \{M_1, \ldots, M_k\}$ {update the mesh}
```

k-Parallel Combined-tree-construction($\mathscr{A}_{\mathscr{T}}$,k) (kPCT) takes as input a set of tree algorithms $\mathscr{A} = \{ \mathscr{A}_{\mathscr{T}1} \ldots \mathscr{A}_{\mathscr{T}k} \}$, and an integer $k > 0$. kPCT then concurrently forks k executions, that is, each $\mathscr{A}_{\mathscr{T}i}$, $i \le k$, on an input graph. Similar to kPT, kPCT also waits until all k executions are done and merges the trees to one mesh (see algorithm 29). It is optional if the previously chosen tree-edges are excluded among the k executing tree algorithms. For degree-limited algorithms the current available degree may also be reduced according to the previously chosen tree-edges. However, these optional features increases the complexity of the kPCT implementation.

Algorithm 29 k-PARALLEL-COMBINED-TREE-CONSTRUCTION

```
In: $G = (V, E, c)$, a $k = |\mathscr{A}|$, a set of tree algorithms $\mathscr{A} = \{ \mathscr{A}_{\mathscr{T}1} \ldots \mathscr{A}_{\mathscr{T}k} \}$.
Out: Connected subgraph $M = (V_M, E_M)$
for i = 0; i < k; i++ do
    fork algorithm $\mathscr{A}_{\mathscr{T}i}(G,M_i)$ and continue
end for
wait until $k$ algorithms have completed
$M \cup \{M_1, \ldots, M_k\}$ {update the mesh}
```

11.1.2 Enhanced tree subgraph algorithms

Enhanced tree algorithms are subgraph algorithms that typically compute a single connected tree on an input graph, and then add single edges to this tree based on some optimization goal. The optimization goal may, for example, be to reduce the pair-wise latencies, or perhaps the failure tolerance of the subgraph. Wang et al. [137] described enhanced tree algorithms, and proposed an overlay multicast protocol called Tmesh, which adds "short-cut" edges to a pre-constructed tree.

An enhanced tree algorithm comprises a tree algorithm $\mathscr{A}_{\mathscr{T}}$ that produces a tree T, and an edge-selection algorithm that adds edges to the graph T such that it is transformed into a mesh M (cyclic subgraph). The number of edges that are added may be a predefined integer k, or based on some optimization goal, for example, a mesh-diameter below a given bound D. The most common edge-selection strategies add edges that reduce a node's eccentricity or the mesh's diameter. Notice that in a shared-overlay the maximum eccentricity equals the diameter, such that reducing the maximum eccentricity does in effect reduce the diameter of the mesh. Therefore, a strategy for reducing both is to always pick the nodes with the maximum eccentricity (= diameter) and try to reduce their eccentricity.

There are also edge-selection strategies that aim at reducing the pair-wise latencies in an overlay. The average pair-wise latency of a node is the sum of the shortest path latencies between it and every other node, divided by the number of nodes. Both Narada [49] and Tmesh [137] focus on reducing a node's pair-wise latency. Reducing the pair-wise latency or the diameter are very much similar goals. If the diameter is reduced, the pair-wise latency is also reduced, however, if the pair-wise latency is reduced it does not automatically reduce the diameter. In a shared-overlay it is more important to reduce the diameter.

The following interleaved-trees algorithms take as input a tree algorithm $\mathscr{A}_{\mathscr{T}}$ and a configurable integer $k > 0$, which defines the number of edges to add to the constructed tree such that it becomes a mesh.

k-Diameter-Links($\mathscr{A}_{\mathscr{T}}$,k) (kDL) executes the tree algorithm $\mathscr{A}_{\mathscr{T}}$ to produce a tree T, and then adds k diameter links to T to produce a mesh M. For k rounds, the edge-selection strategy in kDL tries to reduce the diameter of the mesh M by adding a shortest-path edge between two nodes in the current diameter path (see algorithm 30 for more details). Ties are broken arbitrarily.

Algorithm 30 k-DIAMETER-LINKS

```
In: $G = (V, E, c)$, a $k > 0$, a tree algorithm $\mathscr{A}_{\mathscr{T}}$
Out: Connected subgraph $M = (V_M, E_M)$
call algorithm $\mathscr{A}_{\mathscr{T}}(G,M)$
for i = 0; i < k; i++ do
    find diameter path $P = (u_1, \ldots, u_n)$, where $u_i \in V_M$
    find an edge $e \in (E - E_M)$ between two nodes $u, v \in P$
    $M \cup \{e = (u, v)\}$ {update the mesh}
end for
```

k-Long-Links($\mathscr{A}_{\mathscr{T}}$,k) (kLL) executes the tree algorithm $\mathscr{A}_{\mathscr{T}}$ to produce a tree T, and then adds k long links to T to produce a mesh M. For k rounds, the edge-selection strategy in kLL chooses the node with the lowest degree in M and adds the longest shortest-path edge to an overlay-node (see algorithm 31 for more details). Ties are broken arbitrarily.

Algorithm 31 k-LONG-LINKS

```
In: $G = (V, E, c)$, a $k > 0$, a tree algorithm $\mathscr{A}_{\mathscr{T}}$
Out: Connected subgraph $M = (V_M, E_M)$
call algorithm $\mathscr{A}_{\mathscr{T}}(G,M)$
for i = 0; i < k; i++ do
    find node $v \in V_M$ with lowest degree $deg_M(v)$
    find the longest edge $e \in (E - E_M)$ to a node $u \in V_M$
    $M \cup \{e\}$ {update the mesh}
end for
```

k-Core-Links($\mathscr{A}_{\mathscr{T}}$,k) (kCL) executes the tree algorithm $\mathscr{A}_{\mathscr{T}}$ to produce a tree T, and then adds k core links to T to produce a mesh M. kCL finds k core-nodes C from M using the core-node selection algorithm k-Median (chapter 7), and then the k nodes W from M that have the largest pair-wise latencies. Then, for k rounds, the edge-selection strategy in kCL finds an edge connecting a core-node $c \in C$ with a node $w \in W$ (see algorithm 32 for more details).

Algorithm 32 k-CORE-LINKS

```
In: $G = (V, E, c)$, a $k > 0$, a tree algorithm $\mathscr{A}_{\mathscr{T}}$
Out: Connected subgraph $M = (V_M, E_M)$
call algorithm $\mathscr{A}_{\mathscr{T}}(G,M)$
$C$ = $k$-MEDIAN-CORE-SELECTION($G$,$M$, $k$) {sorted set}
$W$ = $k$-WORST-CORE-SELECTION($G$,$M$, $k$) {sorted set}
for each $w \in W$ do
    while available degree on $w$ or some maximum $n < k$ do
        find edge $e \in (E - E_M)$ between $w$ and best $c \in C$
        $M \cup \{e\}$ {update the mesh}
    end while
end for
```

11.1.3 Edge pruning subgraph algorithms

Edge pruning algorithms are subgraph algorithms that typically compute a connected mesh on an input graph by applying one or several edge-selection strategies that may be configurable. Edge pruning algorithms include strategies that remove edges from an input graph G based on some goal, and also algorithms that pick single edges from an input graph and constructs a mesh M. These two approaches are essentially different, however, the algorithms share the same goal. Consequently, we call all of them edge pruning algorithms.

The simplest edge pruning algorithm is to add a number of edges randomly to the mesh. Yoid [50] applies this method, with the added step of applying a routing protocol atop of the mesh. In the Narada [49] protocol a node joins a random peer and then slowly moves to more favorable peers as they are discovered. Although random edge pruning algorithms are in use, they are the most naive edge pruning algorithms and we disregard them from this investigation [76].

Minimum-cost Greedy Reconnect Graph heuristic: Some edge pruning algorithms cannot guarantee that a connected subgraph is found. Therefore, a fast procedure for identifying the disconnected groups (subgraph forests) is needed. Identifying one (partitioned) forest can be done by using depth first search (DFS) with running time $O(V + E)$ (section 4.4). Furthermore, connecting two forests through a greedy minimum-cost edge is a $O(n^2)$ procedure, for example, using a minimum-cost algorithm from chapter 12.

One heuristic that connects disconnected graphs through some low-cost edges has a time-complexity of $O(n^3)$. This is the group Steiner-tree heurisic k-mcReconnectTreeHeuristic, which is introduced in chapter 12. This tree-heuristic is adjustable to a group Steiner-subgraph heuristic (section 4.10 introduces the Steiner-subgraph problem formally).

Instead, we introduce the following greedy algorithm 33, which is also a Steiner-subgraph-heuristic, except the groups (forests) are discovered dynamically inside the procedure, and then the groups are greedily connected through some low-cost edges. The greediness reduces the time-complexity to $O(n^2)$, with the cost of potentially adding higher-cost edges. However, the greedy draw-back is very limited since the edge-pruning algorithms produce subgraps with a small number of subgraph forests.

Algorithm 33 MC-GREEDY-RECONNECT-GRAPH-HEURISTIC

```
In: $G = (V, E, c)$, a disconnected subgraph $M = (V, E_M)$
Out: Connected subgraph $M = (V, E_M)$
while $M$ is disconnected do
    $P$ = find a partition in $M$ using DEPTH-FIRST-SEARCH($M$)
    choose one vertex $p \in P$ and connect $p$ to $v \in (V_M - P)$ using minimum-cost edge $e \in E$
end while
```

The following edge pruning algorithms take as input a configurable integer $k \in \mathbb{N}$, which defines the number of edges the edge-selection strategy should add, for example, for each node.

k-Best-Links(k) (kBL) is an edge pruning algorithm that lets each node autonomously add its k minimum weight (best) edges to the mesh (algorithm 34). In kBL, pairs of nodes independently include each other into the mesh, such that the resulting mesh may have only $(k*n)/2$ edges (at most $k*n$). Because of this, kBL has a higher chance of producing a disconnected graph [147]. This is the main reason that few if any protocols use kBL as described.

Algorithm 34 k-BEST-LINKS

```
In: A connected graph $G = (V, E, c)$, and an integer $k > 0$.
Out: A connected subgraph $M = (V, E_M)$, where $E_M \subset E$.
For each node $v \in V$, include its $k$ minimum-cost edges to $E_M \subset E$.
if $M$ is not connected then
    call mcGreedyReconnectGraphHeuristic($G$,$M$)
end if
```

Degree-limited-Best-Links (dlBL) is an edge pruning algorithm, which is a special case of kBL. In dlBL each node autonomously add minimum-weight edges to the mesh until its degree-limit is reached. We propose dlBL such that the node-degree is limited but also exploited to the maximum on each node (algorithm 35).

We have also introduced an edge pruning algorithm [130] that builds meshes that include low weight links and higher weight longer links to a set of pre-selected core-nodes, selected using a multiple core-node selection algorithm (chapter 7). The number of core-nodes to choose is defined by equation 8.1 (chapter 8). Young et al. [147] have described similar randomized

```
Algorithm 35 dl-BEST-LINKS
In: A connected graph $G = (V, E, c)$ and a degree-limit $d(v)$ for each $v \in V$.
Out: A connected subgraph $M = (V, E_M)$, where $E_M \subset E$.
  For each node $v \in V$, accept and include $d(v)$ minimum-cost edges to $E_M \subset E$.
  if $M$ is not connected then
      call mcGreedyReconnectGraphHeuristic($G$,$M$)
  end if
```

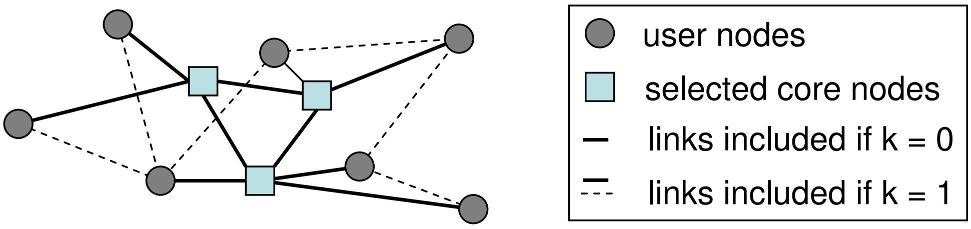


Figure 11.1: A mesh produced by *add*-Core-Links-Optimized.

approaches as short-long strategies. Informally, the next edge-pruning algorithms use a kBL approach and then shortest path links to connect core-nodes to the remaining nodes.

add-Core-Links-Optimized(k, O) (aCLO) takes as input a complete graph G, an integer $k \geq 0$, and a set $O \subset V$, which may be identified by a multiple core-node selection algorithm. In aCLO, each non-core-node (not in O) includes its k minimum-cost edges to the mesh. Then, aCLO builds a full mesh of the nodes in O, and includes to the mesh. Further, aCLO lets each core-node o add a number $s = |V - O|/|O|$ of (disjoint) edges to the non-core-nodes. After these steps, the constructed mesh forms, conceptually, a two-layer graph. Figure 11.1 illustrates a mesh generated by aCLO. Algorithm 36 is the aCLO algorithm when equation 8.1 (section 8.5.4) is used to determine the number of core-nodes, and k-Median identifies these core-nodes among the member-nodes. The algorithm has a time complexity of $O(n^2)$.

```
Algorithm 36 add-CORE-LINKS-OPTIMIZED
In: A complete graph $G = (V, E, c)$, and an integer $k \geq 0$.
Out: A connected subgraph $M = (V, E_M)$, where $E_M \subset E$.
  $O = k$-Median($G, l$)
  {find $l$ core-nodes among $V$, $l$ is obtained from equation 8.1 in section 8.5.4}
  For each node $m \in (V \setminus O)$, include its $k$ minimum-cost edges to $E_M \subset E$.
  For each core node $o \in O$, include an edge to every other node $v \in O$.
  For each core-node $o \in O$ disjointly connect to $l = |V \setminus O|/|O|$ nodes $v \in (V \setminus O)$ through minimum-cost
  edges.
```

degree-limited add-Core-Links-Optimized(k, O) (dl-aCLO) is a degree-limited version of aCLO. dl-aCLO works much the same way, but the core-nodes O do not form a full mesh of shortest paths. Instead, the core-nodes should be "meshified" using only a certain percentage of their full degree-capacity to connect to other core-nodes. Then, the core-nodes use their remaining degree-capacity to connect to the non-core-nodes. dl-*a*CLO sometimes produces a disconnected subgraph, therefore, algorithm 33 is used to connect it. Algorithm 37 is an example of the dl-*a*CLO used in the simulations.

Algorithm 37 dl-*add*-CORE-LINKS-OPTIMIZED

```
In: A graph $G = (V, E, c)$, an integer $k \geq 0$, and for each $v \in V$ there is a degree limit $deg(v) > 0$.
Out: A connected subgraph $M = (V, E_M)$, where $E_M \subset E$, and for each $v \in V$ the $deg_M \leq deg(v)$.
$O$ = k-Median-Selection($G$)
    {find $l$ core-nodes among $V$, $l$ is obtained from equation 8.1}
For each node $m \in (V - O)$, include its $k$ minimum-cost edges to $E_M \subset E$. {kBL}
For each core node $o \in O$, connect to $d = d(o) * 0.33)$ nodes in $v \in O$, and add edges to $E_M$.
For each core node $o \in O$ disjointly connect to $d = d(o) * 0.67$ nodes in $v \in (V - O)$, and add edges to $E_M$.
if $M$ is not connected then
    call mcGreedyReconnectGraphHeuristic($G$,$M$)
end if
```

The spanning subgraph variations of *a*CLO and dl-*a*CLO use member-nodes as the core-nodes, while, the Steiner subgraph variations use Steiner points as the core-nodes. The Steiner subgraph variatons are actually approximating a Terminal Steiner-subgraph (definition 88), in which, all member-nodes are leaf-nodes when $k = 0$.

11.2 Group communication simulations of spanning subgraph algorithms

The following evaluations are of the results obtained from group communication simulations using the spanning subgraph algorithms to build overlay networks for event distribution. Table 11.1 introduces the basics of each spanning subgraph algorithm.

The spanning-tree algorithms $\mathscr{A}_{\mathscr{T}}$ that are used as input to the spanning subgraph algorithms $\mathscr{A}_{\mathscr{M}}$ are introduced in section 9.2, and were found to be the better of them in the evaluations in section 9.3. A summary of their algorithm details is given in table 11.2.

11.2.1 Experiment configurations

In the experiments, we use complete group graphs as input graphs to the subgraph algorithms. Chapter 8 introduced and evaluated the problems that constrained overlay construction heuristics have in finding a solution, especially when the constraints are strict. That is also the case with the constrained spanning-tree heuristics evaluated here. The success rate of the algorithms depend on the constraint and the input graph. We use dynamic relaxation on the degree-limits and the diameter bounds whenever a tree heuristic cannot continue the tree construction.

A low diameter is a target metric, such that in our simulations we use a strict diameter bound of 0.250 to the diameter bounded algorithms. The heuristics are then frequently forced to relax

Algorithm	*Meaning*	*Optimization*	*Input to $\mathscr{A}_{\mathscr{M}}$*	*Complexity*	*Category*	*Problem*	*Reference*
kIT	k-Iterative Tree constructions	$\mathscr{A}_{\mathscr{T}}$ optimization goal	$G, k, \mathscr{A}_{\mathscr{T}}$	$O(k * O(\mathscr{A}_{\mathscr{T}}))$	Interleaved trees	70) k-c subgraph	[147]
kPT	k-Parallel Tree Constructions	$\mathscr{A}_{\mathscr{T}}$ optimization goal	$G, k, \mathscr{A}_{\mathscr{T}}$	$O(k * O(\mathscr{A}_{\mathscr{T}}))$	Interleaved trees	70) k-c subgraph	[147]
kCT	k-Combined Tree Constructions	$\mathscr{A}_{\mathscr{T}}$ optimization goal	$G, k, \mathscr{A}_{\mathscr{T}}$	$O(k * O(\mathscr{A}_{\mathscr{T}}))$	Interleaved trees	70) k-c subgraph	[147]
kDL	k-Diameter Links	latency diameter	$G, k, \mathscr{A}_{\mathscr{T}}$	$O(n^3 + O(\mathscr{A}_{\mathscr{T}}))$	Enhanced-trees	78) k-c MDDL	-
kLL	k-Long Links	pair-wise latency	$G, k, \mathscr{A}_{\mathscr{T}}$	$O(n^2 + O(\mathscr{A}_{\mathscr{T}}))$	Enhanced-trees	78) k-c MDDL	[137]
kCL	k-Core Links	pair-wise latency	$G, k, \mathscr{A}_{\mathscr{T}}$	$O(n^2 + O(\mathscr{A}_{\mathscr{T}}))$	Enhanced-trees	78) k-c MDDL	[137]
kBL	k-Best-Links	minimum cost	G, k	$O(n^2)$	Edge pruning	72) k-c min-cost	[147]
dl-BL	degree-limited Best-Links	pair-wise latency	G	$O(n^2)$	Edge pruning	73) k-c dl min-cost	-
aCLO	add-Core-Links-Optimized	pair-wise latency	$G, k, O \subset V$	$O(n^2)$	Edge pruning	74) MD subgraph	[130]
dl-aCLO	degree-limited add-Core-Links-Optimized	pair-wise latency	$G, k, O \subset V$	$O(n^2)$	Edge pruning	77) MDDL subgraph	[130]

Table 11.1: Mesh construction algorithms ($\mathscr{A}_{\mathscr{M}}$).

Algorithm	*Meaning*	*Optimization*	*Constraints*	*Complexity*	*Problem*	*Reference*
MST	Prim's minimum-spanning tree	total cost	-	$O(n^2)$	MST	[58]
SPT	Dijkstra's shortest-path tree	core/destination cost	-	$O(n^2)$	SPT	[58]
md-OTTC	Minimum diameter one-time tree construction	diameter	-	$O(n^3)$	MDST	[131]
OTTC	One-time tree construction	total cost	diameter	$O(n^3)$	BDMST	[1]
RGH	Randomized greedy heuristic	total cost	diameter	$O(n^2)$	BDMST	[104]
mddl-OTTC	Minimum diameter degree-limited one-time tree construction	diameter	degree	$O(n^3)$	MDDL	[131]
dl-OTTC	Degree-limited one-time tree construction	total cost	diameter and degree	$O(n^3)$	BDDLMST	[131]
dl-RGH	Degree-limited randomized greedy heuristic	total cost	diameter and degree	$O(n^2)$	BDDLMST	[131]
dl-SPT	Degree-limited Dijkstra's shortest-path tree	core/destination cost	degree	$O(n^2)$	d-SPT	[95]
dl-MST	Degree-limited Prim's minimum-spanning tree	total cost	degree	$O(n^2)$	d-MST	[95]

Table 11.2: Tree construction algorithms ($\mathscr{A}_{\mathscr{T}}$).

Description	*Parameter*
Placement grid	$100x100$ milliseconds
Number of nodes in the network	1000
Degree limit	5 and 10
Diameter bound	250 milliseconds

Table 11.3: Spanning subgraph experiment configuration.

the bound. The experiment configuraions are listed in table 11.3.

11.2.2 Evaluated target metrics

A spanning subgraph algorithm is considered good if it can produce overlays with a low diameter, a low average pair-wise distance, within a reasonable time that does not add unreasonable cost to the network. For our evaluation of the overlays and algorithms, we therefore consider four metrics to be very important: overlay diameter, average pair-wise distance, algorithm execution time, and total network cost. In addition, given the fairly limited resources available on average clients in the Internet, the algorithm should obey degree-limitations such that the stress on each node in the overlay is bounded.

In the following, we evaluate the results from our simulations. In the evaluation, we focus on the target metrics and also evaluate the different spanning subgraph algorithms against each other.

11.2.3 Results from one group size range

Our main plots are figure 11.2 and 11.3, which includes a complete comparison of the tree and mesh construction algorithms evaluated towards our target metrics. It includes statistics from overlays of sizes between 100 and 120. The tree-heuristics are plotted as interleaved tree algorithms with $k = 1$.

We observe that the best tree-heuristics achieve a *diameter* of around 0.3 seconds, while the degree-limited tree-heuristics achieve 0.5 seconds. For the degree-unlimited algorithms there is almost no gain in going from a tree to a mesh. For the degree-limited algorithms we can see a larger reduction in the diameter, but even here it is not significant. Among the enhanced-tree algorithms we see that kDL reduces the diameter more than kLL. Comparatively, we see that it is only the mesh algorithms that use MST or dl-MST as input that reduce the diameter significantly in a mesh (compared to a tree). When these minimum-cost overlays are transformed from a tree to a mesh, they become more competetive with the close-to-minimal diameter overlays. The edge pruning algorithms all produce low diameter meshes, with the exception of kBL($k < 3$). We observe similar trends for the average *pair-wise* distance (seconds).

When we analyze the *total cost* of the overlays it is clear that the mesh algorithms build more costly overlays compared to the tree construction algorithms. The enhanced-tree algorithms only slightly increases the total cost of the overlays, due to the rather small k we used. The edge pruning algorithms kBL and dl-aCLO also yield a reasonable total cost. We see that the interleaved tree algorithms kIT(dl-MST,k) and kIT(dl-SPT,k) construct overlays with a low total cost. However, this is due to the fact that dl-MST and dl-SPT often fail to construct degree-limited trees in sparse graphs [130].

The *edge count* is the number of links in an overlay, and the plots confirm this observation. We see that the edge-count for the kIT(dl-MST,2) and kIT(dl-SPT,2) overlays are much lower than the remaining interleaved trees algorithms. The edge pruning algorithm aCLO clearly adds too many edges to the overlay. For the interleaved-trees algorithms we can see that the edge-count increases in certain intervals for each k increase. The degree-unlimited algorithms merge a new full tree for each round, whereas, the degree-limited algorithms are not able to construct a full spanning tree for each round, in particular when $k > 2$. For example, in the kIT algorithm the already chosen tree-edges are excluded from the input graph, and the available degree is reduced accordingly. This is the main reason why the degree-limited algorithms increasingly fail to construct full spanning trees.

The *execution time* shows that the degree-limited interleaved-tree algorithms struggle to find trees when the k increases. The kDL algorithm is quite time-consuming when $k > 40$; however, our implementation may be sub-optimal for kDL. The fastest algorithms are the edge-pruning algorithms, which are barely visible in the graphs.

11.2.4 Results with varying group sizes

The group size may influence the performance of an overlay construction algorithm. In figure 11.4, 11.5 and 11.6 we plot the diameter and pair-wise latencies and hop-count, then the total cost and overlay edge count for selected algorithms with group sizes between 10 and 150. We include results from the interleaved- and enhanced-trees algorithms using the tree algorithms dl-MST and mddl-OTTC as input.

The *diameter* achieved is plotted in figure 11.4(a). We observe that the reduction in the diameter between mddl-OTTC and kIT(mddl-OTTC,2), is very small. The other mddl-OTTC mesh algorithm variations perform very similar. dl-MST is not plotted but produces a diameter

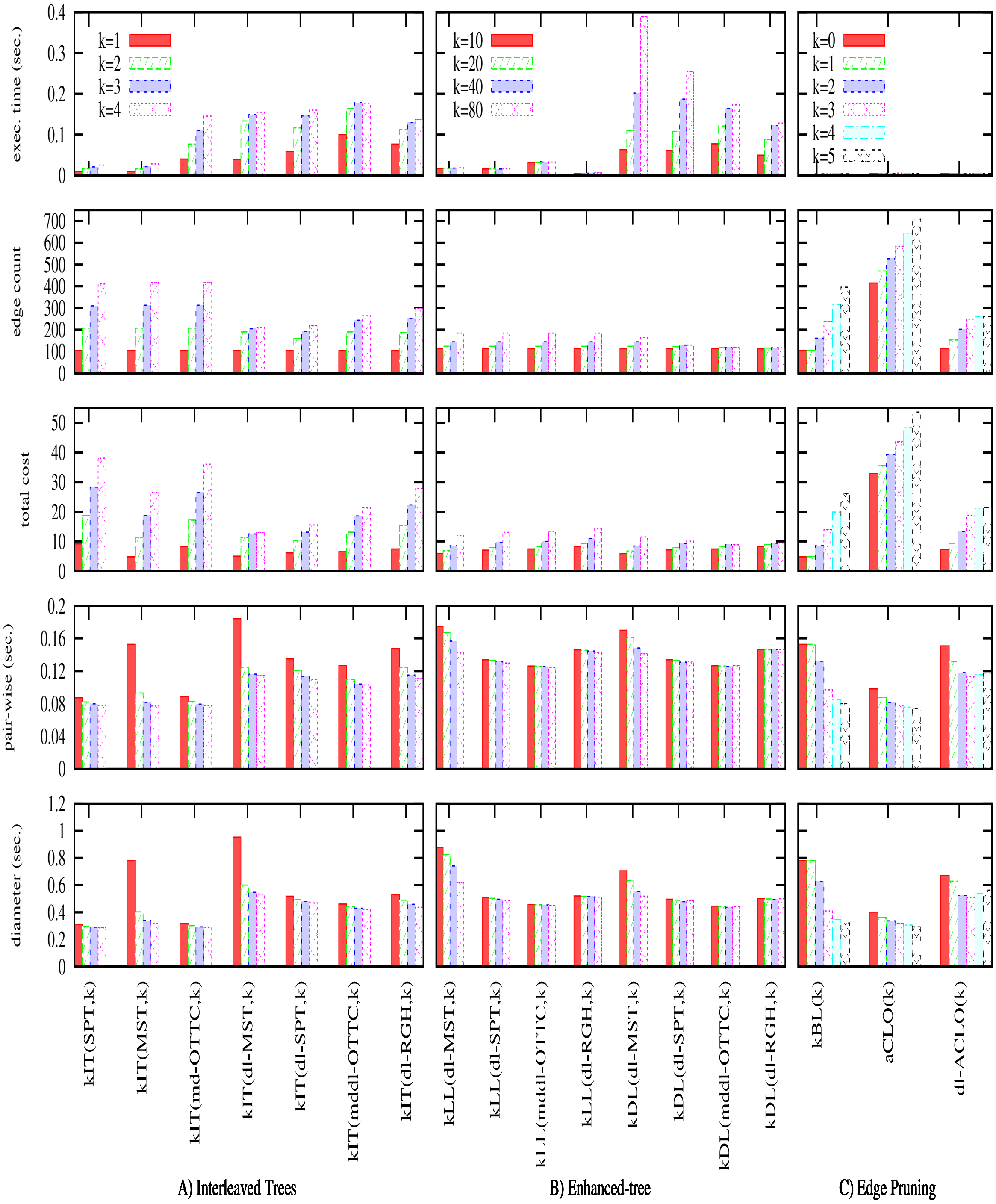


Figure 11.2: Comparison of all spanning-tree and mesh algorithms.

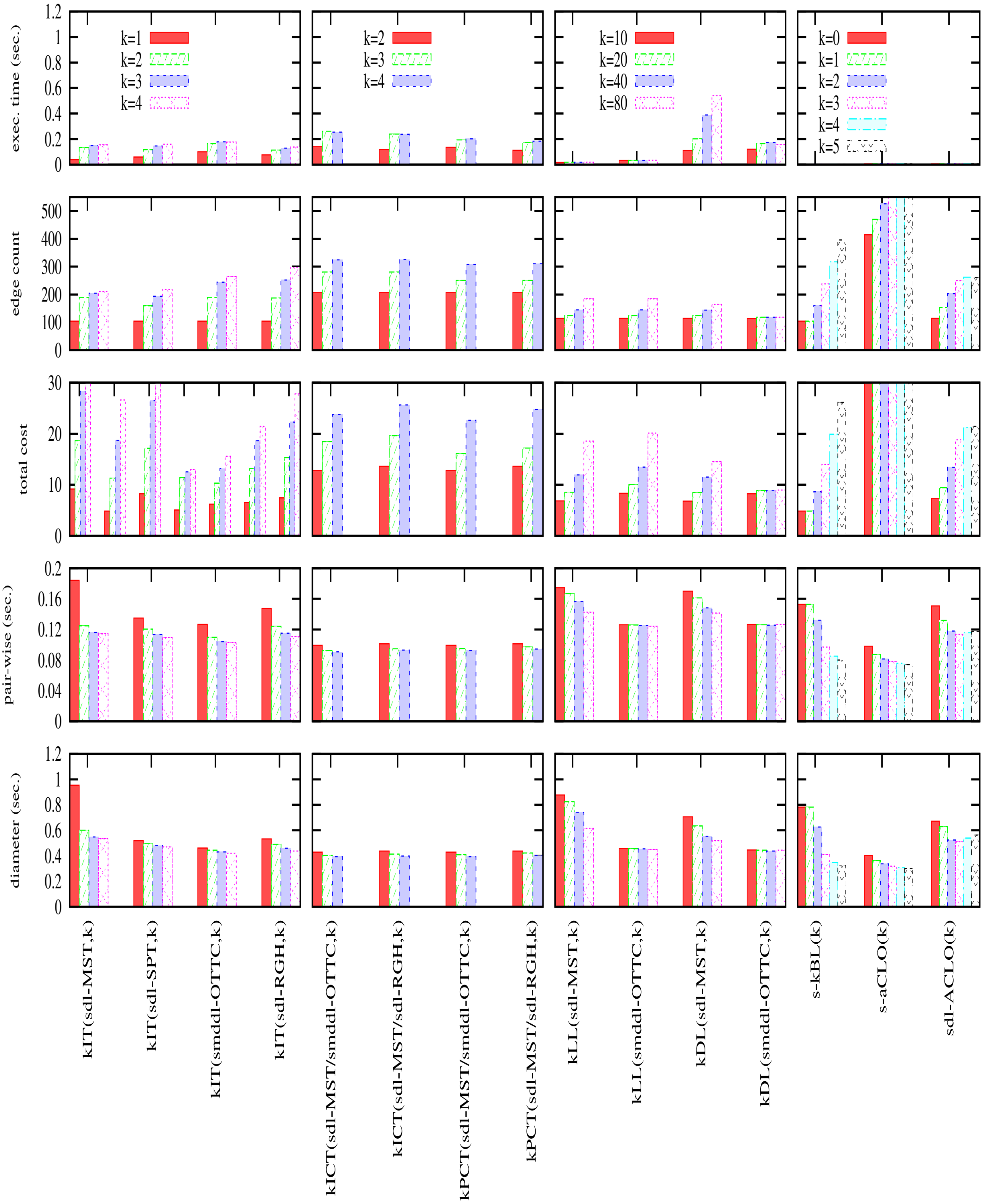


Figure 11.3: Comparison of all spanning-tree and mesh algorithms.

which is 30 % higher than kIT(dl-MST,2), and for $k = 3$ the reduction is only about 5 %. kLL(dl-MST,80) yields overlays with a lower diameter than kIT(dl-MST,2) for group sizes < 110, while kDL(dl-MST,80) achieves about 25 % better throughout our group size range. Among all the subgraph algorithms, it was kICT(dl-MST/dl-RGH,2) and kDL(mddl-OTTC,80) that yield the lowest diameter. We observed very similar trends for the average *pair-wise latency* in figure 11.5(a).

The *hop-diameter* achieved by the algorithms is plotted in figure 11.4(b). The hop-diameter reduction between mddl-OTTC and kIT(mddl-OTTC,2) is about 25 %. kDL(mddl-OTTC,80) does not reduce the hop-diameter much, although it reduces the diameter quite significantly. kLL(mddl-OTTC,80) and kLL(dl-MST,80) yield similar hop-diameter with group sizes < 80, and are both better than kIT(mddl-OTTC,2) in that group range. For group sizes > 60, kCIT(dl-MST/dl-RGH,2) is the better, with kIT(mddl-OTTC,2) only slightly higher. We observed very similar trends for the average *pair-wise hop-count* in figure 11.5(b).

For the *total cost* plot in figure 11.6(b) wee see that the enhanced-trees algorithm KLL constructs the most expensive overlays. kIT(mddl-OTTC,2) builds overlays that are slightly more expensive than kIT(dl-MST,2). The kDL(mddl-OTTC,80) is not able to add 80 links to each overlay, therefore its total cost is lower than expected. Not surprisingly, the trees constructed by dl-MST and mddl-OTTC are the cheapest.

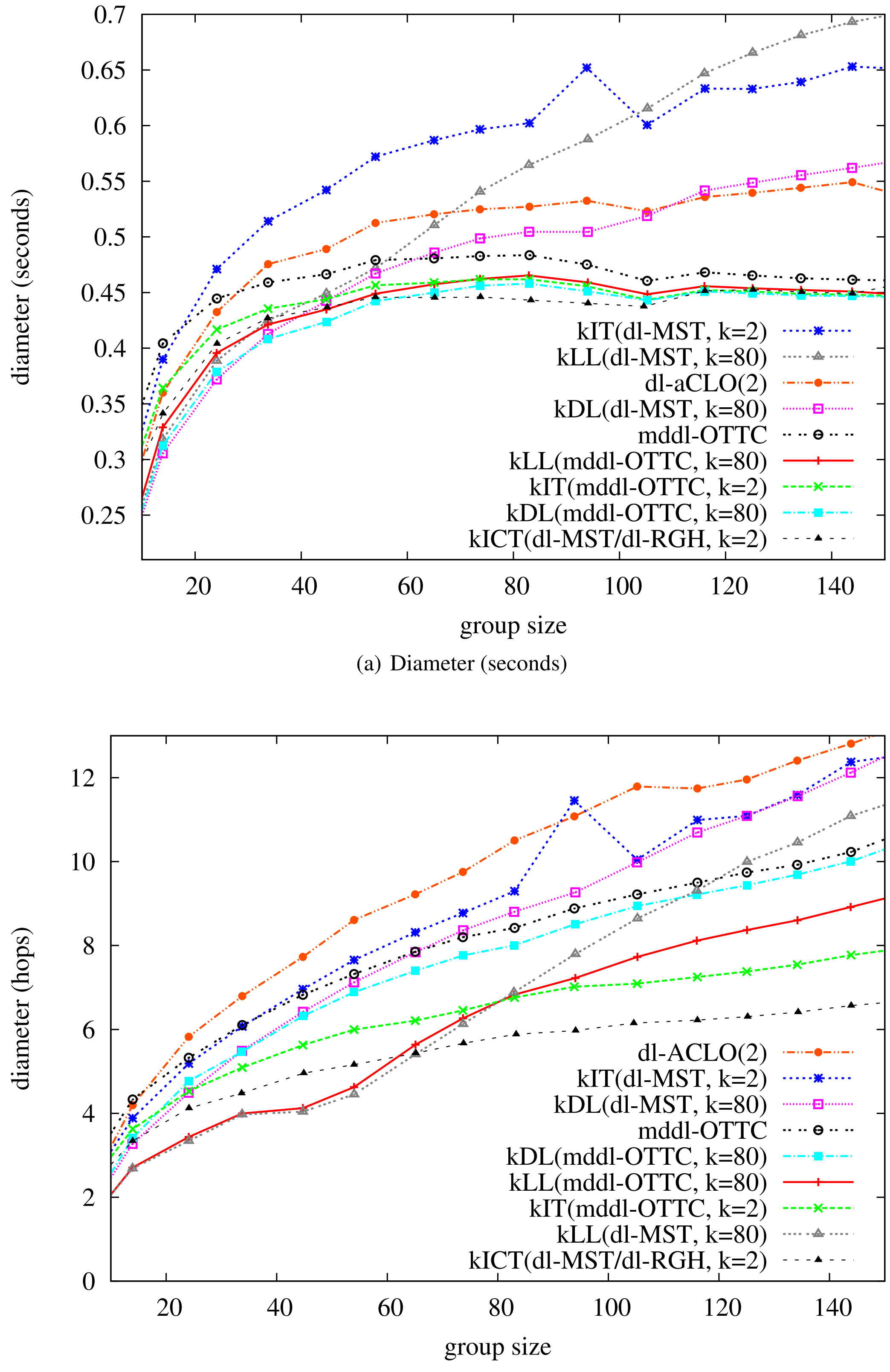
The *edge count* in figure 11.6(a) confirms that kDL(mddl-OTTC,80) fails to add 80 links, due to the degree-limitation, but rather only manages to add about 15 links before it gives up. kDL(dl-MST, 80), on the other hand, has a much higher success rate in adding links. Close-to-minimum diameter trees are more star shaped (leafy trees), where inner nodes have a high degree. Therefore, kDL struggles to add edges to the diameter path because the degree capacity quickly gets exhausted. kICT(dl-MST/dl-RGH,2), on the other hand, is able to include two full trees in the entire group range.

We observed from the results that many of the subgraph algorithms struggled to include edges due to the rather strict degree-limit of 5. In the following, we include and evaluate results from using a higher degree-limit of 10.

11.2.5 Results with varying group sizes and degree-limits

The previous evaluation used a static degree-limit of 5 to analyze the results. Here, we use degree limits of 5 and 10 to see the effect on the subgraph algorithms and the constructed subgraphs.

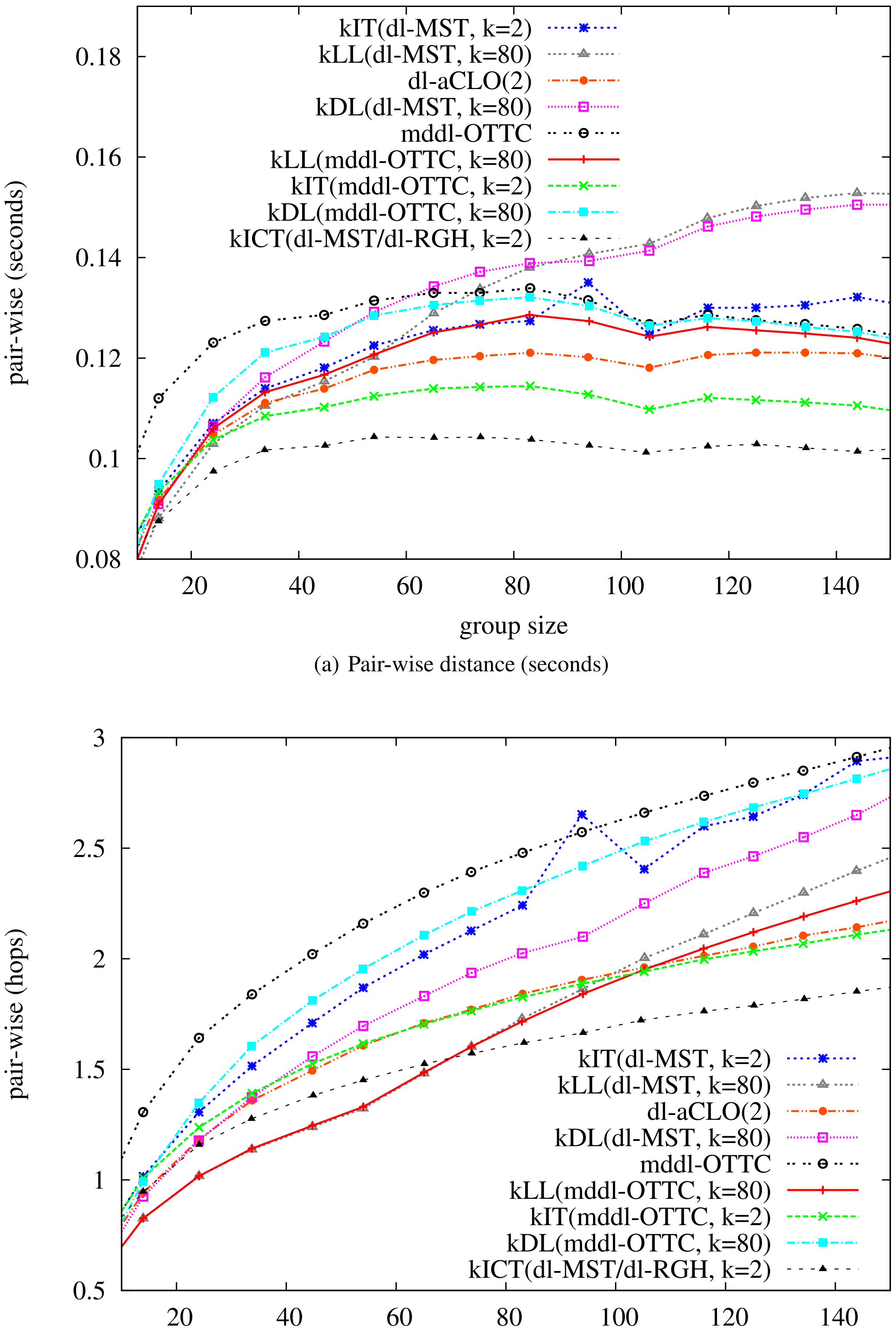
The *diameter (seconds)* of the subgraphs is plotted in figure 11.7(a). The plotted subgraph algorithms were found to perform the best in the previous evaluation. It is quite clear that the doubled degree capacity has enabled the subgraph algorithms to decrease the diameter of the subgraphs quite significantly. The diameter reduction is consistently 90 milliseconds among the



(a) Diameter (seconds)

(b) Diameter (hops)

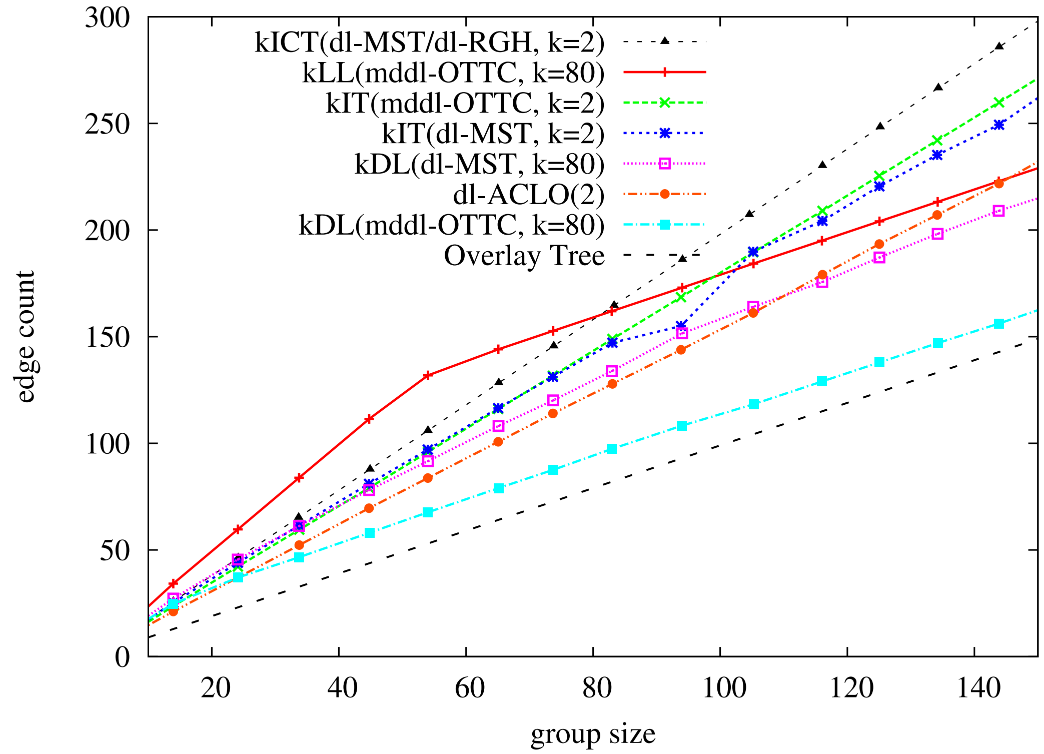
Figure 11.4: Diameter (seconds and hop-count) of the subgraphs, constructed by selected subgraph algorithms (degree-limit = 5).



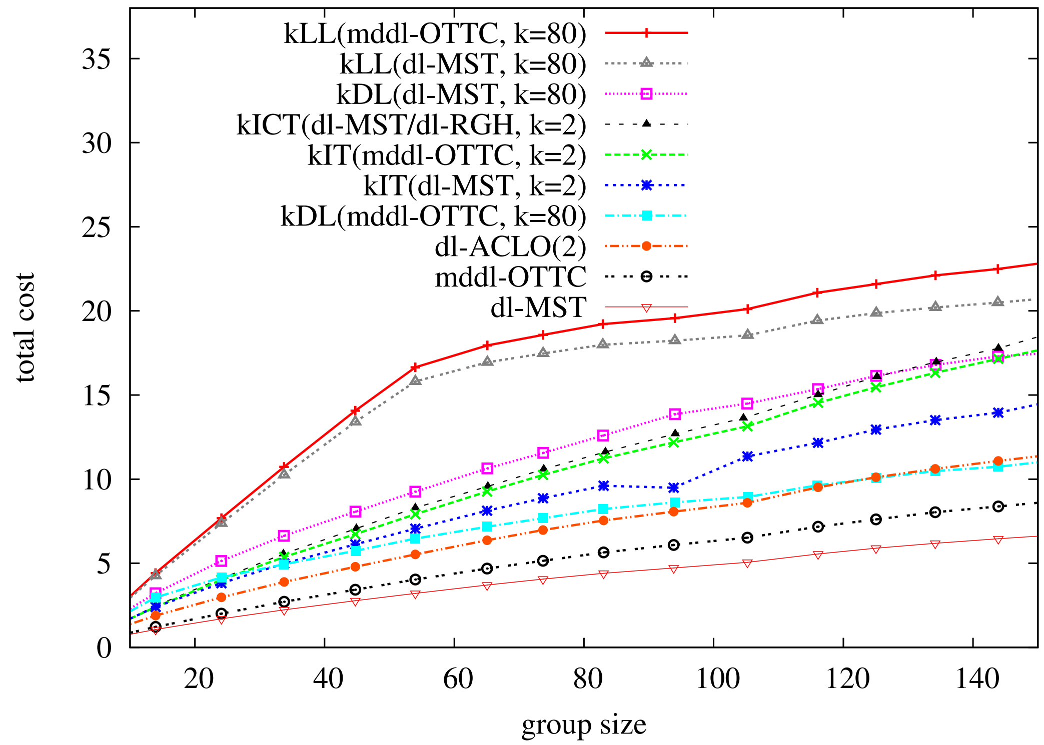
(a) Pair-wise distance (seconds)

(b) Pair-wise distance (hops)

Figure 11.5: Pair-wise latencies and hop-count of the subgraphs, constructed by selected subgraph algorithms (degree-limit = 5).



(a) Edge count



(b) Total cost

Figure 11.6: Edge-count and total-cost of the subgraphs, constructed by selected subgraph algorithms (degree-limit = 5).

algorithms. However, kICT(dl-MST/dl-RGH,2) does not reduce the diameter with more than 50 milliseconds when the degree capacity is doubled. When the degree-limit was 5, the randomized dl-RGH still managed to complete its construction, while the other subgraph algorithm combinations struggled to complete (please preview figure 11.8(a) for edge-count). The higher degree-limits enable almost every subgraph algorithm combination to complete their construction (that is, for the evaluated *k*-values). Therefore, the combinations kIT(mddl-OTTC,2) and kDL(mddl-OTTC,80) are the ones that yield the lowest diameter when the degree limit is 10.

The *diameter (hop-count)* of the subgraphs is plotted in figure 11.7(b). Similar tendencies are revealed, and we see that kDL(mddl-OTTC,80) and mddl-OTTC reduce the hop-diameter by about 25 % with a doubled degree capacity. kIT(mddl-OTTC,2) reduces the hop-diameter by 15 %, while kICT(dl-MST/dl-RGH) only marginally reduces it.

The subgraph *edge-count* is plotted in figure 11.8(a). We observe that when the degree limit is 10, the subgraph algorithms manage to complete their construction. However, kDL(mddl-OTTC,80) is still not able to enhance the tree with 80 edges, but rather it only manages to add 50 edges. But, we observed from figure 11.7(a) and 11.7(b) that kDL(mddl-OTTC,80) yield a significantly lower diameter (seconds and hop-count) with a doubled degree capacity.

The *total cost* of the subgraphs is plotted in figure 11.8(b). The increased success rate of kDL(mddl-OTTC,80) when the degree-limit is 10 is noticable in the total cost, and is consistently very costly. Among the remaining subgraph algorithms there is only a marginal total cost increase.

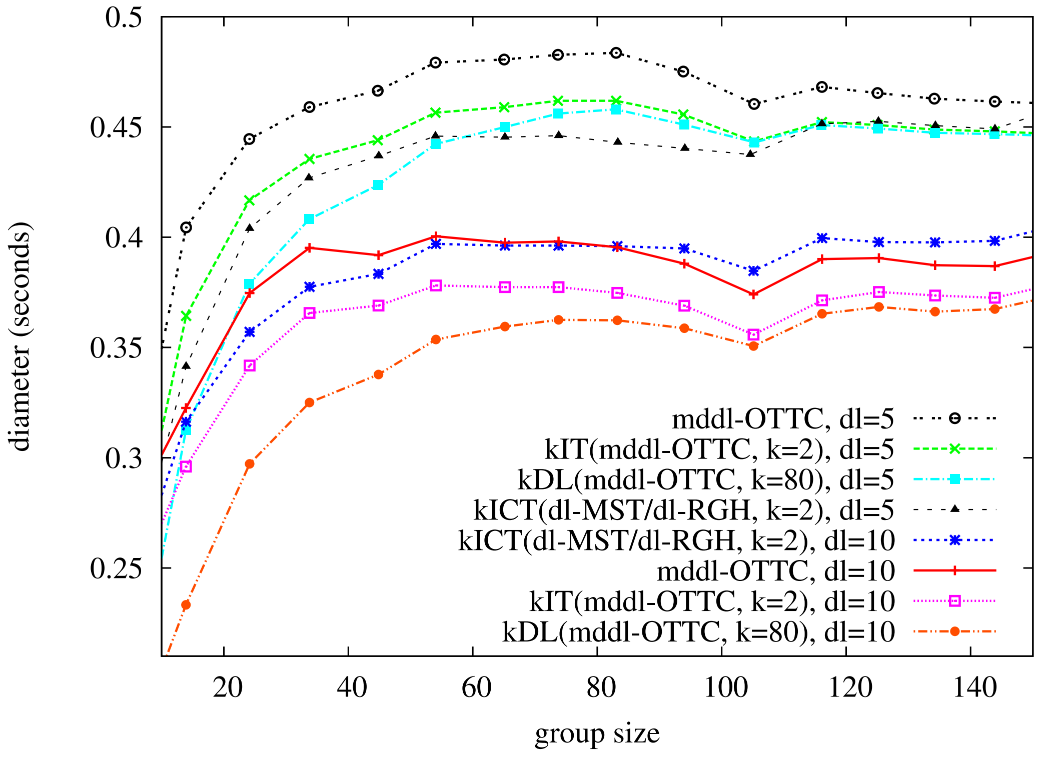
To summarize, we found that doubling the degree-limit from 5 to 10, enabled the subgraph algorithms to construct subgraphs with a lower diameter. The main reason for this is that their success rates were increased, and the increased degree capacity enabled the degree-limited algorithms to add more edges adjacent to centrally located (well-placed) nodes (see chapter 9). Chapter 8 introduced the success rate to be the rate of which a constrained algorithm successfully completes, for example, an overlay construction.

11.2.6 Discussions on spanning-subgraph algorithms

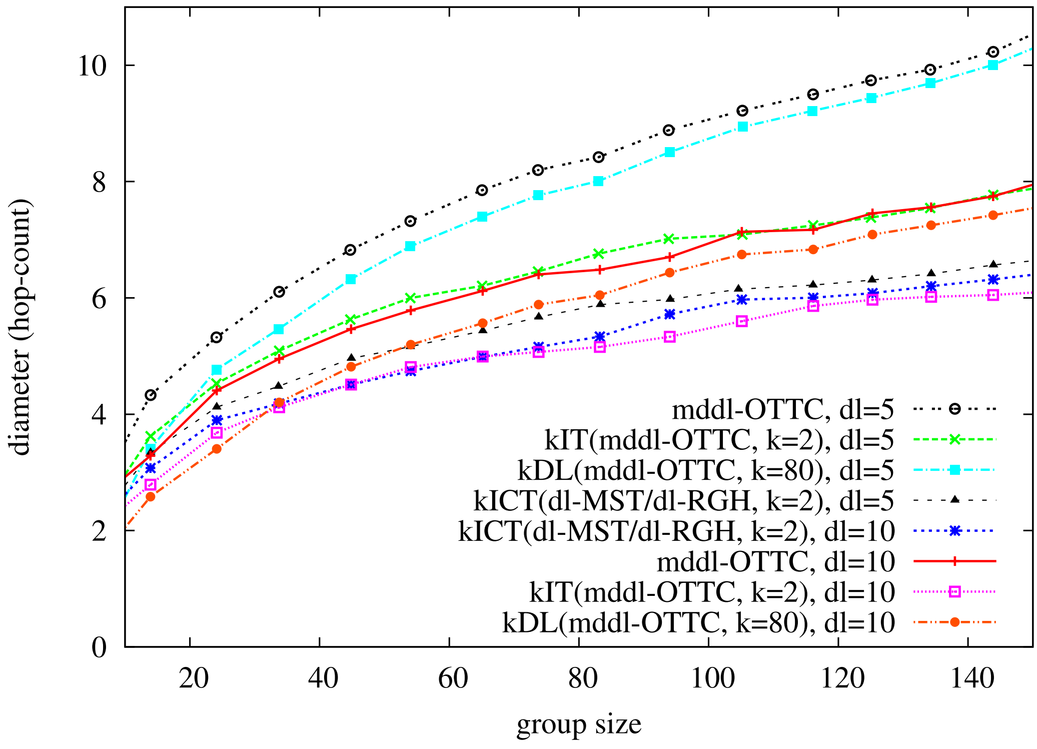
From our results we see that a tree is able to compete with a mesh when it comes to the latency diameter in an overlay. However, a mesh does have advantages in the hop-diameter. The main drawback of meshes is the added cost they incur, whereas the upside is the added failure tolerance. A tree is much cheaper, but is more failure prone.

Multicasting shared events in a tree is cheaper than multicasting events in a mesh. The extra bandwidth consumption in a mesh is not desirable, and may force some packet-routing on top of the mesh. In such cases it may be just as well to use a source-based tree for each client.

Among the mesh algorithms that use tree algorithms as input it is the dl-MST algorithm that has the largest diameter reduction. Furthermore, the edge pruning algorithm dl-aCLO produces

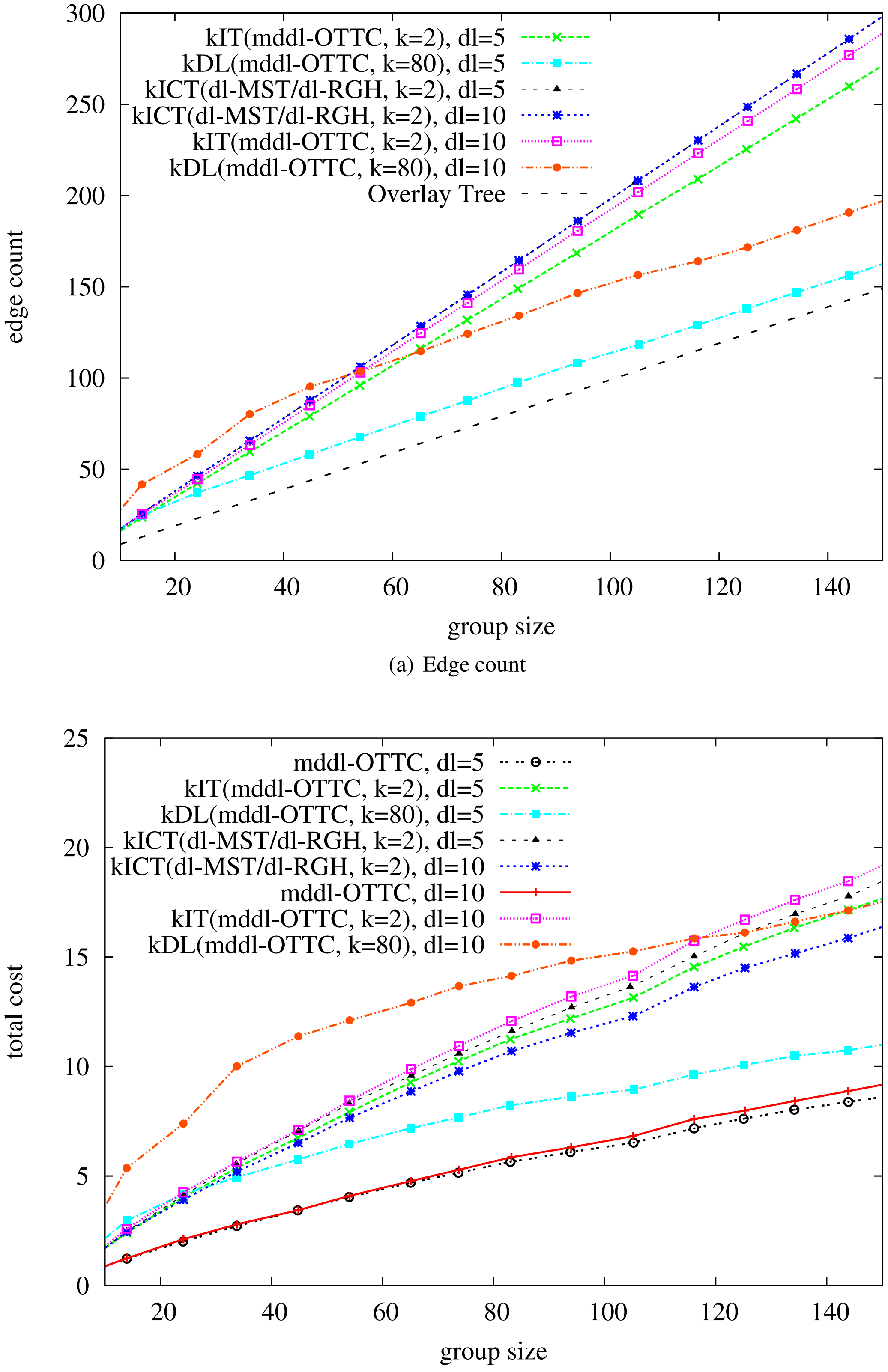


(a) Diameter (seconds)



(b) Diameter (hops)

Figure 11.7: Diameter (seconds and hop-count) of the subgraphs, constructed by selected subgraph algorithms with degree-limits (dl) 5 and 10.



(a) Edge count

(b) Total cost

Figure 11.8: Edge-count and total-cost of the subgraphs, constructed by selected subgraph algorithms with degree-limits (dl) 5 and 10.

meshes with a low diameter and total cost, and is very fast; however, it cannot be configured into a tree structure.

Table 11.4 illustrates our subjective opinions on how to configure some selected mesh algorithms with the optimal k value. For each target metric an optimal k value is listed. These are averaged into the proposed k, which is what we consider the better way to configure the mesh algorithm. The mesh algorithm combinations with mddl-OTTC and dl-SPT use very low k-values, whereas the dl-MST combinations use higher k-values. The difference between dl-MST, dl-SPT and mddl-OTTC mesh algorithm combinations is reduced significantly with these configurations. For example, in the kIT algorithms, a tree structure is what we consider the most optimal for dl-SPT and mddl-OTTC.

Algorithm			Optimal k			Proposed k
Mesh	Tree	k-range	*Total cost*	*Diameter*	*Hop-diam.*	
kIT	mddl-OTTC	$[1,4]$	1	2	2	1
	dl-SPT	$[1,4]$	1	2	2	1
	dl-MST	$[1,4]$	1	3	3	2
kDL	mddl-OTTC	$[0,80]$	0	15	15	7
	dl-SPT	$[0,80]$	0	15	15	7
	dl-MST	$[0,80]$	0	80	80	40
kLL	mddl-OTTC	$[0,80]$	0	40	40	20
	dl-SPT	$[0,80]$	0	40	40	20
	dl-MST	$[0,80]$	0	80	80	40
kBL		$[1,5]$	1	5	5	3
dl-aCLO		$[0,5]$	0	4	4	2

Table 11.4: Proposed k-configurations for a few selected algorithms, with degree-limit 5.

11.3 Group communication simulations of Steiner subgraph algorithms

The following evaluations are of the results obtained from group communication simulations using the Steiner subgraph algorithms to build overlay networks for event distribution.

We evaluated Steiner-tree algorithms in section 10.3 and found that Steiner-tree spanning-heuristics can be efficiently used to find close-to-optimal Steiner-trees of minimum diameter. We also showed that many existing Steiner tree heuristics that only consider shortest paths as a way to include Steiner points, do not work on a complete graph (full mesh) made of shortest paths. Therefore, the following experiments do not use such Steiner tree heuristics, but rather uses the Steiner tree spanning-heuristics.

Steiner-tree spanning-heuristics are regular spanning-tree algorithms that construct a spanning-tree on an input graph that includes Steiner points, and then removes leaf Steiner points. Table 11.5 summarizes the algorithm details for the Steiner-tree spanning-heuristics that are used in the group communication exeperiments on Steiner subgraph algorithms.

Algorithm	*Meaning*	*Optimization*	*Constraints*	*Complexity*	*Problem*	*Reference*
smddl-OTTC	Steiner minimum diameter degree-limited OTTC	diameter	degree	$O(n^3)$	Steiner-MDDL	[131]
sdl-OTTC	Steiner degree-limited OTTC	total cost	diameter and degree	$O(n^3)$	BDDLSMT	[131]
sdl-RGH	Steiner degree-limited RGH	total cost	diameter and degree	$O(n^2)$	BDDLSMT	[131]
sdl-SPT	Steiner degree-limited Dijkstra's SPT	core/destination cost	degree	$O(n^2)$	Steiner-MRDL	[95]
sdl-MST	Steiner degree-limited Prim's MST	total cost	degree	$O(n^2)$	d-SMT	[95]

Table 11.5: Steiner tree construction algorithms ($\mathscr{A}_{\mathscr{T}}$).

11.3.1 Experiment configurations

In the experiments, the complete group-graphs include k Steiner-points. The Steiner-points are added to the input graph to enable the Steiner subgraph algorithms to reduce the diameter and increase their success rate. We use equation 8.1 to calculate the number of Steiner-points to include. More specifically, we use the degree limit d in the current experiment and the current group size $|V|$: $|O| = |V|/d$. The function approximates the number of Steiner-points that are needed to ensure that the degree-limited subgraph algorithms are able to build a subgraph. The k Steiner points are chosen by k-Median(k) from 100 Steiner-points (super-nodes) that are identified at the beginning among the 1000 nodes in the overlay network. These well-placed super-nodes (chapter 8) are found by the multiple core-node selection algorithm k-Center($k = 100$). If not otherwise noted, we use k-Median($k = 1$) to select a source node for every Steiner-tree and subgraph heuristic (see chapter 7.6.1).

A low diameter is a target metric, such that in our simulations we use a strict diameter bound of 0.250 to the diameter bounded algorithms. The heuristics are then frequently forced to relax the bound. Table 11.6 summarizes the experiment parameters.

Description	*Parameter*
Placement grid	$100x100$ milli-seconds
Number of nodes in the network	1000
Degree limit	5 and 10
Diameter bound	250 milli-seconds
Super-nodes found by k-Center-Selection(k)	$k = 100$
Steiner-point size	(group-size/degree-limit)

Table 11.6: Experiment configuration.

11.3.2 Evaluated target metrics

A Steiner subgraph algorithm is considered good if it can produce overlays with a low diameter, a low average pair-wise distance, within a reasonable time that does not add unreasonable cost to the network. For the evaluation of the overlays and algorithms, there are four metrics considered to be very important: overlay diameter, average pair-wise distance, algorithm execution time, and total network cost. In addition, given the fairly limited resources available to average clients in the Internet, the algorithm should obey degree-limitations such that the stress on each node in the overlay is bounded.

11.3.3 Comparison of spanning- and Steiner-subgraph algorithms

Chapter 11.2 evaluated spanning-subgraph algorithms and it was found that with a fairly limited amount of added edges (from a tree) the subgraph has a close-to-optimal diameter and pair-wise latency. In the following, the evaluation include the same subgraph algorithms, only applied to a Steiner subgraph scenario. The major difference is that these Steiner subgraph algorithms include Steiner points to the input graph.

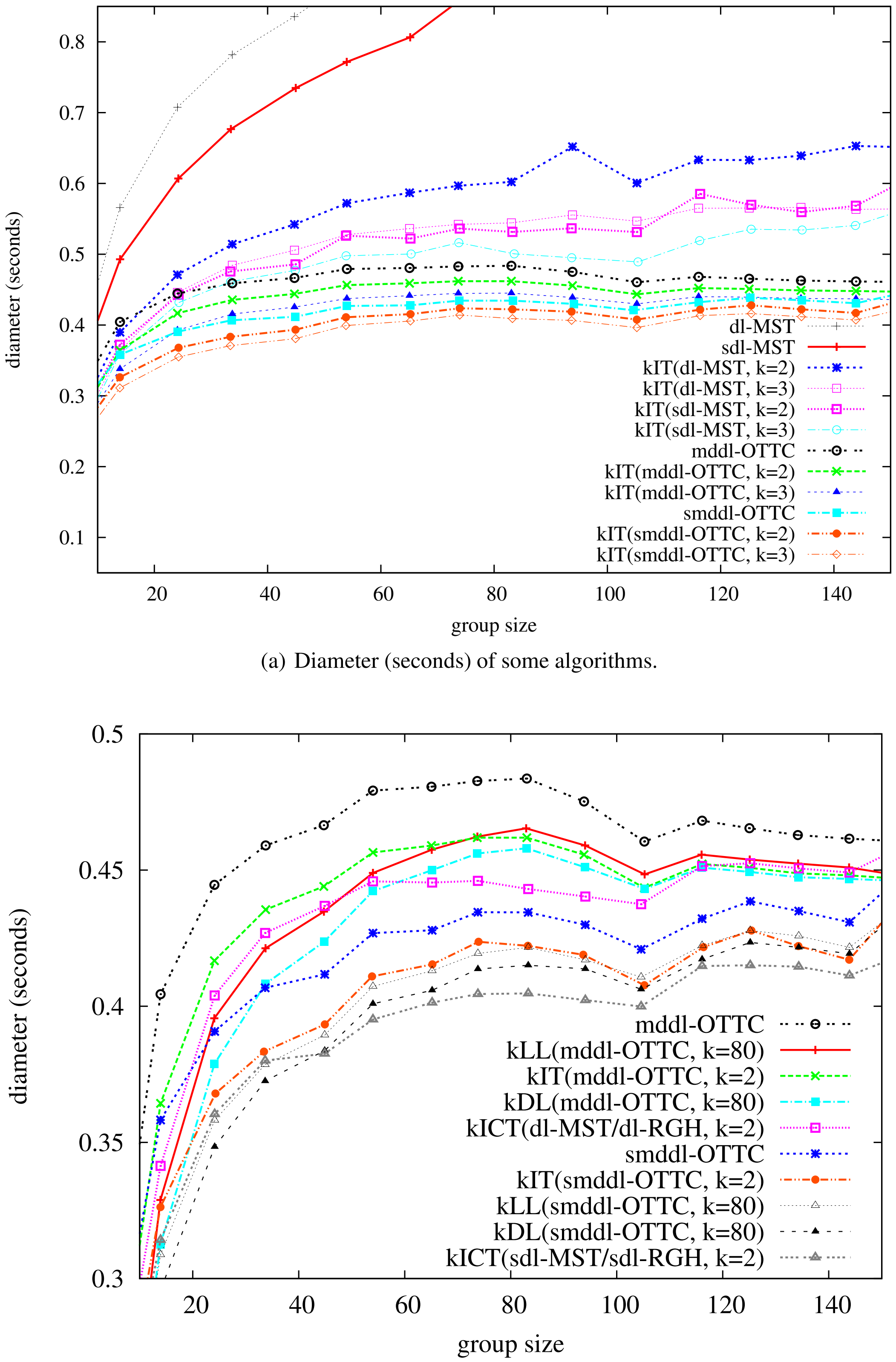
The following sections first compare spanning- and Steiner-subgraph algorithms using a static degree-limit of 5, then we add a degree-limit of 10 and observe the results.

Comparisons with a degree-limit of 5

Figure 11.9(a) plots the diameter (seconds) for some selected spanning-subgraph and Steiner-subgraph algorithms. It is evident that both tree-algorithms dl-MST and Steiner dl-MST (sdl-MST) produce trees with too high diameter. The main observation, is that the diameter is lower for Steiner-subgraph algorithms than spanning-subgraph algorithms. It is interesting that Steiner mddl-OTTC (smddl-OTTC) produce subgraphs with a lower diameter than spanning-subgraph kIT(mddl-OTTC,2) and almost as low as Steiner-subgraph kIT(smddl-OTTC,2). Figure 11.9(b) further compares the diameter (seconds) for the top performing spanning subgraph and Steiner subgraph algorithms. It is quite clear that all the Steiner subgraph algorithms perform better than spanning-subgraph algorithms. The main reason for this is the increased degree capacity available among the Steiner points.

Figure 11.11(a) and 11.11(b) compares the diameter (hop-count), corresponding to Figure 11.9(a) and 11.9(b). From these figures we see that the hop-count diameter suffers when Steiner points are added to the input graph. However, it is the tree-algorithms that suffer most, for the Steiner subgraph algorithms kIT and kDL there is only a slight difference.

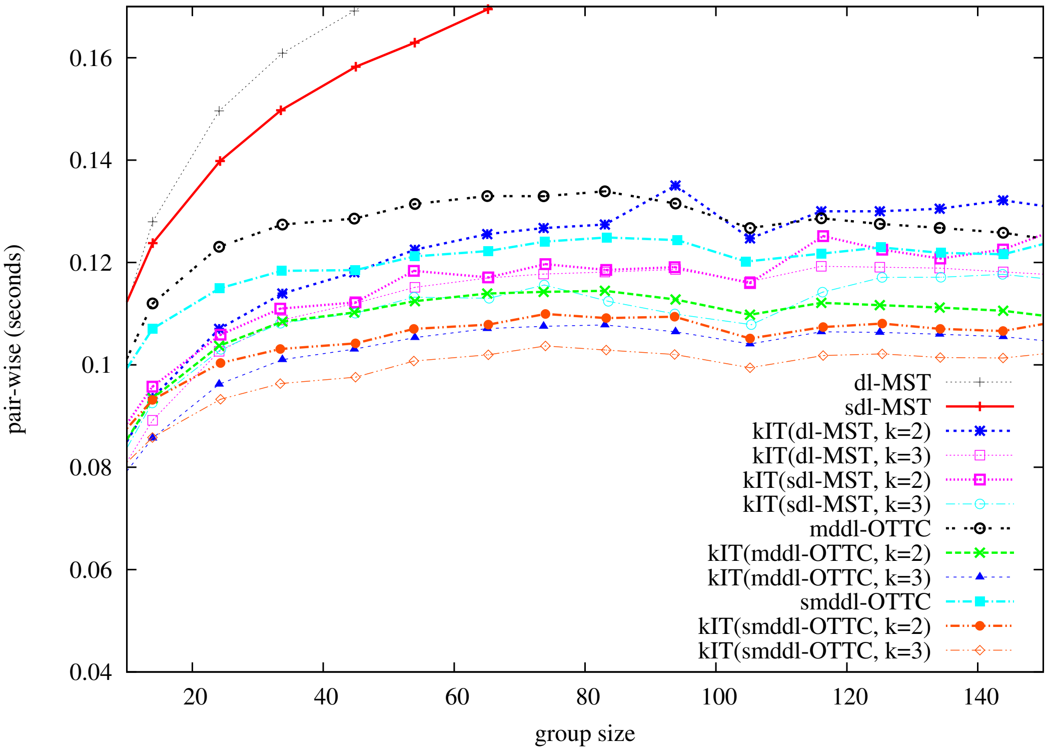
Figure 11.12(a) and 11.12(b) compares the total cost of the subgraphs (corresponding to previous figures). We see that adding Steiner points to the input graph does not significantly add to the total cost of the subgraph. We see that smddl-OTTC produce very low cost trees



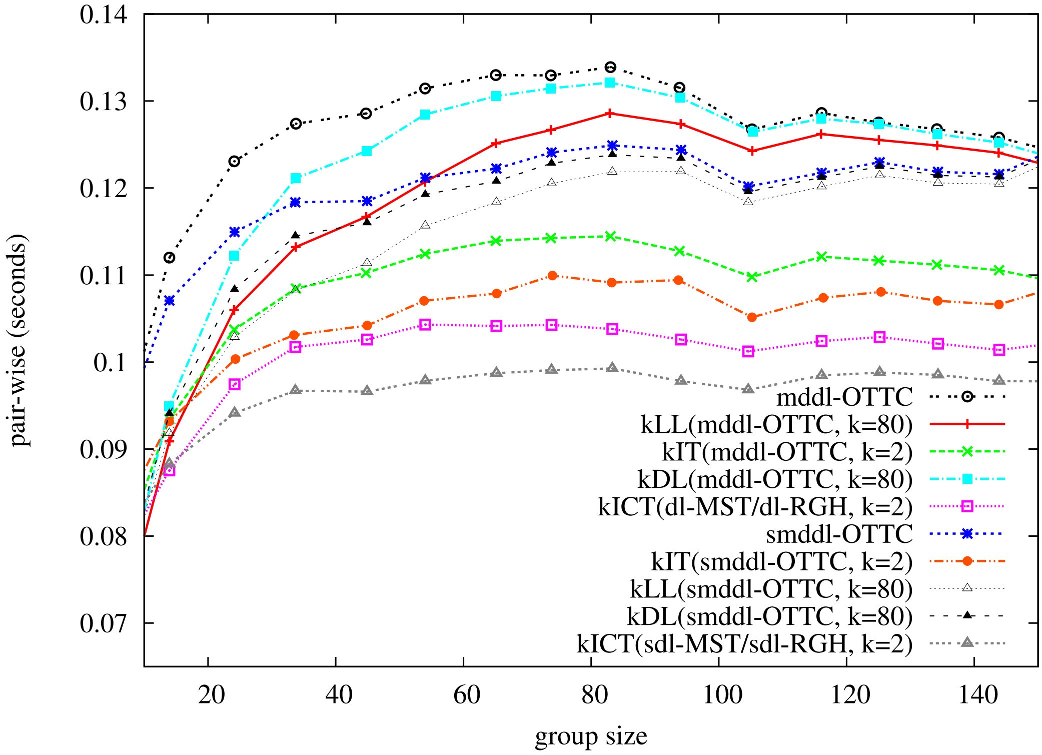
(a) Diameter (seconds) of some algorithms.

(b) Diameter (seconds) of some algorithms.

Figure 11.9: Spanning-subgraph and Steiner subgraph algorithms (diameter (seconds)).

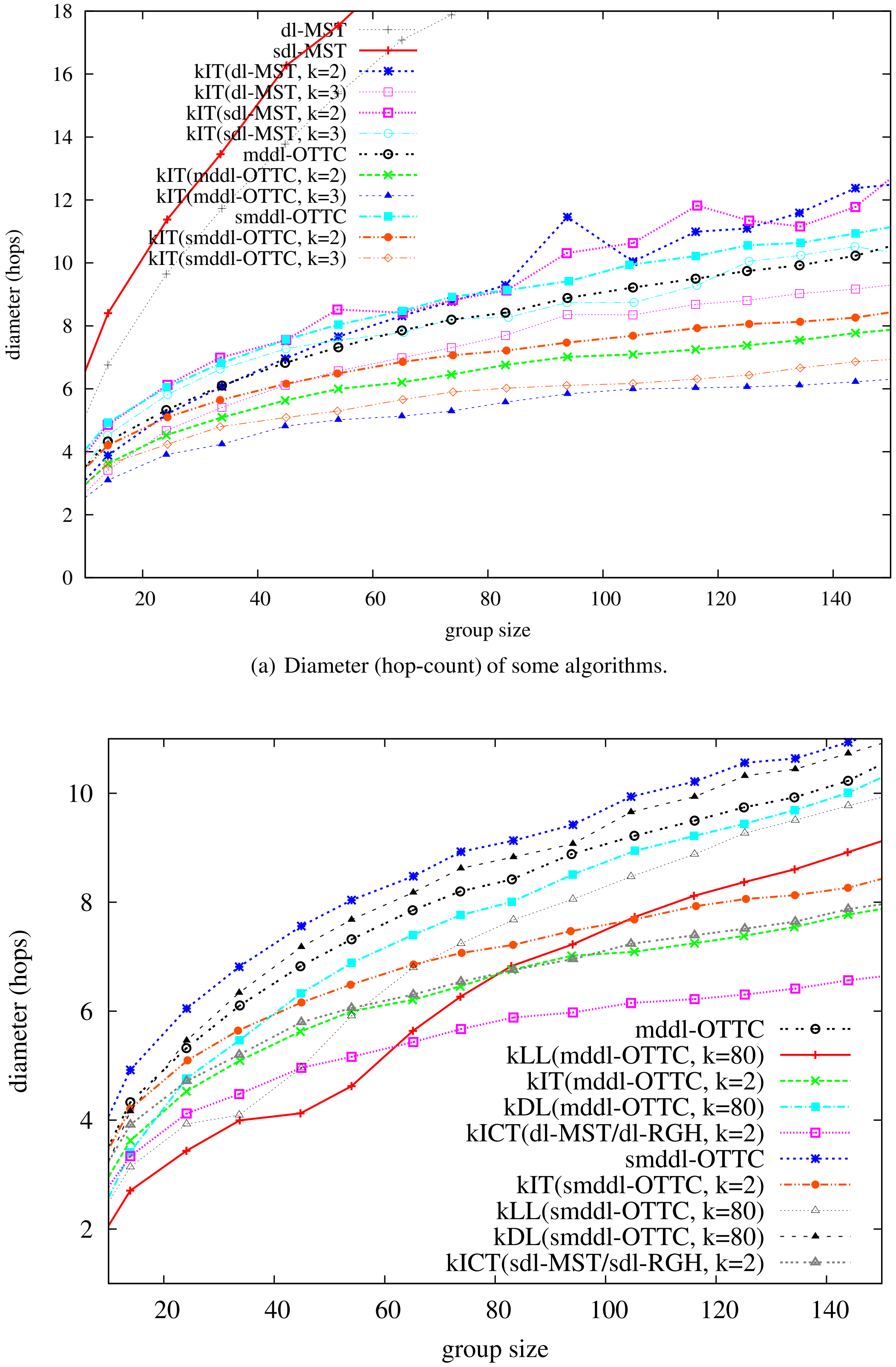


(a) Pair-wise (seconds) of some algorithms.



(b) Pair-wise (seconds) of some algorithms.

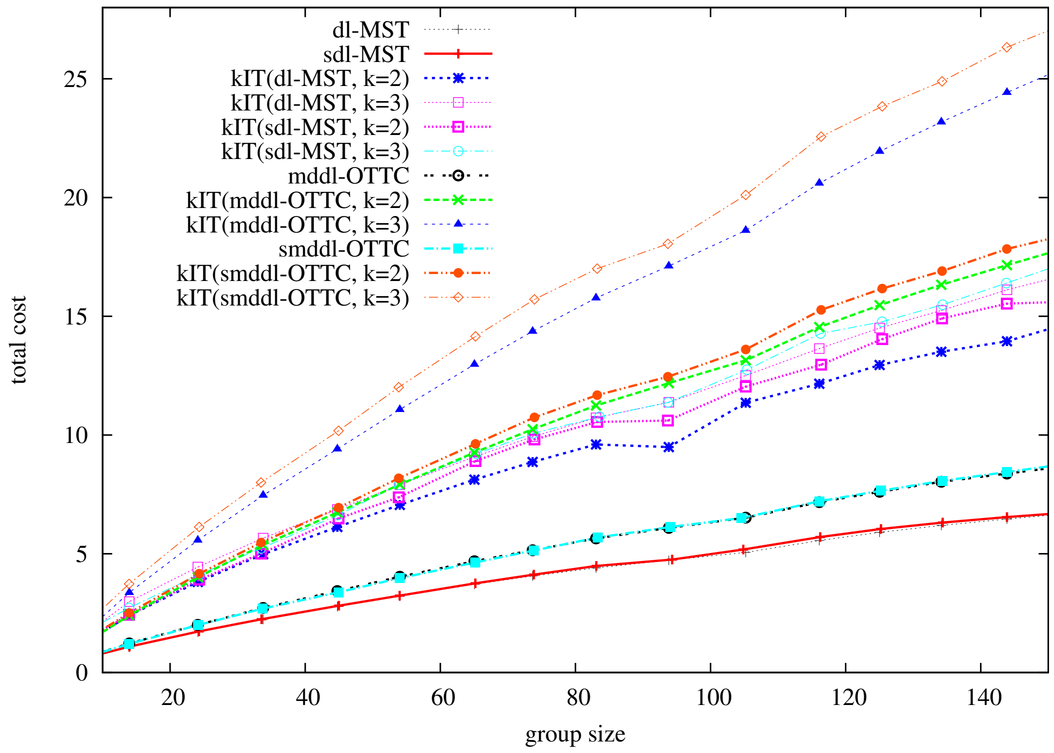
Figure 11.10: Spanning-subgraph and Steiner subgraph algorithms (pair-wise (seconds)).



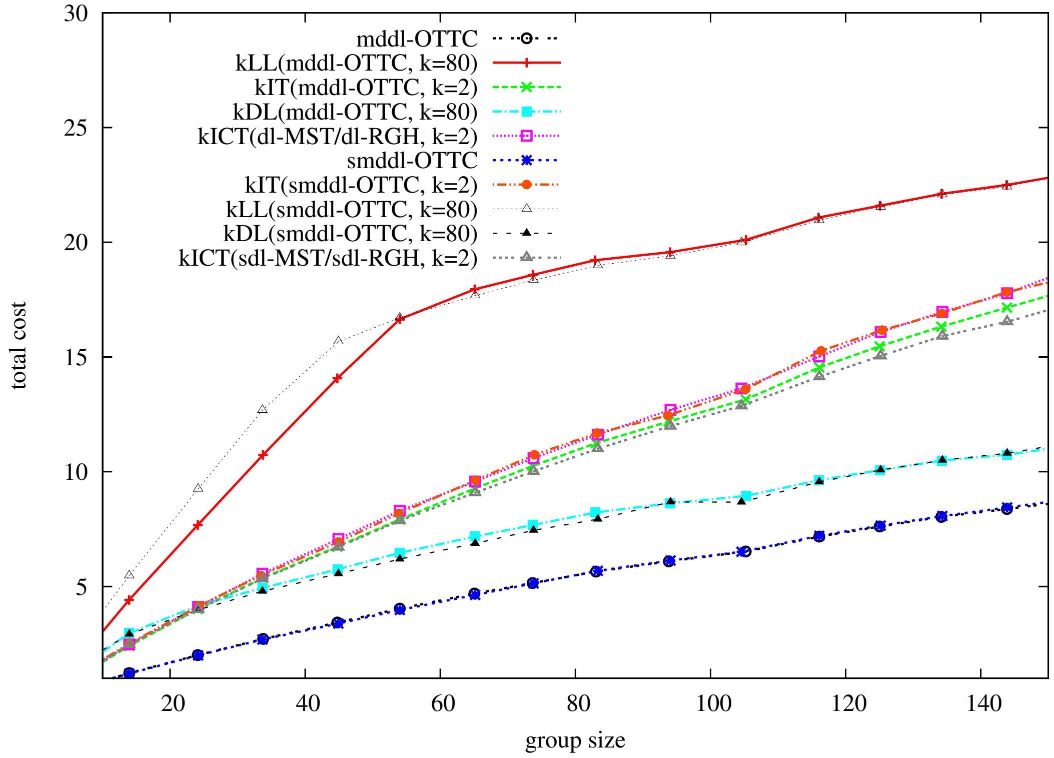
(a) Diameter (hop-count) of some algorithms.

(b) Diameter (hop-count) of some algorithms.

Figure 11.11: Spanning-subgraph and Steiner subgraph algorithms (diameter (hop-count)).

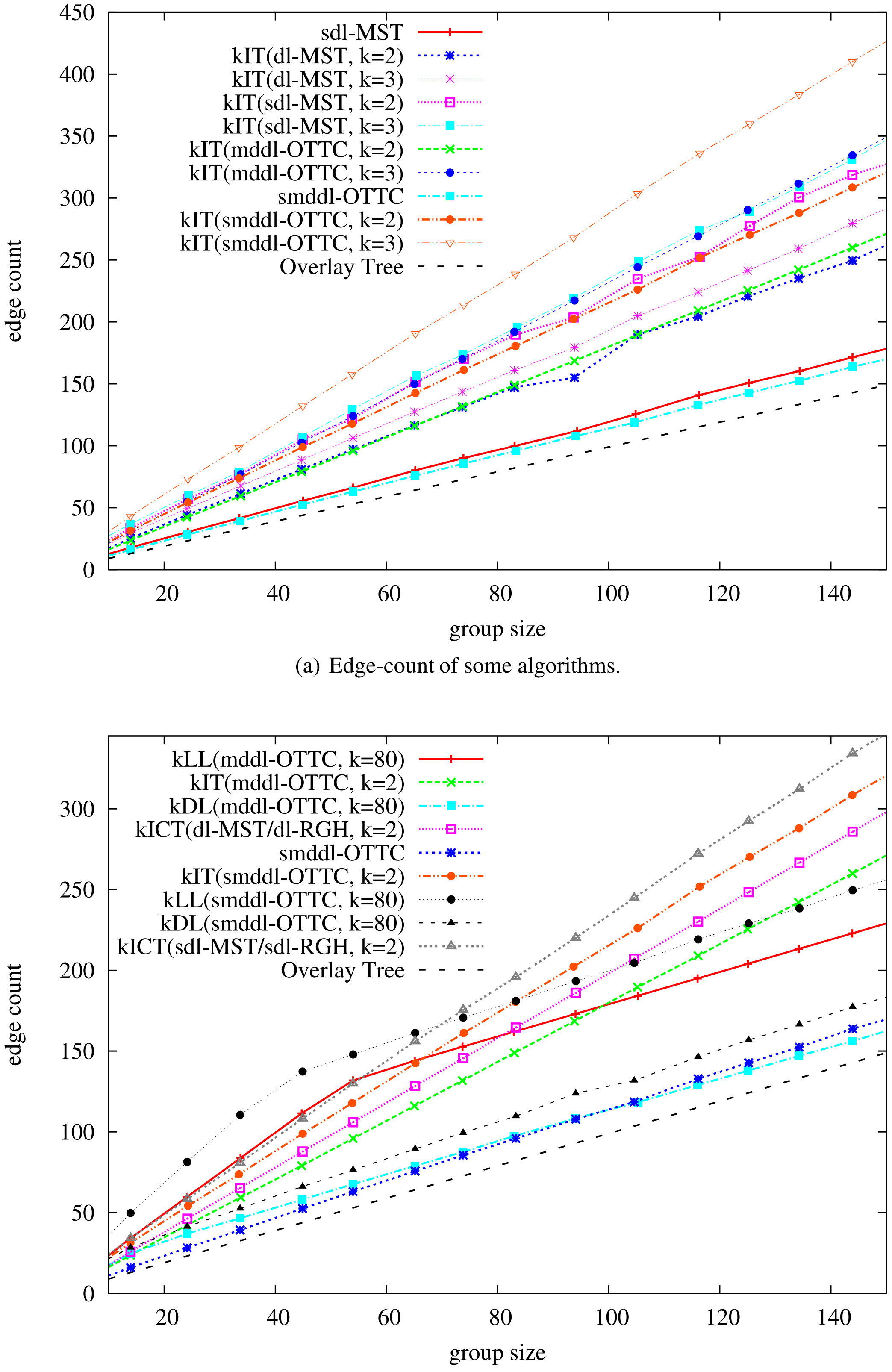


(a) Total cost of some algorithms.



(b) Total cost of some algorithms.

Figure 11.12: Spanning-subgraph and Steiner subgraph algorithms (total cost).



(a) Edge-count of some algorithms.

(b) Edge-count of some algorithms.

Figure 11.13: Spanning-subgraph and Steiner subgraph algorithms (edge-count).

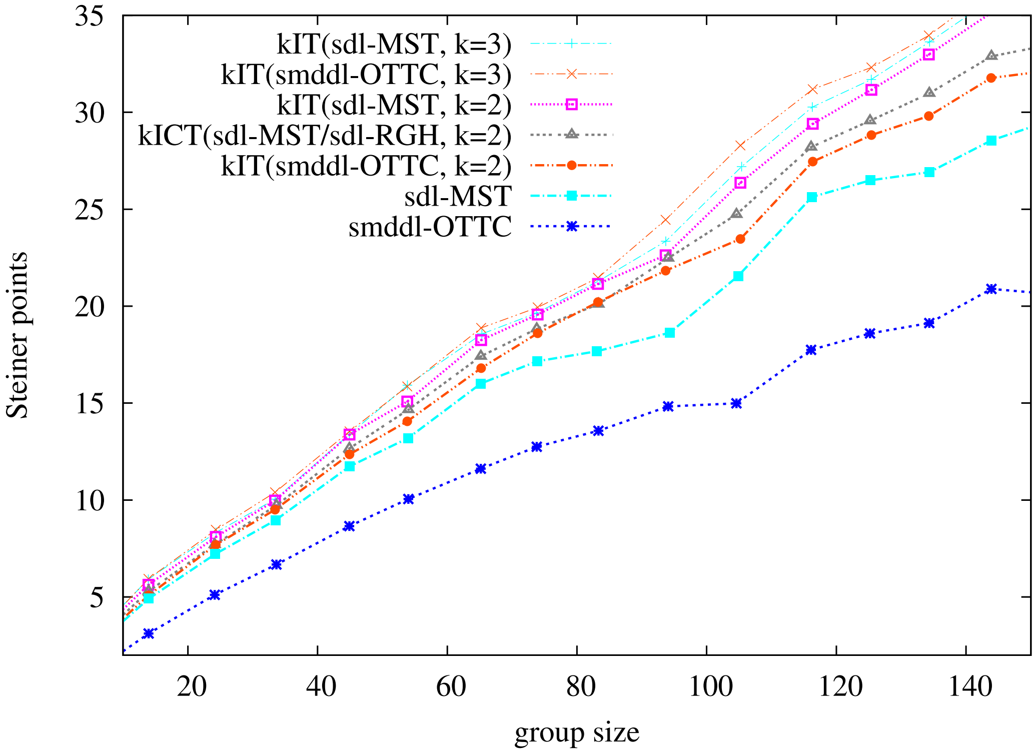


Figure 11.14: Spanning-subgraph and Steiner subgraph algorithms (Steiner points).

compared to kIT(smddl-OTTC, 2). From figure 11.9(a) it was identified that smddl-OTTC only produced subgraphs (trees) with slightly higher diameter than kIT(smddl-OTTC, 2). Hence, this means that smddl-OTTC is a better alternative, even if the total cost is unimportant.

Figure 11.13(b) compares the number of edges the spanning-subgraph and Steiner-subgraph algorithms are able to include. On average, the Steiner-subgraph algorithms manage to include more edges during overlay construction, which is due to the increased degree capacity from the Steiner points. Figure 11.14 plots the number of Steiner points selected to be in the subgraphs for the Steiner-subgraph algorithms. It is the minimum-cost algorithm sdl-MST that includes the most Steiner points, regardless of Steiner subgraph algorithm (kIT, kLL or kDL). smddl-OTTC has a lower number of Steiner points.

Comparisons with degree-limits 5 and 10

In the previous results the degree-limit was set to 5, the following includes a comparison of the results when a degree-limit of 10 is used.

The subgraph *diameter (seconds)* as produced by selected subgraph algorithms is plotted in figure 11.15(a) and 11.15(b). We observe that the behavior of the subgraph algorithms is fairly predictable. A higher degree-limit enables the subgraph algorithms to construct subgraphs of lower diameter, and we observe that an increased degree capacity has the most influence on reducing the subgraph diameter. Adding Steiner-points to the input graph has the second most influence on reducing the subgraph diameter. These Steiner-points enable subgraph algorithms to add adjacent edges to centrally located nodes. Then, thirdly, letting the diameter-reducing tree-heuristics (for example, smddl-OTTC) include more edges to the graph, also has a positive effect and reduces the diameter. The combined, a higher degree-limit, Steiner-points in the input graph and more subgraph edges, yield the best results and the subgraphs with the lowest diameters.

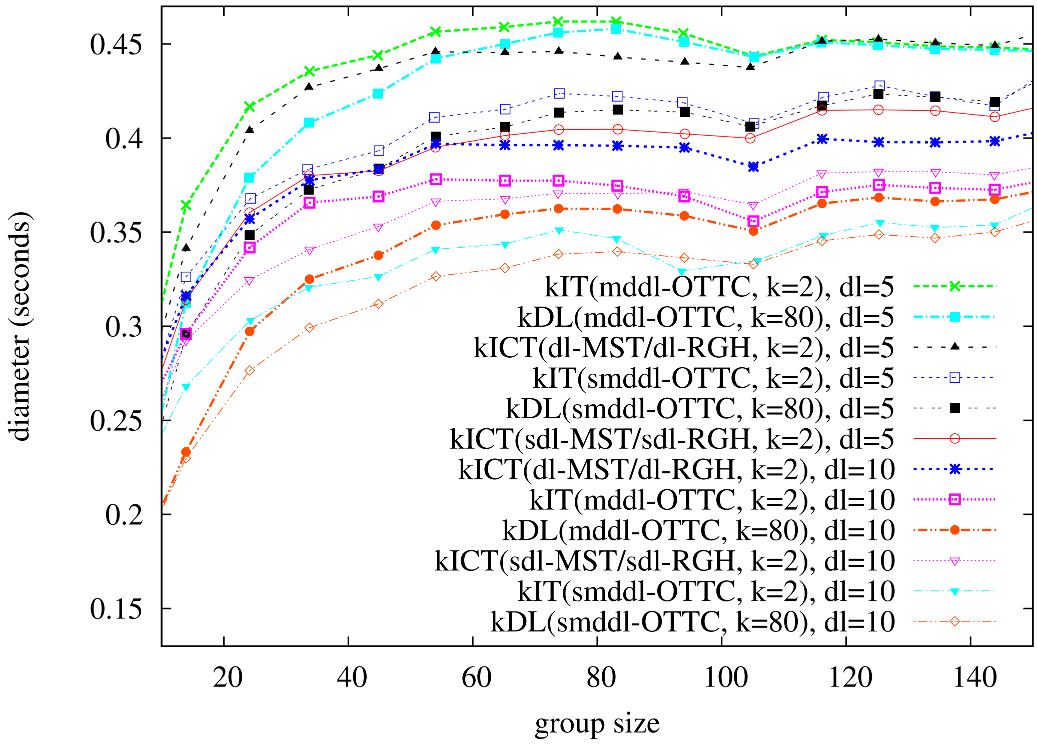
The subgraph *hop-diameter* as produced by selected subgraph algorithms is plotted in figure 11.16(a) and 11.16(b). Generally, we see that an increased degree-limit and more edges to the subgraph reduces the hop-diameter. However, it is the increased degree-limit that has the most effect on reducing the hop-diameter. Adding Steiner-points increases the hop-diameter, as was also found in chapter 10, where Steiner-tree algorithms were evaluated.

The subgraph *edge-count* in figure 11.17 confirms our previous observations in that a higher degree-limit and added Steiner-points increases the success-rates of the subgraph algorithms. The subgraph algorithms are able to included more edges due to this (higher degree-limit and added Steiner points) and a lower diameter is the result. We also see this from the number of *steiner-points* in the subgraphs (figure 11.18).

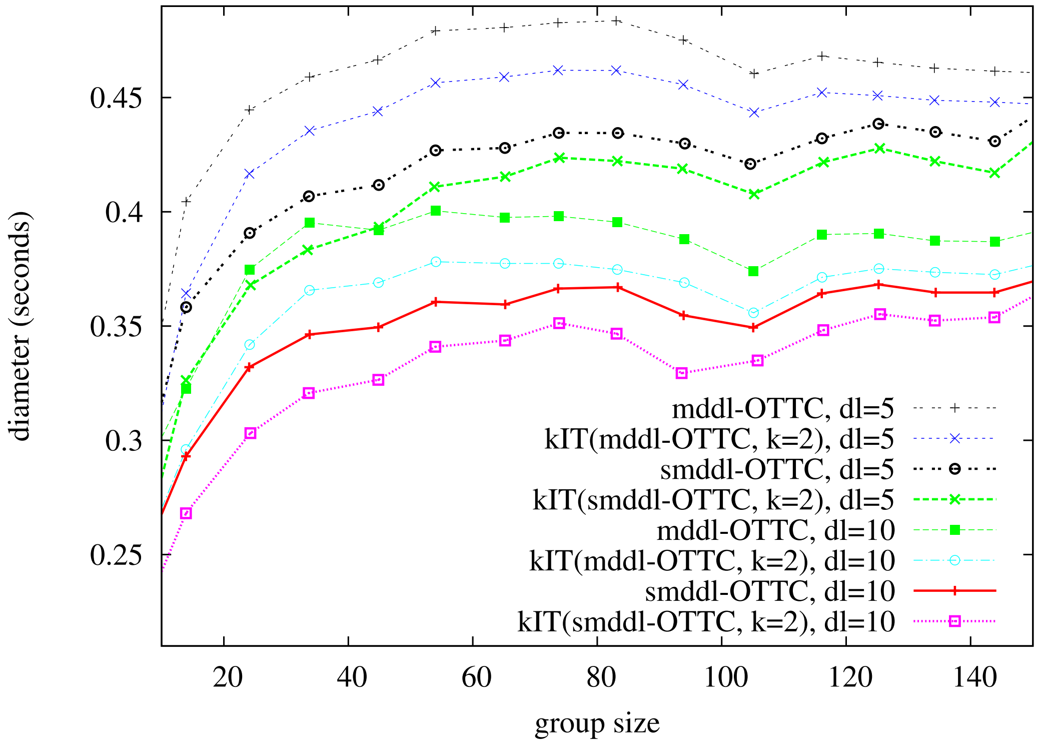
11.3.4 Results from one group size range

Similarly to section 11.2, we evaluate the results from our simulations on Steiner subgraph algorithms. The main plot is figure 11.19, which includes a complete comparison of predominantly degree-limited (dl = 5) Steiner-tree and Steiner-mesh algorithms evaluated towards our target metrics. It includes statistics from overlays of sizes between 100 and 120. The Steiner-tree spanning-heuristics are plotted as interleaved tree algorithms with $k = 1$.

We observe that the better Steiner-tree spanning-heuristics achieve a *diameter* of around 0.5 seconds. For the degree-limited algorithms we can see a larger reduction in the diameter, but even here it is not significant. Among the enhanced-tree algorithms we see that it is only variations with sdl-MST that reduces the diameter, and applying smddl-OTTC to kLL and kDL has little or no effect in this group size range (100-120). The kICT and kPCT Steiner subgraph variations construct subgraphs with the lowest diameter and pair-wise latency. These (kICT and kPCT) combine a diameter-reducing Steiner-tree spanning-heuristic with the minimum-cost Steiner-tree spanning-heuristic sdl-MST. The combination low-cost and low-diameter algorithm fit especially well together because they pick very different tree-edges. Low-cost algorithms minimize the cost between nodes that are close in the input graph, while low-diameter algorithms reduce the maximum-distance between nodes in the input graph. The result is a subgraph that exhibits low pair-wise latency and low diameter. sdl-aCLO is an attempt to automate this process in terms of execution time, both kICT and kPCT are dependent on the complexities

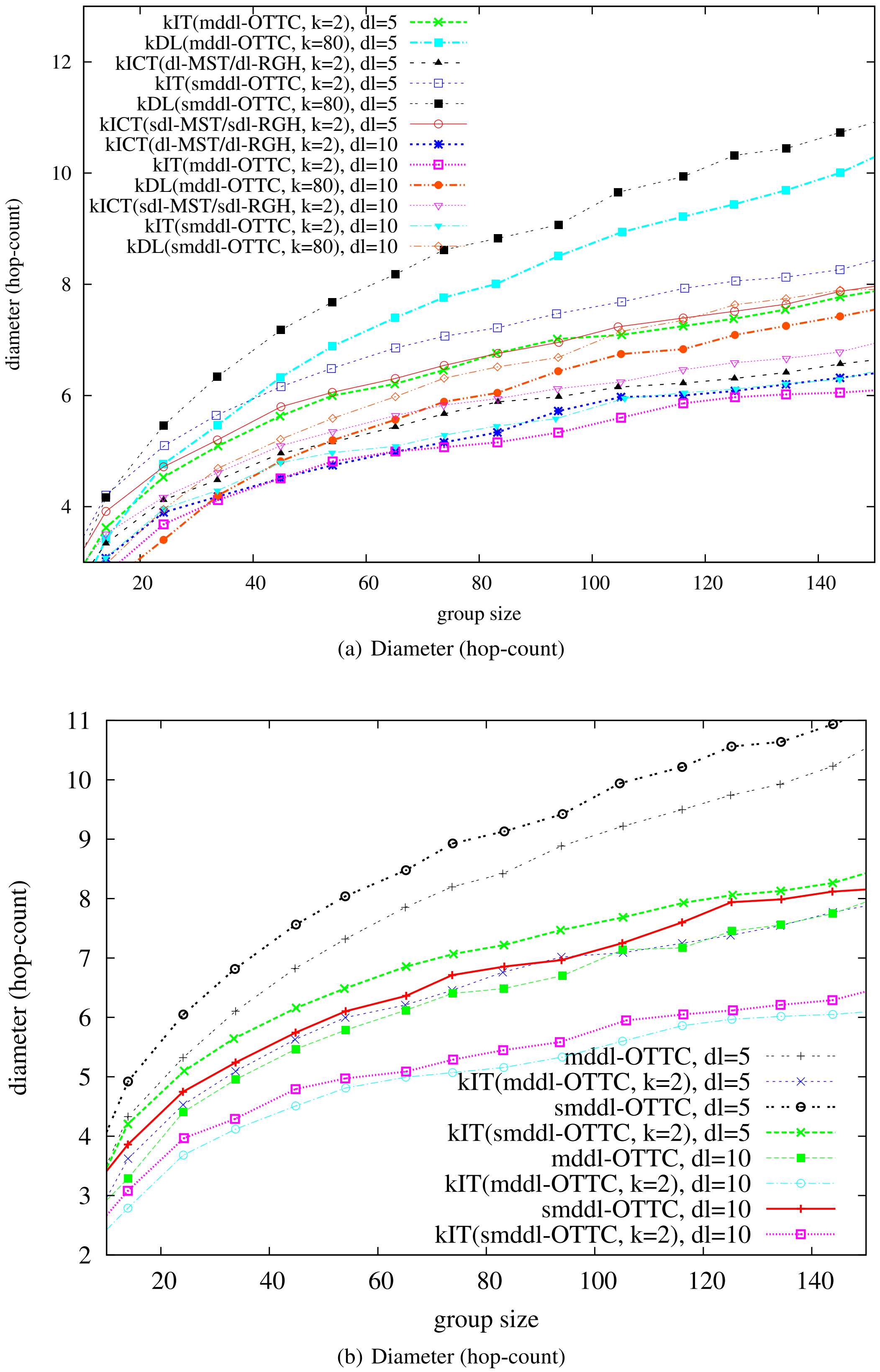


(a) Diameter (seconds)



(b) Diameter (seconds)

Figure 11.15: Spanning-subgraph and Steiner subgraph algorithms (diameter (seconds)).



(a) Diameter (hop-count)

(b) Diameter (hop-count)

Figure 11.16: Spanning-subgraph and Steiner subgraph algorithms (diameter (hop-count)).

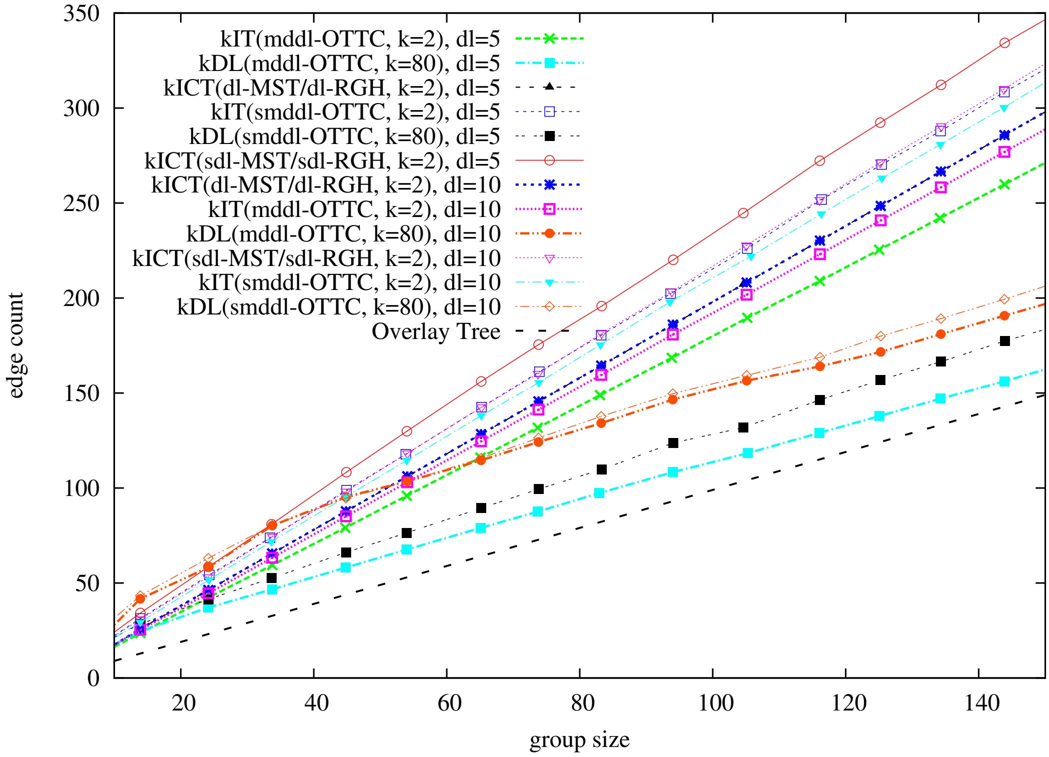


Figure 11.17: Spanning-subgraph and Steiner subgraph algorithms (edge-count).

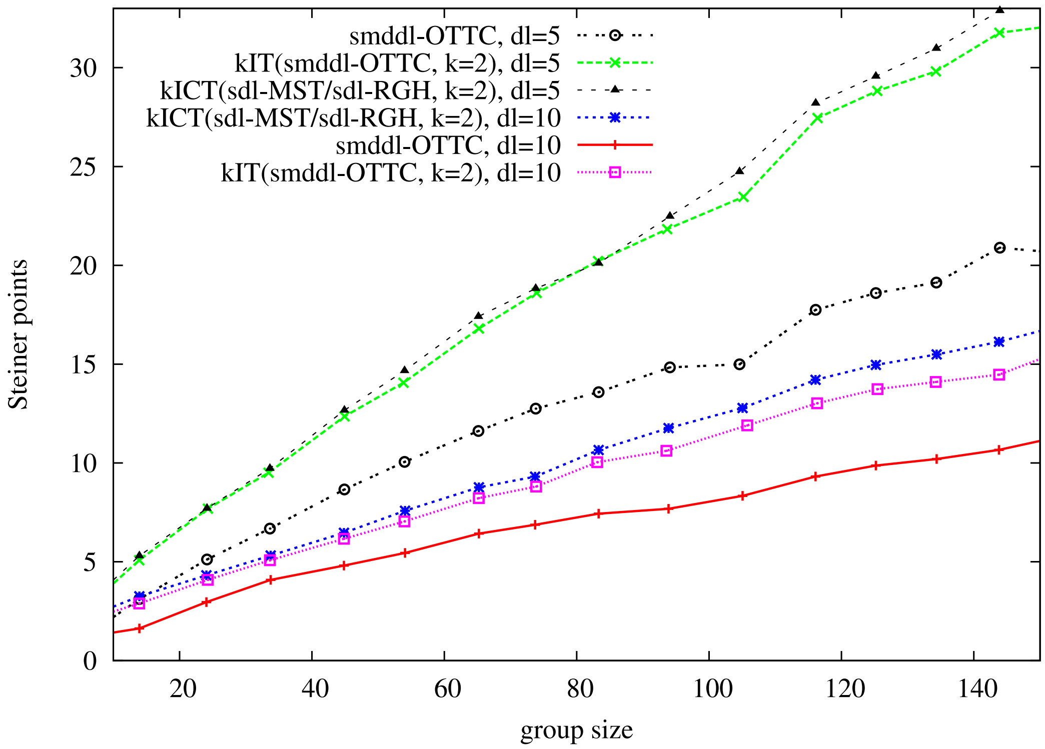


Figure 11.18: Spanning-subgraph and Steiner subgraph algorithms (Steiner points).

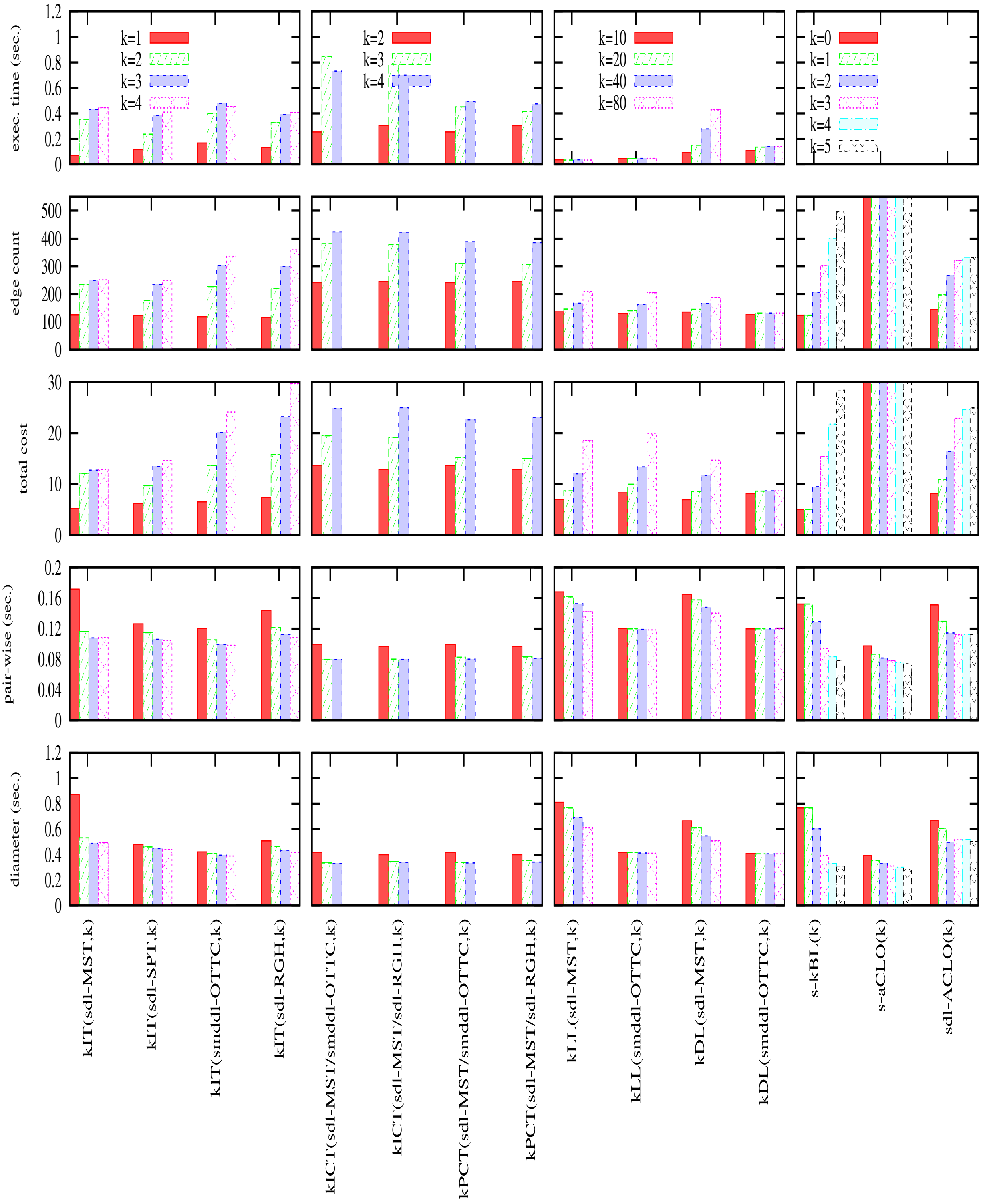


Figure 11.19: Comparison of all tree and mesh algorithms.

of the input algorihtms. Steiner-subgraph sdl-aCLO differs from spanning-subgraph dl-aCLO in that all the core nodes in the sdl-aCLO are Steiner points. We observe similar trends for the average *pair-wise* distance (seconds).

When we analyze the *total cost* of the overlays it is clear that the mesh algorithms build more costly overlays compared to the tree construction algorithms. The enhanced-tree algorithms only slightly increase the total cost of the overlays, due to the rather small k used (maximum 80). The edge pruning algorithms s-kBL and sdl-aCLO also yield a reasonable total cost. We see that the interleaved tree algorithms kIT(sdl-MST,k) and kIT(sdl-SPT,k) construct overlays with a low total cost. However, this is due to the fact that sdl-MST and sdl-SPT often fail to construct degree-limited trees in sparse graphs [130].

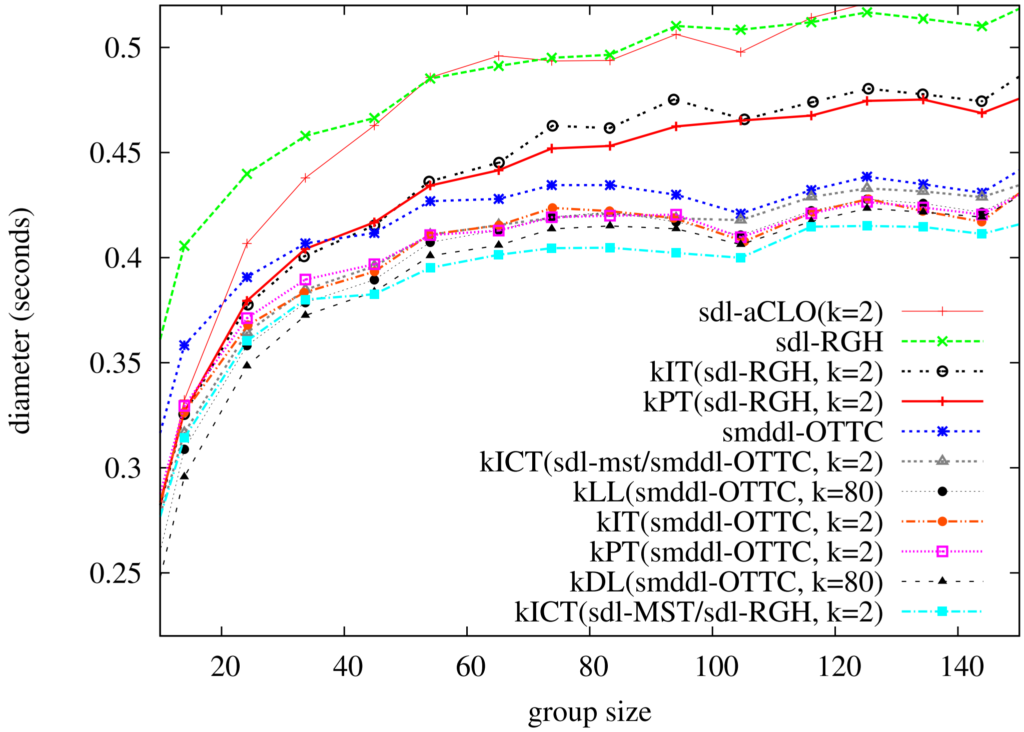
The *edge count* is the number of links in an overlay, and the plots confirm this observation. We see that the edge-count for the kIT(sdl-MST,2) and kIT(sdl-SPT,2) overlays are much lower than the remaining interleaved trees algorithms. The edge pruning algorithm s-aCLO clearly adds too many edges to the overlay. For the interleaved-trees algorithms we can see that the edge-count increases in certain intervals for each k increase. The degree-unlimited algorithms merge a new full tree for each round, whereas, the degree-limited algorithms are not able to construct a full spanning tree for each round, in particular when $k > 2$.

The *execution time* shows that the degree-limited interleaved-tree algorithms struggle to find trees when the k increases. The kDL algorithm is quite time-consuming when $k > 40$; however, our implementation may be sub-optimal for kDL. The fastest algorithms are the edge-pruning algorithms, which are barely visible in the graphs.

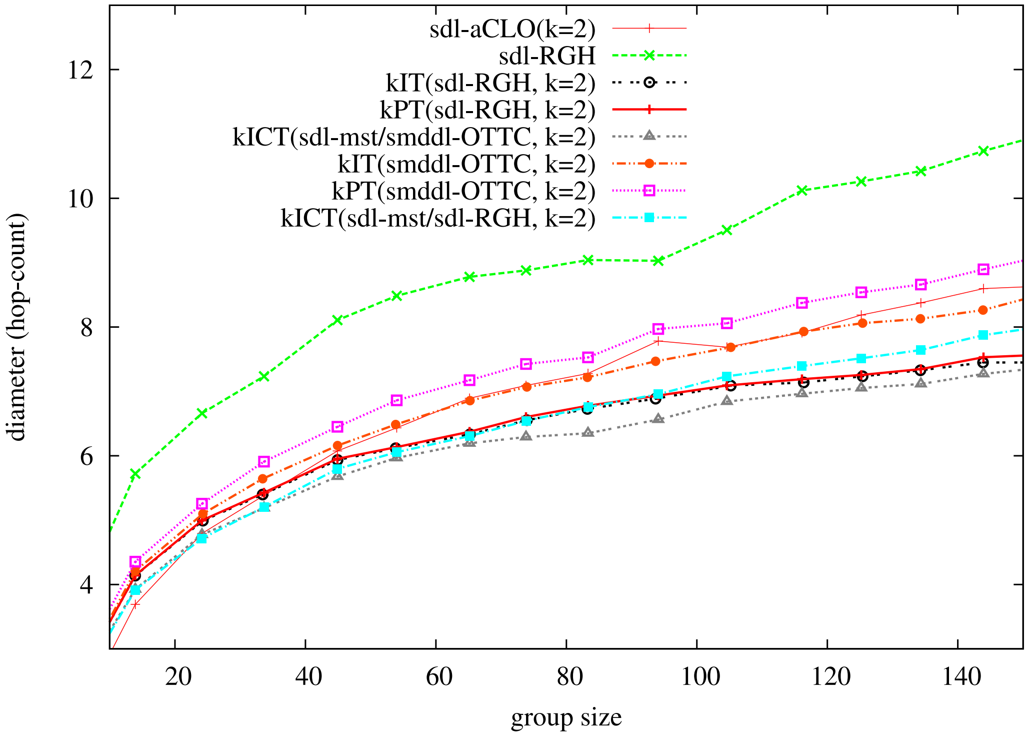
11.3.5 Results with varying group sizes

We saw in section 11.2 from the spanning-subgraph algorithms that the group size may influence the performance of an overlay construction algorithm. In figure 11.20 and 11.21, we plot the diameter, total cost, and overlay edge count for selected algorithms with group sizes between 10 and 150. We include selected results from the interleaved- and enhanced-trees algorithms using the tree algorithms sdl-MST, sdl-RGH and mddl-OTTC as input.

The *diameter* achieved is plotted in figure 11.20(a). We observe that among the plotted subgraph-variations the diameter varies between 0.4 and 0.5 seconds. The variation kICT(sdl-MST/sdl-RGH,2) produces the subgraph with lowest diameter, actually slightly under 0.4 seconds. The randomized bounded-diameter sdl-RGH and the sdl-MST match particularly well for producing low-diameter overlays. This is surprising, since the remaining sdl-RGH variations produce rather high diameters. It is probably because the randomized edge-selection in sdl-RGH picks vastly different edges than dl-MST, enabling the combination to complete 2 overlay constructions using the low degree-limit of 5. The kDL(smddl-OTTC,80) performs second best, and is a good alternative for producing low-diameter subgraphs. Further, wee see that the

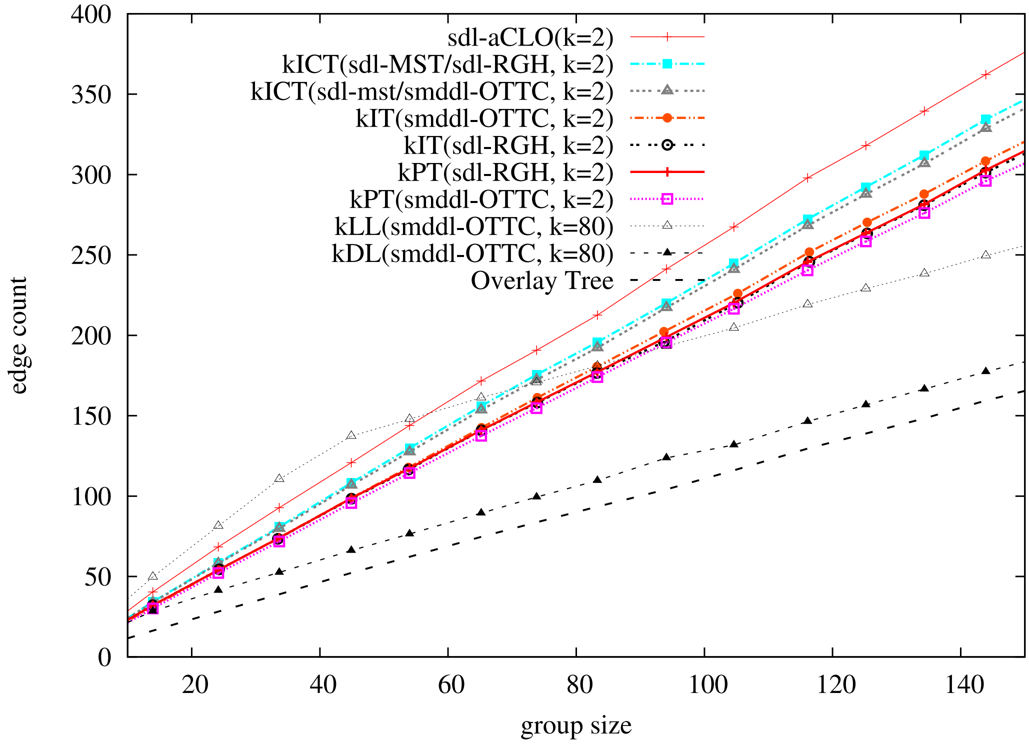


(a) Diameter (seconds)

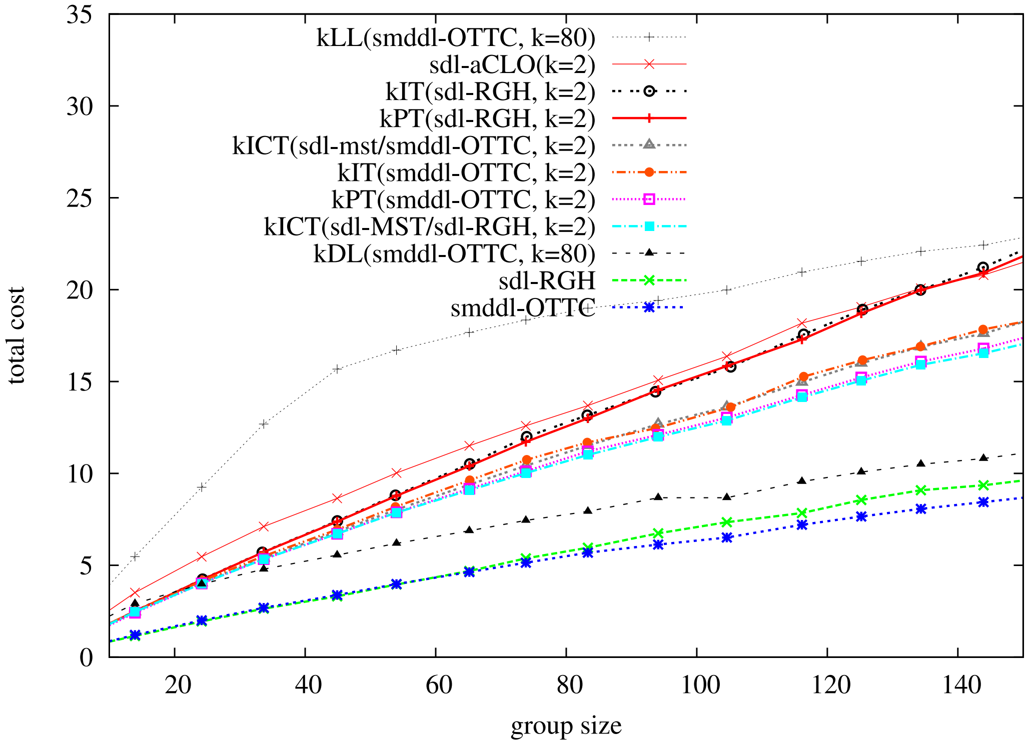


(b) Diameter (hops)

Figure 11.20: Selected overlay construction algorithms (diameter (seconds and hop-count)).



(a) Edge count



(b) Total cost

Figure 11.21: Selected overlay construction algorithms (total cost and edge-count).

reduction in the diameter between smddl-OTTC and kIT(smddl-OTTC,2), is very small. The reamining subgraph construction algorithms combined with smddl-OTTC produce very similar diameter (just above 0.4 seconds). sdl-MST is not plotted but produces a diameter which is around 50 % higher than kIT(sdl-MST,2).

The *hop-diameter* achieved by the algorithms is plotted in figure 11.20(b). Only a few selected subgraph algorithms are plotted, because they all produce very similar hop-diameter. The hop-diameter reduction between sdl-RGH and kIT(sdl-RGH,2) is about 25 % (similarly for smddl-OTTC). It is the kICT(sdl-MST/smddl-OTTC,2) that construct overlays with the lowest hop-diameter, with the remaining variations within firing range. Overall, the hop-diameter stays between 6-9 for group-sizes up to 150, which must be considered good for degree-limited subgraph algorithms.

For the *total cost* plot in figure 11.21(b) wee see that the enhanced-trees algorithm KLL constructs the most expensive overlays. kIT(smddl-OTTC,2) build overlays that are slightly more expensive than kCIT(sdl-MST/sdl-RGH,2). The kDL(smddl-OTTC,80) is not able to add 80 links to each overlay, therefore its total cost is lower than expected. Not surprisingly, the trees constructed by sdl-MST and smddl-OTTC are the cheapest.

The *edge count* in figure 11.21(a) confirms that kDL(mddl-OTTC,80) fails to add 80 links, due to the degree-limitation, but rather only manages to add about 15 links before it gives up. kDL(dl-MST, 80), on the other hand, has a much higher success rate in adding links. Close-to-minimum diameter trees are more star shaped (leafy trees), where inner nodes have a high degree. Therefore, kDL struggles to add edges to the diameter path because the degree capacity quickly gets exhausted.

11.3.6 Discussions on spanning- and Steiner-subgraph results

Chapter 10.3 discussed and compared spanning-tree algorithms and Steiner-tree algorithms, and found that the added Steiner points in the input graph are important for degree-limited algorithms that aim for a low diameter. The added degree capacity in the input graph, helps the degree-limited algorithms to find alternative paths to the member-nodes (targets).

A very similar trend was observed among the evaluated spanning- and Steiner-subgraph algorithms. In this case the trend is also that the added Steiner-points, combined with the Steiner-tree spanning-heuristic smddl-OTTC, minimize the advantage of adding extra edges to reduce the diameter further. The penalty of the added Steiner-points is a slightly increasing hop-diameter. The tree-structures have a higher hop-diameter than the Steiner-subgraph algorithms; however, the difference stays within a couple of hops. The major advantage of tree-structures over cyclic subgraphs is the low cost of trees, which is important in todays bandwidth limited Internet.

Similar to table 11.4, table 11.7 also illustrates our subjective opinions on how to configure some selected mesh algorithms with the optimal k value. For each target metric an optimal k value is listed. These are averaged into the proposed k, which is what we consider the better way to configure the mesh algorithm. However, unlike table 11.4, we here list configurations when the degree-limit is 10. We choose the same low k-values for the the mesh algorithm combinations with smddl-OTTC and sdl-SPT, and also the sdl-MST combinations, which uses a higher k-values. The difference between sdl-MST, sdl-SPT and smddl-OTTC mesh algorithm combinations is reduced significantly with these configurations. However, the enhanced tree algorithm kDL is able to add more edges to the tree when the degree-limit is 10. Therefore, we choose a larger k-value of 20.

Algorithm			Optimal k			Proposed k
Mesh	Tree	k-range	*Total cost*	*Diameter*	*Hop-diam.*	
kIT	smddl-OTTC	$[1,4]$	1	2	2	1
	sdl-SPT	$[1,4]$	1	2	2	1
	sdl-MST	$[1,4]$	1	3	3	2
kDL	smddl-OTTC	$[0,80]$	0	40	40	20
	sdl-SPT	$[0,80]$	0	40	40	20
	sdl-MST	$[0,80]$	0	80	80	40
kLL	smddl-OTTC	$[0,80]$	0	40	40	20
	sdl-SPT	$[0,80]$	0	40	40	20
	sdl-MST	$[0,80]$	0	80	80	40
kBL		$[1,5]$	1	5	5	3
dl-aCLO		$[0,5]$	0	4	4	2

Table 11.7: Proposed k-configurations for a few selected algorithms, with degree-limit 10.

11.4 Summary of the main points

We have compared a range of tree and mesh construction algorithms, and evaluated their applicability to distributed interactive applications. Our investigation focused on quickly constructing overlays with a low diameter and pair-wise distance. The results revealed that tree structures are cheaper than meshes and also yield a competetive diameter when diameter reducing tree heuristics are used. However, with a fairly limited number of added links, a tree may be optimized to be quite a lot better, especially for algorithms such as the MST.

We also compared spanning-subgraph and Steiner-subgraph algorithms, and found that Steiner-points are important for both tree and subgraph algorithms to reduce the diameter. However, from figure 11.15 we observed that a doubled degree-limit from 5 to 10 helps reduce the diameter the most. When the degree-limit is the same, it is the Steiner-points that reduce the diameter the most. And, generally, when Steiner-points are available, an increased number of edges does reduce the diameter even more.

From these observations, we deduce that the following 3 properties are important for a diameter-reducing heuristic, to enable it to find a subgraph of a lower diameter (most important first):

1. A high ***degree-limit*** helps reduce the diameter signficantly compared to a low degree-limit.

2. Having centrally located ***Steiner points*** is important to reduce the diameter even more, especially when the degree-limits are in the low range.

3. Adding more ***edges*** to a subgraph does in effect reduce the average pair-wise latency, but does not automatically reduce the diameter.

Chapter 12

Overlay construction techniques: Dynamic tree algorithms

The overlay network management from section 5.4 includes overlay construction techniques whose task is to construct low-latency overlay networks for distribution of time-dependent events. In that respect, we continue our evaluation of overlay construction techniques and evaluate dynamic tree algorithms. By doing this we address a goal of the thesis:

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

Enabling clients to join and leave ongoing sessions of real-time interaction is also a goal of the thesis, and dynamic tree algorithms address such client dynamicity. A dynamic tree algorithm belongs to the class of dynamic tree problems, as introduced in section 4.7. Generally, dynamic tree algorithms support incoming requests of these two types:

Insert *node m to the tree T, such that m can communicate with the nodes in V_T.*
Remove *node m from the tree T, such that the nodes in V_T can still communicate.*

Based on these two request types (insert and remove) we identify that each dynamic algorithm is comprised of one insert and one remove strategy. Algorithm 38 shows a generic dynamic tree algorithm, in which an incoming insert or remove request is handled by different strategies.

Algorithm 38 GENERIC-DYNAMIC-TREE-ALGORITHM

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, and a request $\rho$
Out: Updated tree $T_Z = (V_T, E_T)$
if $\rho = \{remove, m\}$ then
    REMOVE-TREE-NODE-ALGORITHM($G$,$T_Z$,$m$)
else if $\rho = \{insert, m\}$ then
    INSERT-TREE-NODE-ALGORITHM($G$,$T_Z$,$m$)
end if
```

The chapter introduces many insert and remove strategies that are paired as dynamic-tree algorithms. The strategies include approaches that optimize towards minimum-cost, minimum-radius (shortest-path) and minimum-diameter trees. We evaluate all combinations of insert and remove strategies through group communication simulations, in which dynamic tree algorithms are used to update the overlay networks that are used for event distribution. From the results we found that selected dynamic tree algorithms are able to construct trees with a low diameter, comparable to those obtained from Steiner-tree algorithms that recompute the entire tree for each insert and remove. However, the dynamic tree algorithms are in general not able to maintain trees with a low diameter for larger group sizes. The main advantages of dynamic tree algorithms are the increased tree stability and reduced execution time, where the stability is measured in terms of the number of edges that change across reconfigurations.

The rest of the chapter is organized in the following manner. Section 12.1 introduces some existing dynamic tree algorthm types in the literature. Section 12.2 introduces basic algorithms that are the components in many of the evaluated dynamic tree algorithms. Section 12.3 presents the insert and remove strategies that are evaluated. Section 12.4 presents and discusses the results when the dynamic tree algorithms are used in group communication simulations. Section 12.5 gives a brief summary of the main points.

12.1 Dynamic tree algorithm types

Following are some brief and basic introductions to minimum-cost, minimum-diameter and shortest-path dynamic tree algorithms.

12.1.1 Minimum-cost dynamic tree algorithms

A minimum-cost tree is the tree of least cost on an input graph that, for Steiner-trees, span all member-nodes, and spanning-trees, span all nodes (chapter 10 and 9). Bharath-Kumar and Jaffe [18] (probably) proposed the first minimum-cost dynamic tree algorithm for an insert-only scenario (removal is not allowed). Later on, Waxman and Imase [140] proposed a similar minimum-cost dynamic tree algorithm that allowed insert and remove. The algorithm was called *Greedy* (algorithm 39). Given a request to insert node m to T, Greedy adds m through the minimum-cost edge to a node in V_T. Upon a remove request of node m from T, Greedy only removes m if it is a leaf node, and prunes leaf non-member-nodes. The Greedy algorithm was also studied by Faloutsos [44] on directed graphs.

Waxman and Imase have also proposed an algorithm called Edge Bounded Algorithm (EBA). EBA works as Greedy when node m is inserted to T_Z. In addition, EBA finds the maximum-cost edge (u,v) in T_Z between the two subtrees connected by (u,v). EBA, then checks whether there is an edge (u',v') that is a constant factor B cheaper than (u,v). A remove request removes m if the degree < 3, and also ensures that non-member nodes in T_Z have a degree $>$ 2. The insert strategy of EBA addresses the minimum-reconfiguration dynamic tree problem in definitions 59 and 63, while the remove strategy does not. Hence, the dynamic-tree algorithm actually addresses the reconfigurable dynamic tree problem in definition 58 and 62.

Algorithm 39 GREEDY-DYNAMIC-TREE-ALGORITHM

```
In: $G = (V,E,c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, and a request $\rho$
Out: Updated tree $T_Z = (V_T, E_T)$
if $\rho = \{remove, m\}$ then
        Remove $m$ from $Z$ if it is a leaf
        Remove leaf non-member-nodes $v \notin Z$
else if $\rho = \{insert, m\}$ then
        add $m$ to $T_Z$ through the minimum-cost edge to a node $v \in V_T$
end if
```

12.1.2 Minimum-diameter dynamic tree algorithms

A minimum-diameter tree is the tree of minimum diameter on an input graph that, for Steiner-trees, span all member-nodes, and spanning-trees, span all nodes (chapter 10 and 9). Bharath-Kumar and Jaffe [18] analyzed the problem of minimizing the average pair-wise tree latency in a dynamic tree scenario. The average pair-wise latency is the sum of all shortest paths among the member-nodes in a tree, divided by the number of member-nodes. Minimum-diameter dynamic tree algorithms have also been addressed in the literature [125]. A common minimum-diameter dynamic tree algorithm is to insert a node m to T_Z through the path that minimizes the eccentricity of m in T_Z (see algorithm 40). The eccentricity of a node m in a graph structure is the length of its longest shortest path to a destination node. While, remove requests are similarly handled as in the Greedy (minimum-cost) dynamic tree algorithm (section 12.1.1).

Algorithm 40 MINIMUM-ECCENTRICITY-DYNAMIC-TREE-ALGORITHM

```
In: $G = (V,E,c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, and a request $\rho$
Out: Updated tree $T_Z = (V_T, E_T)$
if $\rho = \{remove, m\}$ then
        Remove $m$ from $Z$ if it is a leaf
        Remove leaf non-member-nodes $v \notin Z$
else if $\rho = \{insert, m\}$ then
        find node $v \in V_T$ that (upon a connect) minimizes the eccentricity of $m$
        connect $m$ to $v \in V_T$ through the shortest-path, and add to $T_Z$.
end if
```

12.1.3 Shortest-path dynamic tree algorithms

A shortest-path tree is the tree on an input graph that has shortest paths from a source s to a number p of destinations. A *Naive* shortest-path dynamic tree algorithm was studied by

Faloutsos [44]. The Naive dynamic tree algorithm finds for each new destination m the shortest path from the source $s \in V_T$ to the destination, and adds to the multicast tree T_Z the part of the path that is not already in the tree (see algorithm 41). Naive then executes a request to remove node m, which is similar to the Greedy (minimum-cost) dynamic tree algorithm (section 12.1.1).

Algorithm 41 NAIVE-DYNAMIC-TREE-ALGORITHM

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a source-node $s \in Z$, and a request $\rho$
Out: Updated tree $T_Z = (V_T, E_T)$
if $\rho = \{remove, m\}$ then
    Remove $m$ from $Z$ if it is a leaf
    Remove leaf non-member-nodes $v \notin Z$
else if $\rho = \{insert, m\}$ then
    find shortest path from $m$ to source-node $s \in Z$
    connect $m$ to $s \in Z$ through the shortest-path, and add to $T_Z$.
end if
```

12.2 Components of the dynamic tree algorithms

The following introduces a number of algorithms that are applied by the evaluated centralized dynamic tree algorithms introdued later in section 12.3. The dynamic tree algorithms presented there consist of insert and remove strategies that are both spanning-tree and Steiner-tree strategies (section 4.7). One insert strategy and one remove strategy are paired to one dynamic tree algorithm.

12.2.1 Strategies for identifying sub-trees for reconfiguration

There are insert and remove strategies that reconfigure parts of the tree upon execution. Several approaches for tree-reconfiguration exist, where a general approach is to identify a part of the existing tree (a sub-tree). This sub-tree is reconfigured based on some strategy that disconnects the tree and then reconnects the sub-trees such that the tree is again connected. Reconnecting the sub-trees is identical to the Group Steiner-tree problem as described by definition 55 in section 4.6.

Two specific strategies are presented for identifying sub-trees in an existing tree T_Z. The *k-Neighbor-Spanning-Subtree* algorithm is given as input a tree T_Z, a source-node $s \in V_T$, and an integer $k \geq 0$ that determines how deep into the tree (or how large) the identified sub-tree should be. The algorithm starts from the source node s, where s may be the node to remove. From the source node s the sub-tree is identified by traversing k nodes along each of its out-edges in T_Z. The *k-Neighbor-Steiner-Subtree* is identical, except it traverses k member-nodes along each of the out-edges from $s \in V_T$.

Algorithms 42 and 43 presents the pseudo-code of k-Neighbor-Spanning-Subtree and k-Neighbor-Steiner-Subtree (respectively). The main difference between the two is that k-Neighbor-Steiner-Subtree is aware of the member-nodes (and Steiner points).

Algorithm 42 k-NEIGHBOR-SPANNING-SUBTREE

```
In: A graph $G$, tree $T = (V_T, E_T)$, $k \geq 0$, and a source-node $s \in V_T$
Out: A sub-tree $T' = (V', E')$
traverse $k$ nodes $v \in V_T$ along each out-edge from $s \in V_T$
include every intercepted edge to $T'$
```

Algorithm 43 k-NEIGHBOR-STEINER-SUBTREE

```
In: A graph $G$, tree $T_Z = (V_T, E_T)$, $k \geq 0$, a set of members $Z \subseteq V_T$, and a source-node $s \in V_T$ a request $\rho$, and a
    cost function $cost \rightarrow \mathbb{R}$
Out: A sub-tree $T' = (V', E')$
traverse $k$ member-nodes $z \in Z$ along each out-edge from $s \in V_T$
include every intercepted edge to $T'$
```

12.2.2 Disconnect and reconnect trees

The sub-tree $T' = (V', E')$ identified by, for example, k-Neighbor-Spanning-Subtree, is used further by some insert and remove strategies to reconfigure the tree T_Z. The tree T_Z is typically disconnected $T_Z = T_Z - T'$ and then reconnected by reconnecting the $|V'|$ sub-trees to one tree T_Z, including or excluding m. The algorithms for reconnecting sub-trees belong to the Group Steiner-tree problem (definition 55).

Algorithm 44 gives a generic algorithm for reconnecting a disconnected tree that contains subsets of vertices, which are the connected sub-trees. The sub-trees are reconnected through single edges that are chosen based on some optimization goal.

Algorithm 44 GENERIC-RECONNECT-TREE-ALGORITHM

```
In: $G = (V, E, c)$, a disconnected tree $T_Z = (V_T, E_T)$, $k$ connected sub-trees (groups) $g_1 \ldots g_k \in V_T$.
Out: Connected tree $T_Z = (V_T, E_T)$
Ensure each group $g_i$ is a connected sub-tree with no overlapping edges
$G = \emptyset$
for each group $g_i$ do
    find and connect to a group $g_j \notin G$ based on some optimization goal
    $G \cup \{g_i, g_j\}$
end for
```

12.2.3 Group Steiner-tree heuristics

The dynamic tree algorithms that disconnect the group trees during tree-updates may apply Group Steiner-tree heuristics. Among our evaluated dynamic tree algorithms there are strategies that apply heuristics for reconnecting sub-trees optimized for minimum-cost and -diameter.

Algorithm 45 presents the $O(n^3)$ *mddl-ReconnectTree* for reconnecting sub-trees, which aims to achieve a low diameter of the reconnected tree. The algorithm iterates through the sub-trees g_i and chooses the vertex $u \in V_{g_i}$, which was a neighbor of the removed node m. Then, for each other group g_j find the node $v \in V_{g_j}$ that minimizes the function:

$$f(v) = MAX\{(eccentricity(v, V_{g_j}) + weight(v, u)), eccentricity(u, V_{g_i})\}$$

where $eccentricity(v, V_{g_j})$ is the eccentricity of $v \in V_{g_j}$, $weight(u, v)$ is the weight of the edge to connect sub-trees g_i and g_j, and $eccentricity(u, V_{g_i})$ is the current eccentricity of $u \in V_{g_i}$. The mddl-ReconnectTree does $|V_{g_j}|$ depth-first searches to retrieve the $eccentricity(v, V_{g_j})$. Therefore, it may prove time-consuming if the number of groups k is close to the number of tree-vertices $|V_T|$.

For the minimum-cost heuristic *mc-ReconnectTree* the function to minimize is simply $f(v) = \{weight(v, u)\}$, that is, the minimum-cost edges. The minimum-cost Group Steiner-tree heuristic is less time-consuming, and performs a $O(n^2)$ to find the minimum-cost edge to connect groups g_i and g_j.

Generally, a minimum-cost goal is easier to solve than a minimum-diameter goal. This is also the case for the Group Steiner-tree problem, and the reason why mddl-ReconnectTree is more time-consuming. It is upon a remove request of a node with a degree > 2 that these heuristics are natural to apply.

12.3 Evaluated dynamic tree algorithms

The following introduces a number of centralized dynamic tree algorithms that are evaluated through simulations and experiments in section 12.4. The dynamic tree algorithms presented here consist of insert and remove strategies that are both spanning-tree and Steiner-tree strategies (section 4.7). However, we do not explicitly distinguish between the two types in the presentation of the evaluated strategies. Rather, we observe the effects in the simulation studies in section 12.4.

A dynamic tree algorithms consists of one insert strategy and one remove strategy. Algorithmically, the dynamic tree algorithm works as such:

Dynamic-Tree-Algorithm($\mathscr{R}_{\mathscr{D}}$, $\mathscr{I}_{\mathscr{D}}$) (DA) takes as input an insert strategy $\mathscr{I}_{\mathscr{D}}$ and a remove strategy $\mathscr{R}_{\mathscr{D}}$, and a request ρ. Furthermore, a global undirected weighted graph $G = (V, E, c)$, where V is the set of vertices ($n = |V|$), E is the set of edges, $c : E \rightarrow \mathbb{R}$ is the edge cost function. Moreover, a group-tree $T_Z = (V_T, E_T)$, where $V_T \subset V$, and $E_T \subset E$ In addition, there is a set of member-nodes $Z \subset V_T$. DA executes $\mathscr{I}_{\mathscr{D}}$ upon an insert request, and $\mathscr{R}_{\mathscr{D}}$ upon a remove request (algorithm 46), and updates the group-tree T_Z accordingly.

Algorithm 45 MDDL/MC-RECONNECT-TREE

```
In: $G = (V, E, c)$, a disconnected tree $T_Z = (V_T, E_T)$, $k$ subsets of vertices, which are connected sub-trees (groups) $g_1 \dots g_k \in V_T$.
Out: Connected tree $T_Z = (V_T, E_T)$.
Ensure each group $g_i$ is a connected sub-tree with no overlapping edges
Start:
for each $g_i$, $i \leq k$ do
    find a vertex $u \in V_{g_i}$ adjacent to $m$ {$m$ is the node that was removed}
    for each $g_j$, $j \neq i$ do
        find the node $v \in V_{g_j}$ that minimizes:
        $f(v)$=MAX{eccentricity($v$, $V_{g_j}$) + weight($v$,$u$), eccentricity($u$, $V_{g_i}$)} {MDDL-RECONNECT}
        OR
        $f(v) = \{weight(v,u)\}$ {MC-RECONNECT}
    end for
end for
if found degree-limited edge($u$,$v$) then
    add edge($u$,$v$) to $T_Z$
    $g_i \cup \{g_i, g_j\}$
    remove $g_j$ from groups
else
    relaxDegree($V_T$)
end if
if tree $T_Z$ is not connected then
    goto Start
end if
```

Algorithm 46 DYNAMIC-TREE-ALGORITHM

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a request $\rho$, one insert strategy $\mathscr{I}_{\mathscr{D}}$ and one remove strategy $\mathscr{R}_{\mathscr{D}}\}$.
Out: Updated tree $T_Z = (V_T, E_T)$
if $\rho = \{remove, m\}$ then
    $\mathscr{R}_{\mathscr{D}}(G, T_Z, m)$
else if $\rho = \{insert, m\}$ then
    $\mathscr{I}_{\mathscr{D}}(G, T_Z, m)$
end if
```

Table 12.1 provides a quick overview of abbreviations and features. An algorithm is non-member node aware if it is able to insert or remove non-member nodes from a group tree. The reconfiguration set R contains the edges that are changed between reconfigurations of a group tree. Furthermore, the time complexity (in big-oh notation) is the worst-case number of computational steps of an algorithm. Following is an introduction of the insert and remove strategies both formally and informally.

12.3.1 Insert strategies

An insert strategy $\mathscr{I}_{\mathscr{D}}$ inserts a new member m into the group tree and assures that m can communicate with the remaining member nodes [133]. Formally, an insert strategy works like this:

Given $G = (V, E, c)$, a tree $T = (V_T, E_T)$, a set of members $Z_T \subseteq V_T$, and a new member $m \in V$. Update T, such that $Z_T \cup \{m\}$ are connected.

The new member m may be inserted into T in many different manners. We have devised several insert strategies that bound the size of the reconfiguration set R. It is possible to add m to T through a single edge, while other strategies use the degree limit as a bound. Algorithm 47 illustrates a generic insert strategy.

Algorithm 47 GENERIC-INSERT-TREE-ALGORITHM

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a new member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
Find a tree node $v \in V_T$ optimizing some cost function
connect $m$ to $T_Z$ through edge($m$ ,$v$)
```

Insert strategies considering the minimum-cost

The evaluated 8 insert strategies that consider the cost of a tree tries to include the node m such that the tree's total cost is kept low.

I-MC (minimum cost) includes the joining member m to the tree, through the minimum-cost edge. I-MC is a $O(n)$ algorithm and is the most common insert strategy in the literature.

Algorithm 48 INSERT-MINIMUM-COST

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a new member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
Find a tree node $v \in V_T$ with minimum-cost edge to $m$
connect $m$ to $T_Z$ through edge($m$ ,$v$)
```

ITR-MC (try reconfiguration, minimum-cost) first finds the minimum-cost edge from m to a node in the tree, $e = (m, v)$. Then, ITR-MC checks three different choices of reconfigurations (see figure 12.1):

1) Add m to the tree through the minimum-cost edge $e = (m, v)$.
2) Use m as the new intersection vertex for the neighbors of the minimum-cost target node v.
3) Find a well-placed core-node c with k-Median using $\{m, v, neighbor(v)\}$ as the member node set, and then use c as the new intersection point for this member-node set.

ITR-MC chooses the reconfiguration that reduces the *total cost* the most. ITR-MC is a $O(n^2)$ algorithm, and is a contribution of the thesis.

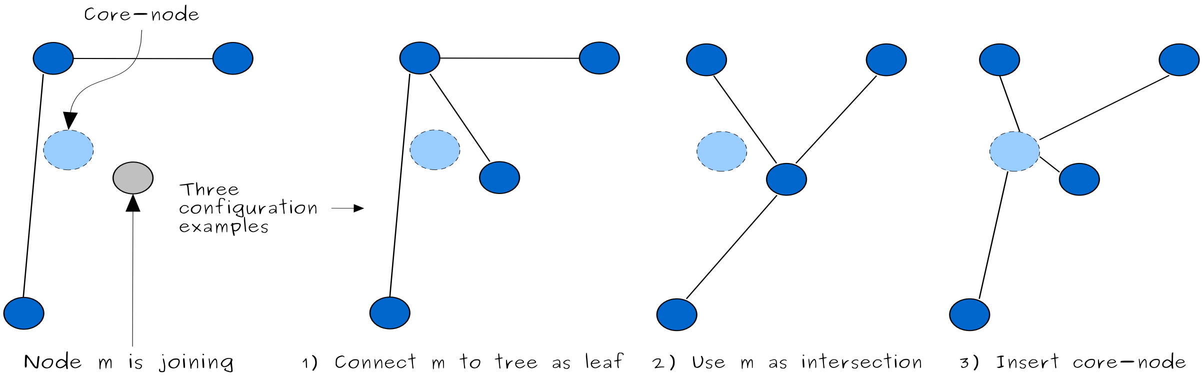


Figure 12.1: ITR-MC configuration options when m is joining.

Algorithm 49 INSERT-TRY-RECONFIGURATION-MINIMUM-COST

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a new member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
  find $v \in V$ with minimum-cost edge to $m$
  $T'$ = $k$-Neighbor-Spanning-Subtree($T_Z$, $v$, $k = 1$)
  $c$ = $k$-Median(k=1, $V'(T')$)
  {insert neighbors of $v$ in $T_Z$, pluss $v$, $c$, and $m$ to $V_N$}
  $V_N = v + c + V'(T')$
  {Compare: use one node $V_N$ as intersection for all member-nodes in $V_N$.}
  find node $s \in V_N$ for which the sum of edge weights to all member-nodes $z \in V_N$ is the minimum
  $T_Z = T_Z$ - $T'$
  reconnect $T_Z$ using $s$ as intersection.
```

Insert strategies considering the diameter

We consider 4 insert strategies that aim to keep the diameter low upon inserting a new member-node.

I-MDDL (minimum diameter degree limited) includes the new member-node m to the tree, such that it achieves the minimum eccentricity (see algorithm 50). I-MDDL is a $O(n^2)$ algorithm and uses $|V_T|$ depth-first searches to find the tree-node $v \in V_T$ that m can connect to, such that its eccentricity is minimized.

Algorithm 50 INSERT-MINIMUM-DIAMETER-DL

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a new member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
  Find an edge to a tree node $v \in V_T$ such that the eccentricity of $m$ is minimized
  connect $m$ to $T_Z$ through edge($m$ ,$v$)
```

I-CN (center node) makes each group elect a center node among the nodes in the current tree. Upon election, the center node has the smallest eccentricity related to the member nodes, and has not reached the degree limit. A new node m is (always) connected to it. Whenever the current center node has exhausted its degree limitation, a new one is elected (see algorithm 51).

I-CN's center-election is a $O(n^2)$ procedure, which applies $|V_T|$ depth-first searches to get the node with the smallest eccentricity. The node with the smallest eccentricity is by definition 11, the center vertex of a graph. If the center-node has already been elected, I-CN is a $O(1)$ algorithm.

Algorithm 51 INSERT-CENTER-NODE

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a new member $m$ , a center-node $c$
Out: Updated tree $T_Z = (V_T, E_T)$
if $c \notin V_T$ or $deg_{T_Z}(c) \geq deg(c)$ then
    $c$ = getNodeWithSmallestEccentricity($T_Z$) {elect new center-node}
end if
connect $m$ to $T_Z$ through edge($m$ ,$c$)
```

IBD-MC (bounded diameter, minimum cost) inserts a new member-node m to the tree, through a low-cost edge such that the eccentricity of m is lower than a given diameter bound (see algorithm 52). IBD-MC is a $O(n^2)$ algorithm that first applies $|V_T|$ depth-first searches to retrieve the eccentricities of all nodes $v \in V_T$, then a $O(n)$ procedure connects m to T_Z through the least cost edge that does not violate a given diameter bound B. If it is not possible, a dynamic relaxation procedure relaxes the diameter bound B until m can be connected.

Algorithm 52 INSERT-BOUNDED-DIAMETER-MINIMUM-COST

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a new member $m$, and a diameter bound $B \in \mathbb{R}$
Out: Updated tree $T_Z = (V_T, E_T)$
Find a tree node $v \in V_T$, such that it is the least cost-edge that makes the eccentricity of $m \leq$ to $B$
connect $m$ to $T_Z$ through edge($m$ ,$v$)
```

ITR-MDDL (try reconfiguration, minimum diameter degree limited) first finds the minimum-eccentricity edge for m to a node in the tree, $e = (m, v)$. Then, ITR-MDDL checks three different choices of reconfigurations:

1) Add m to the tree through the minimum-eccentricity edge $e = (m, v)$.
2) Use m as the new intersection for the neighbors of the min-eccentricity connect node v.
3) Find a well-placed core-node c with k-Median using $\{m, v, neighbor(v)\}$ as the member node set, and then use c as the new intersection point for this member-node set.

ITR-MDDL is a $O(n^2)$ algorithm that uses $|V_T|$ depth-first searches to retrieve the eccentricites of all nodes $v \in V_T$, then it tries the three combinations and chooses the reconfiguration that reduces the *diameter* the most. ITR-MDDL is a contribution of the thesis.

Algorithm 53 INSERT-TRY-RECONFIGURATION-MDDL

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a new member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
find $v \in V$ with minimum-eccentricity edge to $m$
$T'$ = $k$-Neighbor-Spanning-Subtree($T_Z$, $v$, $k = 1$)
$c$ = $k$-Median(k=1, $V'(T')$
{insert neighbors of $v$ in $T_Z$, pluss $v$, $c$, and $m$ to $V_N$}
$V_N = v + c + V'(T')$
{Compare: use one node $V_N$ as intersection for all member-nodes in $V_N$.}
find node $s \in V_N$, such that when $s$ is used to interconnect the member-nodes $z \in V_N$ the diameter of $T_Z$ is the minimum
$T_Z$ = $T_Z$ - $T'$
reconnect $T_Z$ using $s$ as intersection.
```

Insert strategies considering the radius

We consider 2 insert strategies that consider the radius upon inserting a new member-node. The strategies also reduce the diameter because of this.

I-MRDL (minimum radius degree limited) finds the node s in the tree that has the smallest eccentricity and an available degree capacity, and then connects m to the tree through the shortest-path to s (see algorithm 54). I-MRDL is a $O(n^2)$ algorithm and applies $|V_T|$ depth-first searches to find the least eccentricity node.

Algorithm 54 INSERT-MINIMUM-RADIUS-DL

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a new member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
Find the node $s \in V_T$ with the smallest eccentricity and $deg_{T_Z}(s) < deg(s)$.
connect $m$ to $T_Z$ through edge($m$ ,$s$)
```

IBR-MC (bounded radius, minimum cost) finds the node s in the tree that has the smallest eccentricity. Then it connects m to the tree through the lowest cost edge that does not violate a radius bound to s (see algorithm 55). IBR-MC is a $O(n^2)$ algorithm that first applies $|V_T|$ depth-first searches to retrieve the node s with the smallest eccentricity of all the nodes $v \in V_T$. Then a $O(n)$ procedure connects m to T_Z, as mentioned, through the least cost edge while not violating the radius bound R to s. If it is not possible, a dynamic relaxation procedure relaxes the radius bound R until m can be connected.

Algorithm 55 INSERT-BOUNDED-RADIUS-MINIMUM-COST

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a new member $m$, and a radius bound $R \in \mathbb{R}$
Out: Updated tree $T_Z = (V_T, E_T)$
Find the node $s \in V_T$ with the smallest eccentricity
Find the least cost-edge to a node $v \in V_T$, such that the distance($m$,$s$) is $\leq$ to $R$.
connect $m$ to $T_Z$ through edge($m$ ,$v$)
```

12.3.2 Remove strategies

A remove strategy $\mathscr{R}_{\mathscr{D}}$ removes a member m from the multicast tree and assures that the group members are reconnected. Formally, a remove strategy works like this:

Given $G = (V, E, c)$, a tree $T = (V_T, E_T)$, a set of members $Z_T \subseteq V_T$, and a member $m \in V$. Update T, such that $Z_T \setminus \{m\}$ are connected.

The number of direct neighbor nodes of m (its degree) in T influences the necessary actions. If the degree $d_T(m) = 1$, m is a leaf, which is simply removed along with the edge to its only neighbor. If it is greater than 1, a removal of m partitions the group if no additional steps are taken, and $d_T(m)$ unconnected subtrees is the result. The neighbors of the leaving node are connected directly in the simple case $d_T(m) = 2$. The basic goal of remove strategies is the reconnection of subtrees into a single tree when $d_T(m) > 2$. Algorithm 56 illustrates a generic remove strategy.

Algorithm 56 GENERIC-REMOVE-TREE-ALGORITHM

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
disconnect $m$ from tree $T_Z$
if disconnected tree $T_Z$ then
    reconnect subtrees optimizing some cost function
end if
```

Remove strategies considering the minimum-cost

We consider 11 remove strategies that try to keep the total cost of a tree low after servicing a remove request.

RK (keep) does not actually remove the leaving node m from the tree but requires it to forward traffic (as a non-member node), until its degree has dropped to 2. RK is a $O(1)$ algorithm, and is most commonly used in the literature today. The authors have previously shown that it can degrade for larger groups [60].

Algorithm 57 REMOVE-KEEP

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
Remove $m$ from $Z$ if it is a leaf
Remove leaf non-member-nodes $v \notin Z$
```

R-MC (minimum-cost) removes the node m and reconnects the sub-trees using the immediate neighbors of m, through minimum-cost edges. For this, it uses the mc-ReconnectTree Group Steiner-tree heuristic. However, if R-MC is unable to reconnect the sub-trees due to exhausted

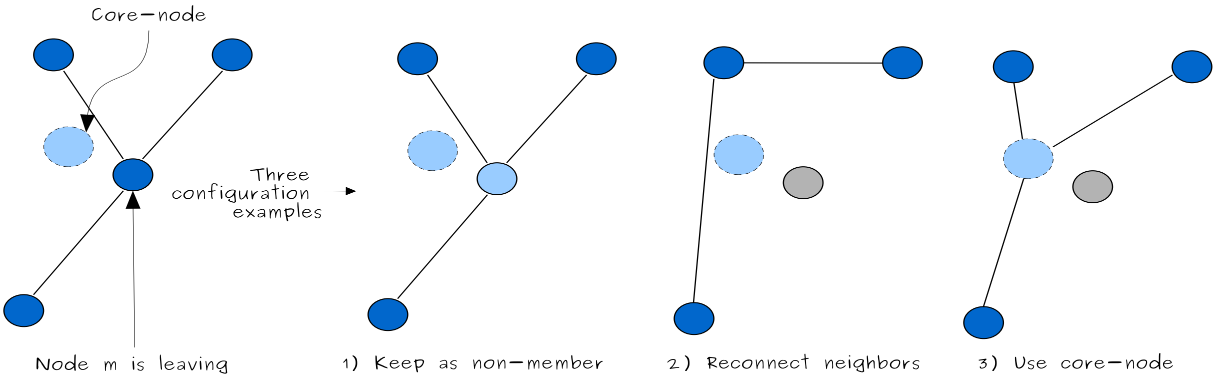


Figure 12.2: RTR-MC configuration options when m is leaving.

degree-limits, R-MC then leaves m in the tree as a non-member-node. R-MC is a $O(n^3)$ algorithm due to the complexity of mc-ReconnecTree. However $n = |V'|$, where V' are the neighbors of m. Therefore, the current degree-limit bounds the number of neighbors, and thus the complexity of R-MC.

Algorithm 58 REMOVE-MINIMUM-COST

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
$T'$ = $k$-Neighbor-Spanning-Subtree($T_Z$, $m$, $k = 1$)
$E_T \setminus E'$ and $Z \setminus \{m\}$
if feasible to reconnect tree using $V'$ then
    mc-ReconnectTree($T_Z$, $V'$)
end if
```

RS-MC (search minimum-cost) is identical to R-MC, except that it does not limit the reconnection points to the immediate neighbors of m. Rather, the sub-trees may be reconnected using any node in the neighbor sub-trees. RS-MC is a $O(n^3)$ algorithm due to the complexity of mc-ReconnecTree, were $n = |V_T|$, and V_T are all the nodes in the tree, except the removed node m. Therefore, RS-MC is more time-consuming than its simpler (previous) version R-MC.

Algorithm 59 REMOVE-SEARCH-MINIMUM-COST

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
$T'(V', E')$ = $k$-Neighbor-Spanning-Subtree($T_Z$, $m$, $k = 1$)
$E_T \setminus E'$ and $Z \setminus \{m\}$
if feasible to reconnect tree using $V_T$ then
    mc-ReconnectTree($T_Z$, $V_T$)
end if
```

RTR-MC (try reconfiguration, minimum-cost) is similar to ITR-MC, and also checks three different choices of reconfigurations (see figure 12.2):

1) Leave m in the tree.
2) Use a neighbor of m as the new intersection vertex for all m's neighbors.
3) Find a well-placed core-node c with k-Median using $\{neighbor(m)\}$ as the member node set, and then use c as the new intersection point for this member-node set.

RTR-MC chooses the reconfiguration that reduces the *total cost* the most. RTR-MC is a $O(n^2)$ algorithm, and is a contribution of the thesis.

Algorithm 60 REMOVE-TRY-RECONFIGURATION-MINIMUM-COST

```
In: $G = (V,E,c)$, a tree $T_Z = (V_T,E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T,E_T)$
$T'$ = $k$-Neighbor-Spanning-Subtree($T_Z$, $m$, $k = 1$)
$V_N = V'(T') - m$
$c$ = $k$-Median(k=1,$V_N$)
$V_N$ = $m$ + $c$ + neighbors($m$,$T_Z$)
{Compare: keep $m$ in tree as non-MN, replace $m$ with $c$, or use $v \in V_N$ as intersection.}
find node $s \in V_N$ for which the sum of edge weights to all member-nodes $z \in V_N$ is the minimum
$T_Z$ = $T_Z$ - $T'$
reconnect $T_Z$ using $s$ as intersection.
```

RTR-P-MC (try reconfiguration, prune, minimum-cost) tries to reduce (prune) the number of non-member-nodes in the neighborhood of m, and then reconnect in a minimum-cost fashion.

It uses k-Neighbor-Steiner-Subtree(m,T_Z,$k = 1$) to identify m's neighborhood with the member-nodes and non-member-nodes. (The neighbors are found from all member-nodes that are either directly connected to m or that are only separated by non-members.)

Then, RTR-P-MC calculates the minimum number K of non-member-nodes that are required to reconnect the sub-trees that are created when disconnecting m. For this, it uses equation 8.1 (section 8.5.4) to calculate the number of Steiner points.

Then, it chooses K non-member-nodes (Steiner-points) from the neighborhood of m (already in the tree). If it cannot find K nodes, it uses k-Median to find additional well-placed core-nodes to use as Steiner-points. When all of this is done, RTR-P-MC removes m, and uses the mc-ReconnectTree Group Steiner-tree heuristic to reconnect the tree. However, it limits the reconnection options to be the member-nodes of m and the identified Steiner-points.

RTR-PS-MC (try reconfiguration, prune, search minimum-cost) is identical to RTR-P-MC in all phases but one. In the reconnection of the sub-trees it does not limit the reconnection options to the neighbors of m. Rather, RTR-PS-MC applies mc-ReconnectTree to search among every node in the subtrees to identify low cost edges to reconnect the sub-trees to one.

Algorithm 61 REMOVE-TRY-RECONFIGURATION-PRUNE-MINIMUM-COST

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
$T'$ = $k$-Neighbor-Steiner-Subtree($T_Z$, $m$, $k = 1$)
$K$ = $|V'|/deg(v')$ {use equation 8.1 }
if number of non-member nodes in $V' < K$ then
    use $k$-Median($V'$) to identify more Steiner-points
else if number of non-member nodes in $V > K$ then
    identify $K$ nodes from $V'$ using $k$-Median
end if
$T_Z$ = $T_Z$ - $T'$
mc-ReconnectTree($T_Z$, $V'$)
```

Algorithm 62 REMOVE-TRY-RECONFIGURATION-PRUNE-SEARCH-MINIMUM-COST

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
$T'$ = $k$-Neighbor-Steiner-Subtree($T_Z$, $m$, $k = 1$)
$K$ = $|V'|/deg(v')$ {use equation 8.1 }
if number of non-member nodes in $V' < K$ then
    use $k$-Median($V'$) to identify more Steiner-points
else if number of non-member nodes in $V > K$ then
    identify $K$ nodes from $V'$ using $k$-Median
end if
$T_Z$ = $T_Z$ - $T'$
mc-ReconnectTree($T_Z$, $V_T$)
```

Remove strategies considering the diameter

We consider 5 remove strategies that try to keep the diameter of a tree low after servicing a remove request.

R-MDDL (minimum diameter degree limited edge reconnects) removes the node m and reconnects the sub-trees through edges that keep the diameter of the tree low. It uses mddl-ReconnectTree Group Steiner-tree heuristic to reconnect the sub-trees using the immediate neighbors of m. If R-MDDL is unable to reconnect the sub-trees due to exhausted degree-limits, R-MDDL then leaves m in the tree as a non-member-node. R-MDDL is a $O(n^3)$ algorithm due to the complexity of mddl-ReconnecTree. However $n = |V'|$, where V' are the neighbors of m. Therefore, the current degree-limit bounds the number of neighbors, and thus the complexity of R-MDDL.

Algorithm 63 REMOVE-MINIMUM-DIAMETER-DEGREE-LIMITED

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
$T'$ = $k$-Neighbor-Spanning-Subtree($T_Z$, $m$, $k = 1$)
$E_T \setminus E'$ and $Z \setminus \{m\}$
if feasible to reconnect tree using $V'$ then
    mddl-ReconnectTree($T_Z$, $V'$)
end if
```

RS-MDDL (search for minimum diameter degree limited edges) is identical to R-MDDL, except that it does not limit the reconnection points to the immediate neighbors of m. Rather, the sub-trees may be reconnected using any node in the neighbor sub-trees. RS-MDDL is a $O(n^3)$ algorithm due to the complexity of mddl-ReconnecTree, were $n = |V_T|$, and V_T are all the nodes in the tree, except the removed node m. Therefore, RS-MDDL is more time-consuming than its simpler (previous) version R-MDDL.

Algorithm 64 REMOVE-SEARCH-MINIMUM-DIAMETER-DEGREE-LIMITED

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
if feasible to reconnect tree using $V'$ then
$T'$ = $k$-Neighbor-Spanning-Subtree($T_Z$, $m$, $k = 1$)
$E_T \setminus E'$ and $Z \setminus \{m\}$
if feasible to reconnect tree using $V'$ then
    mddl-ReconnectTree($T_Z$)
end if
```

RTR-MDDL (try reconfiguration, minimum diameter) is similar to ITR-MDDL and RTR-MC, and also checks three different choices of reconfigurations:

1) Leave m in the tree.

2) Use a neighbor of m as the new intersection vertex for all m's neighbors.

3) Find a well-placed core-node c with k-Median using $\{neighbor(m)\}$ as the member node set, and then use c as the new intersection point for this member-node set.

RTR-MDDL is a $O(n^2)$ algorithm that uses $|V_T|$ depth-first searches to retrieve the eccentricites of all nodes $v \in V_T$, then it tries the three combinations and chooses the reconfiguration that reduces the *diameter* the most. RTR-MDDL is not found in the literature, and is a contribution of the thesis.

Algorithm 65 REMOVE-TRY-RECONFIGURATION-MINIMUM-DIAMETER-DEGREE-LIMITED

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
$T'$ = $k$-Neighbor-Spanning-Subtree($T_Z$, $m$, $k = 1$)
$V_N = V'(T') - m$
$c$ = $k$-Median(k=1,$V_N$)
$V_N$ = $m$ + $c$ + neighbors($m$,$T_Z$)
{Compare: keep $m$ in tree as non-MN, replace $m$ with $c$, or use $v \in V_N$ as intersection.}
find node $s \in V_N$ that reconnects all member-nodes $z \in V_N$, such that the diameter of $T_Z$ is the lowest
$T_Z$ = $T_Z$ - $T'$
reconnect $T_Z$ using $s$ as intersection.
```

RTR-P-MDDL (try reconfiguration, prune, minimum-diameter) tries to reduce (prune) the number of non-member-nodes in the neighborhood of m, and then reconnect such that the tree has a low diameter. It is identical to RTR-P-MC, except it uses mddl-ReconnectTree Group

Steiner-tree heuristic to reconnect the neighbors of m and the identified Steiner-points such that the tree is connected. RTR-P-MDDL is a $O(n^3)$ algorithm due to the complexity of mddl-ReconnecTree. In RTR-P-MDDL, $n = |V'|$, where V' are the neighbor nodes of the removed node m.

Algorithm 66 REMOVE-TRY-RECONFIGURATION-PRUNE-MINIMUM-DIAMETER-DEGREE-LIMITED

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
$T' = k$-Neighbor-Steiner-Subtree($T_Z$, $m$, $k = 1$)
$K = |V'|/deg(v')$ {use equation 8.1 }
if number of non-member nodes in $V' < K$ then
    use $k$-Median($V'$) to identify more Steiner-points
else if number of non-member nodes in $V > K$ then
    identify $K$ nodes from $V'$ using $k$-Median
end if
$T_Z = T_Z - T'$
mddl-ReconnectTree($T_Z$, $V'$)
```

RTR-PS-MDDL (try reconfiguration, prune, search minimum-diameter) is identical to RTR-P-MDDL in all phases but one. In the reconnection of the sub-trees it does not limit the reconnection options to the neighbors of m, rather, RTR-PS-MDDL searches among every node in the subtrees to identify edges that make the reconnected tree yield a low diameter. RTR-PS-MDDL is the most time consuming remove strategy, and is a $O(n^3)$ algorithm due to the complexity of mddl-ReconnecTree. In RTR-PS-MDDL, $n = |V_T|$, where V_T are all the nodes in the tree, except the removed node m.

Algorithm 67 REMOVE-TRY-RECONFIGURATION-PRUNE-SEARCH-MINIMUM-DIAMETER-DEGREE-LIMITED

```
In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a leaving member $m$
Out: Updated tree $T_Z = (V_T, E_T)$
$T' = k$-Neighbor-Steiner-Subtree($T_Z$, $m$, $k = 1$)
$K = |V'|/deg(v')$ {use equation 8.1 }
if number of non-member nodes in $V' < K$ then
    use $k$-Median($V'$) to identify more Steiner-points
else if number of non-member nodes in $V > K$ then
    identify $K$ nodes from $V'$ using $k$-Median
end if
$T_Z = T_Z - T'$
mddl-ReconnectTree($T_Z$, $V_T$)
```

Figure 12.3 illustrates an example where m is leaving a tree, and RK, RTR-MDDL and RTR-P-MDDL are used to reconfigure and then reconnect the tree. From the figure we observe that the remove strategy RK converts m to a non-member node and leaves it in the tree. RTR-MDDL reconfigures the immediate neighborhood of m and includes a well-placed core-node in

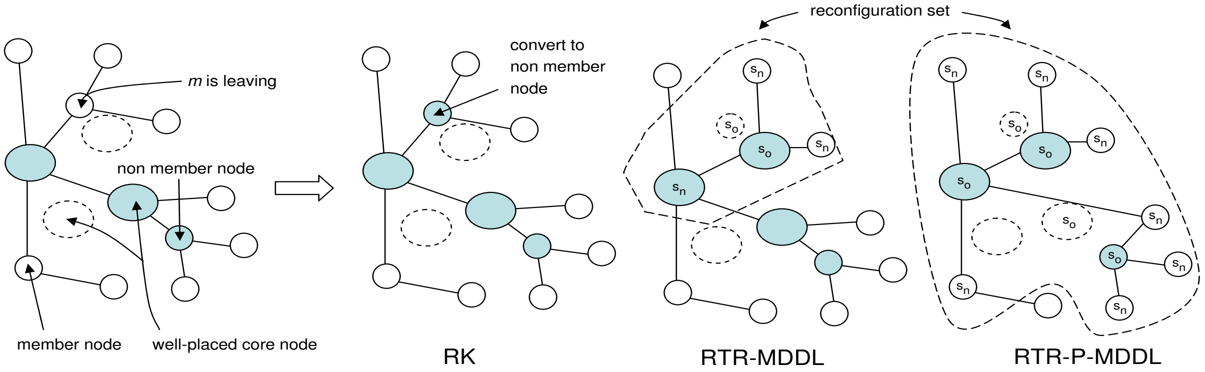


Figure 12.3: Remove reconfiguration examples when m is leaving and RK, RTR-MDDL and RTR-P-MDDL reconfigures.

the place of m. RTR-P-MDDL reconfigures a larger portion of the tree and reduces the number of non-member-nodes in the tree. It is clear the RK has the least influence on the tree, while RTR-P-MDDL has the largest.

12.4 Group communication simulations and dynamic tree algorithms

Chapter 9, 10, and 11 evaluated overlay construction algorithms that construct overlay networks from scratch. Although these algorithms are useful in a variety of situations, including distributed interactive applications, they do lack the option of inserting and removing single member-nodes from an existing overlay.

As mentioned, a primary goal is to enable clients to join and leave ongoing sessions of distributed interactive applications in a timely fashion. In that respect, the dynamic tree algorithms introduced previously are designed for the specific purpose of enabling group dynamics of online sessions. Following are evaluations of the dynamic tree algorithms, table 12.1 summarizes all insert and remove strategies with some details.

12.4.1 Experiment configurations

In the experiments, we use equation 8.1 (section 8.5.4) to approximate the number k of Steiner-points that are needed to ensure that the degree-limited dynamic tree algorithms are able to reconnect a group-tree. The k Steiner points are chosen by k-Median(k) from 100 Steiner-points (super-nodes) that are identified at the beginning among the 1000 nodes in the overlay network. These well-placed super-nodes (chapter 8) are found by the multiple core-node selection algorithm k-Center($k = 100$).

		MN				*Time*	
Algorithm	*Meaning*	*aware*	*Optimization*	*Constraints*	*Reconfiguration set R*	*complexity*	*Problem*
I-MDDL	Insert miminum diameter degree limited edge	×	diameter	degree	$\|R\| = 1$	$O(n^2)$	def. 64
I-MRDL	Insert miminum radius degree limited edge	×	radius	degree	$\|R\| = 1$	$O(n^2)$	def. 64
I-CN	Insert center node	×	diameter	degree	$\|R\| = 1$	$O(n)$	def. 64
ITR-MDDL	Insert try reconfiguration and MDDL-edge	✓	diameter	degree	$\|R\| \leq d(m) * 2$	$O(n^2)$	def. 65
IBD-MC	Insert bounded diameter minimum cost edge	×	total cost	degree/diameter	$\|R\| = 1$	$O(n^2)$	def. 64
IBR-MC	Insert bounded radius minimum cost edge	×	total cost	degree/rad.	$\|R\| = 1$	$O(n^2)$	def. 64
I-MC	Insert miminum cost degree limited edge	×	total cost	degree	$\|R\| = 1$	$O(n)$	def. 64
ITR-MC	Insert try reconfiguration and MC-edge	✓	total cost	degree	$\|R\| \leq d(m) * 2$	$O(n^2)$	def. 65
RK	Remove keep as non-member node	×	shortest path	degree	$\|R\| = 3 \rightarrow d_T(m) \leq 2$	$O(1)$	def. 68
			-	-	$\|R\| = 0 \rightarrow d_T(m) > 2$	-	-
R-MDDL	Remove miminum diameter degree limited edge	×	diameter	degree	$\|R\| \leq d(m) * 2$	$O(n^3)$	def. 68
RS-MDDL	Remove search miminum diameter degree limited edge	×	diameter	degree	$\|R\| \leq d(m) * 2$	$O(n^3)$	def. 68
RTR-MDDL	Remove try reconfiguration and MDDL-edge	✓	diameter	degree	$\|R\| \leq d(m) * 2$	$O(n^2)$	def. 68
RTR-P-MDDL	Remove try reconfiguration and prune	✓	diameter	degree	$\|R\| \leq \|E_T\| * 2$	$O(n^3)$	def. 69
RTR-PS-MDDL	Remove try reconfiguration, prune and search MDDL	✓	diameter	degree	$\|R\| \leq \|E_T\| * 2$	$O(n^3)$	def. 69
R-MC	Remove miminum cost degree limited edge	×	total cost	degree	$\|R\| \leq d(m) * 2$	$O(n^3)$	def. 68
RS-MC	Remove search miminum cost degree limited edge	×	total cost	degree	$\|R\| \leq d(m) * 2$	$O(n^3)$	def. 68
RTR-MC	Remove try reconfiguration and MC-edge	✓	total cost	degree	$\|R\| \leq d(m) * 2$	$O(n^2)$	def. 68
RTR-P-MC	Remove try reconfiguration and prune	✓	total cost	degree	$\|R\| \leq \|E_T\| * 2$	$O(n^3)$	def. 69
RTR-PS-MC	Remove try reconfiguration, prune and search MDDL	✓	total cost	degree	$\|R\| \leq \|E_T\| * 2$	$O(n^3)$	def. 69

[1] The algorithm is able to add and remove non member-nodes (MNs).

Table 12.1: Algorithm descriptions and a set of properties.

Description	*Parameter*
Placement grid	$100x100$ milliseconds
Number of nodes in the network	1000
Degree limits	5 and 10
Super-nodes found by k-Center(k)	$k = 100$
Diameter bound	250 milliseconds

Table 12.2: Experiment configuration.

12.4.2 Target metrics

A dynamic tree algorithm is considered good if it can, in a timely fashion, insert and remove nodes from an existing overlay network such that the overlay yields a low diameter, a low average pair-wise latency, and does not add unreasonable cost to the network. For the evaluation of the overlays and algorithms, there are five metrics considered to be very important: overlay diameter, average pair-wise latency, algorithm execution time, total network cost and stability. Stability is measured in terms of how many edges it is that change in an overlay network upon an insert or remove of a node. In addition, given the fairly limited resources available to average clients in the Internet, the algorithm should obey degree-limitations such that the stress on each node in the overlay is bounded.

12.4.3 Experiment results

The *execution time* of the dynamic tree algorithms is very low compared to the tree algorithms from the previous chapters. Figure 12.4 plots the execution time of the fastest and best Steiner-tree algorithms from chapter 10.3 (sdl-RGH, sdl-OTTC, smddl-OTTC and sdl-SPT), compared to the worst-case remove strategies. In addition, the Steiner minimum-cost tree (SMT) heuristic average-distance heuristic (ADH) is plotted as a reference [1]. We observe that it is only RTR-PS-MDDL that needs more than 0.01 seconds to finish the reconfiguration in the group-range (0-160). The remaining insert and remove strategies executes in less than 0.002 seconds. Among the Steiner-tree algorithms it is sdl-RGH that is the fastest (< 0.1 seconds for group-sizes up to 130), while the three others are much slower. The main reason for the fast execution times for insert and remove strategies is the size of the reconfiguration set (see figure 12.3). Ordinary tree algorithms rebuild the entire tree for each insert and remove, while, the dynamic algorithms reconfigure smaller parts of the tree. Hence, the time complexity of the dynamic algorithms does not greatly influence the execution time because the reconfiguration set size is small.

The *total cost* of selected dynamic-tree and Steiner-tree algorithms are plotted in figure 12.5. We observe that the dynamic tree algorithms that yield a low diameter (combinations of RTR-MDDL and RK) have a total cost which is twice the total cost of the tree produced from the

[1] The evaluation in chapter 10 identified that ADH was the only evaluated SMT heuristic that could identify Steiner-points given a full mesh of shortest paths.

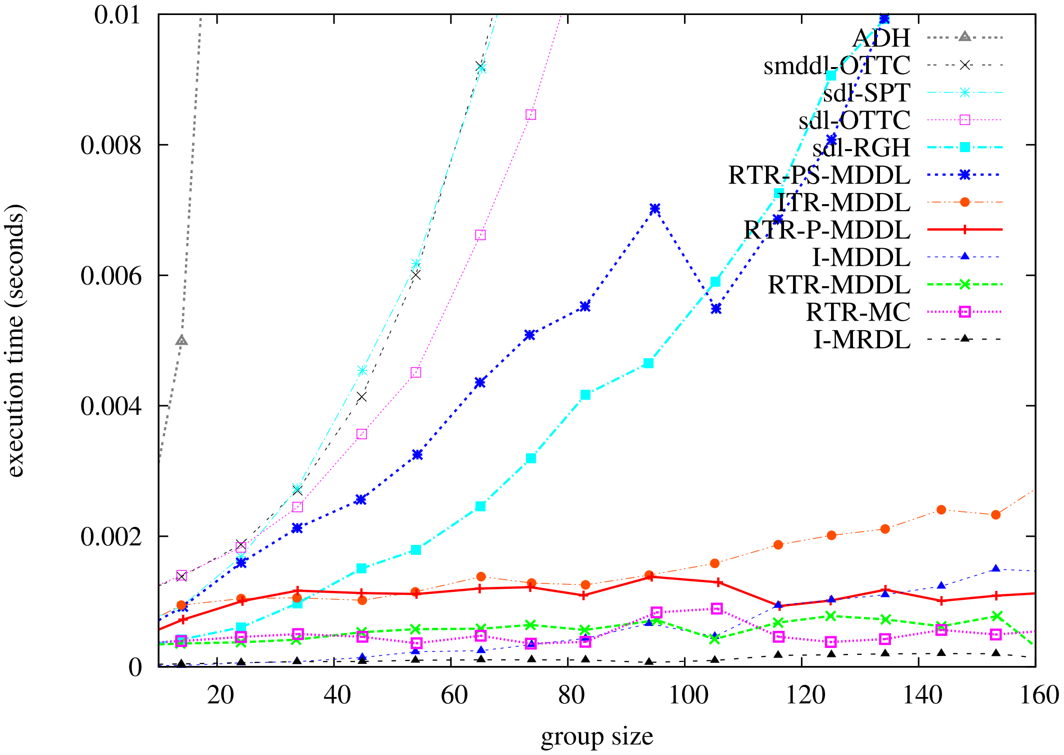


Figure 12.4: The execution times (seconds) of selected dynamic tree algorithms.

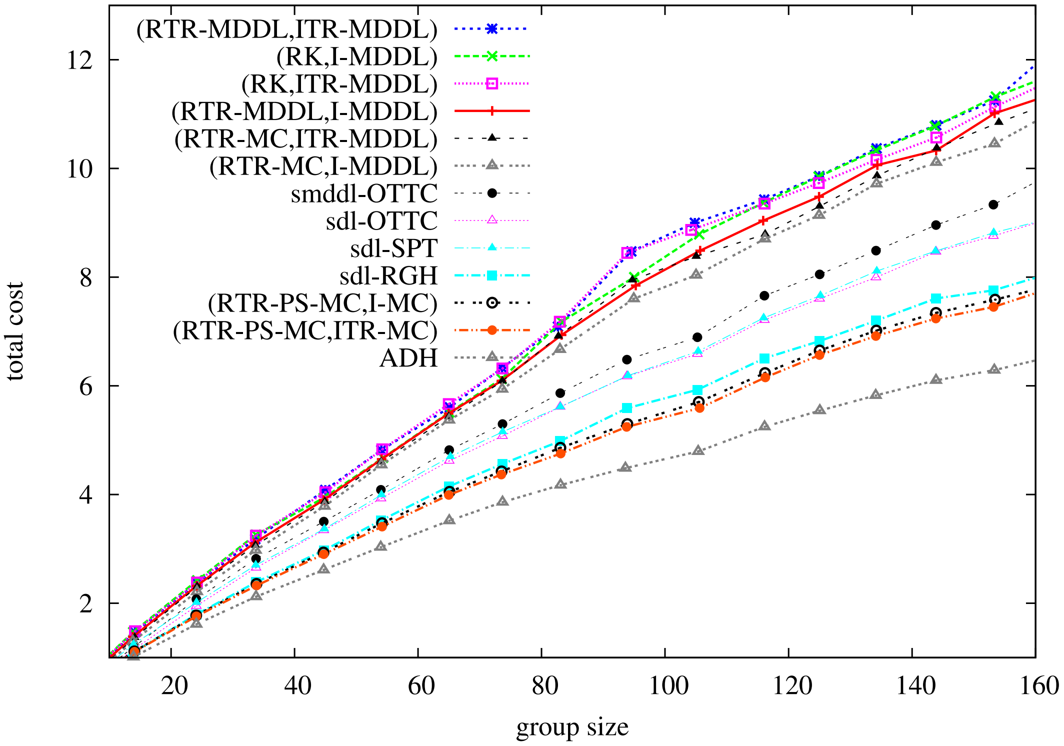


Figure 12.5: The total cost of the trees, as produced by selected dynamic tree algorithms.

SMT-heuristic ADH (chapter 10). Furthermore, the same dynamic tree algorithms also yield a total cost, which is around 30 % higher than the diameter-optimizing Steiner-tree algorithms. The cheapest dynamic tree algorithms (combinations of RTR-PS-MC) produce group trees that are the only about 20 % more costly than ADH.

The *diameter* (seconds) achieved by every combination of insert and remove strategy is plotted in figure 12.6. We observe that insert and remove strategies that optimize for minimum-cost yield a high diameter. The diameter-optimizing remove strategies that cannot add Steiner-points to the group-tree (R-MDDL and RS-MDDL) also perform badly. The remove strategies that prune non-member nodes (RTR-P-MDDL, etc) also produce group-trees with large diameters. The 3 remove strategies, RK, RTR-MC and RTR-MDDL, perform well with almost all the insert strategies that consider the diameter or radius while updating group-trees. The remove strategy RK is very naive (see chapter 12) and only removes non-member-nodes with degree < 3 from the group-tree. RTR-MC and RTR-MDDL, on the other hand, have procedures that check if it is beneficial for the tree-cost or -diameter (respectively) to i) keep the leaving node as a non-member, ii) add a well-placed Steiner-point to the tree, iii) or reconnect using one neighbor as intersection point. It is apparent that these remove-procedures fit diameter-optimizing insert strategies particularly well.

Figure 12.7 and 12.8 plots the best-case dynamic tree algorithm combinations (insert,remove) from the simulations. They are compared to the best diameter-reducing Steiner-tree algorithm from chapter 10.3, namely, smddl-OTTC. We observe that the best dynamic algorithms (RTR-MC,I-MDDL) and (RTR-MC,I-MRDL) produce group trees that yield a diameter, which is 30 % larger than the close-to-optimal smddl-OTTC. However, we saw above that the execution time of smddl-OTTC quickly closes in on 100 milliseconds. The dynamic algorithms (RK,I-MDDL) and (RK,I-MRDL) yield low diameter group-trees for small group-sizes. However, the diameter gets up to 0.5 seconds for group sizes larger than 100. The main observation is that the dynamic tree algorithms perform comparatively better for smaller group sizes, than larger group sizes. In addition, from the results it is clear that the dynamic tree algorithms are more unpredictable in their behavior. Especially, the group-tree diameter varies quite alot more than for the Steiner-tree algorithms that reconstruct the entire tree for each membership change.

Figure 12.9 plots the diameter for selected dynamic algorithm combinations. We observe that the dynamic algorithms (R-MDDL,I-MDDL) and (RS-MDDL,I-MDDL) that are unable to add Steiner-points yield high diameter trees. The remove strategies with pruning (RTR-P-MDDL and RTR-PS-MDDL) perform similarly with any of the insert strategies. Further work is needed to adjust the aggresiveness of the pruning step. The remove strategies RK, RTR-MDDL and RTR-MC all perform well with ITR-MDDL and I-MDDL. As noted previously, there is a slight tendency that (RK, I-MDDL) degrades for larger groups. From the figure it is apparent that non-member-nodes (Steiner-points) are very important for the performance of dynamic algorithms. In particular, the remove strategies RTR-MDDL and RTR-MC looks to

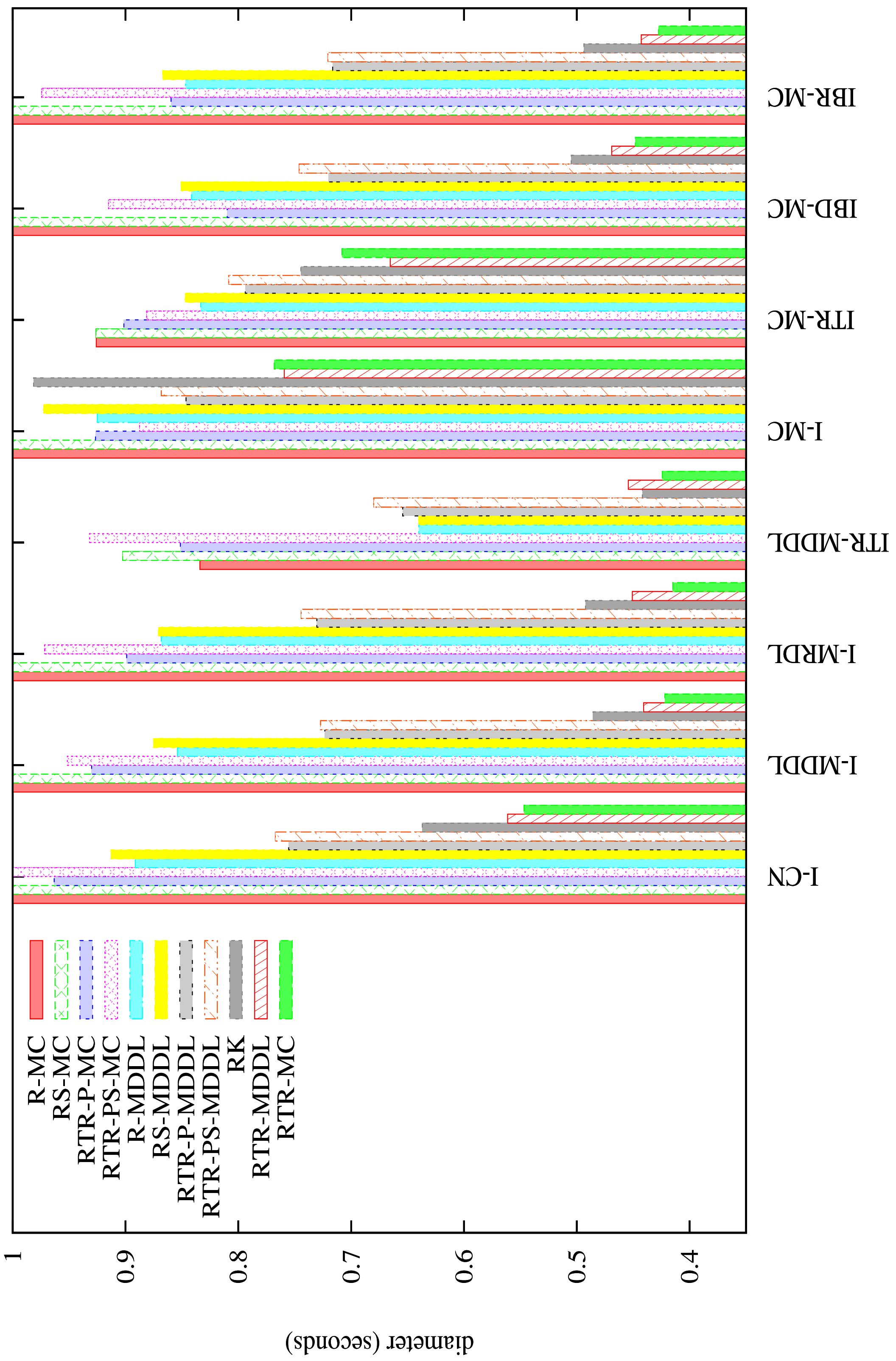


Figure 12.6: Diameter (seconds) of trees (every insert and remove combination).

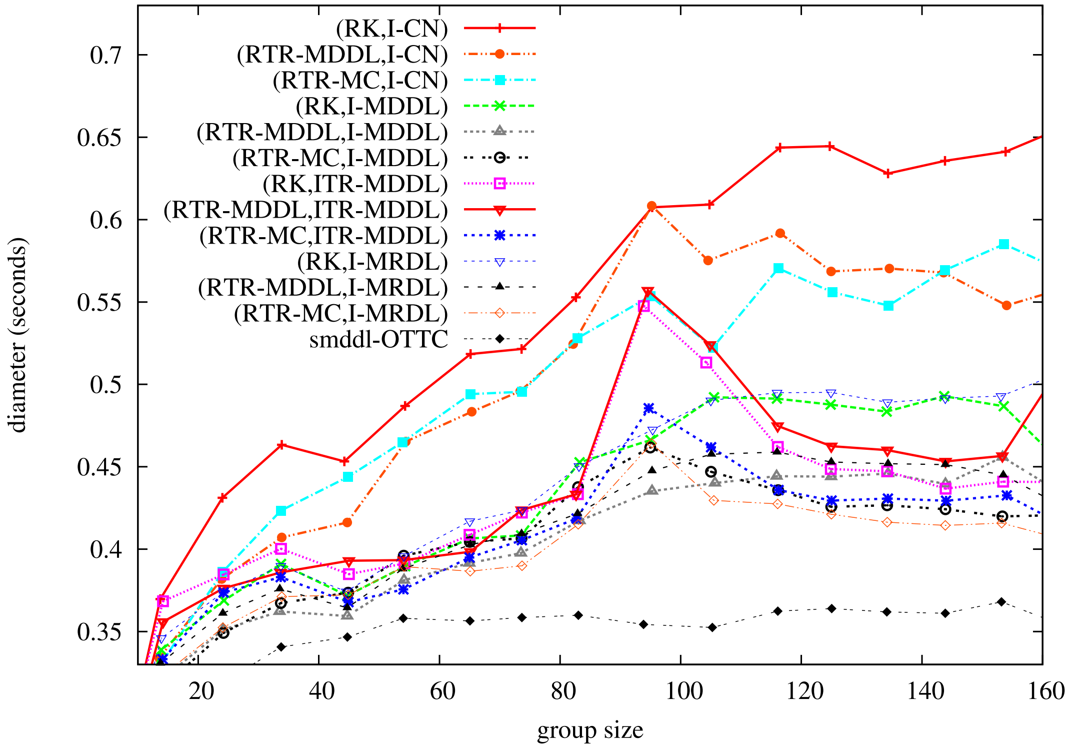


Figure 12.7: Diameter (seconds) of trees as constructed by the better performing selected dynamic tree algorithms.

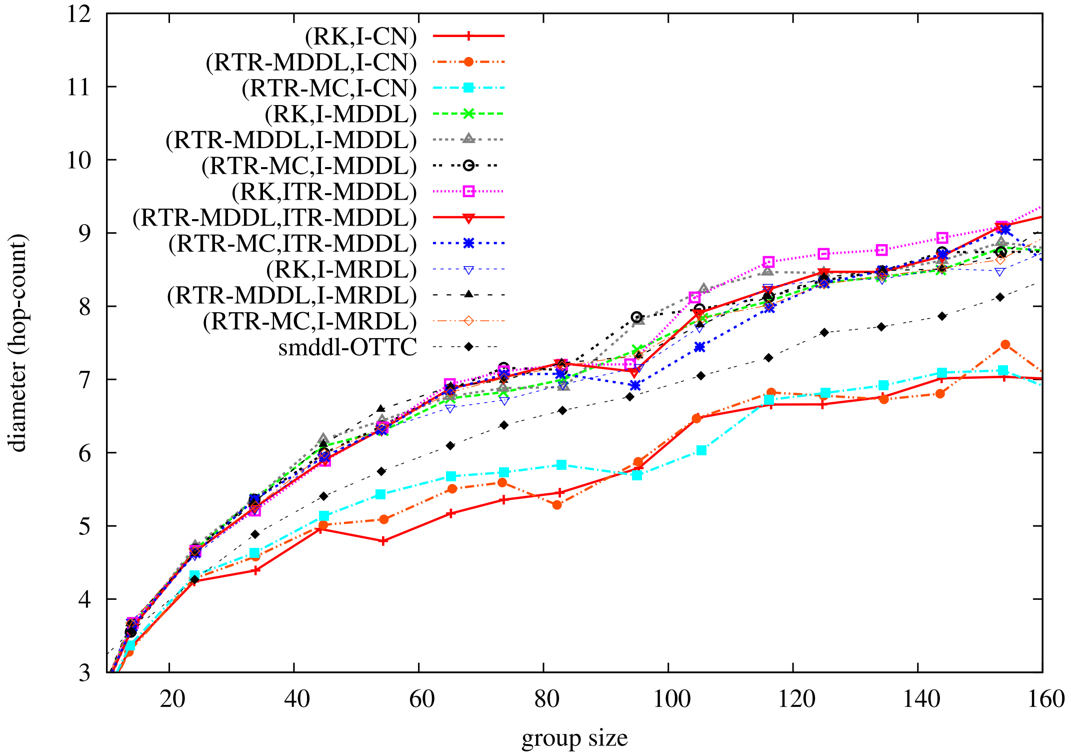


Figure 12.8: Diameter (hop-count) of trees as constructed by the better performing selected dynamic tree algorithms.

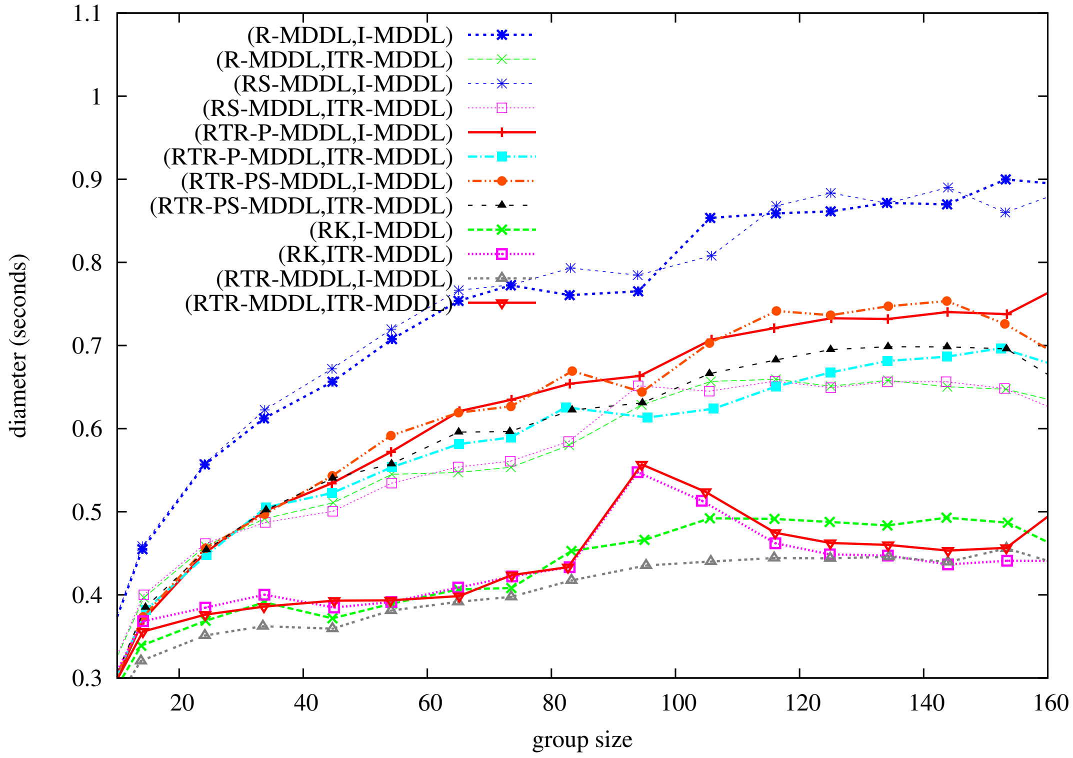


Figure 12.9: Diameter (seconds) of trees as constructed by various dynamic tree algorithms.

work well with I-MDDL. These remove strategies are able to choose what benefits the tree more for each remove request.

The reason for why well placed non-member-nodes are important to let remain in the tree, is the increased degree capacity centrally. Observe that the diameter-reducing insert strategies ITR-MDDL, I-MDDL, and I-MRDL exhaust the degree of nodes that reduce the diameter of the tree. Commonly, such nodes are centrally located well placed nodes. The result is that many member nodes are leaf-nodes that have a well-placed node as a neighbor. When a centrally located member-node, with many member-nodes attached to it, is leaving the tree, its neighbors must be reconnected by the remove strategy. It is often the case that these neighbors have maxed out their degree-limitations. Therefore, some neighbors are forced to reconnect to poorly placed neighbors that results in a higher diameter. This sequence of events is mainly due to i) degree-limitations that results in bounded capacity on well-placed nodes, ii) the goal of not reconfiguring large portions of the tree, iii) the resulting simplicity of the remove strategies.

For the pruning strategies, the case when a node m tries to leave, is that they include many non member nodes to the reconfiguration set and reduce the number of non members (including well-placed nodes). Thus, a lower degree capacity is the result, which increases the diameter. RTR-MDDL and RTR-MC do not prune non-member nodes (other than m), but may include well-placed nodes. Thus, RTR-MDDL and RTR-MC are able to produce trees with a lower diameter, because they have a higher degree capacity in their trees.

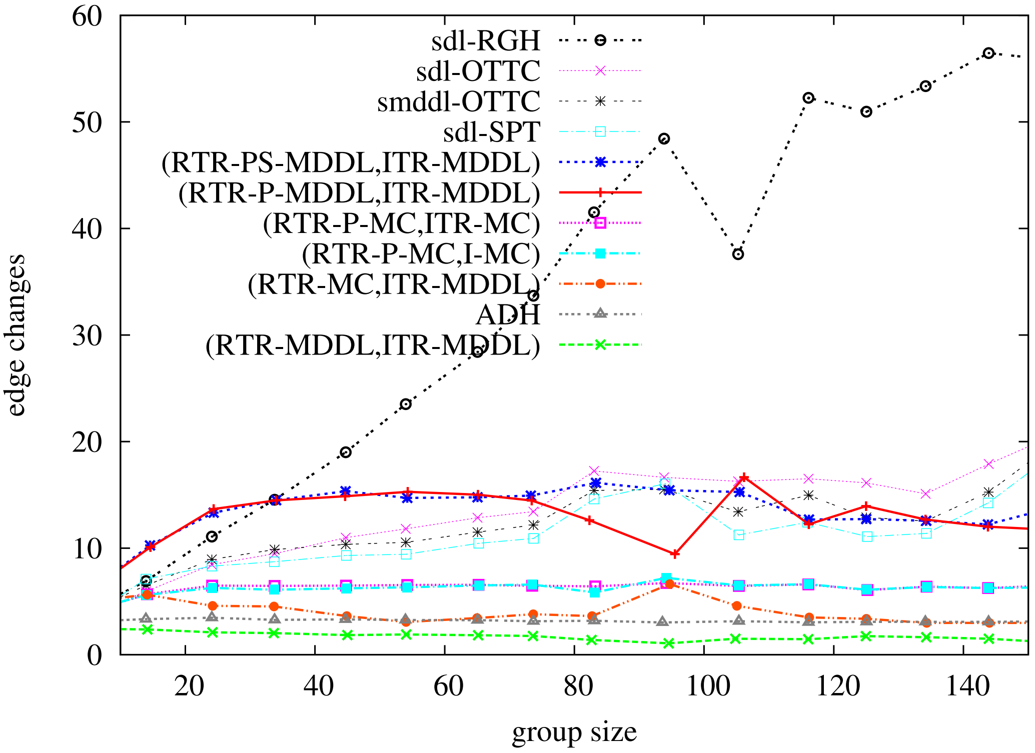


Figure 12.10: The mot unstable dynamic tree algorithms in terms of edge-changes in a tree-reconfiguration.

To summarize, we observed that the diameter-reducing insert strategies (ITR-MDDL, I-MDDL, I-MRDL) in combination with the remove strategies with pruning (RTR-P, etc) yield trees with a higher diameter than the same insert strategies in combination with RK, RTR-MDDL and RTR-MC. The pruning strategies have a powerful pruning mechanism (larger reconfiguration set) that makes the insert strategies less influential. RTR-MDDL and RTR-MC are less influential (smaller reconfiguration set) and thus cooperates better. RK performs well combined with insert strategies that optimize for the diameter, but looks to degrade when the group sizes increase beyond 100. However, RK and ITR-MDDL couples well, and yield low diameter trees. In this combination, the insert strategy is more powerful and ensures that well-placed member-nodes or non-member-nodes are strongly connected upon execution.

The *stability* of the dynamic tree algorithms is compared with that of the Steiner-tree spanning-heuristics sdl-RGH, sdl-OTTC, smddl-OTTC and sdl-SPT in figure 12.10. A higher stability is indicated by a lower number of edges that are changed across reconfigurations. It is clear that the randomized sdl-RGH is much too unstable (high number of edge changes) to be an option for dynamic scenarios. The remaining Steiner-tree spanning-heuristics are fairly stable around 10-15 edge changes. We also see that the stability of the pruning strategies is actually worse than the best case diameter reducing Steiner-tree heuristics (for example, sdl-SPT).

The SMT-heuristic ADH is very stable throughout the group-size range. The pruning strategies have the lowest stability when they are combined with insert strategies that optimize for the diamter (ITR-MDDL, I-MDDL, etc.). The reason for this is that all of these insert strategies are biased towards making connections between well-placed nodes that are potentially non-member-nodes. Remove strategies with pruning collapse these areas in their reconfigurations to reduce the number of non-member-nodes. Conversely, the stability is higher for RTR-P-MC in combination with ITR-MC and I-MC because they optimize for total cost and make few connections between well-placed non-member nodes. The (RTR-MDDL,ITR-MDDL) dynamic tree algorithm is consistenly very stable, and has an average of 3 edge changes for each configuration. (RTR-MC,ITR-MC) is only slightly worse, with an average of 5 edge changes. Many of the insert strategies only change (add) one edge when joining a new node to a group-tree. ITR-MDDL and ITR-MC have procedures that try to fit a well-placed joining node inside the group-tree. Their number of edge-changes are comparable to that of RTR-MDDL and RTR-MC. Figure 12.11 adds the observation that the stability of RK and RTR-MDDL is fairly unaffected by group size. And that the combination (RK,I-CN) is extremely stable with a number of edge-changes closing in to zero. Figure 12.12 shows the stability of all remove strategies in combination with different insert strategies. It is quite clear that the remove strategies with pruning are more unstable with diameter-optimizing insert strategies, than minimum-cost insert strategies. Furthermore, we see that RK and RTR-MDDL are both very stable with any of the insert strategies.

12.4.4 Discussing and comparing the results

We have tested several combinations of insert and remove strategies for efficiency, in terms of the tree metrics introduced in chapter 12.4.2. We have also compared the dynamic tree algorithms with Steiner-tree spanning-heuristics. These Steiner-tree heuristics are approximation algorithms for the Steiner-MDDL, Steiner-MRDL and BDDLSMT problems (see chapter 4.6). These heuristics outperform our dynamic-tree algorithms in many situations, but we saw that their practical use is limited in a dynamic scenario, because of their high execution time (tree-reconfiguration time) (figure 12.4). In addition, their stability is low, in that they change quite a few edges upon a reconfiguration, which is only natural for algorithms that construct trees from scratch (figure 12.10).

From the results we find that there are insert and remove strategies that do not fit together. For example, an insert strategy that tries to minimize the diameter does not fit best with a remove algorithm that reconfigures with the aim of minimizing the total cost. This effect can be seen clearly in the figures. RTR-P-MC optimizes for total cost and we can see that the consequence is that the combination of RTR-P-MC with I-MDDL produces trees with lower total cost (figure 12.5) but higher diameter (figure 12.7). Additionally, RK does not include

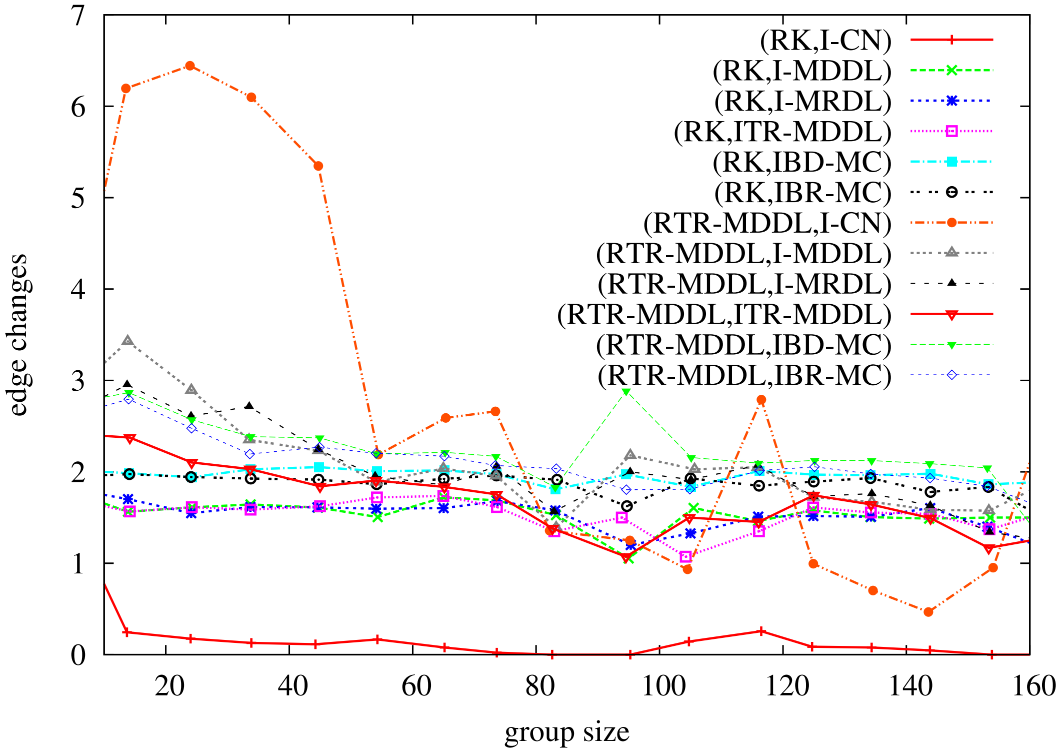


Figure 12.11: Selected dynamic tree algorithms and their reconfiguration stability.

well-placed non-member-nodes but leaves members with degree > 2 in the tree. We see that RK performs better with respect to the diameter in combination with the diameter-optimizing strategies ITR-MDDL, I-MDDL and I-MRDL. RK combined with insert strategies that are oriented towards a low total cost, ITR-MC and I-MC, makes the group-trees degrade.

When the diameter-oriented strategy I-CN is used, it is likely that most members are either leaves or have a higher outdegreE. This leads to a small number of edge changes when it is combined with RK, as shown in figure 12.11. Since RK only removes a leaving group member if it has a $degree \leq 2$, it leaves inner nodes untouched. The negative effect that this has on the diameter can be seen in figure 12.7. RTR-MDDL and RTR-MC avoid this penalty to the diameter, because they remove poorly placed (leaving) non-member nodes with higher degree.

Figure 12.13 plots the number of non-members-nodes (Steiner points) in the trees, i.e., the *non-member node degradation*, using remove strategies RK and RTR-MDDL combined with ITR-MDDL, I-MDDL and IBR-MC. We see that RK and RTR-MDDL combined with IBR-MC has a higher number of Steiner points, than the reamining combinations. IBR-MC is a bounded diameter insert strategy that choose low-cost edges if the inserted node has an eccentricity below the diameter-bound (0.25 seconds). We also see that combinations with RK has around 25 % more non-member-nodes than the RTR-MDDL combinations. Ths Steiner-tree spanning-heuristics sdl-SPT and sdl-OTTC have both fairly small numbers of Steiner-points (non-member-nodes) in their trees. Figure 12.14 plots every (remove,insert) strategies and their

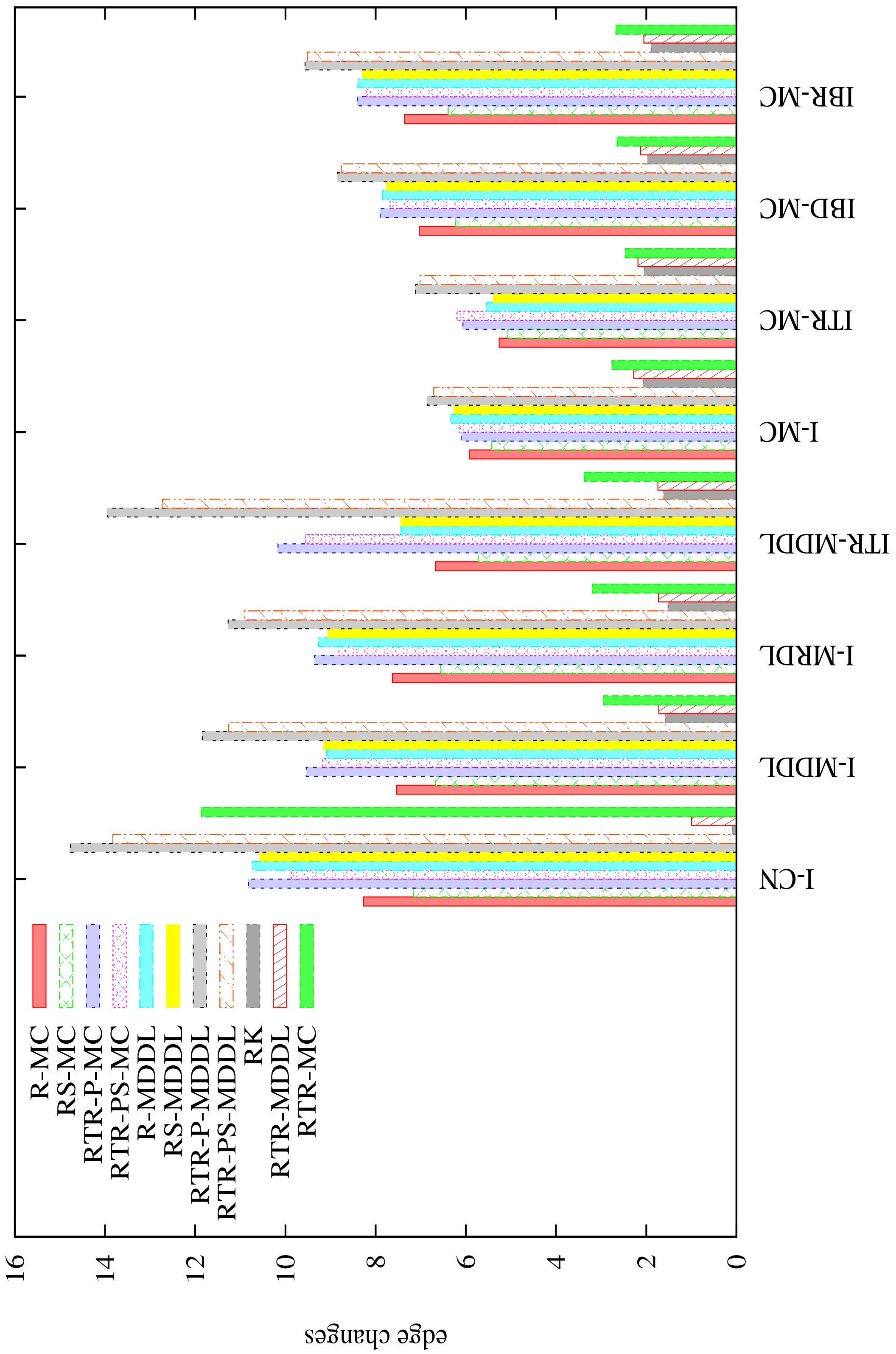


Figure 12.12: The edge-changes in a reconfiguration (group size 110-130).

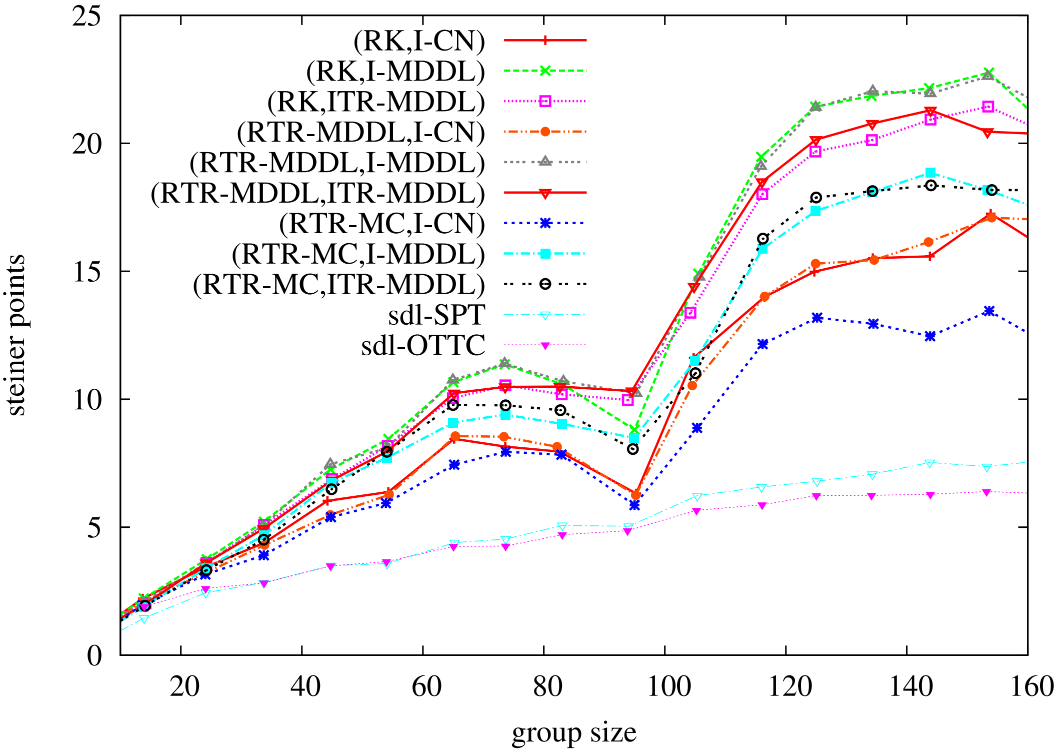


Figure 12.13: The non-member-node degradation, measured by the number of Steiner-points in a tree.

stability in terms of the number of edge changes in the tree reconfigurations. It is clear that the RK and RTR-MDDL combinations have much larger number of Steiner-points in their group-trees than any other combination.

Figure 12.15 plots the edge changes for selected remove strategies for each reconfiguration as a scatter-plot (insert strategy is I-MDDL). Each point in the graph is the number of edge-changes after a remove request and the following tree-reconfiguration. We can see that RTR-P-MC have a much higher maximum numbers of edge changes than RTR-MC. RK is never above three edge changes.

12.4.5 Results with varying degree limits

Figure 12.16 plots the diameter achieved by selected dynamic tree algorithms, and degree-limits 5 and 10. It is quite clear that the dynamic-tree algorithms do not perform as good when the degree-limit is reduced to 5, compared to the close-to-optimal smddl-OTTC. The hop-diameter and the stability, however, is not being affected, as seen in figure 12.17 and 12.18.

Figure 12.19 plots the non-member-node degradation in terms of the number of Steiner-points in the group-trees. For larger group-sizes the number of Steiner-points increases quite rapidly to above 30 for the dynamic-tree algorithms.

The non-member-node degradation may be a reason for why the dynamic-tree algorithms

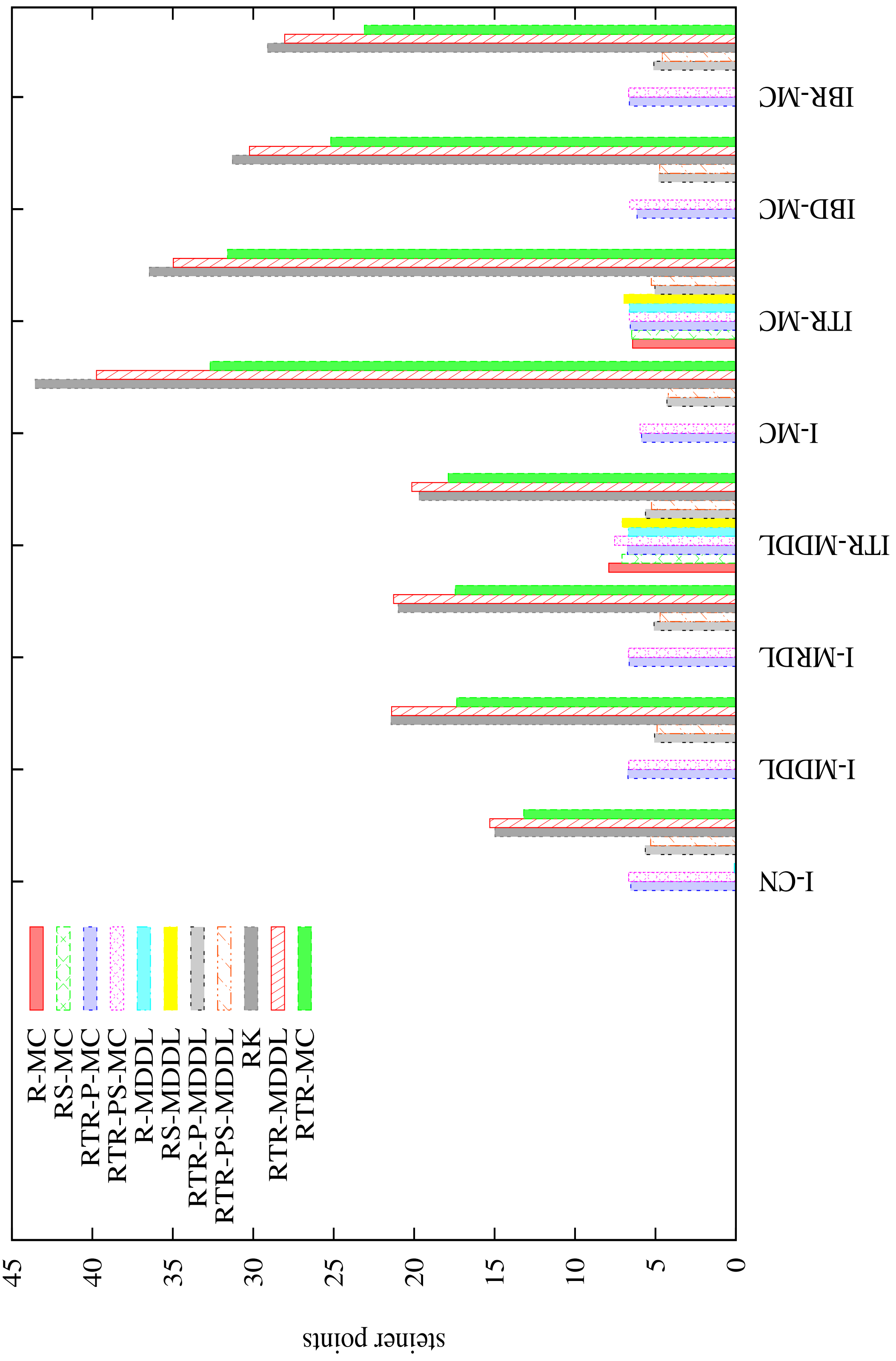


Figure 12.14: The non-member-node degradation for every insert and remove combination.

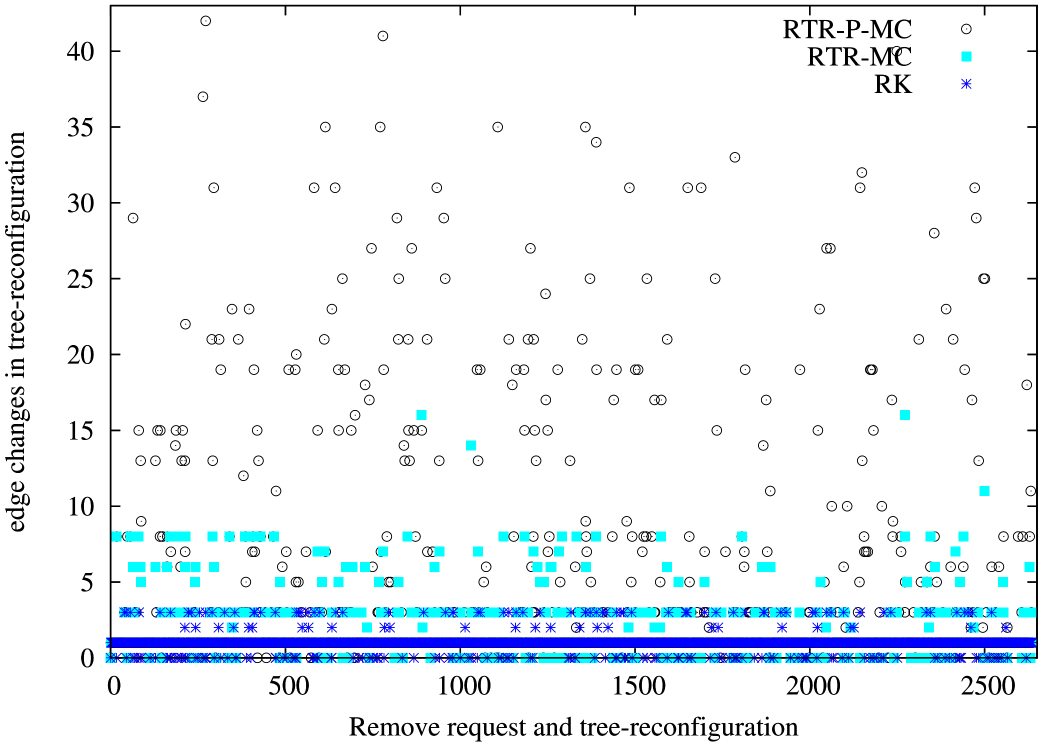


Figure 12.15: Scatter plot of the stability of remove strategies combined with I-MDDL. Each plot is the number of edge-changes after a remove request and the tree-reconfiguration.

are less competetive in terms of the tree-diameter (seconds) when the degree-limit is reduced to 5. Other reasons include the fact that dynamic tree algorithms are very simplistic in nature, and a low degree bound is much harder to solve than higher degree bounds. It is clear that these dynamic tree algorithms do no maintain group-trees with a sufficiently low diameter when the degree-limit is low.

12.4.6 Discussions on a few of the better algorithms

In the previous section, we identified that there exist insert and leave strategies that do not fit together, and some that fit especially well together. With this in mind, we have summarized all our observations in table 12.3. In the table, every combination of insert and remove strategy as well as the tree heuristics are evaluated towards our target metrics. The evaluation is based on the results from our experiments, but (inevitably) it is also based on subjective opinions from the authors. If an algorithm behaves positively towards a target metric it is represented with a $" + "$, correspondingly, if negatively with a $" - "$. The *performance index* is the sum of the positives ($"+"$) of an algorithm, where it is possible to get a maximum performance index of 4. (The optimization goals low *total cost* and small *diameter* are contradictory.)

Overall, we observe that RK, RTR-MC and RTR-MDDL performs well in combination

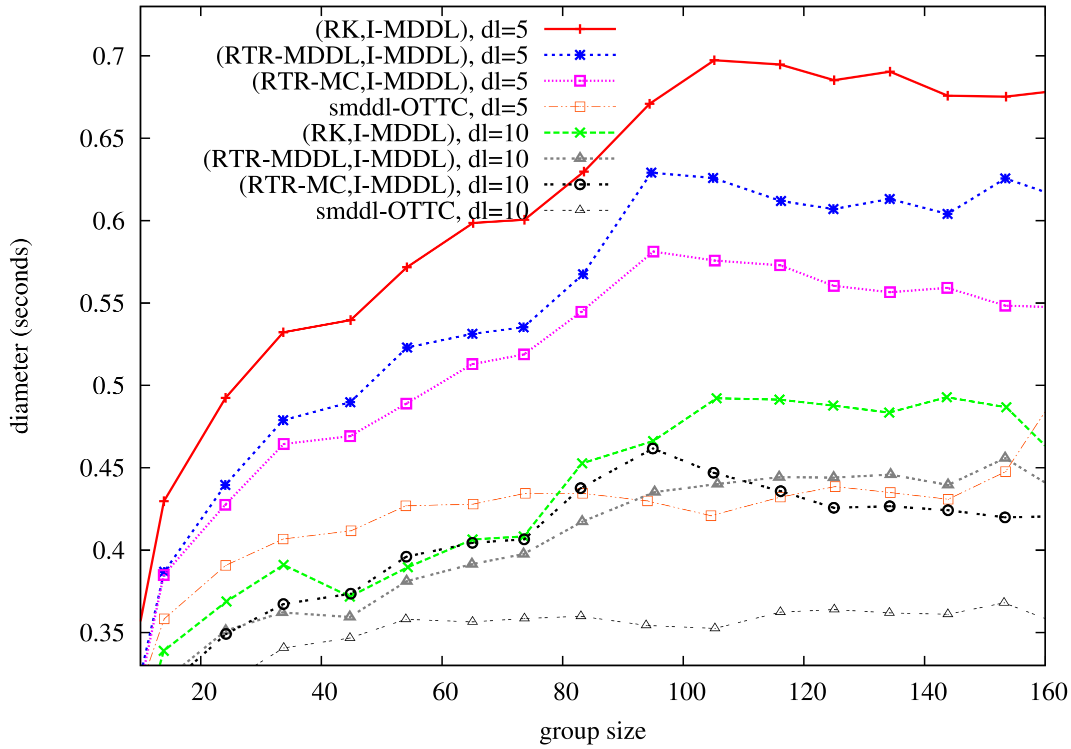


Figure 12.16: Diameter (seconds) of trees as constructed by the better performing selected dynamic tree algorithms.

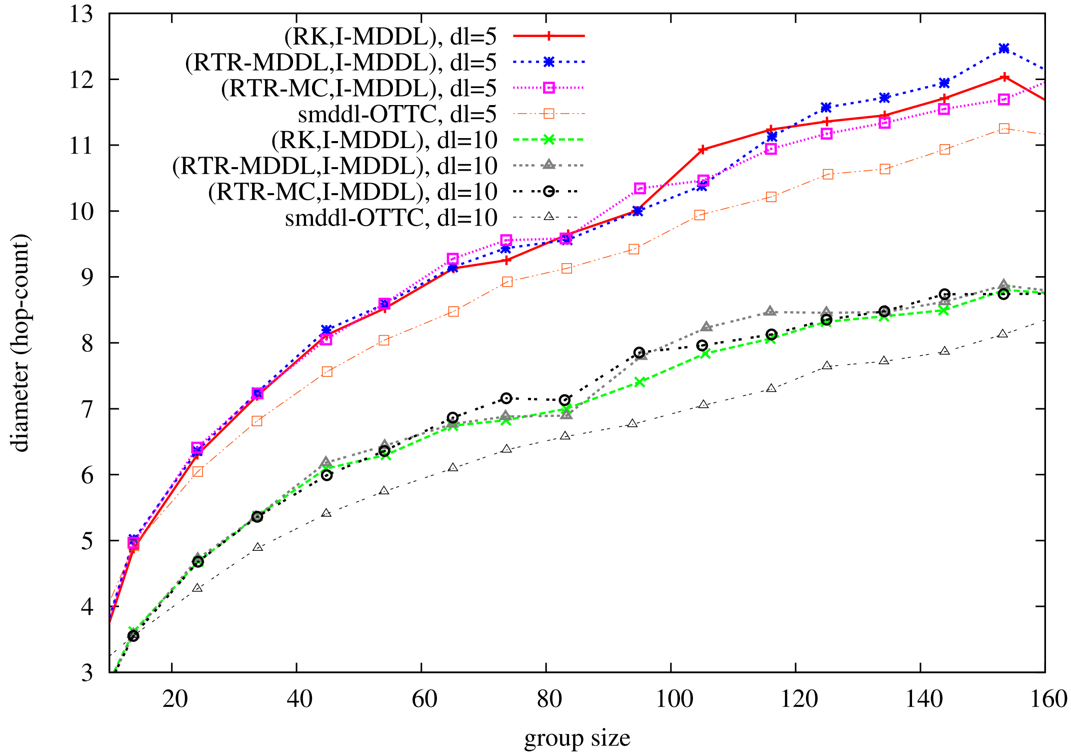


Figure 12.17: Diameter (hop-count) of trees as constructed by the better performing selected dynamic tree algorithms.

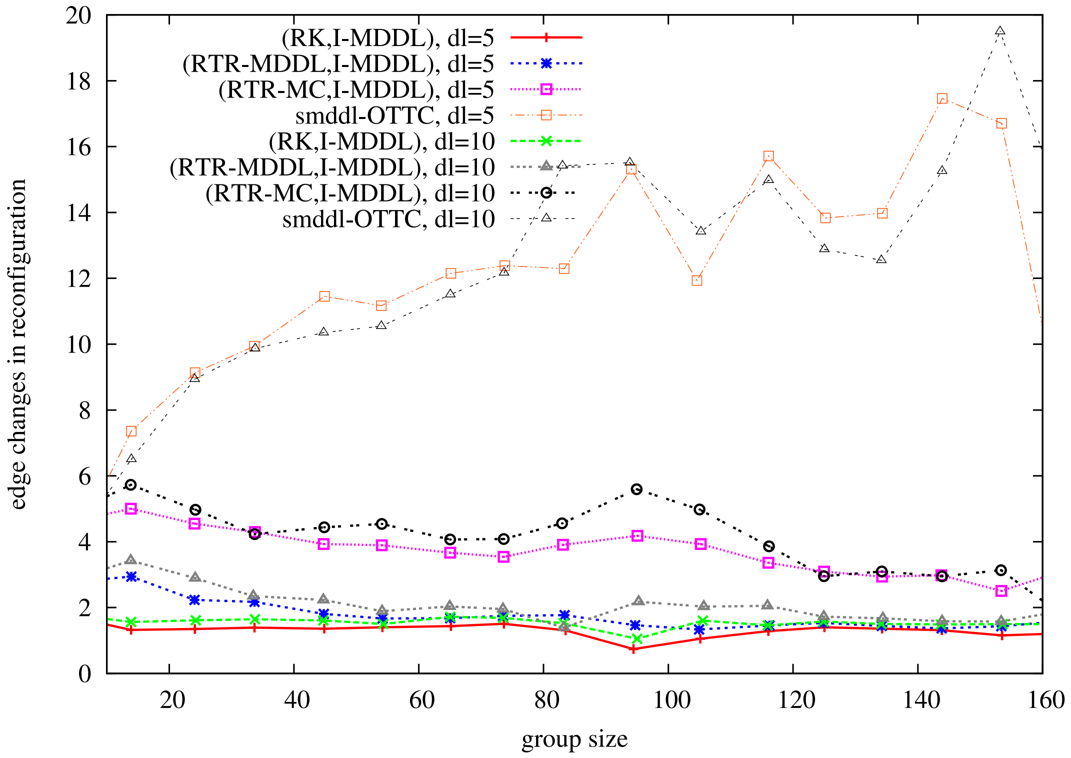


Figure 12.18: Selected dynamic tree algorithms and their reconfiguration stability.

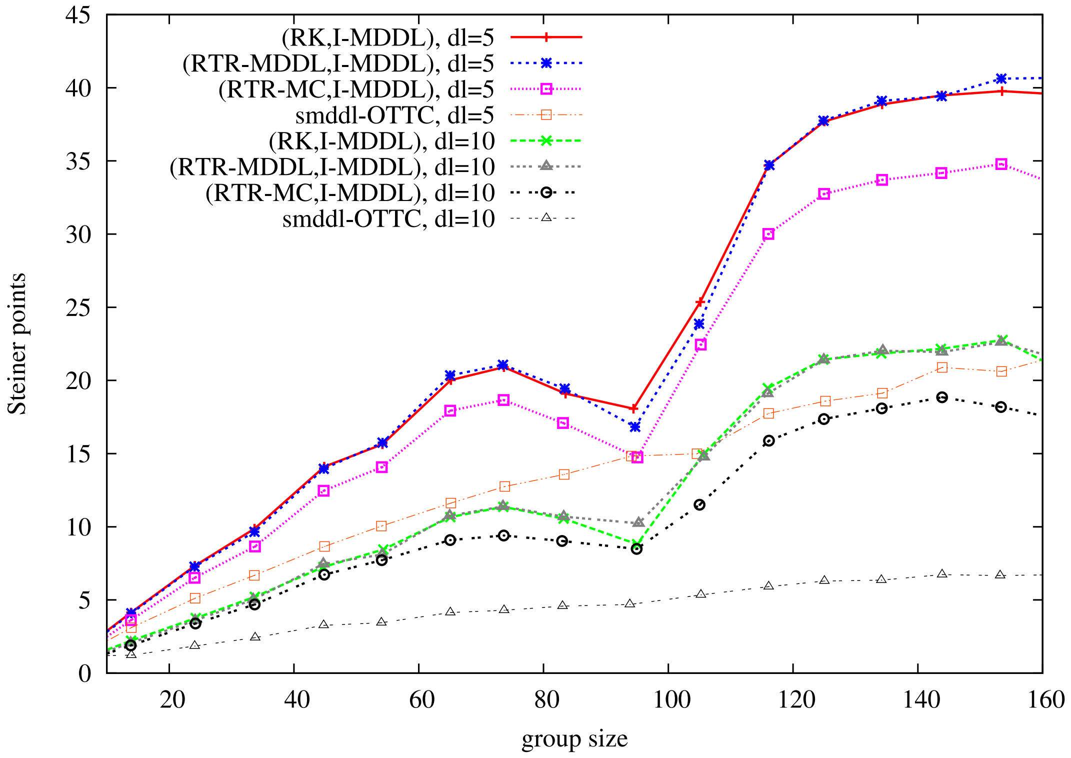


Figure 12.19: The non-member-node degradation, measured by the number of Steiner-points in a tree.

with insert strategies that optimize for the *diameter*. Insert strategies that optimize for *total cost*, for example I-MC, combined with these remove strategies suffer from non-member node degradation, i.e., increasing number of non-member nodes in the trees. However, it is RK that suffer most from non-member-node degradation. RTR-MC and RTR-MDDL are the remove strategies that perform best with insert strategies that opt for a low diameter. The reason for this may be that they keep the reconfiguration sets small, and, hence, does not greatly interfere the insert strategy. Remove strategies with pruning (for example, RTR-P-MC) fit well with insert strategies that optimize for the *total cost*. They also perform satisfactory combined with the remaining insert strategies, but the *diameter* does not get as good as with RTR-MC and RTR-MDDL. Most importantly, the number of *edge changes* is higher with RTR-P-MC. One approach may be to run remove with pruning only periodically.

The Steiner-tree heuristics that were compared to the dynamic tree algorithms produced trees with the lowest diameter, and did not suffer from non-member-node degradation. However, their exeuction times are much higher and their stability is not satisfactory, because the number of edge-changes in a tree-reconfiguration is quite high. Especially the randomized sdl-RGH had a very high number of edge-changes across tree-reconfigurations.

The best dynamic-tree algorithm are (RTR-MC,I-MRDL) and (RTR-MDDL, I-MDDL), and they yielded tree-diameters that were about 30 % higher than the close-to-optimal smddl-OTTC (0.45 seconds vs. 0.35 seconds) for group sizes > 80. For group sizes < 80 the difference was around 15 % (0.40 seconds vs. 0.35 seconds). Overall we saw that dynamic-tree algorithms have a tendency to degrade for larger group sizes.

12.5 Summary of the main points

The previous investigation evaluated dynamic tree algorithms and compared them to selected close-to-optimal Steiner-tree heurisitc. All the tested algorithms were centralized where a given central entity executed the group management (see chapter 5). The algorithms optimized, independently, for *total cost* and *diameter*, under given degree limits ($= 10$).

The results showed that the dynamic tree algorithms have very low *execution time*, and that they produce low diameter trees that can compete with the Steiner-tree heuristics smddl-OTTC, sdl-SPT and sdl-OTTC. In addition, we were able to minimize the number of *edge changes* for each reconfiguration. However, we also showed that there is still room for improvements, for example, reducing the number of non-member nodes in a tree.

Table 12.3 highlighted that RK, RTR-MC and RTR-MDDL performed very well combined with all of the diameter-reducing insert strategies I-MDDL, ITR-MDDL, I-MRDL and IBR-MC. Remove strategies with pruning (for example, RTR-P-MC and RTR-P-MDDL), on the other hand, did not yield low diameter trees with any of the insert strategies they were combined

Algorithm		Performance metrics					Performance index
Remove	Insert	*MN ratio*[1]	*Total cost*	*Diameter*	*Stability*	*Reconfig. time*	
RK	I-MC	−	−	−	+	+	2
	ITR-MC	−	−	−	+	+	2
	I-MDDL	+	−	+	+	+	4
	I-CN	+	−	−	+	+	3
	IBR-MC	+	−	+	+	+	4
RTR-MC	I-MC	−	+	−	+	+	3
	ITR-MC	−	+	−	+	+	3
	I-MDDL	+	−	+	+	+	4
	I-CN	+	−	+	+	+	4
	IBR-MC	+	−	+	+	+	4
RTR-MDDL	I-MC	−	−	−	+	+	2
	ITR-MC	−	−	−	+	+	2
	I-MDDL	+	−	+	+	+	4
	I-CN	+	−	+	+	+	4
	IBR-MC	+	−	+	+	+	4
RTR-P-MC	I-MC	+	+	−	+	+	4
	ITR-MC	+	+	−	+	+	4
	I-MDDL	+	−	+	−	+	3
	I-CN	+	−	+	−	+	3
	IBR-MC	+	−	+	−	+	3
ADH		+	+	−	+	−	3
smddl-OTTC		+	−	+	−	−	2

[1] The member node (MN) ratio in a tree should be high, if not it is crowded with non-member nodes and degrade.

Table 12.3: Algorithm performance.

with. However, the combination I-MC and RTR-P-MC/RTR-PS-MC produces trees with low *total cost*, even compared to the close to optimal ADH. The center based insert strategies are fast and simple, but still gave good results in terms of the *diameter*.

In summary, our results show that the remove strategies RK, RTR-MC and RTR-MDDL that use smaller reconfiguration sets, fit better with insert strategies that optimize for the *diameter*. In addition, it was apparent that non-member-nodes (Steiner-points) are very important for the performance of the dynamic tree algorithms. The Steiner-points enabled better reconfiguration options for RTR-MC and RTR-MDDL. Remove strategies with pruning use a (potentially) larger reconfiguration set and fit best with insert strategies that optimize for the *total cost*. The results also showed that the insert and remove strategies perform very differently with respect to our target metrics.

We shall further investigate reconfigure strategies that are run independently of insert and remove strategies (chapter 14). Further work is also needed for the group Steiner-tree heuristic mddl-ReconnectTree, as we were unable to find a suitable remove strategy for it. In addition, the pruning steps of RTR-P-MC, RTR-P-MDDL, etc, needs adjustment, and should be tested with a less aggressive pruning approach.

The next investigation (chapter 13) is of dynamic subgraph algorithms, which are identical to dynamic tree algorithms, except they can build cyclic connected subgraphs.

Chapter 13

Overlay construction techniques: Dynamic subgraph algorithms

The overlay network management from section 5.4 includes overlay construction techniques whose task is to construct low-latency overlay networks for distribution of time-dependent events. In that respect, we continue our evaluation of overlay construction techniques and evaluate dynamic subgraph algorithms. By doing this we address a goal of the thesis:

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

A dynamic subgraph algorithm belongs to the class of dynamic subgraph problems, as introduced in section 4.11. These problems address client dynamicity, which should be supported by distributed interactive applications (section 2.3). The dynamic subgraph algorithms introduced in the thesis are able to handle incoming requests of type:

Insert *node m to the subgraph M, such that m can communicate with the nodes in V_M.*
Remove *node m from the subgraph M, such that the nodes in V_M can still communicate.*

Like the dynamic tree algorithms from chapter 12, a dynamic subgraph algorithm must support both insert and remove requests. In addition, the dynamic subgraph algorithms allows the construction of cyclic subgraphs, which increases the failure tolerance of the subgraph. Algorithm 68 shows a generic dynamic subgraph algorithm, in which an incoming insert or remove request is handled by different strategies.

The chapter introduces a range of insert and remove strategies that are paired as dynamic-subgraph algorithms. The strategies include approaches that optimize for minimum-cost and minimum-diameter subgraphs. All of the combinations of insert and remove strategies are evaluated in a group communication simulator. The results from the group communication simulations show that remove strategies that remove non-member nodes are necessary to prevent the subgraphs from non-member-node degradation. Non-member-node degradation occurs

Algorithm 68 GENERIC-DYNAMIC-SUBGRAPH-ALGORITHM

```
In: $G = (V, E, c)$, a connected subgraph $M_Z = (V_M, E_M)$, a set of members $Z \subseteq V_M$, and a request $\rho$
Out: Updated connected subgraph $M_Z = (V_M, E_M)$
if $\rho = \{remove, m\}$ then
    REMOVE-NODE-SUBGRAPH-ALGORITHM($G$,$M_Z$,$m$)
else if $\rho = \{insert, m\}$ then
    INSERT-NODE-SUBGRAPH-ALGORITHM($G$,$M_Z$,$m$)
end if
```

when too many non-member-nodes are left in the subgraphs, such that the subgraph degrades in terms of a larger diameter, total-cost, etc. However, we found that a simple minimum-cost insert strategy combined with a minimum-cost remove strategy with pruning are able to maintain a subgraph such that the subgraph diameter is consistently low throughout the evaluated group-size range (0-170).

The rest of the chapter is organized in the following manner. Section 13.1 presents a few common types of dynamic subgraph algorithms. Section 13.2 presents the insert and remove strategies that are paired as dynamic subgraph algorithms. Section 13.3 evaluates the results from the group communication simulations using the dynamic subgraph algorithms. Finally, section 13.4 gives a brief summary of the main points.

13.1 Dynamic subgraph algorithm types

There are many possible types of dynamic subgraph algorithms, their similarity is that they insert and remove nodes from a connected subgraph, based on incoming requests. Following, are some examples of possible dynamic subgraph algorithms that optimize for a low-cost and minimum-diameter subgraph.

13.1.1 Low-cost dynamic subgraph algorithms

A pure minimum-cost connected subgraph is equal to the minimum-cost connected spanning-tree. However, a minimum-cost dynamic subgraph algorithm should exhibit some configurable property that allows for fault tolerance in the subgraph. The dynamic subgraph algorithm should address the dynamic subgraph problems from section 4.11, and relate the construction to the problems of k-connectivity (section 4.9). A k-connected graph has the property that the removal of any $k - 1$ nodes leave the subgraph in a connected state. For example, a k-connected minimum-cost dynamic subgraph algorithm is given a sequence of insert and remove requests, and for each request the algorithm updates the current subgraph such that it is k-connected and has minimum-cost. However, the k-connected minimum-cost dynamic subgraph algorithm (definition 24) is NP-complete, such that it is not a goal to solve this exactly.

13.1.2 Minimum-diameter dynamic subgraph algorithms

A minimum-diameter dynamic subgraph algorithm may, for example, produce a minimum-diameter subgraph within a given total cost bound. The algorithm should address the dynamic subgraph problems (section 4.11), and relate the construction requirements to the problems of constructing a minimum diameter spanning subgraph (definition 74) and a k-connected minimum-diameter spanning subgraph (definition 76). However, this problem is NP-complete, and in a dynamic scenario it is especially difficult to approximate a close-to-optimal solution, while at the same time service incoming join and leave requests sufficiently fast.

13.2 Evaluated dynamic subgraph algorithms

The following introduces a number of centralized dynamic subgraph algorithms that are evaluated through simulations and experiments in section 13.3. The dynamic subgraph algorithms presented here, consist of insert and remove strategies that are both spanning-subgraph and Steiner-subgraph strategies (section 4.11). One insert strategy and one remove strategy are paired to one dynamic subgraph algorithm. A dynamic subgraph algorithm works as such:

k-Dynamic-Subgraph-Algorithm($\mathscr{R}_{\mathscr{D}}$, $\mathscr{I}_{\mathscr{D}}$, k) (kDA) takes as input an insert strategy $\mathscr{I}_{\mathscr{D}}$ and a remove strategy $\mathscr{R}_{\mathscr{D}}$, an integer $k \geq 1$, and a request ρ. Furthermore, a global undirected weighted graph $G = (V, E, c)$, where V is the set of vertices ($n = |V|$), E is the set of edges, $c : E \rightarrow \mathbb{R}$ is the edge cost function. Moreover, a connected subgraph $M_Z = (V_M, E_M)$, where $V_M \subset V$, and $E_M \subset E$. In addition, there is a set of member-nodes $Z \subset V_M$.

kDA executes $\mathscr{I}_{\mathscr{D}}$ upon an insert request, and $\mathscr{R}_{\mathscr{D}}$ upon a remove request, and updates the subgraph M_Z accordingly (algorithm 69). It is only the insert strategy that adds k edges to the subgraph. The remove strategy removes m and reconnects the subgraph as if it was a tree. The main reason for this is that we do not want to crowd the subgraph with too many costly edges.

Algorithm 69 k-DYNAMIC-SUBGRAPH-ALGORITHM

```
In: $G = (V, E, c)$, a subgraph $M_Z = (V_M, E_M)$, a set of members $Z \subseteq V_M$, an integer $k \geq 1$, a request $\rho$, one insert strategy $\mathscr{I}_{\mathscr{D}}$ and one remove strategy $\mathscr{R}_{\mathscr{D}}\}$.
Out: Updated subgraph $M_Z = (V_M, E_M)$
if $\rho = \{remove, m\}$ then
    $\mathscr{R}_{\mathscr{D}}(G, M_Z, m, k = 1)$
else if $\rho = \{insert, m\}$ then
    $\mathscr{I}_{\mathscr{D}}(G, M_Z, m, k)$
end if
```

13.2.1 Insert strategies

An insert strategy $\mathscr{I}_{\mathscr{D}}$ inserts a new member m into the subgraph and assures that m can communicate with the remaining member nodes. Formally, an insert strategy works like this:

Given $G = (V, E, c)$, a connected subgraph $M = (V_M, E_M)$, a set of members $Z_M \subseteq V_M$, and a new member $m \in V$. Update M, such that $Z_M \cup \{m\}$ are connected.

The new member m may be inserted into M in many different manners. We have devised 3 insert strategies that connect m to the subgraph using k edges. Thus, they bound the size of the reconfiguration set R to these k edges.

I-MC(k) (minimum cost) connects m to the subgraph using the k least cost edges to nodes in the subgraph. I-MC(k) is a $O(n)$ algorithm that optimizes for *total cost*, and approximates k-connected subgraphs of minimum-cost that resemble spanning subgraphs.

Algorithm 70 INSERT-MINIMUM-COST(k)

```
In: $G = (V, E, c)$, a subgraph $M_Z = (V_M, E_M)$, a set of members $Z \subseteq V_M$, a new member $m$ , and an integer $k \geq 1$
Out: Updated subgraph $M_Z = (V_M, E_M)$
1: Find $k$ least-cost edges from $m$ to subgraph nodes $v \in V_M$
2: connect $m$ to $M_Z$ through the $k$ edges
```

I-MDDL(k) (minimum diameter degree limited) connects m to the subgraph using the k edges that results in the lowest eccentricity to m in the subgraph. I-MDDL(k) approximates a k-connected subgraph of minimum-diameter. I-MDDL(k) is a $O(n^3)$ algorithm and uses $|V_T|$ Dijkstra's shortest-path searches to find the k tree-nodes $v \in V_T$ that m can connect to, such that its eccentricity is minimized.

Algorithm 71 INSERT-MINIMUM-DIAMETER-DL(k)

```
In: $G = (V, E, c)$, a subgraph $M_Z = (V_M, E_M)$, a set of members $Z \subseteq V_M$, a new member $m$ , and an integer $k \geq 1$
Out: Updated subgraph $M_Z = (V_M, E_M)$
1: Find $k$ edges to subgraph nodes $v \in V_M$ such that the eccentricity of $m$ is minimized
2: connect $m$ to $M_Z$ through the $k$ edges
```

I-MC-MDDL(k) (minimum-cost, minimum diameter degree limited) connects m to the subgraph using the minimum-cost and minimum-eccentricity edges alternately. For example, if $k = 1$, it inserts m using the minimum-cost edge, if $k = 2$ it uses one minimum-cost edge and one minimum-eccentricity edge, $k = 3$ two minimum-cost edges and one minimum-eccentricity edge, and so on. I-MC-MDDL(k) approximates a k-connected subgraph of low-cost and low-diameter. I-MDDL(k) is a $O(n^3)$ algorithm, because it uses $|V_T|$ Dijkstra's shortest-path searches to find the tree-nodes $v \in V_T$ that m can connect to, such that its eccentricity is minimized. The search for the minimum-cost edge, is a cheap $O(n)$ procedure.

Algorithm 72 INSERT-MINIMUM-DIAMETER-DL(k)

```
In: $G = (V,E,c)$, a subgraph $M_Z = (V_M,E_M)$, a set of members $Z \subseteq V_M$, a new member $m$ , and an integer $k \geq 1$
Out: Updated subgraph $M_Z = (V_M,E_M)$
1: Find $k$ edges to subgraph nodes $v \in V_M$ such that the eccentricity of $m$ is minimized
2: connect $m$ to $M_Z$ through the $k$ edges
```

13.2.2 Remove strategies

A remove strategy $\mathscr{R}_{\mathscr{D}}$ removes a member m from the subgraph and assures that the member-nodes are connected. Formally, a remove strategy works like this:

Given $G = (V,E,c)$, a connected subgraph $M = (V_M,E_M)$, a set of members $Z_M \subseteq V_M$, and a member $m \in V$. Update M, such that $Z_M \setminus \{m\}$ are connected.

The number of direct neighbor nodes of m (its degree) in M influences the necessary actions. If the degree $d_M(m) = 1$, m is a leaf that is simply removed along with the edge to its only neighbor. If it is greater than 1, a removal of m may partition the subgraph, unless it is a k-connected subgraph with $k > 1$. It is a hard problem to ensure that a graph stays k-connected for $k > 1$. Therefore, the basic goal of the remove strategies is to ensure that the subgraphs stay connected after removing m.

Evaluated remove strategies: In our evaluation, it is the remove strategies from chapter 12 that are paired with the subgraph insert strategies. The remove strategies are applicable to cyclic subgraphs (meshes) as well as trees, however their performances in a mesh may be very different from a tree. The main difference is that depth-first searches in a cyclic subgraph do not find the longest shortest path. Instead, the remove-strategies apply shortest-path searches using Dijkstra's SPT approach [58]. Therefore, the time complexity of the remove-strategies for subgraphs are all one degree higher than their correspondent remove strategy for reconnecting trees.

For subgraphs it is expected that the number of nodes with a degree less than 3 is fairly low. Therefore, it is very much important that the remove strategies are able to remove non-member-nodes from the subgraph. Due to this, we regard RK as a bad alternative and do not evaluate it. Figure 13.1 illustrates how RK, RTR-MC and RTR-P-MC reconfigure a given tree when a node m leaves. RK only removes leaf-nodes, which may be a problem in highly connected subgraphs because there are potentially few leaf-nodes. Moreover, RTR-MC is able to reconfigure locally, but may face degradation issues when the group-size increases. Finally, RTR-P-MC reduces the number of links and the number of non-member-nodes upon reconfiguration, and, as we shall see in the following, is able to avoid the degradation issues of RK and RTR-MC.

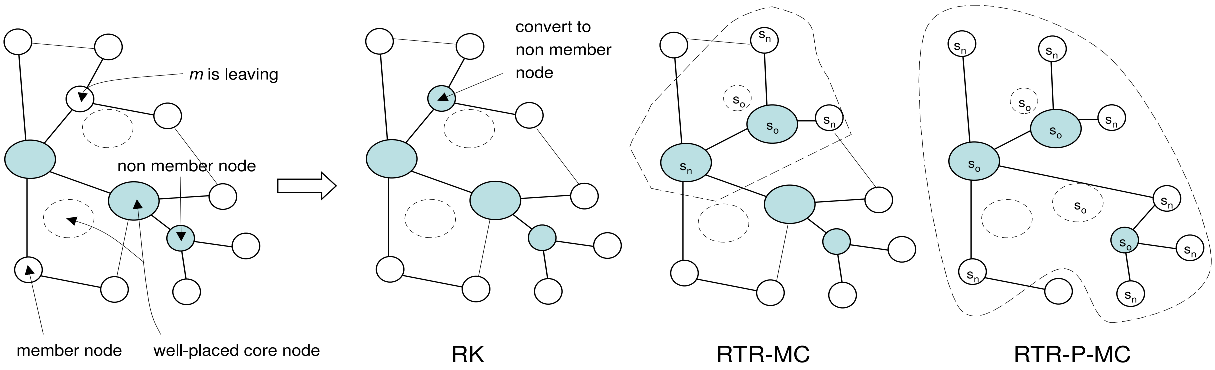


Figure 13.1: Remove subgraph reconfiguration examples. RTR-P-MC avoids non-member-node degradation by aggressive pruning.

13.3 Group communication simulations of dynamic subgraph algorithms

Following are evaluations of the dynamic subgraph algorithms introduced previously. They address the graph theoretical problems from section 4.11. There are many different combinations of insert and remove strategies possible, however, we shall focus on a few combinations that work well, and mention the combinations that should not be used.

Before we start the evaluation, we summarize a few of the past experiences. Chapter 11 evaluated spanning subgraph algorithms and Steiner subgraph algorithms. In those evaluations, it was identified that trees optimized for the diameter can compete with subgraphs with a relatively high connectivity. It was also found that subgraph algorithms that optimize for total cost also yield fairly low diameters and average pair-wise distances. Moreover, when Steiner points were added to input graphs, every subgraph algorithm yielded a lower diameter than without Steiner points.

The following evaluation of dynamic subgraph algorithms starts out from these experiences and try to find similar approaches. Table 13.1 has tabulated information about the evaluated dynamic subgraph algorithms.

13.3.1 Experiment configurations

In the experiments, we use equation 8.1 (section 8.5.4) to approximate the number k of Steiner-points that are needed to ensure that the degree-limited dynamic subgraph algorithms are able to reconnect a subgraph. The k Steiner points are chosen by k-Median(k) from 100 Steiner-points (super-nodes) that are identified at the beginning among the 1000 nodes in the overlay network. These well-placed super-nodes (chapter 8) are found by the multiple core-node selection algorithm k-Center($k = 100$).

Algorithm	*Meaning*	*MN aware*	*Optimization*	*Constraints*	*Reconfiguration set R*	*Time complexity*	*Problem*
I-MDDL(k)	Insert miminum diameter degree limited edge	×	diameter	degree	$\|R\| = k$	$O(n^3)$	def. 98
I-MC(k)	Insert miminum cost degree limited edge	×	total cost	degree	$\|R\| = k$	$O(n)$	def. 98
I-MC-MDDL(k)	Insert miminum cost and minimum diameter degree limited edge	×	total cost	degree	$\|R\| = k$	$O(n^3)$	def. 98
R-MDDL	Remove miminum diameter degree limited edge	×	diameter	degree	$\|R\| \leq d(m) * 2$	$O(n^3)$	def. 102
RS-MDDL	Remove search miminum diameter degree limited edge	×	diameter	degree	$\|R\| \leq d(m) * 2$	$O(n^3)$	def. 102
RTR-MDDL	Remove try reconfiguration and MDDL-edge	✓	diameter	degree	$\|R\| \leq d(m) * 2$	$O(n^2)$	def. 102
RTR-P-MDDL	Remove try reconfiguration and prune	✓	diameter	degree	$\|R\| \leq \|E_T\| * 2$	$O(n^3)$	def. 103
RTR-PS-MDDL	Remove try reconfiguration, prune and search MDDL	✓	diameter	degree	$\|R\| \leq \|E_T\| * 2$	$O(n^3)$	def. 103
R-MC	Remove miminum cost degree limited edge	×	total cost	degree	$\|R\| \leq d(m) * 2$	$O(n^3)$	def. 102
RS-MC	Remove search miminum cost degree limited edge	×	total cost	degree	$\|R\| \leq d(m) * 2$	$O(n^3)$	def. 102
RTR-MC	Remove try reconfiguration and MC-edge	✓	total cost	degree	$\|R\| \leq d(m) * 2$	$O(n^2)$	def. 102
RTR-P-MC	Remove try reconfiguration and prune	✓	total cost	degree	$\|R\| \leq \|E_T\| * 2$	$O(n^3)$	def. 103
RTR-PS-MC	Remove try reconfiguration, prune and search MDDL	✓	total cost	degree	$\|R\| \leq \|E_T\| * 2$	$O(n^3)$	def. 103

[1] The algorithm is able to add and remove non member-nodes (MNs).

Table 13.1: Algorithm descriptions and a set of properties.

Description	*Parameter*
Placement grid	$100x100$ milliseconds
Number of nodes in the network	1000
Degree limits	5 and 10
Super-nodes found by k-Center(k)	$k = 100$
Diameter bound	250 milliseconds

Table 13.2: Experiment configuration.

13.3.2 Target metrics

A dynamic subgraph algorithm is considered good if it can, in a timely fashion, insert and remove nodes from an existing overlay network such that the overlay yields a low diameter, a low average pair-wise latency, and does not add unreasonable cost to the network. For the evaluation of the overlays and algorithms, there are five metrics considered to be very important: overlay diameter, average pair-wise latency, algorithm execution time, total network cost and stability. Stability is measured in terms of how many edges change in an overlay network upon an insert or remove of a node. In addition, given the fairly limited resources available to average clients in the Internet, the algorithm should obey degree-limitations such that the stress on each node in the overlay is bounded.

13.3.3 Results from group communication simulations

Figure 13.2 plots the diameter (seconds) achieved by all combinations of insert and remove strategies that together form a single dynamic subgraph algorithm. From the figure, we see that there is little or no improvement with a $k > 2$, hence, we regard these as bad choices of k for most algorithms. In chapter 12.4 we learnt that combining insert and remove strategies that optimize for minimum-cost yield high tree-diameters. However, we observe that when $k = 2$, there are a number of combinations of I-MC that yield a low diameter. The remove strategies that prune (RTR-P-MC and RTR-P-MDDL) are performing well with I-MC when $k \geq 2$, in contrast to when $k = 1$.

Figure 13.3 plots the *total cost* of the dynamic subgraph algorithm combinations when $k = 2$. From the results it is clear that the remove strategies R-MC, RTR-MC and RTR-MDDL are not able to keep the cost of the subgraphs at a resonable level and degrade. RTR-P-MC and RTR-P-MDDL, on the other hand, yield a reasonable cost when $k = 2$. We also see that kDA(RTR-P-MC, I-MC, $k = 3$) yield subgraphs of reasonable total cost. kDA(RTR-P-MC,I-MC,$k = 1$) is plotted as a point of reference for the algorithms. The remove with pruning strategies remove excess non-member nodes (Steiner points) for each remove request. They use k-Neighbor-Steiner-Subtree($k = 1$) (chapter 12) to identify the reconfiguration set, and then adds the minimum amount of Steiner points neccessary to reconnect the reconfiguration set to the subgraph. The remaining remove strategies only consider neighboring nodes of the leaving node (k-Neighbor-Spanning-Subtree($k = 1$)) and cannot identify whether neighbors are Steiner-points or not. However, most importantly, *they do not remove a leaving node if the available degree is not sufficient to add the same amount of edges back to the subgraph.*

Figure 13.4 highlights the reason why the remove strategies R-MC, RTR-MC and RTR-MDDL yield such high subgraph-costs. These strategies suffer very much from non-member-node degradation. Too many leaving member-nodes are left in the subgraphs as Steiner-points, and the remove strategies are not able to reduce this number as the groups grow larger and larger. Hence, there is a massive non-member-node degradation, which results in a massive total cost of the subgraph. The figure also confirms that dynamic subgraph algorithms with the remove strategies RTR-P-MC and RTR-P-MDDL, have a rather small amount of Steiner points compared to the remaining dynamic subgraph algorithms. From these observation it is concluded to disregard the remove strategies R-MC, R-MDDL, RTR-MC and RTR-MDDL as alternatives when $k > 1$. We are left with the remove strategies RTR-P-MC and RTR-P-MDDL as alternatives for $k \geq 2$. The insert strategies are all performing well, since they only add k edges to the subgraphs for each insert.

13.3.4 Discussions and comparison of best combinations

The previous results and observations concluded that it is little point in considering a $k > 3$ for most algorithms. For $k = 2, 3$, the only remove strategies that yield a reasonable total cost without non-member-node degration were RTR-P-MC and RTR-P-MDDL. Combinations of RTR-P-MC and RTR-P-MDDL for $k > 1$, performed similarly, but RTR-P-MC was always slightly better in terms of total cost and almost identical for the diameter. Therefore, only RTR-P-MC is plotted in the following figures. These figures consider the better dynamic subgraph algorithm combinations.

Figure 13.5 plots the diameter (seconds) for the dynamic subgraph algorithms that performs best among the evaluated algorithms. The results show that the dynamic tree algorithm kDA(RTR-MDDL,I-MDDL,$k = 1$) yields subgraphs of a lower diameter (seconds) than every other dynamic algorithm combination for smaller group sizes (< 90). kDA(RTR-P-MC, I-MC, $k = 3, 4$) yields a slightly higher diameter than kDA(RTR-MDDL, I-MDDL, $k = 1$) for smaller group sizes, but has a lower diameter for larger group sizes (> 90). The diameter is consistently below 0.4 seconds for both combinations. RTR-P-MC combinations with $k = 2$ are also performing consistently well, and has a diameter just above 0.4 seconds. The Steiner-tree spanning-heuristic is plotted as the reference point, and we see that it produces the lowest diameter throughout the group-range. However, for smaller group-sizes (< 80) the dynamic subgraph algorithm kDA(RTR-MDDL, I-MDDL, $k = 1$) yields almost an identical diameter to that from smddl-OTTC. Figure 13.6 plots the hop-diameter, and shows a similar trend; however, the better dynamic subgraph algorithms yield a lower hop-diameter or similar to smddl-OTTC.

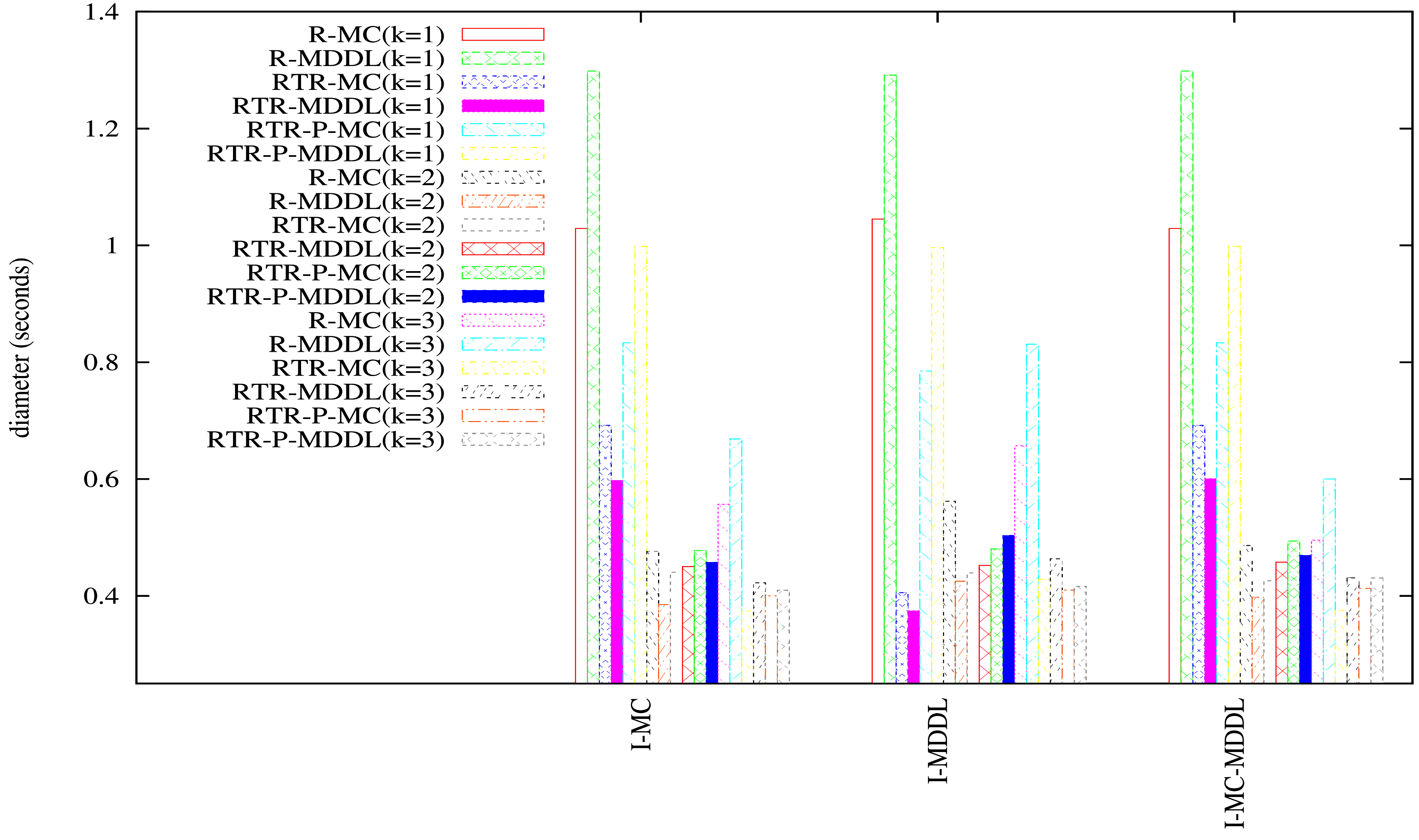


Figure 13.2: Comparison of all dynamic subgraph algorithms (insert,remove and k=1,2,3).

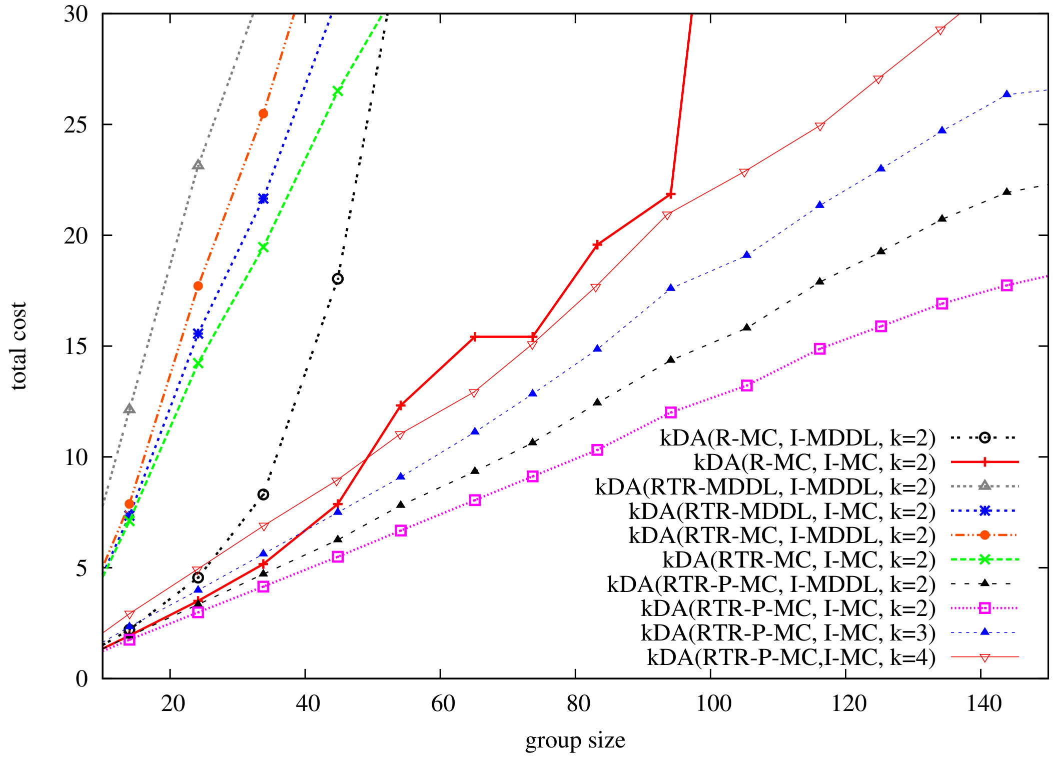


Figure 13.3: The total cost of the subgraphs as produced by selected dynamic subgraph algorithms.

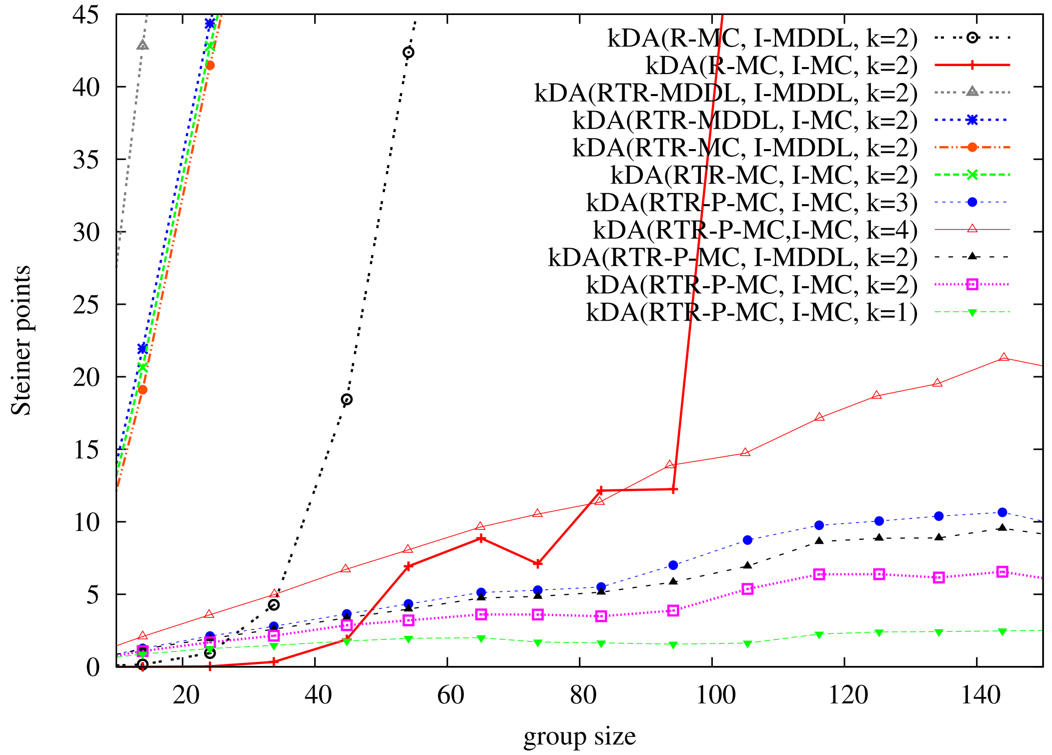


Figure 13.4: The number of Steiner-points in the subgraphs as produced by selected dynamic subgraph algorithms.

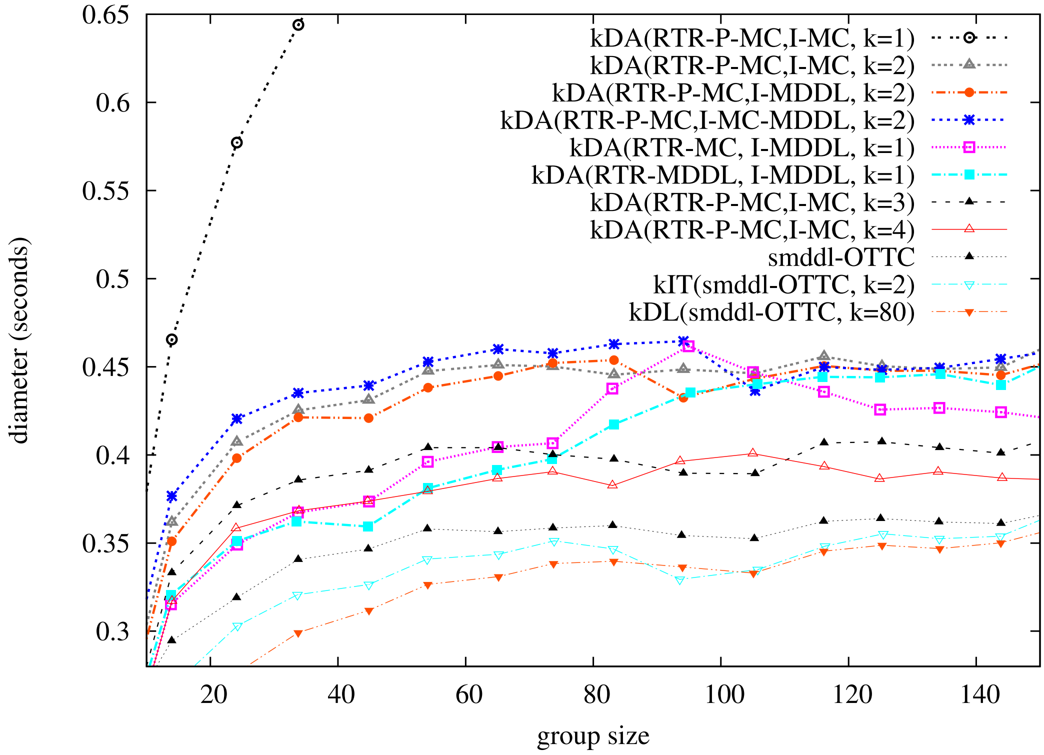


Figure 13.5: The diameter (seconds) of the better combinations of dynamic subgraph algorithms.

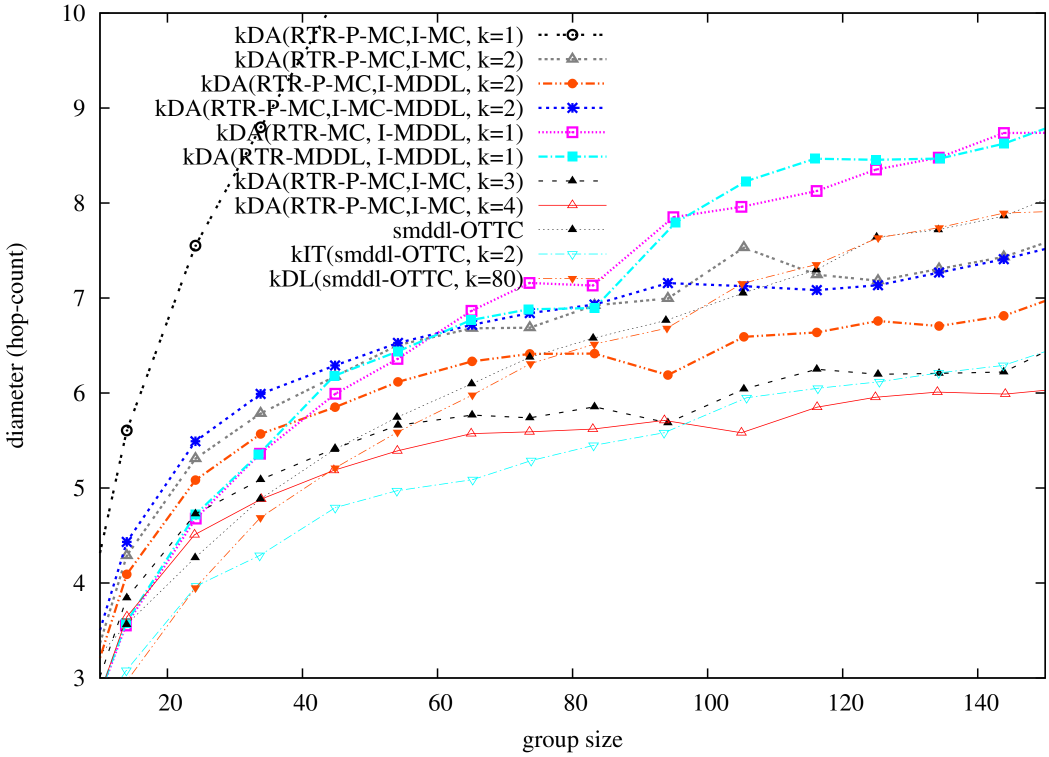


Figure 13.6: The hop-diameter of the subgraphs, as produced by dynamic subgraph algorithms and smddl-OTTC.

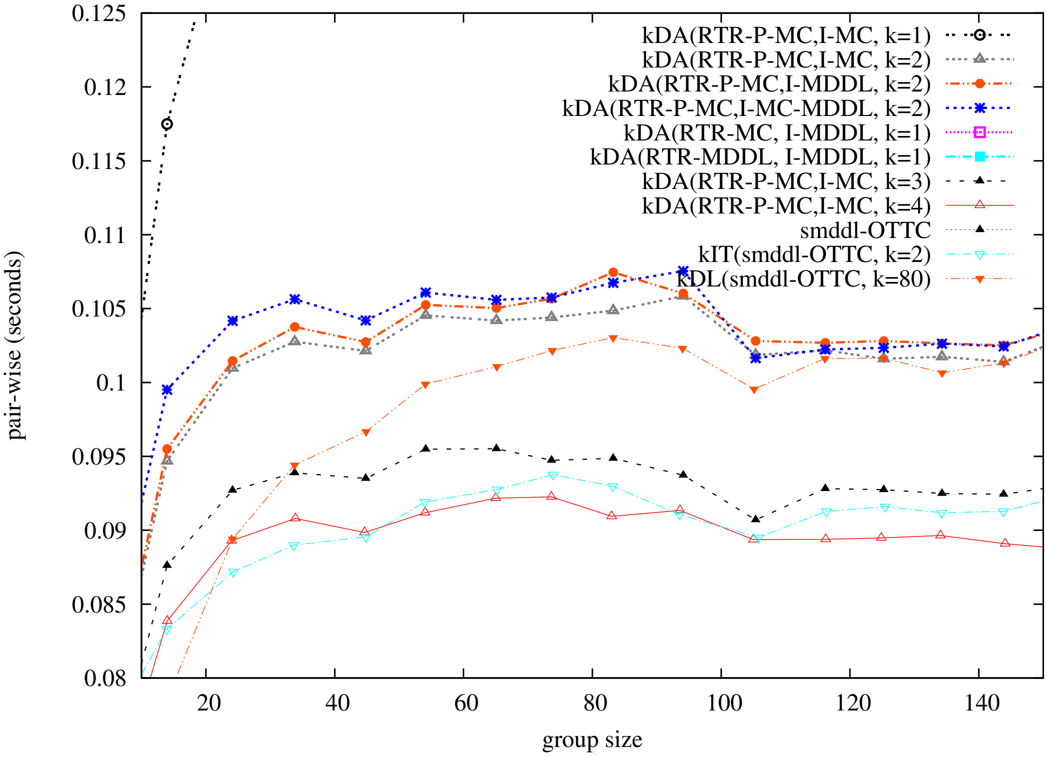


Figure 13.7: The average pair-wise latency (seconds) in the subgraphs.

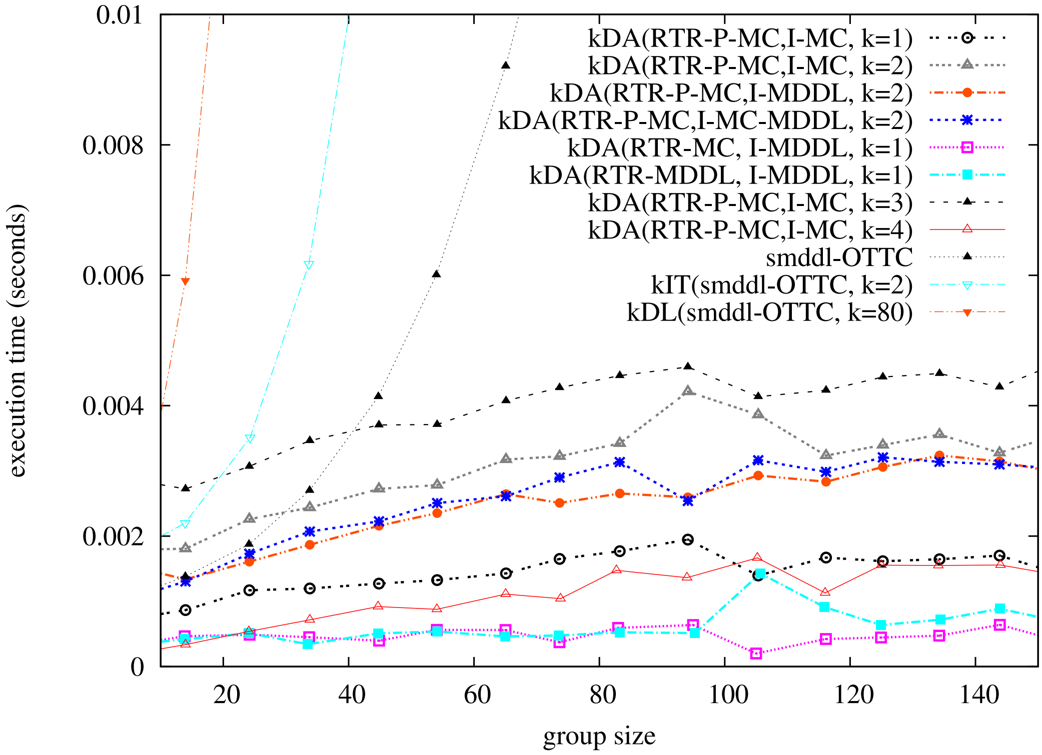


Figure 13.8: Execution time of the remove-strategy of the dynamic subgraph algorithms.

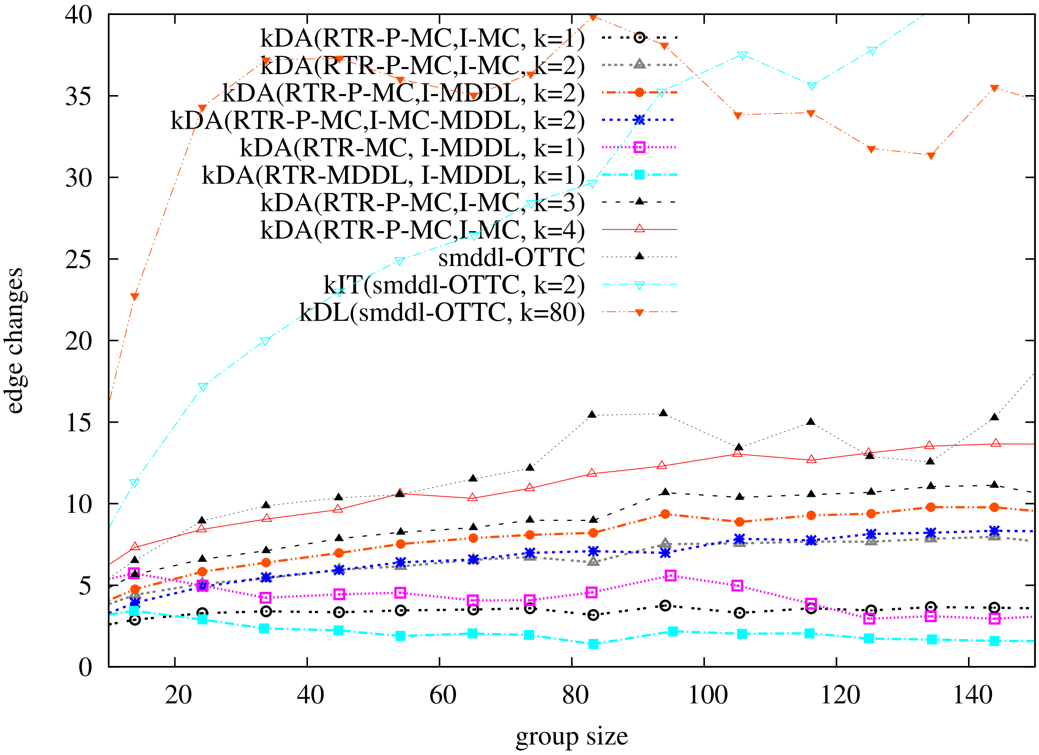


Figure 13.9: The average number of edge changes for the remove strategies (insert strategies always include k edges).

Figure 13.7 plots the average pair-wise latencies (seconds) achieved by the dynamic subgraph algorithms and smddl-OTTC. It is clear that the RTR-P-MC combinations with k $>$ 1 produce subgraphs with the lowest pair-wise latencies. This is to be expected when multiple paths to the destinations exist. The better dynamic tree algorithm combinations ($k = 1$) RTR-MC and RTR-MDDL, have an increasingly higher average pair-wise latency when the group size increases. The kDA(RTR-P-MC, I-MC, $k = 3, 4$) yields the lowest average pair-wise latency, which is constistently at 0.09 seconds.

Figure 13.8 plots the execution times of the remove strategy of the dynamic subgraph algorithms. The execution-time of the plotted insert strategies is always lower than the remove strategy, due to lower algorithm complexity. It is quite clear that that dynamic subgraph algorithms are extremely fast, compared to any other Steiner-subgraph algorithm.

The stability of a subgraph is measured in terms of how many edges change upon a membership change. Figure 13.9 plots the average number of edge changes upon a remove request. We observe that the remove strategy RTR-P-MC combined with any of the insert strategies has consistently a higher number of edge changes after a remove request has been executed. kDA(RTR-P-MC, I-MC, $k = 3$) breaks an average of 10 edge changes when the group size is > 90. However, the Steiner-tree algorithm smddl-OTTC is more unstable even though it is a tree, which has fewer edges to change. The dynamic-tree algorithms with remove strategies

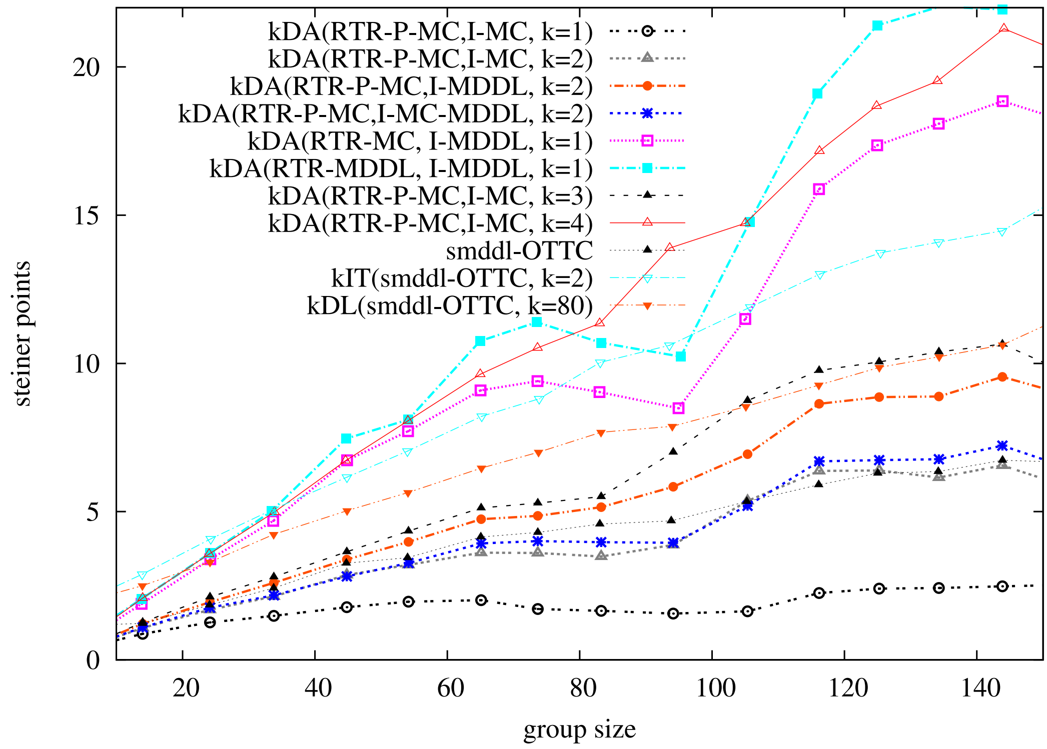


Figure 13.10: Comparison of selected dynamic mesh algorithms (insert,remove and k=1,2).

RTR-MC and RTR-MDDL, have an average number of edge changes of less than 3.

We plotted the total cost and non-member-node degradation in terms of the number of Steiner points in the previous (figure 13.4). Figures 13.11 and 13.10 give these results for the better choices of dynamic subgraph algorithms. We see that the kDA(RTR-P-MC, I-MC, $k = 3$) have comparatively few Steiner-points in the subgraphs compared to the dynamic tree algorithms ($k = 1$) that do not apply pruning. This is also reflected in the total cost, for which the pruning strategies manage to keep the total cost low even with a $k = 3$.

13.3.5 Results from varying degree-limits

The previous results used a static degree-limit of 10, the next evaluations compare the performance of the dynamic subgraph algorithms when a degree-limit of 5 is used.

Figure 13.12 plots the subgraph diameter (seconds) as built by selected algorithms. We observe that for the lower degree-limit of 5, the dynamic subgraph algorithm kDA(RTR-P-MC,I-MC,$k = 2$) outperforms the dynamic-tree algorithm kDA(RTR-MC,I-MDDL,$k = 1$). However, for a degree-limit 10 it is kDA(RTR-MC,I-MDDL,$k = 1$) that is the better of the two. We also see that kDA(RTR-P-MC,I-MC,$k = 3$) is unstable in its performance, which is because it struggles to reconnect the subgraphs due to the strict degree-limit combined with a subgraph with many links.

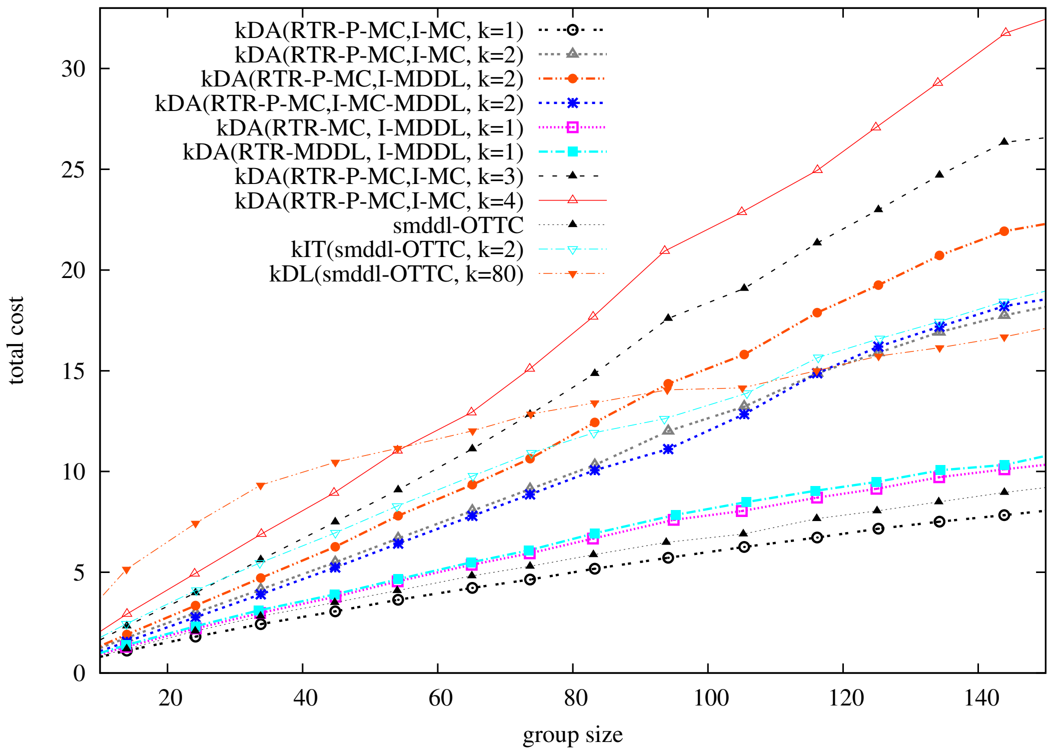


Figure 13.11: Comparison of selected dynamic mesh algorithms (insert,remove and k=1,2).

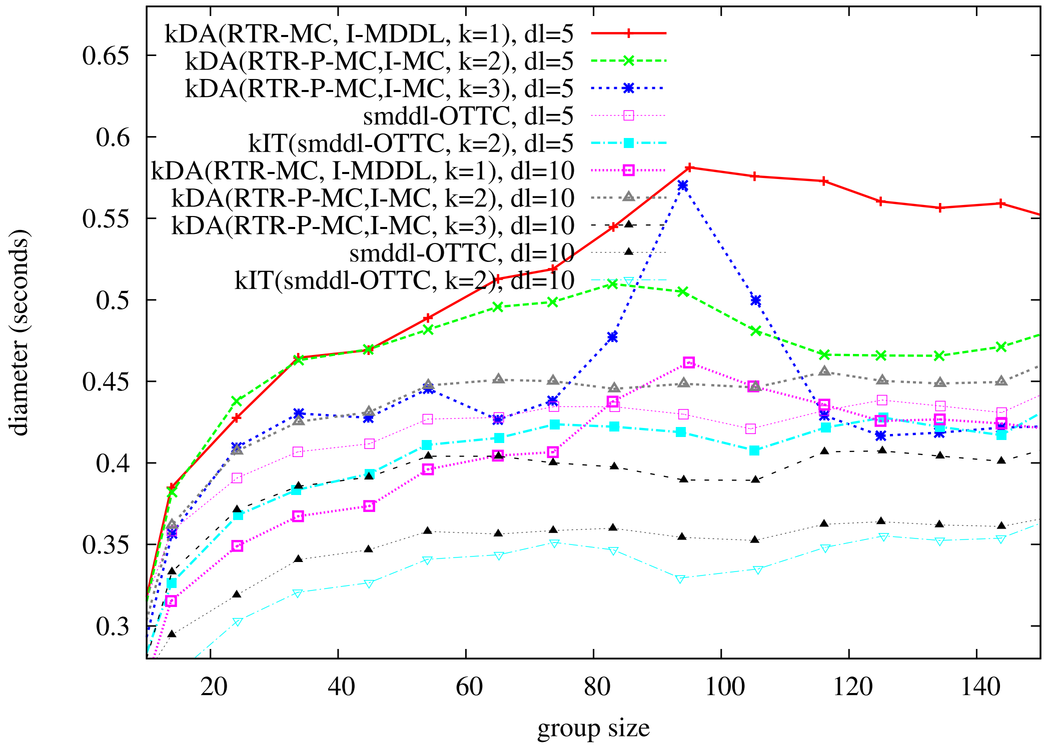


Figure 13.12: The diameter (seconds) of the better combinations of dynamic subgraph algorithms.

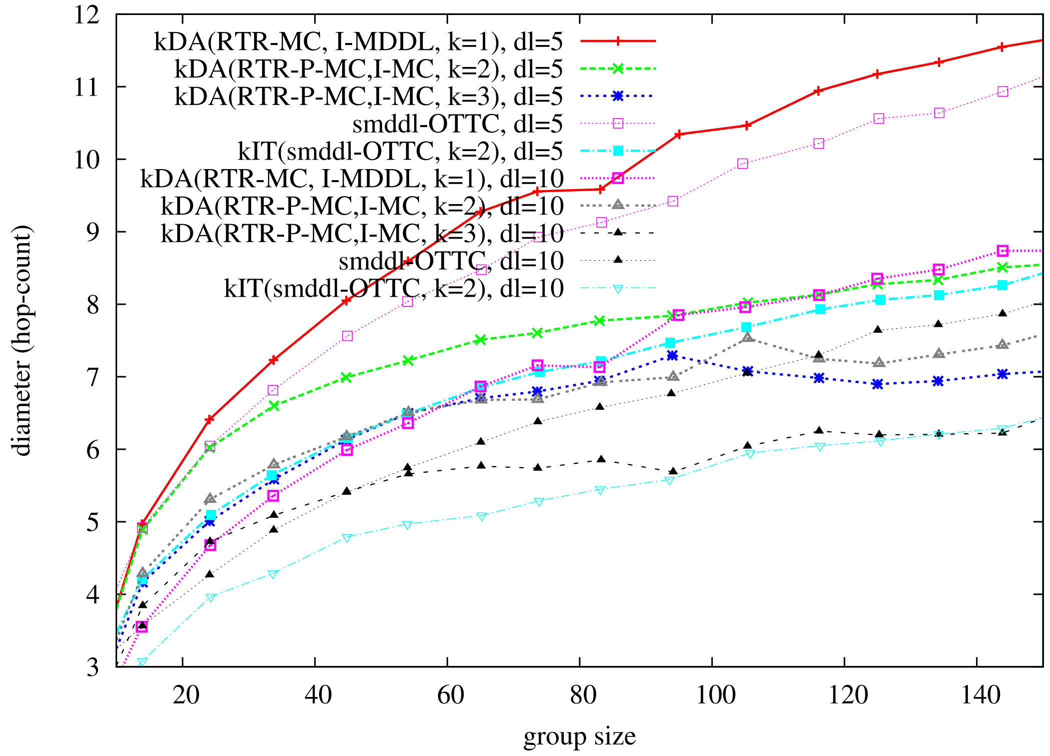


Figure 13.13: The diameter (hop-count) of the better combinations of dynamic subgraph algorithms.

The hop-diameter of the subgraphs is shown in figure 13.13. We observe that for kDA(RTR-P-MC, I-MC,$k = 2,3$) the hop-diameter is only slightly increased upon a lower degree-limit of 5. It also looks as though kDA(RTR-P-MC, I-MC, $k = 3$) with a degree-limit of 5 is forced to keep more and more Steiner-points in the subgraph in order to reconnect the subgraphs during remove reconfigurations.

These observations are confirmed by figure 13.14, which plots the non-member-node degradation in terms of the number of Steiner-points that are in the subgraphs. A lower degree-limit of 5 forces kDA(RTR-P-MC,I-MC,$k = 2,3$) to keep more Steiner-points in the subgraphs.

The total cost of the subgraphs is shown in figure 13.15. We see that a lower degree-limit of 5 has increased the total cost slightly among the dynamic subgraph algorithms. This is mainly due to the increased number of Steiner-points that are kept in the subgraphs.

From the previous results we observed that when the degree-limit was 5 the algorithm kDA(RTR-P-MC, I-MC,$k = 2$) consistently maintained a subgraph with a lower diameter than the better dynamic-tree algorithms (DA(RTR-MC,I-MDDL)). Therefore, in the context of our evaluated algorithms, we conclude that when the degree-limits are low there is a need for dynamic subgraph algorithms, rather than dynamic tree algorithms, to keep a consistently low diameter.

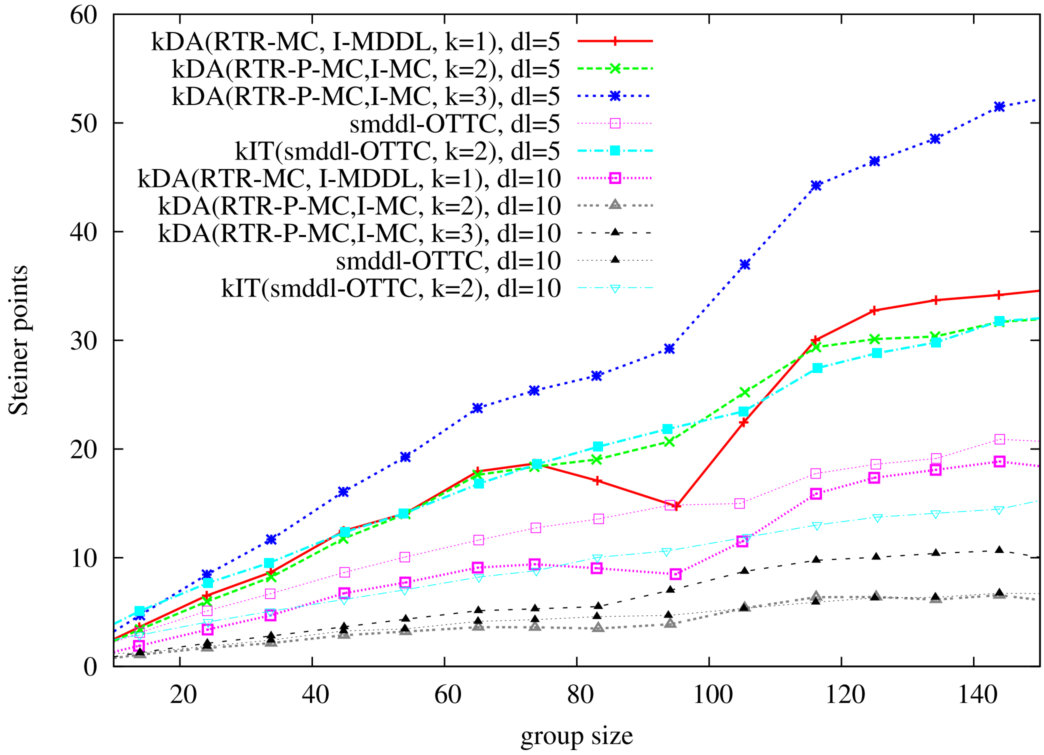


Figure 13.14: The number of Steiner-points in the subgraphs as produced by selected dynamic subgraph algorithms.

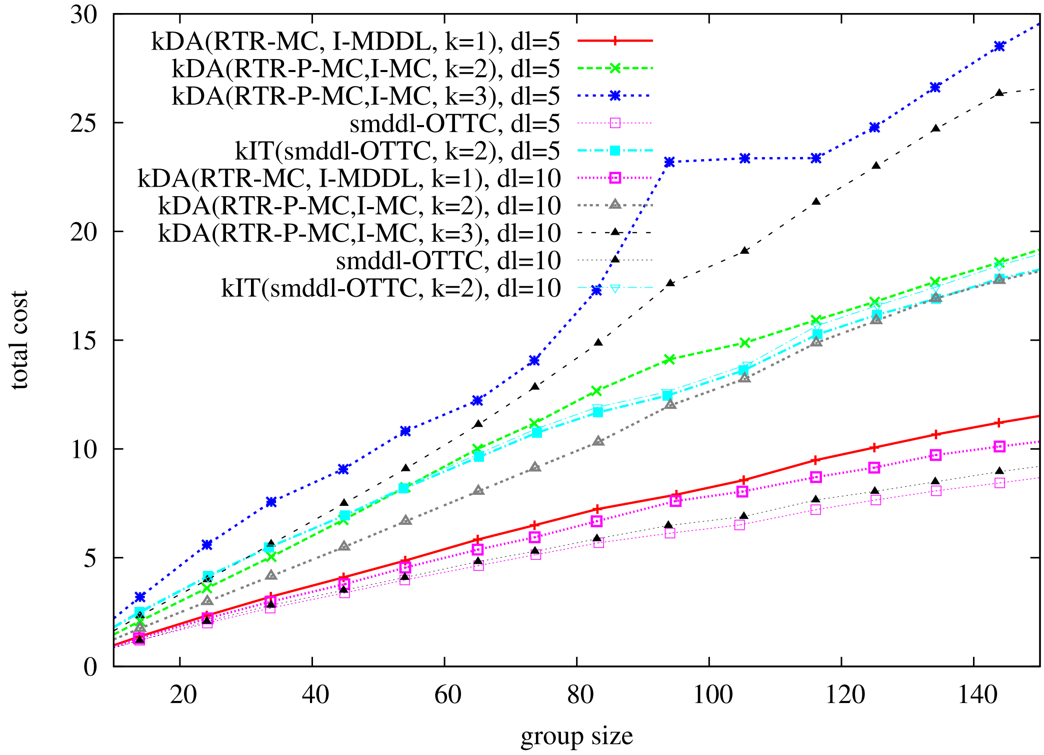


Figure 13.15: The total cost of the subgraphs as produced by selected dynamic subgraph algorithms.

Algorithm			Optimal k					Proposed k
Insert	Remove	k-range	*MN ratio*[1]	*Total cost*	*Diameter*	*Stability*	*Exec.time*	
I-MC(k)	RTR-P-MC	[1,4]	1	4	4	1	1	3
I-MDDL(k)	RTR-MDDL	[1,4]	1	2	2	1	1	1
I-MC-MDDL(k)	RTR-P-MC	[1,4]	1	4	4	1	1	3
Mesh	Tree	k-range						
kIT	smddl-OTTC	[1,4]	1	2	2	1	1	1
kIT	sdl-MST	[1,4]	1	3	3	2	1	2
kCIT	sdl-MST/sdl-RGH	[1,4]	1	3	3	2	1	2

[1]The member node (MN) ratio in a tree should be high, if not it is crowded with non-member nodes and degrade.

Table 13.3: Proposed k-configurations for a few selected algorithms.

Algorithm		Performance metrics					Performance index
Remove	Insert	*MN ratio*[1]	*Total cost*	*Diameter*	*Stability*	*Reconfig time*	
I-MC(2)	RTR-P-MC	+	−	+	+	+	4
I-MDDL(1)	RTR-MDDL	+	+	+	+	+	5
I-MC-MDDL(2)	RTR-P-MC	+	−	+	+	+	4
Mesh	Tree						
kIT(1)	smddl-OTTC	+	+	+	−	−	3
kIT(2)	sdl-MST	+	+	+	−	−	3
kCIT(2)	sdl-MST/sdl-RGH	+	+	+	−	−	3

[1]The member node (MN) ratio in a tree should be high, if not it is crowded with non-member nodes and degrade.

Table 13.4: Algorithm performance.

13.4 Summary of the main points

From the results we observed that dynamic subgraph algorithms must have mechanisms that reduces the number of Steiner-points in the subgraphs, that is, the non-member-node degradation. The degradation was massive when the remove strategies R-MC, R-MDDL, RTR-MC and RTR-MDDL were used in combination with the three insert strategies (I-MC, I-MDDL, and I-MC-MDDL). Instead, we observed that the remove strategies that applied pruning, RTR-P-MC and RTR-P-MDDL performed well for k = 2, and even kDA(RTR-P-MC, I-MC, k=3) yielded a reasonable total cost. We also observed that when the degree-limits were low there was a need for dynamic subgraph algorithms, rather than dynamic tree algorithms, to keep a consistently low diameter.

Table 13.3 illustrates our subjective opinions on how to configure some selected dynamic subgraph algorithms with the optimal k value. For each target metric an optimal k value is listed. These are averaged into the proposed k, which is what we consider the better way to configure the dynamic subgraph algorithm. The combinations that use RTR-P-MC have a proposed $k = 3$, while RTR-MDDL should use $k = 1$ and as such be a dynamic tree algorithm. We also list combinations of Steiner-subgraph algorithms for comparison.

Table 13.4 further gives our subjective opinions on how subgraph algorithms perform with regards to our target metrics. The *performance index* is the sum of the positives ($" + "$) of an algorithm, where it is possible to get a maximum performance index of 5. Even though the optimization goals low *total cost* and small *diameter* are contradictory, we consider the dynamic tree algorithm (RTR-MDDL,I-MDDL) to yield a reasonable total-cost to the network. In general, we observe that the main difference between dynamic algorihms and the Steiner-subgraph algorithms is the stability, which is enabled by local reconfigurations.

One issue that remains, is the fact that all of the evaluated dynamic-tree algorithms ($k = 1$) are unable to maintain a consistently low diameter throughout the group-size range ($0 - 170$). Therefore, we shall investigate how this issue may be addressed using total reconfigurations when the diameter increases beyond a pre-set bound.

Chapter 14

Overlay construction techniques: Combining overlay construction algorithms

Chapters 9 through 13 introduced and evaluated a wide range of overlay construction algorithms with the specific goal of identifying algorithms that construct low-latency overlay networks for distribution of events that are generated by a distributed interactive application. By doing that we addressed a goal of the thesis:

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

In the investigations we found a number of overlay construction algorithms that performed well. However, we did not find one single algorithm that performed well for all of our target metrics throughout the evaluated group size range ($0 - 170$). The goal of the next investigations is therefore to identify algorithms that are more desirable in some group ranges than others, and identify ways that their algorithm ideas can be combined.

The chapter recapitulates some comparisons of the better overlay construction tree and subgraph algorithms. Then, we introduce two reconfiguration algorithms. The first combines a dynamic tree algorithm with a Steiner-tree algorithm, and the second combines a dynamic subgraph algorithm with a Steiner subgraph algorithm. The combination is not algorithmic, but is rather to reconfigure an overlay with a Steiner-tree or Steiner-subgraph algorithm when the overlay has degraded due to the simplicity of the dynamic tree and subgraph algorithms. We show that a simple reconfiguration may dramatically increase the overall tree-quality. The tree-quality is measured in terms of our overlay target metrics; tree-diameter, stability in terms of the number of edge changes in a reconfiguration, and execution time of reconfiguration. The conclusion was that it is a good idea to combine dynamic tree and subgraph algorithms with Steiner-tree and subgraph algorithms to achieve all of these metrics.

The rest of the chapter is organized in the following manner. Section 14.1 gives a summary of the better tree algorithms we found in our previous investigations. Section 14.2 proposes a reconfiguration algorithm for tree structures, and evaluates it through group communication simulations. Section 14.3 gives a summary of the better subgraph algorithms we found in our previous investigations. Section 14.4 proposes a similar reconfiguration algorithm for subgraph structures, and evaluates it through group communication simulations. Finally, section 14.5 gives a brief summary of the main points.

14.1 Summary of the better tree algorithms

Chapter 9, 10 and 12 respectively investigated spanning-tree algorithms, Steiner-tree algorithms and dynamic tree algorithms.

14.1.1 The better tree algorithms

The investigations of the spanning-tree and Steiner-tree algorithms identified that the diameter-reducing spanning-tree algorithms yield trees with a lower diameter when Steiner points are included to the input graph. These were referred to as Steiner-tree spanning-heuristics (section 10.1.1).

Similar tendencies were found among the dynamic tree algorithms. It was the insert and remove strategies that aggressively left non-member-nodes in the trees as Steiner-points and also included additional Steiner-points that performed the best.

Algorithm	*Meaning*	*Optimization*	*Constraints*	*Complexity*	*Problem*
smddl-OTTC	Steiner minimum diameter degree-limited OTTC	diameter	degree	$O(n^3)$	48) Steiner-MDDL
sdl-OTTC	Steiner degree-limited OTTC	total cost	diam./deg.	$O(n^3)$	49) BDDLSMT
sdl-SPT	Steiner degree-limited Dijkstra's SPT	path cost	degree	$O(n^2)$	52) Steiner-MRDL
I-MDDL	Insert miminum diameter degree limited edge	diameter	degree	$O(n^2)$	Definition 64
I-MRDL	Insert miminum radius degree limited edge	radius	degree	$O(n^2)$	Definition 64
ITR-MDDL	Insert try reconfiguration and MDDL-edge	diameter	degree	$O(n^2)$	Definition 65
RK	Remove keep as non-member node	shortest path	degree	$O(1)$	Definition 68
RTR-MDDL	Remove try reconfiguration and MDDL-edge	diameter	degree	$O(n^2)$	Definition 68
RTR-MC	Remove try reconfiguration and MC-edge	total cost	degree	$O(n^2)$	Definition 68

Table 14.1: Steiner tree and dynamic-tree algorithms.

Table 14.1 contains the better performing Steiner-tree algorithms in our evaluations of chapter 10. In addition, it contains the better performing dynamic tree algorithms in the evaluations of chapter 12.

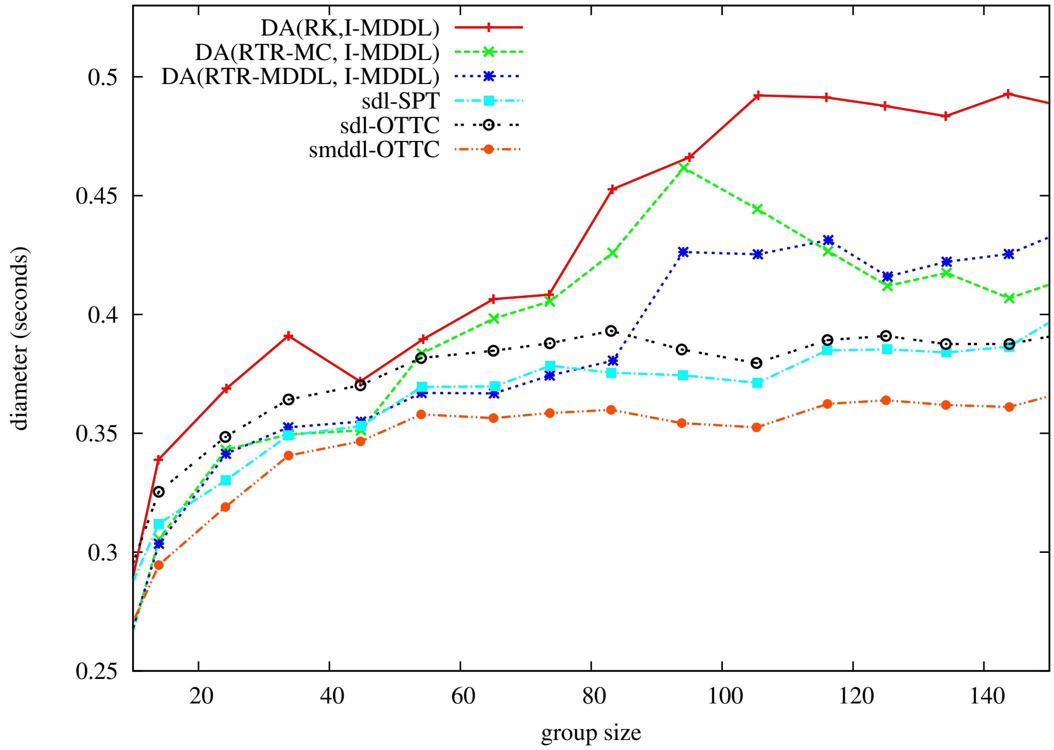


Figure 14.1: Diameter (seconds) of the better tree algorithms.

14.1.2 Simulation results from the better tree algorithms

Figure 14.1 plots the diameter (seconds) of trees as produced by the tree-algorithms in table 14.1. What is clear is that the overlays produced by dynamic-tree algorithms, degrade for larger group sizes, which is due to the simplicity of these algorithms. The dynamic-tree algorithms only do local reconfigurations and are not aware of the state of the tree, for example, the trees current diameter. The Steiner-tree algorithms, on the other hand, produce consistent results throughout the group range, with smddl-OTTC as the one that yields trees with the lowest diameter.

Figure 14.2 compares the stability of the tree algorithms, in terms of the number of edge changes that are made across a reconfiguration. A reconfiguration occurs whenever a node is inserted or removed from a tree. It is quite clear that the Steiner-tree algorithms are much more unstable than the dynamic tree algorithms. This comes as a result of the local reconfigurations of the dynamic tree algorithms, while the Steiner-tree algorithms recompute the entire tree for each time a node is inserted or removed.

Figure 14.3 compares the execution times of the tree reconfigurations. Here it is also very clear that the dynamic tree algorithms are much faster than the Steiner-tree algorithms. Which is also due to local reconfigurations of dynamic tree algorithms, and total reconstruction of the Steiner-tree algorithms.

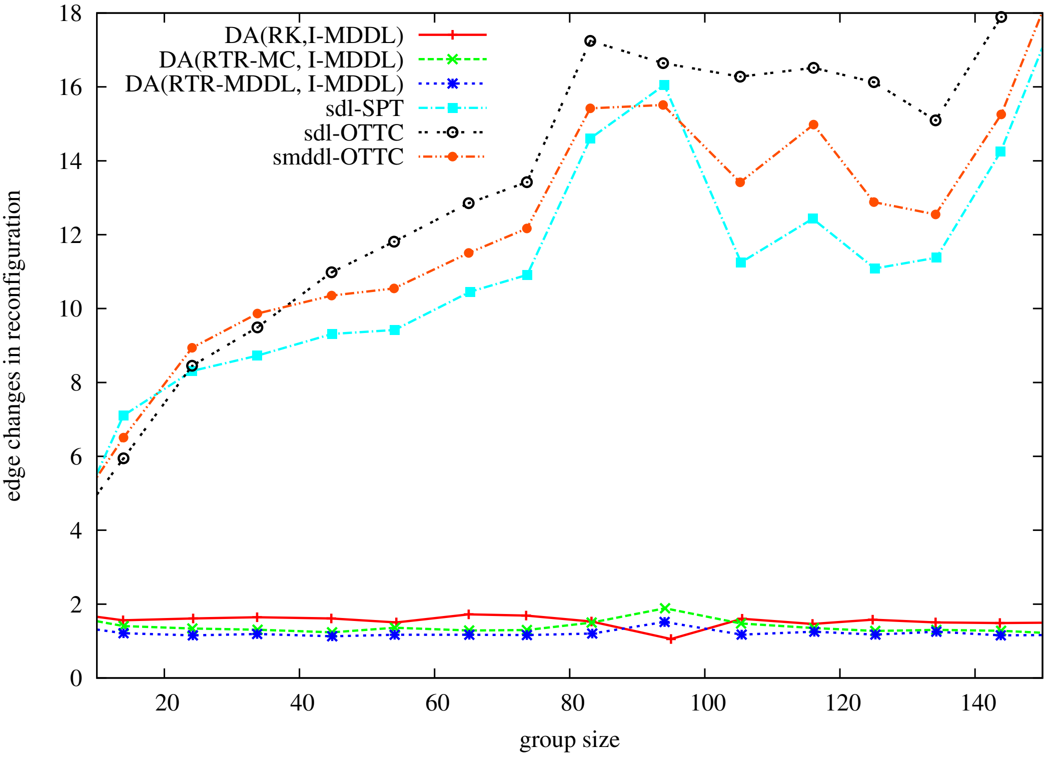


Figure 14.2: Edge changes in reconfigurations of the better tree algorithms.

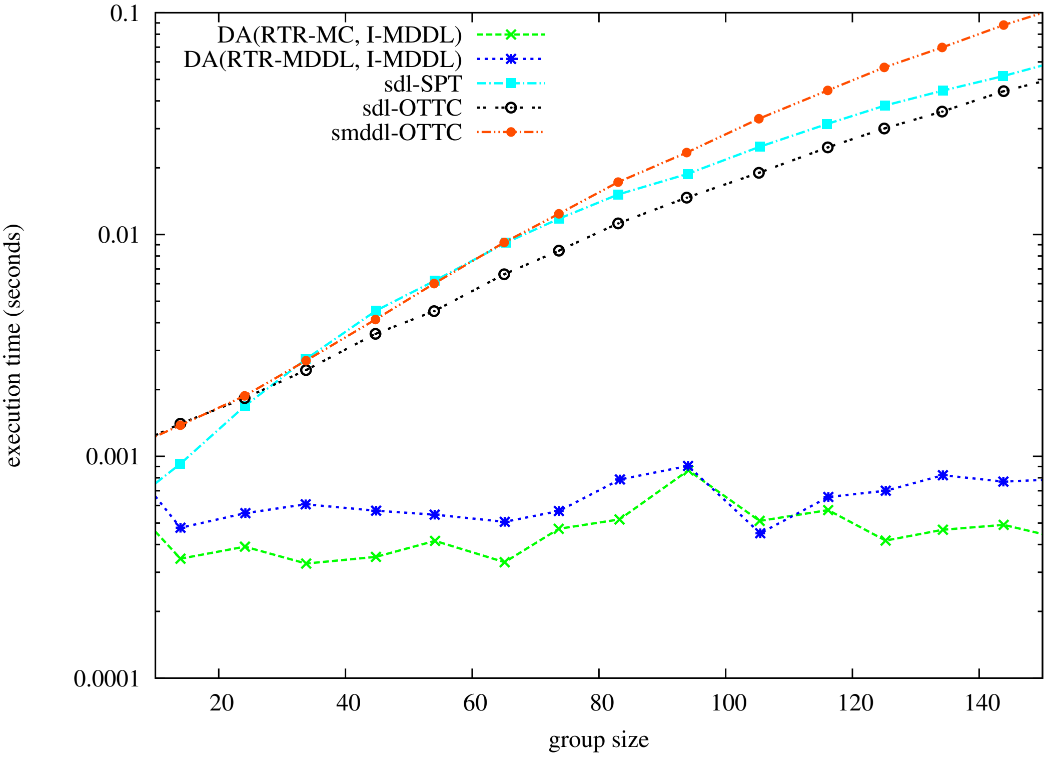


Figure 14.3: Execution time (seconds) in reconfigurations of the better tree algorithms.

Description	*Parameter*
Placement grid	100x100 milliseconds
Number of nodes in the network	1000
Degree limit	10
Super-nodes found by k-Center(k)	$k = 100$
Diameter bounds	$500, 450, 400$ and 350 milliseconds

Table 14.2: Experiment configuration.

From these observations it is clear that the dynamic tree algorithms exhibit very desirable properties when it comes to high stability and low reconfiguration time. This was measured in terms of low number of edge changes across reconfigurations and a low execution time of a tree reconfiguration. The dynamic tree algorithms also yield a low tree diameter for smaller group sizes. However, for larger group sizes the dynamic tree algorithms are not able to keep the tree-diameter low. The Steiner-tree algorithms, on the other hand, are slower and more unstable, but they yield a tree-diameter which is consistently low. This is because they reconstruct the entire tree for each time a node is inserted or removed.

14.2 Reconfiguration of trees

Based on the previous observations, we devised an algorithm for which the goal is to take advantage of the stability and quickness of the dynamic tree algorithms, and the consistency of the Steiner-tree algorithms, where the consistency is in terms of a low tree diameter.

14.2.1 Reconfigure tree algorithm

We propose a reconfiguration algorithm, in which a cost bound decides when a total reconfiguration of the tree should be initiated.

Reconfigure-tree Dynamic-Algorithm($\mathscr{R}_{\mathscr{D}}, \mathscr{I}_{\mathscr{D}}, \mathscr{A}_{\mathscr{T}}, B$) (RDA) takes as input a dynamic tree algorithm, in terms of, a remove strategy $\mathscr{R}_{\mathscr{D}}$ and an insert strategy $\mathscr{I}_{\mathscr{D}}$. Further, it takes as input a tree algorithm $\mathscr{A}_{\mathscr{T}}$ and a cost related bound B, where B may, for example, define a bounded diameter. RDA uses the insert and remove strategies to insert and remove nodes to and from a tree. For each time a tree is reconfigured, RDA checks whether the bound B is satisfied. If B is violated, RDA discards the current tree, and calls the tree algorithm to reconstruct the entire tree. If B is satisfied, RDA does nothing and continues to serve insert and remove requests (see algorithm 73).

Algorithm 73 RECONFIGURE-TREE-DYNAMIC-TREE-ALGORITHM

In: $G = (V, E, c)$, a tree $T_Z = (V_T, E_T)$, a set of members $Z \subseteq V_T$, a request ρ, one insert strategy $\mathscr{I}_{\mathscr{D}}$, one remove strategy $\mathscr{R}_{\mathscr{D}}\}$, one tree algorithm $\mathscr{A}_{\mathscr{T}}$, and an upper cost bound B.

Out: Updated tree $T_Z = (V_T, E_T)$

```
if $\rho = \{remove, m\}$ then
    $\mathscr{R}_{\mathscr{D}}(G, T_Z, m)$
else if $\rho = \{insert, m\}$ then
    $\mathscr{I}_{\mathscr{D}}(G, T_Z, m)$
end if
$B_{T_Z}$ = getCurrentCostOfTree($T_Z$)
if $B_{T_Z} > B$ then
    $G_g$ = createGroupGraph($G$, $Z$)
    $T_Z$ = $\mathscr{A}_{\mathscr{T}}(G_g)$
end if
```

14.2.2 Experiment configurations

In the experiments, we use equation 8.1 (section 8.5.4) to approximate the number k of Steiner-points that are needed to ensure that the degree-limited dynamic tree algorithms are able to reconnect a tree. The k Steiner points are chosen by k-Median(k) from 100 Steiner-points (super-nodes) that are identified at the beginning among the 1000 nodes in the overlay network. These well-placed super-nodes (chapter 8) are found by the multiple core-node selection algorithm k-Center($k = 100$).

We ran the group communication simulations using the RDA algorithm. RDA was given as input RTR-MDDL, I-MDDL, smddl-OTTC and a diameter bound B. We varied the diameter bound B to see the effect it had on our target metrices: diameter, stability in reconfiguration and execution time of reconfiguration. Table 14.2 summarizes the test setup.

14.2.3 Reconfigure tree results

Figure 14.4 compares the diameter achieved by the RDA algorithm given different diameter bounds. The smddl-OTTC and DA(RTR-MC,I-MDDL) are plotted as references. It is clear that the tree-reconfiguration in RDA prevents the tree degradation that occurs for the dynamic tree algorithm. The RDA algorithms now perform very well, even compared to the close-to-optimal smddl-OTTC. Figure 14.5 plots the CDF for the diameter, and we observe a similar pattern.

Figure 14.6 is a CDF of the edge-changes in the tree-reconfigurations. The smddl-OTTC is clearly the most unstable, for which 10 % of the tree-reconfigurations have more than 15 edge changes. We observe that the RDA algorithm that use a diameter bound $B = 0.400$ has a stability which is almost identical to that of DA(RTR-MC,I-MDDL). The more tight bounds of 0.350 and 0.300 decreases the stability somewhat, and clearly illustrates that for RDA the diameter bound has a major influence on the stability.

Figure 14.7 is a CDF of the exexution times of the tree-reconfigurations. As expected, the DA(RTR-MC,I-MDDL) is the fastest, and smddl-OTTC is clearly the slowest. The RDA

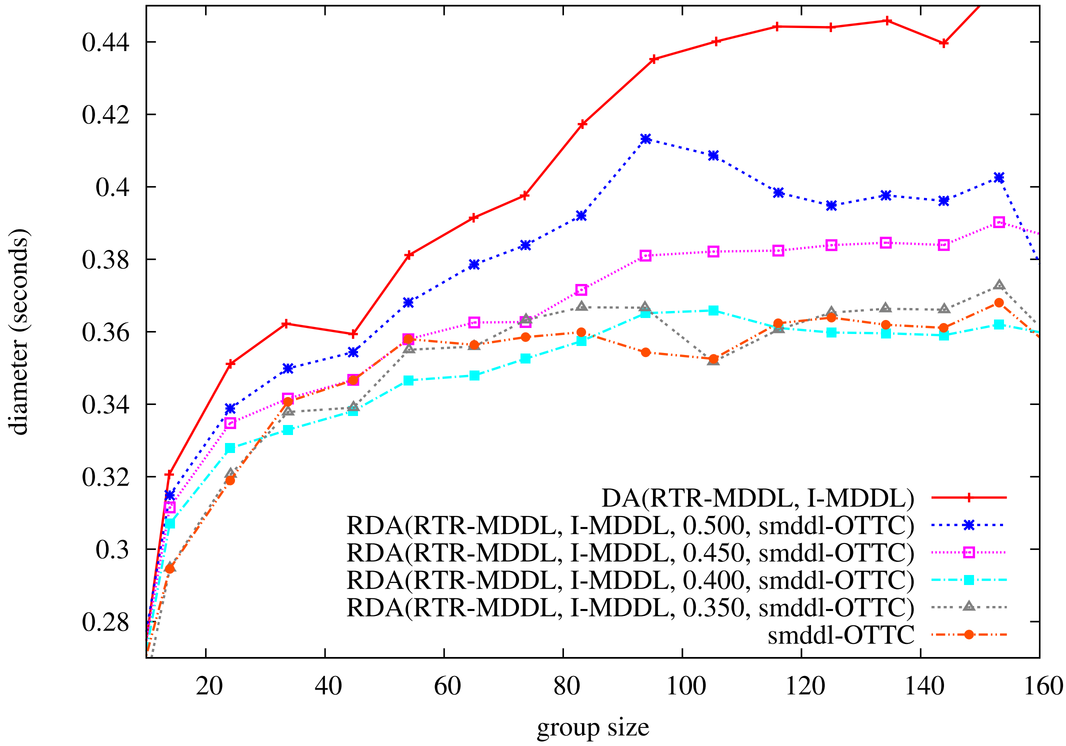


Figure 14.4: Diameter (seconds) of trees using reconfigurations.

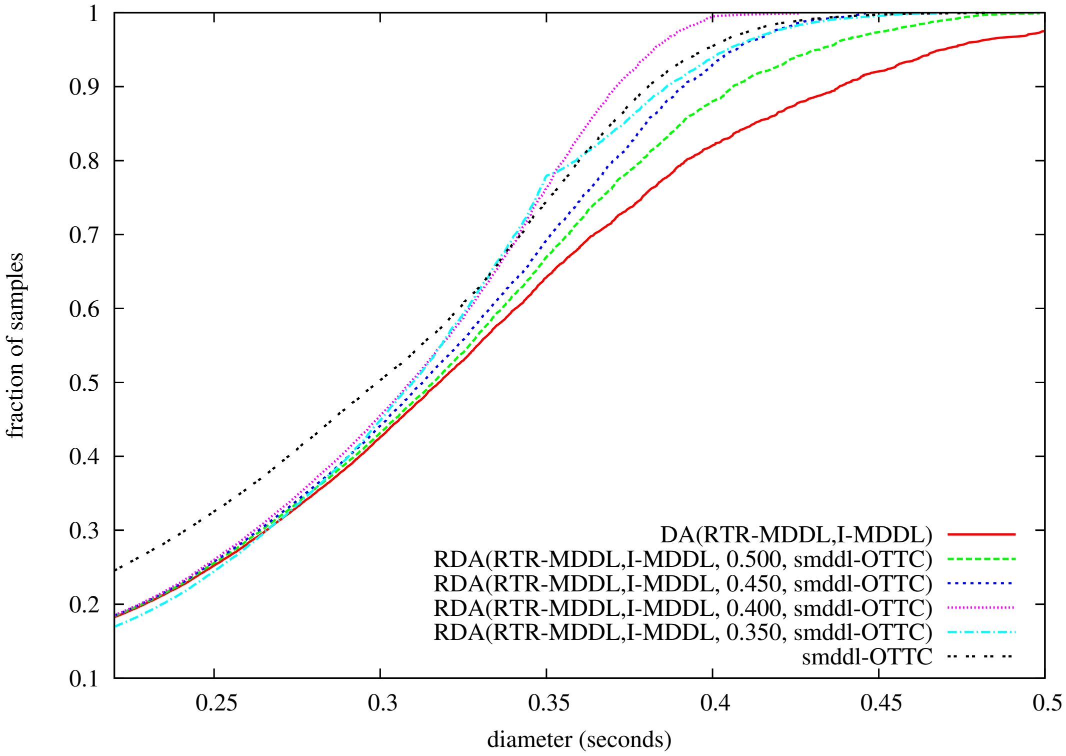


Figure 14.5: CDF of diameter (seconds) of the trees.

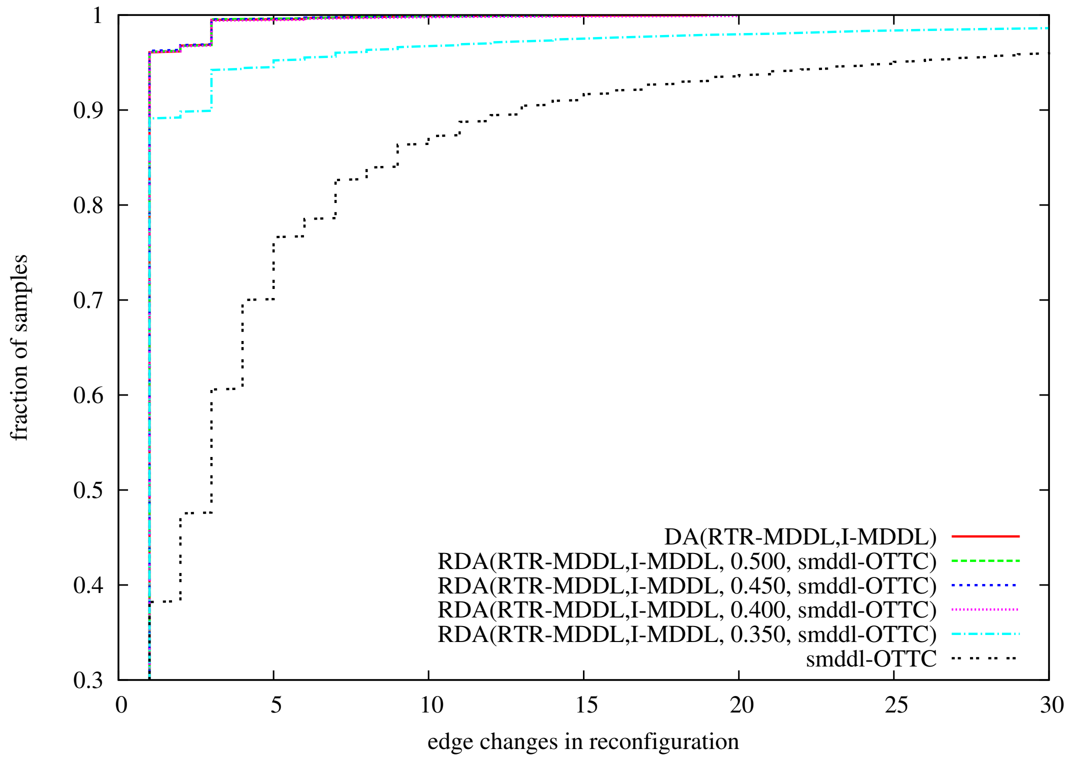


Figure 14.6: CDF of stability, in terms of edge-changes in the reconfigurations.

configuration with a diameter bound $B = 0.400$ is almost identical to DA(RTR-MC,I-MDDL), while the tighter bounds increases the tree-reconfiguration times more.

In summary, we observed that RDA(RTR-MC,I-MDDL,smddl-OTTC, 0.400) yield trees with a consistently low diameter, which is only fractions higher than the close-to-optimal smddl-OTTC. In addition, the RDA configuration kept the stability and quickness of a dynamic tree algorithm. However, we also observed that the initial diameter bound influences the performance of RDA. A tight bound reduces the stability and quickness, while a loose bound removes the advantage of a consistent low diameter.

14.3 Summary of the better subgraph algorithms

Chapter 11 and 13 respectively investigated spanning-subgraph and Steiner-subgraph algorithms, and dynamic subgraph algorithms.

14.3.1 The better subgraph algorithms

The investigations of the spanning-subgraph and Steiner-subgraph algorithms identified that the diameter-reducing subgraph algorithms were able to produce subgraphs that yielded a lower diameter than their respective tree-algorithm counterparts could. However, the reduction in

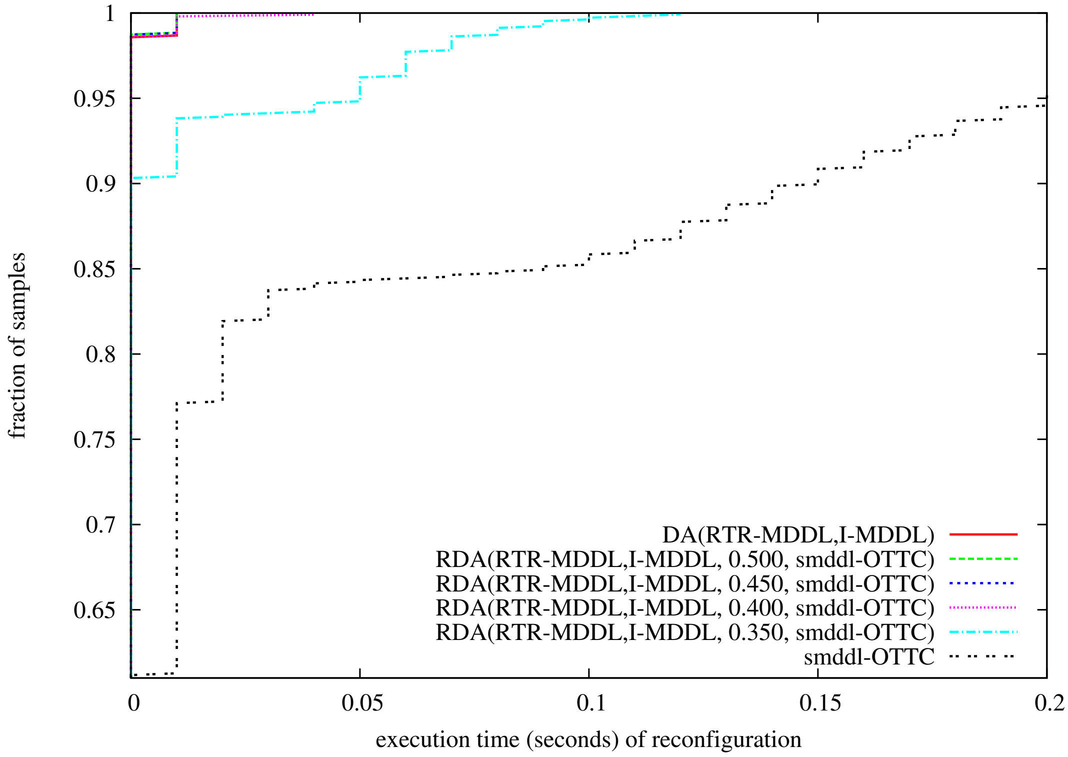


Figure 14.7: CDF of the execution time the reconfigurations.

the diameter was only slight, with the drawback of added network cost. Among the subgraph algorithms, it was the Steiner-subgraph algorithms that overall constructed subgraphs with the lowest diameters.

Algorithm	*Meaning*	*Optimization*	*Input to $\mathscr{A}_\mathscr{M}$*	*Complexity*	*Problem*
kIT	k-Iterative Tree constr.	$\mathscr{A}_\mathscr{T}$ goal	G, k, $\mathscr{A}_\mathscr{T}$	$O(k * O(\mathscr{A}_\mathscr{T}))$	70) k-c subgraph
kDL	k-Diameter Links	diameter	G, k, $\mathscr{A}_\mathscr{T}$	$O(n^3 + O(\mathscr{A}_\mathscr{T}))$	78) k-c MDDL
kDA	k-Dynamic-subgraph alg.	$\mathscr{I}_\mathscr{D}, \mathscr{R}_\mathscr{D}$ goal	G, k, $\mathscr{I}_\mathscr{D}, \mathscr{R}_\mathscr{D}$	$O(\mathscr{I}_\mathscr{D})$, $O(\mathscr{R}_\mathscr{D})$	90) Dynamic Subgraph
kIT	k-Iterative Tree constr.	diameter	G, 2, smddl-OTTC	$O(2 * n^3)$	70) k-c subgraph
kDL	k-Diameter Links	diameter	G, 80, smddl-OTTC	$O(n^3 + n^3)$	78) k-c MDDL
kDA	k-Dynamic-subgraph alg.	minimum-cost	G, 3, I-MC(3), RTR-P-MC	$O(n)$, $O(n^3)$	90) Dynamic Subgraph

Table 14.3: Subgraph construction algorithms

The details of the subgraph algorithms are listed in table 14.3. For the kIT subgraph algorithm, we found that the better input parameters were the Steiner-heuristic smddl-OTTC and $k = 2$. Moreover, for the kDL subgraph algorithm it was also smddl-OTTC, but with a $k = 80$. For the kDA algorithm, we found the remove strategy RTR-P-MC, insert strategy I-MC and $k = 3$ to be the best configuration. In addition to these configurations, we found that a degree-limit of 10, enabled the algorithms to find trees with a lower diameter.

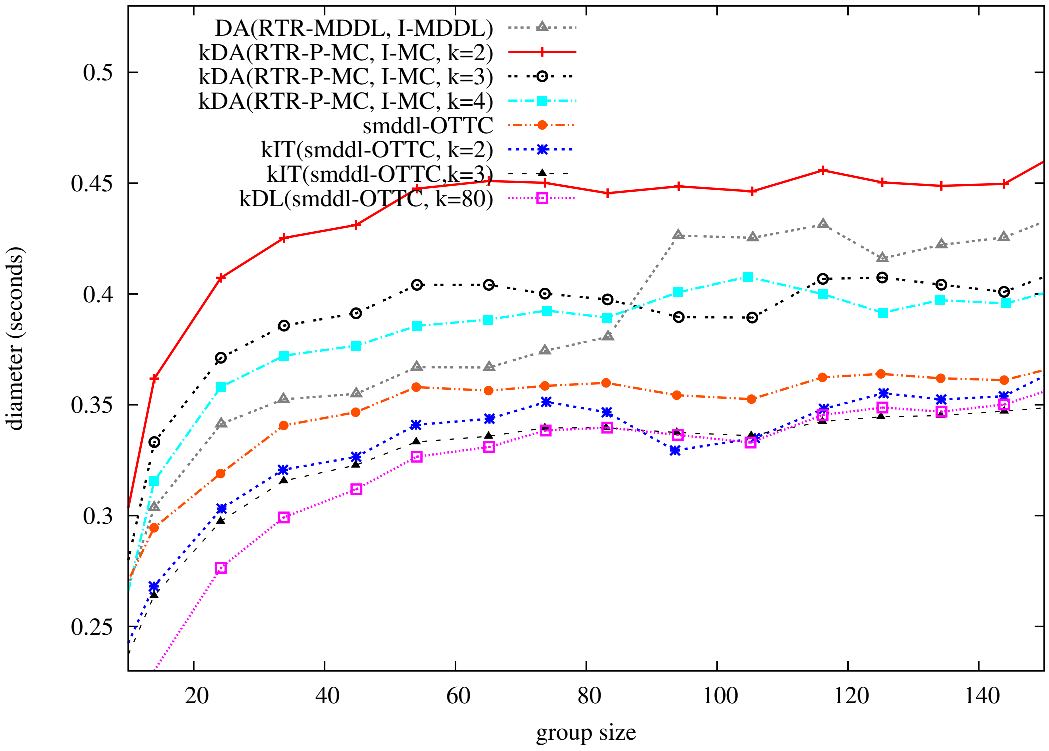


Figure 14.8: Diameter (seconds) of the better mesh algorithms.

14.3.2 Simulation results from the better subgraph algorithms

Figure 14.8 compares the subgraph-diameter (seconds) achieved by the better subgraph algorithms found in our previous investigations. The main observation is that the diameter is not reduced very much, when going from a tree to subgraph (cyclic graph). The best performing dynamic mesh algorithm DA(RTR-P-MC,I-MC,$k = 3/4$) yield a consistently low diameter throughout the group range. However, it does not outperform the best performing dynamic tree algorithm DA(RTR-MDDL,I-MDDL) for smaller group sizes (< 80).

Figure 14.9 compares the average pair-wise latencies in the subgraphs constructed by the subgraph algorithms. It is expected that the subgraphs from the subgraph algorithms yield lower pair-wise latencies than the trees constructed by the tree algorithms. Subgraphs have more routes between the nodes, while a tree has only single routes. The kDA(RTR-P-MC,I-MC,$k = 2, 3, 4$) yield a consistently low pair-wise latency throughout the group-range, while the DA(RTR-MDDL,I-MDDL) still yield overlays that degrade for larger group-sizes (> 80) and is fairly high for smaller group-sizes.

Figure 14.10 compares the total cost of the subgraphs. We observe that the kDA(RTR-P-MC,I-MC,$k = 2, 3, 4$) yield subgraphs with a total cost which is competetive to that of kIT(smddl-OTTC,$k = 2$) and kDL(smddl-OTTC,$k = 80$). The tree algorithms DA(RTR-MDDL,I-MDDL) and smddl-OTTC yield, as expected, much cheaper trees. This is only nat-

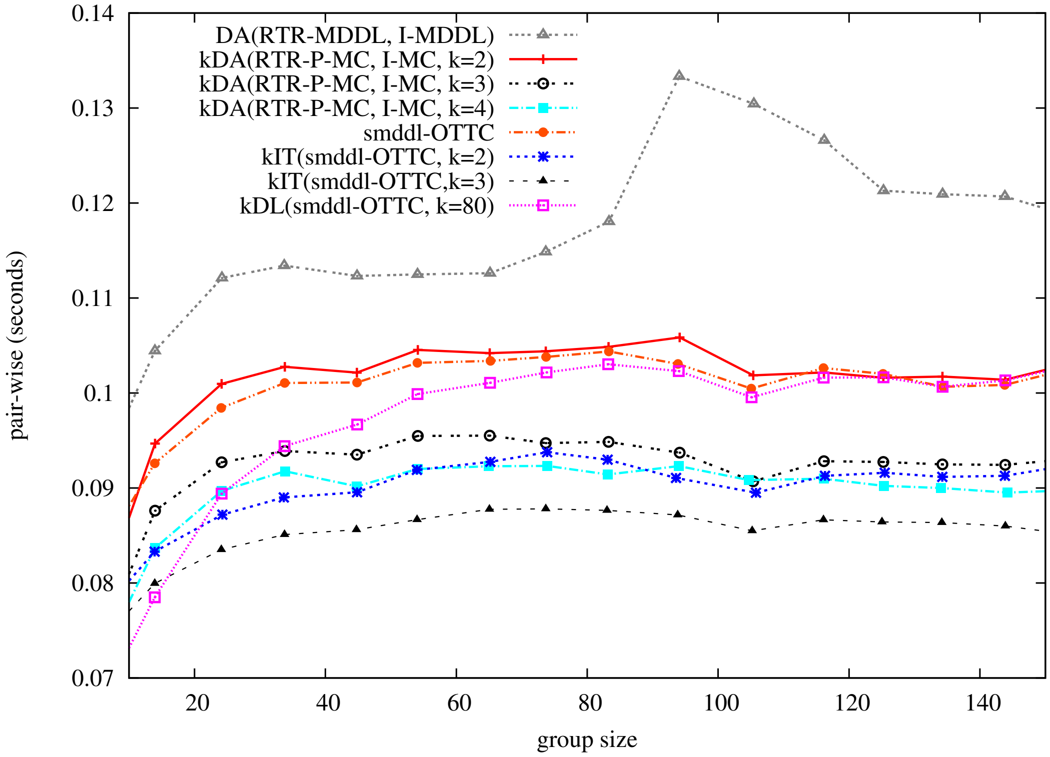


Figure 14.9: Pair-wise (seconds) of the better mesh algorithms.

ural, because trees have the minimum amount of edges possible for a group of nodes to stay connected.

Figure 14.11 compares the stability of the subgraph algorithms, in terms of the number of edge-changes across sugraph reconfigurations. As expected, the Steiner subgraph algorithms have many edge changes, the dynamic subgraph algorithms have much fewer edge-changes, and dynamic tree algorithms have the least. The main reasons are that subgraphs have more edges than a tree, and it is only natural that the subgraph algorithms have more edge-changes. The advantage of subgraphs is that there are potentially more than one route to each destination. Therefore, the reconfiguration instability may not influence the overall delivery rate as much as in a tree, for which there exist only one route to each destination.

Figure 14.12 compares the execution times of subgraph reconfigurations. It is clear that the Steiner subgraph algorithms are very much slower than the dynamic subgraph algorithms. The dynamic subgraph algorithms DA(RTR-P-MC,I-MC,$k = 2, 3, 4$) are still extremely fast, and executes a subgraph reconfiguration in less than 4 milliseconds.

From these results and observations we see that the dynamic subgraph algorithm kDA(RTR-P-MC,$k = 2, 3, 4$) is able to maintain a consistently low diameter of the subgraphs. However, DA(RTR-MDDL,I-MDDL) yield a lower subgraph diameter for smaller group sizes (< 80). kDA(RTR-P-MC,$k = 2, 3, 4$) also achieved very low pair-wise latencies in the subgraphs, even compared to the Steiner subgraph algorithm kIT(smddl-OTTC,$k = 2$). Lower pair-wise laten-

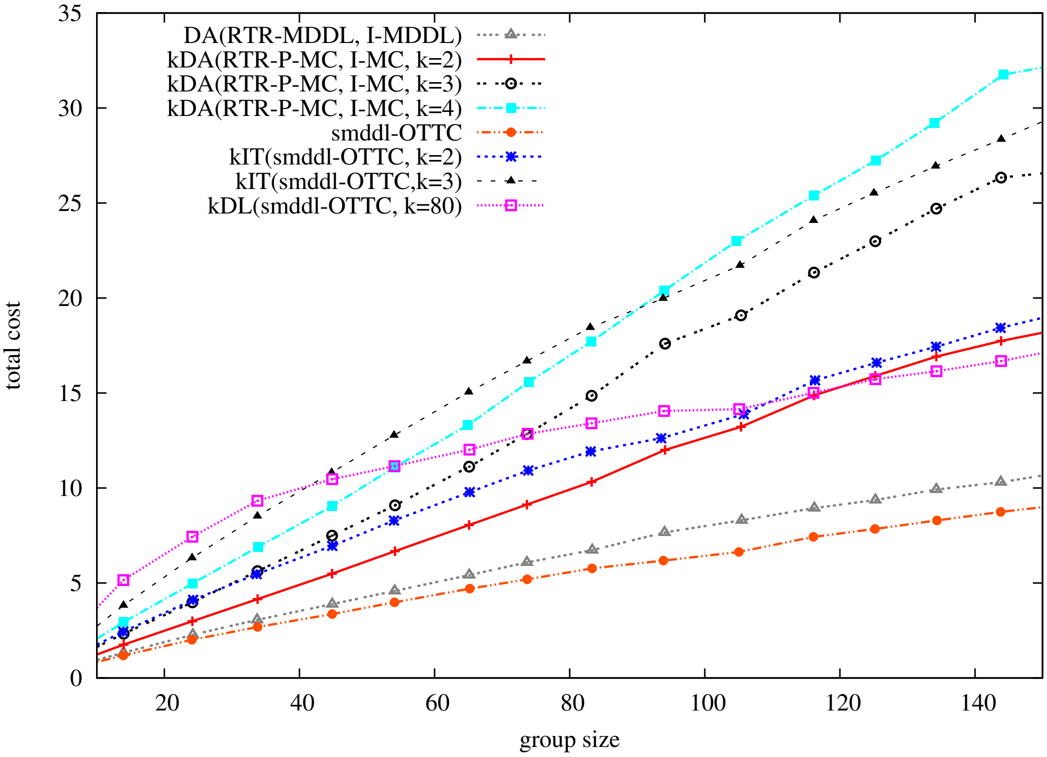


Figure 14.10: Edge changes in reconfigurations of the better mesh algorithms.

cies is the main advantage of a subgraph over a tree. Finally, kDA(RTR-P-MC,$k = 2, 3, 4$) was very stable and fast compared to the Steiner subgraph algorithms.

14.4 Reconfiguration of subgraphs

Similar to the tree-algorithm reconfiguration, we devised an algorithm for which the goal is to take advantage of the stability and quickness of the dynamic subgraph algorithms, and the low-diameter produced by the Steiner-subgraph algorithms

14.4.1 Reconfigure subgraph algorithm

The reconfigure subgraph algorithm is almost identical to the reconfigure tree algorithm (RDA) from section 14.2, expect there is a configurable integer k that describes the connectivity of the subgraph.

k-Reconfigure-subgraph Dynamic-Algorithm($\mathscr{R}_{\mathscr{D}}, \mathscr{I}_{\mathscr{D}}, \mathscr{A}_{\mathscr{T}}, k, B$) (kRDA) takes as input a dynamic subgraph algorithm, in terms of a remove strategy $\mathscr{R}_{\mathscr{D}}$ and an insert strategy $\mathscr{I}_{\mathscr{D}}$. Further, it takes as input a tree algorithm $\mathscr{A}_{\mathscr{T}}$, a configurable integer $k \geq 1$, and a cost related bound B, where B may, for example, define a bounded diameter. kRDA uses the insert and remove strate-

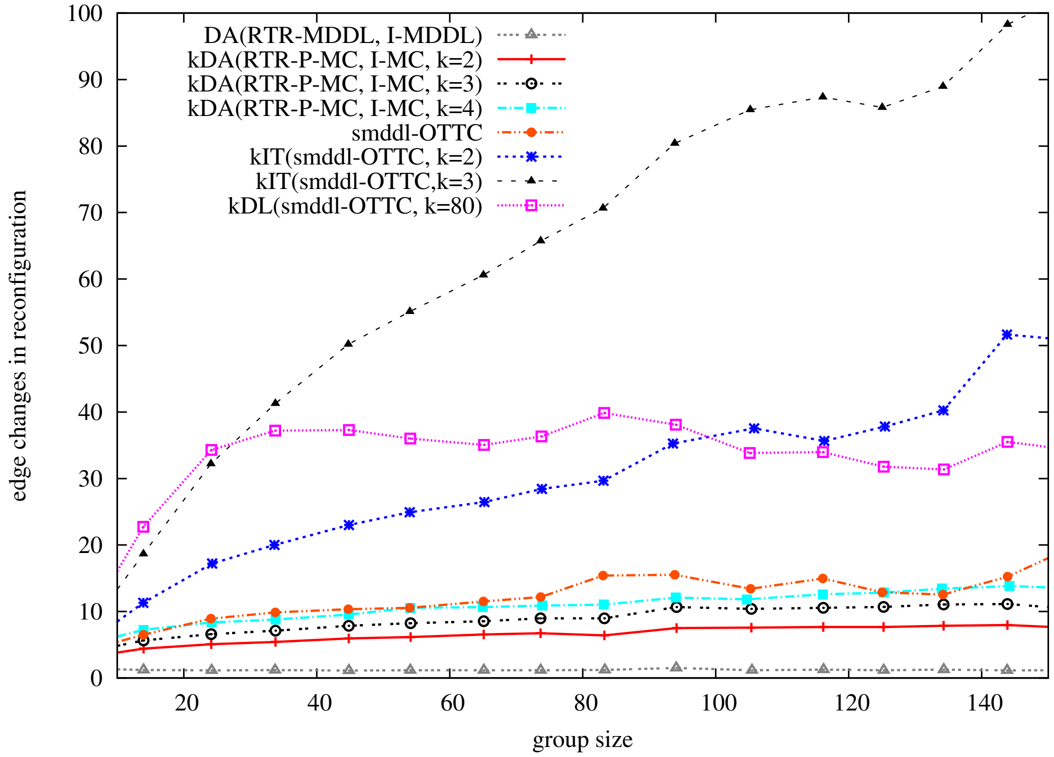


Figure 14.11: Edge changes in reconfigurations of the better mesh algorithms.

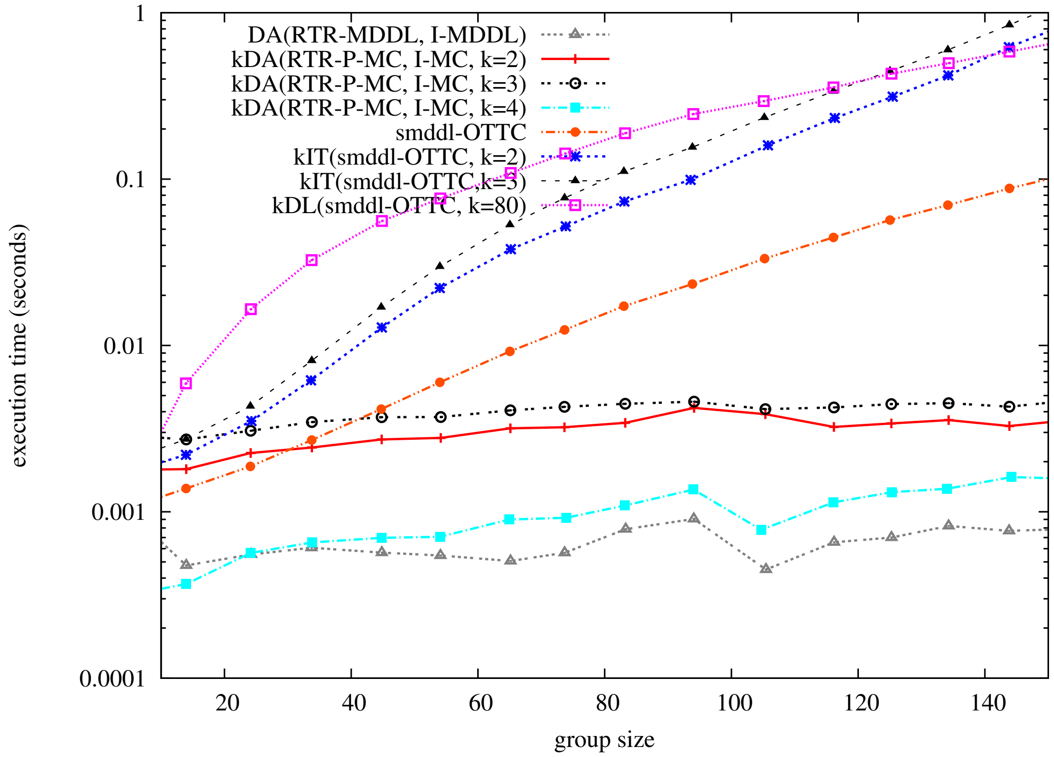


Figure 14.12: Execution time (seconds) in reconfigurations of the better mesh algorithms.

Description	*Parameter*
Placement grid	$100x100$ milliseconds
Number of nodes in the network	1000
Degree limit	10
Super-nodes found by k-Center(k)	$k = 100$
Diameter bounds	$500, 450, 400$ and 350 milliseconds

Table 14.4: Experiment configuration.

gies to insert and remove nodes to and from a subgraph. The insert strategy includes a node m, and adds k links from it to the subgraph. The remove strategy removes m and reconnects the subgraph as if it was a tree. For each time a subgraph is reconfigured, kRDA checks whether the bound B is satisfied. If B is violated, kRDA discards the current subgraph, and calls the kIterativeTree (kIT) subgraph algorithm using the tree algorithm $\mathscr{A}_{\mathscr{T}}$ and k as input. If B is satisfied, kRDA does nothing and continues to serve insert and remove requests (see algorithm 74).

Algorithm 74 k-RECONFIGURE-SUBGRAPH-DYNAMIC-SUBGRAPH-ALGORITHM

In: $G = (V, E, c)$, a subgraph $M_Z = (V_M, E_M)$, a set of members $Z \subseteq V_M$, a request ρ, one insert strategy $\mathscr{I}_{\mathscr{D}}$, one remove strategy $\mathscr{R}_{\mathscr{D}}\}$, one tree algorithm $\mathscr{A}_{\mathscr{T}}$, an integer $k \geq 1$, and an upper cost bound B.

Out: Updated subgraph $M_Z = (V_M, E_M)$

```
if $\rho = \{remove, m\}$ then
    $\mathscr{R}_{\mathscr{D}}(G, M_Z, m, k = 1)$
else if $\rho = \{insert, m\}$ then
    $\mathscr{I}_{\mathscr{D}}(G, M_Z, m, k)$
end if
$B_{M_Z}$ = getCurrentCostOfSubgraph($M_Z$)
if $B_{M_Z} > B$ then
    $G_g$ = createGroupGraph($G$, $Z$)
    $M_Z$ = kIT($\mathscr{A}_{\mathscr{T}}$, $k$, $G_g$)
end if
```

14.4.2 Experiment configurations

We ran group communication simulations using the kRDA algorithm. kRDA was given as input RTR-P-MC, I-MC, smddl-OTTC, $k = 3$, and a diameter bound B. We varied the diameter bound to see the effect it had on our target metrices: diameter, stability in reconfiguration and execution time of reconfiguration. The remaining experiment configurations are listed in table 14.4, and section 14.2.2 gives further details to the experiments.

14.4.3 Reconfigure subgraph results

Figure 14.13 plots the diameter (seconds) of the subgraphs. We observe for the kRDA combinations that the subgraph diameter is consistently reduced step by step as the diameter bound B is tightened. kRDA with $B = 0.350$ seconds is actually performing as good as kIT(smddl-

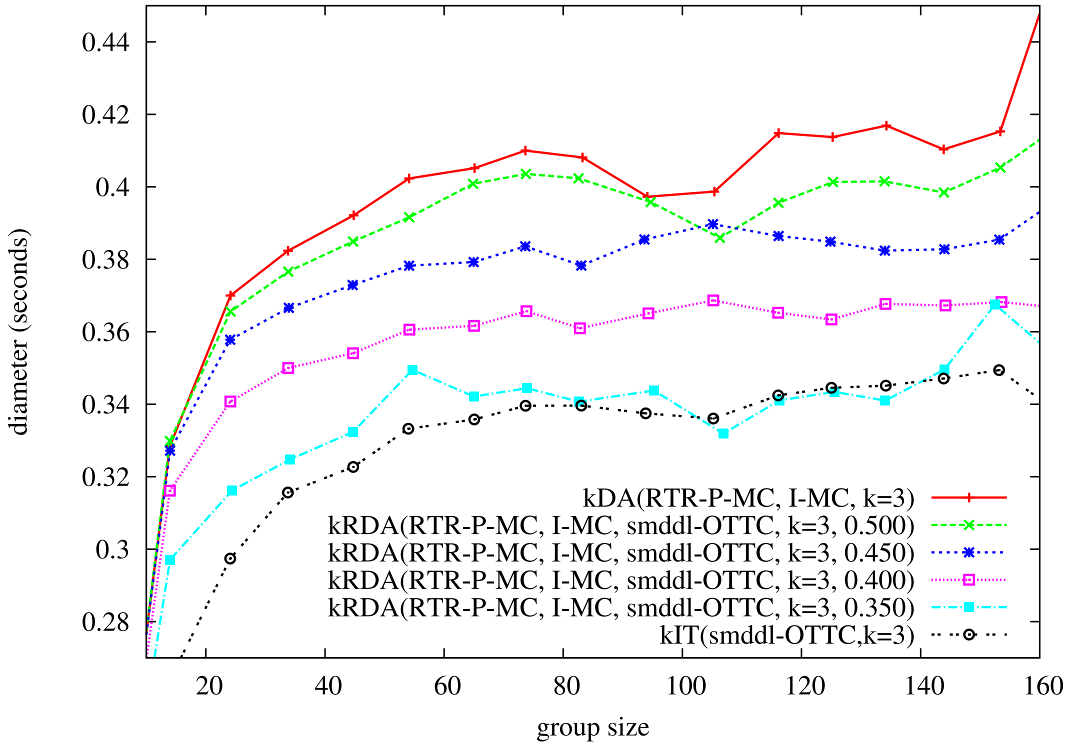


Figure 14.13: Diameter (seconds) of subgraphs.

OTTC,$k = 3$), throughout the group-range. This tells that the local reconfigurations of the dynamic subgraph algorithms is paying off, and the reconfigurations help the dynamic subgraph algorithms to get a "fresh" start. A similar pattern can be seen in the average pair-wise latencies in figure 14.14. Figure 14.15 plots the CDF of the achieved diameters of the different algorithms. The graph clearly visualize that kRDA with tighter bounds is forced to reconfigure and thus achieves a lower subgraph diameter.

The stability is important to maintain, and figure 14.16 plots the CDF of the edge-changes in reconfigurations. The kIT(smddl-OTTC,$k = 3$) is clearly the most unstable, while KDA(RTR-P-MC,I-MC,$k = 3$) is the most stable. The kRDA algorithm combinations vary in their stability depending on the diameter bound B. A lower bound yields a lower stability in terms of a larger number of edge-changes, and a higher bound, yields a higher stability in terms of fewer number of edge changes. Overall we see that it is possible to keep the stability using kRDA, but it depends on the diameter bound. The CDF of the execution times (seconds) of the reconfigurations is plotted in figure 14.17. We observe that a similar pattern emerge as in CDF plot for the edge-changes (figure 14.16).

From the previous results we observed that it is possible to maintain the stability and quickness of the dynamic subgraph algorithms and reduce the diameter further by reconfiguring the subgraph occasionally. In the tests we saw that the diameter bound 0.450 seconds was the bound that overall fit these algorithms the best. The bound, however, is hard to obtain and is

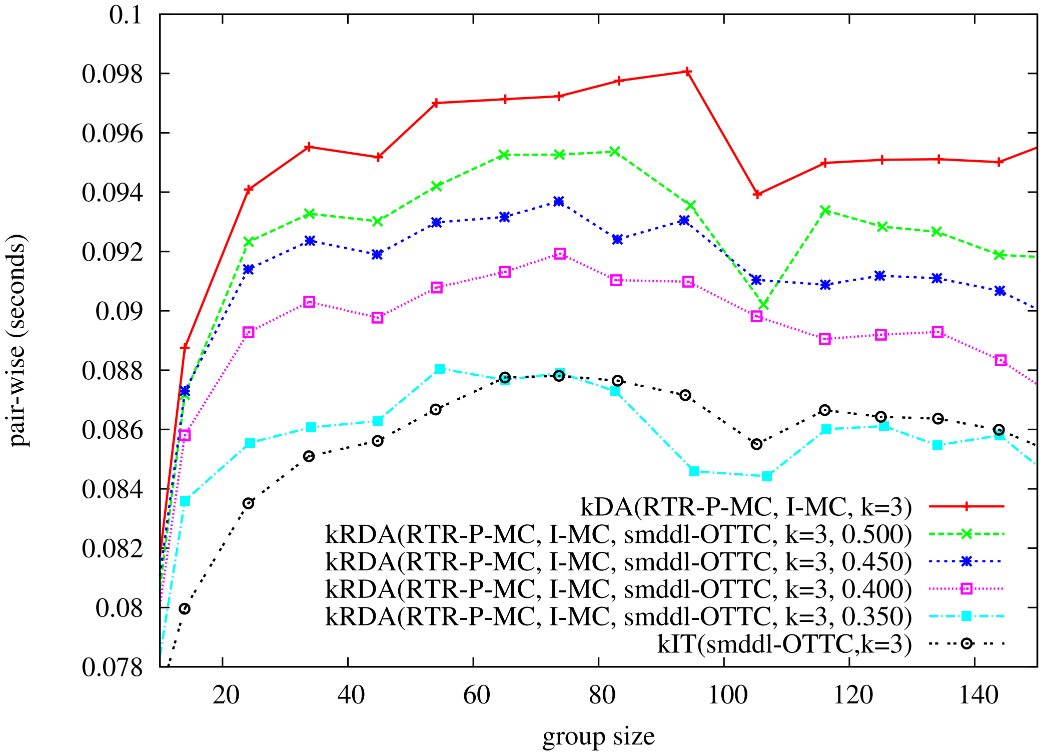


Figure 14.14: Pair-wise latency (seconds) of subgraphs.

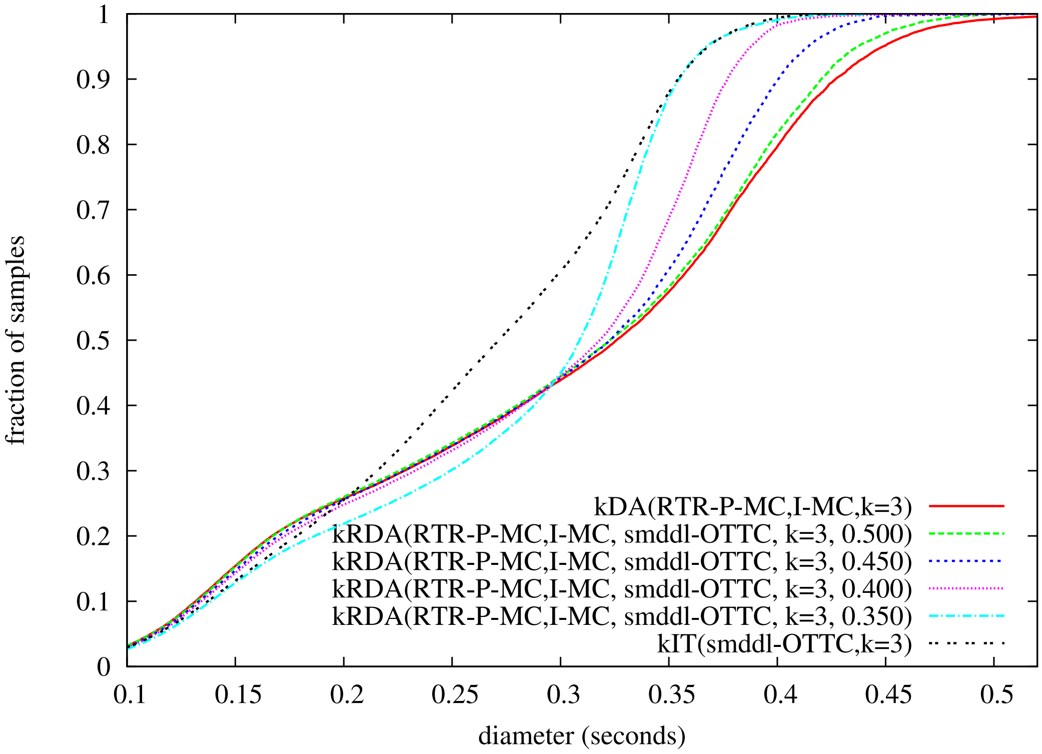


Figure 14.15: CDF of diameter (seconds) of the group-subgraphs.

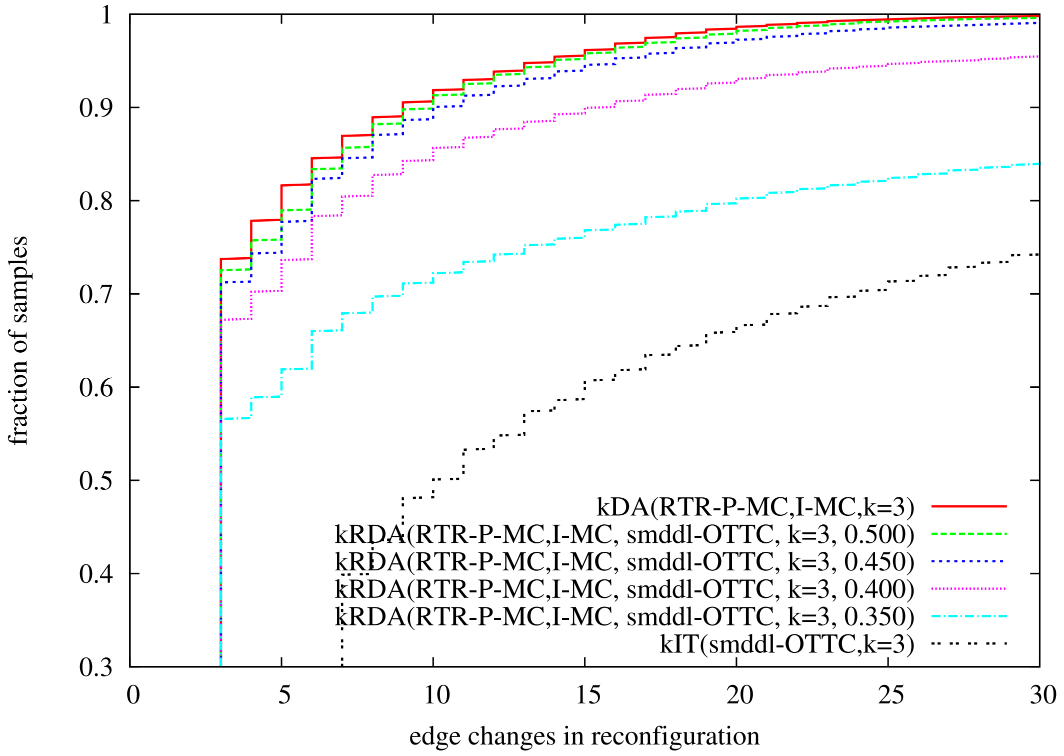


Figure 14.16: CDF of stability, in terms of edge-changes in reconfigurations.

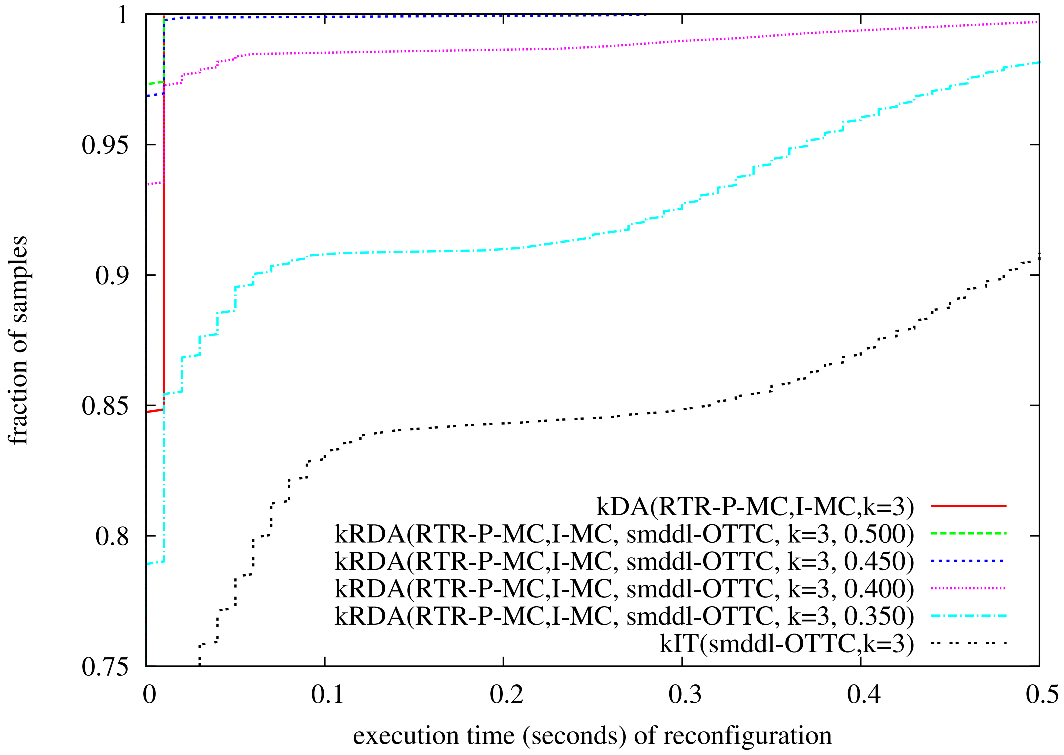


Figure 14.17: CDF of execution time (seconds) of the subgraph reconfigurations.

sometimes not desirable. But, in distributed interactive applications there are often absolute minimum requirements to the latency, hence, it is therefore possible to set a minimum latency bound that the subgraphs must obey.

14.5 Summary of the main points

In the previous sections we gave a summary of the better performing tree and subgraph algorithms that were found in our investigations in previous chapters.

We identified that dynamic-tree algorithms yield overlay trees that degrade for larger group sizes, but that they have desirable properties linked to a high stability and low execution time. The stability is measured in terms of the number of edge-changes in a reconfiguration. We also found that the Steiner-tree heuristic smddl-OTTC yield trees with a consistently low diameter. Therefore, we devised an algorithm that took as input a dynamic tree algorithm, in terms of an insert strategy and a remove strategy, and a Steiner-tree heuristic. In addition, there was a configurable diameter bound that decided if a tree had degraded and a total reconfiguration using the Steiner-tree heuristic was necessary. The goal was to preserve the high stability and low execution time of the dynamic tree algorithms, and still manage to create trees with a consistently low diameter. The results showed that with a well-set diameter bound, we were able achieve these target metrics.

We further investigated dynamic-subgraph algorithms and the subgraph algorithm k-Iterative tree construction (kIT). Also here, we found that the dynamic-subgraph algorithms are stable and fast, while the subgraph algorithms are more unstable and slower, but yield a lower diameter. We devised a similar algorithm for reconfiguration of subgraphs that included a configurable integer k, which described the connectivity of the subgraphs. The results showed that we were able to achieve a higher stability and lower execution time, and also keep the diameter and pair-wise latencies very low.

The main conclusion of the investigations is that dynamic algorithms are well-suited for updating overlay networks that are used for real-time interaction, due to their stability and quickness. However, we also found that when the trees and subgraphs degrade, it is well-advised to reconfigure them by applying a close-to-optimal heuristic, which optimizes towards a low-latency overlay.

Chapter 15

Group communication experiments: Overlay construction algorithms

The group management techniques that were introduced in chapter 5 formed the foundation for the investigations in chapters 6 through 14. These investigations identified algorithms that were suited for the resource management and the overlay management.

In order to test the interoperability between the techniques, we implemented a group communication system that we tested on PlanetLab. The evaluation of the system began in chapter 6, which evaluated the latency estimation techniques Vivaldi and Netvigator in terms of their accuracy in retrieving all-to-all path latencies. The results showed that Netvigator is the most accurate but may be harder to setup, Vivaldi is inaccurate but easier to setup. The all-to-all latency estimates obtained by the latency estimation techniques in the network information mananagment, are used by centralized graph algorithms in the resource management and the overlay network management. More specifically, they are used by core-node selection algorithms (chapter 7) and overlay construction algorithms (chapter 9 through 14).

The interoperability between latency estimation techniques and the core-node selection algorithms was evaluated in section 7.7. It was clear that the Netvigator estimates enabled the core-node selection algorithm k-Median to choose core-nodes that were the close-to-optimal ones. The more inaccurate Vivaldi estimates resulted in a penalty to the core-node selection, and made k-Median choose non-optimal core-nodes. However, the penalty was within reasonable bounds and the conclusion was that both Netvigator and Vivaldi yield sufficiently accurate all-to-all latency estimates.

The following investigation evaluates the interoperability between latency estimation techniques that obtain all-to-all latency estimates, and centralized graph algorithms in terms of core-node selection algorithms and overlay construction algorithms.

More precisely, we apply the latency estimates from Vivaldi and Netvigator to selected spanning-tree and dynamic-tree algorithms, and evaluate the usability of the trees. A few of the spanning-tree algorithms rely on applying the k-Median core-node selection algorithm to find the source-node, from which the tree-construction starts. Some of the dynamic-tree algorithms also try to identify well-placed core-nodes they can include to the tree when nodes are either inserted or removed. Therefore, the tree-construction is also influenced by how well k-Median is able to identify core-nodes.

The results showed that the all-to-all latency estimates from Netvigator are sufficiently accurate to be used in the tree construction, compared to using all-to-all ping measurements. The Vivaldi estimates yield a larger penalty, but were also found to be usable. However, when k-Median was actively used by tree algorithms, the inaccuracy of the trees increased when using Vivaldi estimates, while it remained stable when using Nevigator estimates.

The rest of the chapter is organized in the following manner. Section 15.1 introduces how the overlay network management and network information management cooperates. Section 15.2 presents results from the PlanetLab experiments, where the latency estimates are used by the tree algorithm dl-SPT, and we evaluate their accuracy. Section 15.3 further presents results from the PlanetLab experiments, where the latency estimates are used by multiple tree algorithms, and we evaluate and compare their accuracy. Finally, section 15.4 gives a brief summary of the main points.

15.1 Centralized group communication system

The chapter addresses a goal of the thesis, stated in section 5.5, which was a goal for the network information management:

4) Identify techniques that are able to obtain accurate all-to-all Internet path latencies.

The motivation behind this goal is that centralized graph algorithms are desirable because they are fast, and these algorithms use latency in their search routines. Therefore, in order to properly address the goal we use the centralized group management system from chapter 5, and implement all the techniques introduced there in a real-world system. The centralized group management system includes techniques for membership management, resource management, overlay network management and network information management.

The investigation started in section 7.7, which investigated the interoperability between the network information management and the resource management, in terms of latency estimation techniques and centralized core-node selection algorithms.

In this investigation, we only evaluate the network information management and overlay network management. The network information management holds the latency estimation techniques Vivaldi and Netvigator. The overlay network management includes graph manipulation algorithms and overlay construction algorithms. The graph manipulation algorithm we evaluate is the core-node selection algorithm k-Median, which finds well-placed nodes to be used in the

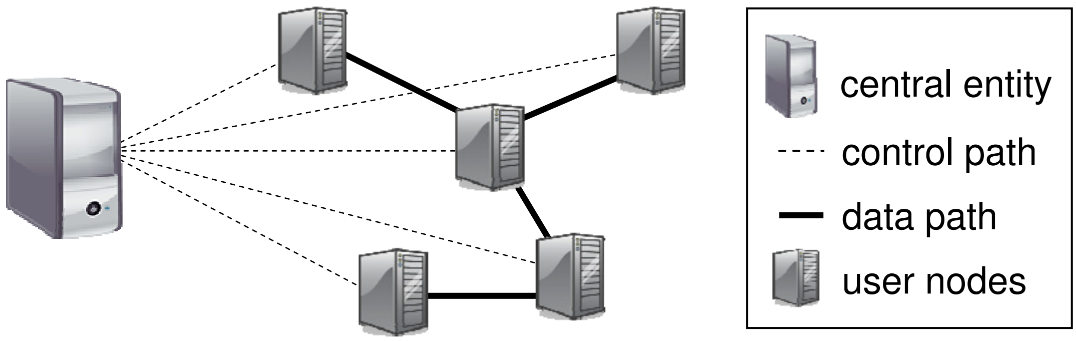


Figure 15.1: Central entity executes the group management teqhniques.

tree construction. The overlay construction algorithms we evaluate are selected spanning-tree and dynamic tree algorithms, from chapter 9 and 12.

The centralized group management system is run by a central entity, and the cooperation between the three parts of the overlay management is as follows:

1. A latency estimation technique estimates the link latencies between the clients in the application. A latency estimation technique continuously updates the link latencies in a global graph that holds all the current clients in the application's network.

2. A graph manipulation algorithm uses the group information from the membership management to group the clients into complete graphs. In this complete graph, each node is a client, and each edge holds the link latency as estimated from the latency estimation technique. Then, it uses a core-node selection algorithm to identify well-placed nodes in the group. This is useful for many overlay construction algorithms that are dependent on the source-node from which it starts building the overlay. Moreover, it is also useful for many dynamic-tree algorithms that are dependent on finding well-placed Steiner-points to include to the trees.

3. The overlay construction technique is given as input a group graph from the graph manipulation and constructs an overlay network on it. This overlay network is to be used for the application's event distribution.

Of the three steps, it is the overlay construction step that actually builds the overlay networks, and the following investigations use selected tree algorithms to execute the overlay construction. The performance of some of these tree algorithms is influenced by the initial source node that is used to start the tree construction. In addition, we evaluate dynamic-tree algorithms that aggressively tries to include well-placed Steiner points to the group trees. All of these well-placed nodes are chosen by the core-node selection algorithm k-Median.

The performance of k-Median was evaluated through simulations in chapter 7, and found to be a very good algorithm for finding well-placed nodes. Then it was used in section 7.7,

Descriptions	*Configurations*
Group sizes	g = 4, 8, 12 clients
RTT measures	*tcpinfo*, *ping*
Packet rates	high (100 packets/sec.), low (2 packets/sec.)
Log times	t = 4, 8, 12, 16, 20 minutes

Table 15.1: Vivaldi experiment configurations.

to evaluate whether the latency estimates from Netvigator and Vivaldi enabled k-Median to find close-to-optimal core-nodes. The conclusion was that both yield latency estimates accurate enough for core-node selection.

In the following investigations we wish to observe the cumulative effects that the accuracy (or inaccuracy) of the latency estimates have on tree algorithms, many of which that rely on core-node selection.

15.2 PlanetLab experiments: Single tree algorithm

The initial construction experiments use *dl-SPT* and a degree limit of 5 for all nodes, which is due to somewhat limited client capacities in the Internet [130]. Furthermore, we use the same 100 PlanetLab nodes that were used for the core-node selection experiments in section 7.7.1. In general, a network of 100 PlanetLab nodes and group sizes below 40 do not give a good enough foundation for a conclusive evaluation of our tree algorithms. However, firstly we are primarily testing the applicability of latency estimation techniques and not the performance of our tree building algorithms. Secondly, in the previous investigations we have performed the experiments on BRITE-generated graphs of size 1000, and the results largely correlate with our findings on this PlanetLab network with fewer nodes.

Our distributed application mimics group communication in distributed interactive applications (e.g., online games). The 100 nodes join and leave groups dynamically throughout our experiments by sending join and leave requests to a central entity. The group popularity is distributed according to a Zipf distribution. The central entity uses the latency measurements or estimations to choose the most appropriate core node using the k-Median core-node selection algorithm (chapter 7), and recomputes the multicast tree for the group whenever membership changes. For the Vivaldi estimates, we allowed a period of 4 minutes to let the node coordinates stabilize. The tests were run each day for a 10 day period. Table 15.1 gives the Vivaldi configurations.

15.2.1 Tree experiment metrics

We evaluated the quality of the trees that were created based on estimates from Vivaldi and Netvigator by examing the resulting *diameters*. The *diameter* expresses the worst case latency between any pair of group members, and is a particularly important metric when all-to-all group communication is required by an interactive application. Since applications make decisions based on the assumed or measured diameter, it is not only important that it is low but also that the application can know it with a high degree of trust. We look at and compare diameters from three different angles in our evaluation:

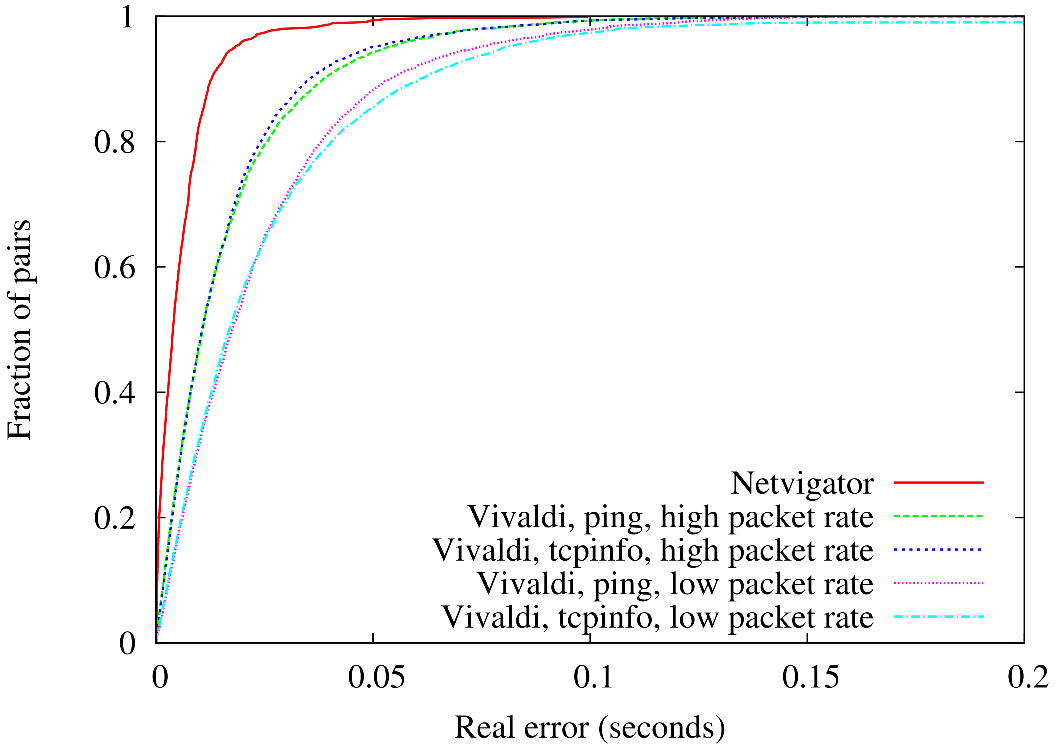
- The *reported diameter* is the tree diameter that is obtained by using the estimated latencies in all steps. That is, by running the group manipulation and overlay construction based on the results of the network identification phase. The value of the reported diameter is an estimated value.
- The *real diameter* is the tree diameter that is obtained by applying the path of the reported diameter on the close-to-real all-to-all ping measurements. Thus, the real diameter is obtained by running the manipulation and construction steps based on estimation. The value of the real diameter, however, is calculated using the close-to-real all-to-all ping measurements on the reported diameter path.
- The *optimal diameter* is the tree diameter that is obtained by using the close-to-real all-to-all ping measurements in all steps. That is, by running manipulation and construction step based on the all-to-all ping measurements. The optimal diameter is taken from these measurements.

It is important that an application can trust the group tree diameter it is reported. Therefore, we measured the difference between the *reported diameter* from the estimates and the *real diameter* from the all-to-all Ping latencies. In addition, we measured the difference between the real diameter and the *optimal diameter*.

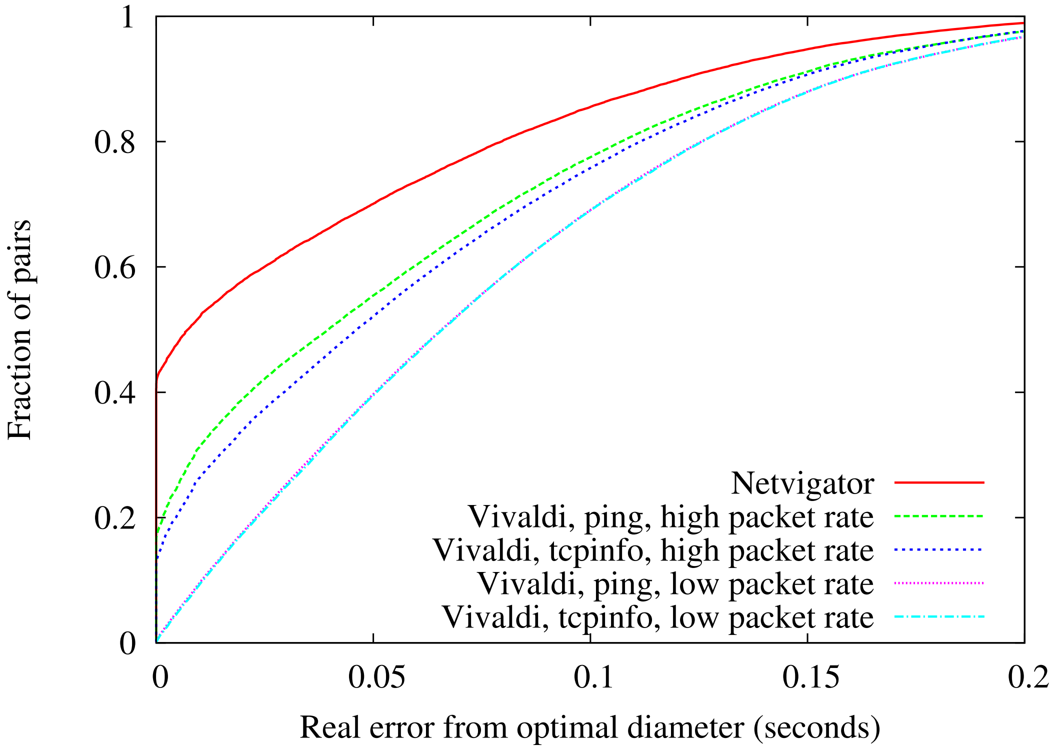
For comparison, we also measured the number of *non-member-nodes* that are present in the trees, and the number of *edge changes* that are made upon tree reconfigurations. Finally, we measured the *tree migration* time, which is the time it takes from the tree updates are sent until every client in the tree has received it.

15.2.2 Tree experiment results

The following are evaluations of the results we got from our group communication experiments performed on PlanetLab. Figure 15.2(a) shows the CDF of the error (in seconds) between the reported tree diameter from the estimates and the real diameter. The group sizes were > 20 for increased confidence and we exhausted the group membership combinations possible. *sdl-SPT* applied to Netvigator yields 96 % of the estimated tree diameters within a 20 milli-second error margin from the real diameter. While, the estimates from Vivaldi with high packet rates using tcpinfo and ping RTTs yields 85 % of the tree diameters within a 25 milli-seconds error



(a) CDF of real error between reported diameter and real diameter.



(b) CDF of real error between real diameter and optimal diameter.

Figure 15.2: Tree metrics for diameter.

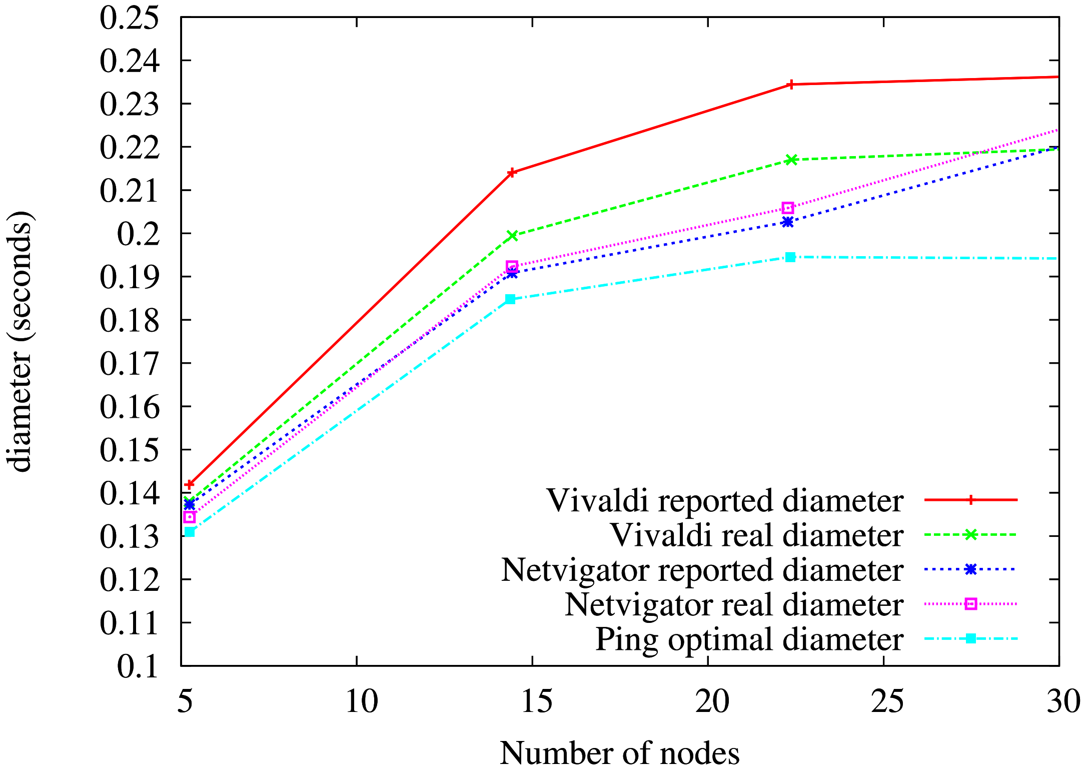


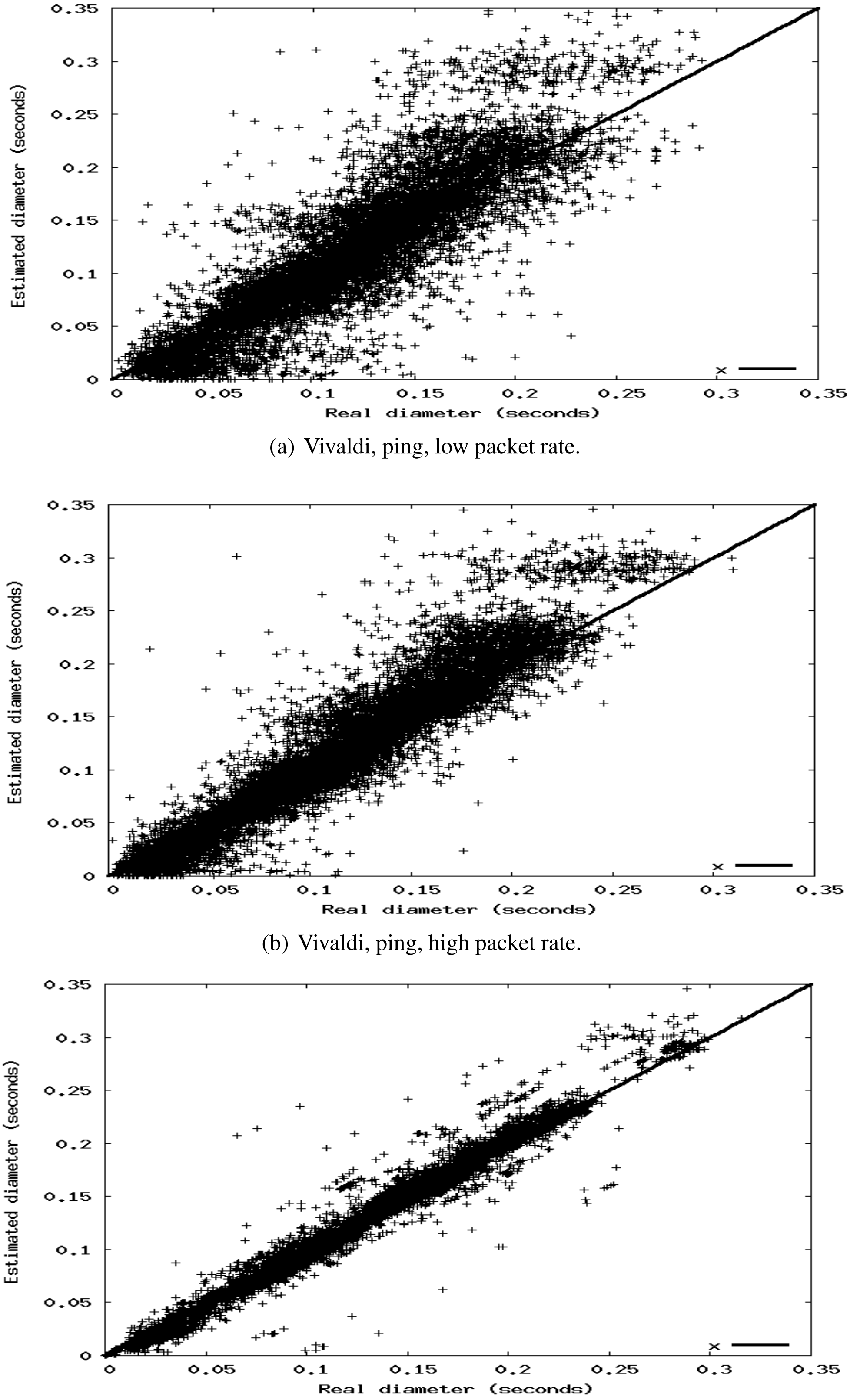
Figure 15.3: Diameter of selected tree algorithms using estimations and real latencies.

margin. Vivaldi exhibits the poorest performance with low packet rates, which yields 80 % of the tree diameters within a 35 milli-seconds error margin. However, the discrepancy between the reported diameter from the estimates and the real diameter is on average reasonably low for Vivaldi.

Figure 15.2(b) shows the CDF of the error (seconds) between the real diameter and the optimal diameter. We see that the error margin between them becomes quite high. This may be due to a cumulative error effect when tree algorithms build a tree from the estimates.

Figure 15.3 shows the diameter (seconds) produced from the estimates compared to the real-world for various group sizes using the tree algorithm *sdl-SPT*. We observe that for group size 30 the difference between the reported diameter from the estimates and the real diameter is 5 milli-seconds for Netvigator and 15 milli-seconds for Vivaldi (ping RTTs and high packet rate). However, the difference is bigger when compared to the optimal diameter. In that case, it is 25 milli-seconds for both Netvigator and Vivaldi.

In figures 15.4 and 15.5 we complete the evaluation of the estimates and visualize the discrepancy between the reported diameter and the real diameter. Not surprisingly, the same tendency that we saw from the core eccentricities emerge. Netvigator performs best, Vivaldi with high packet rates is better than with low packet rates, and ping RTTs is slightly better than tcpinfo RTTs.

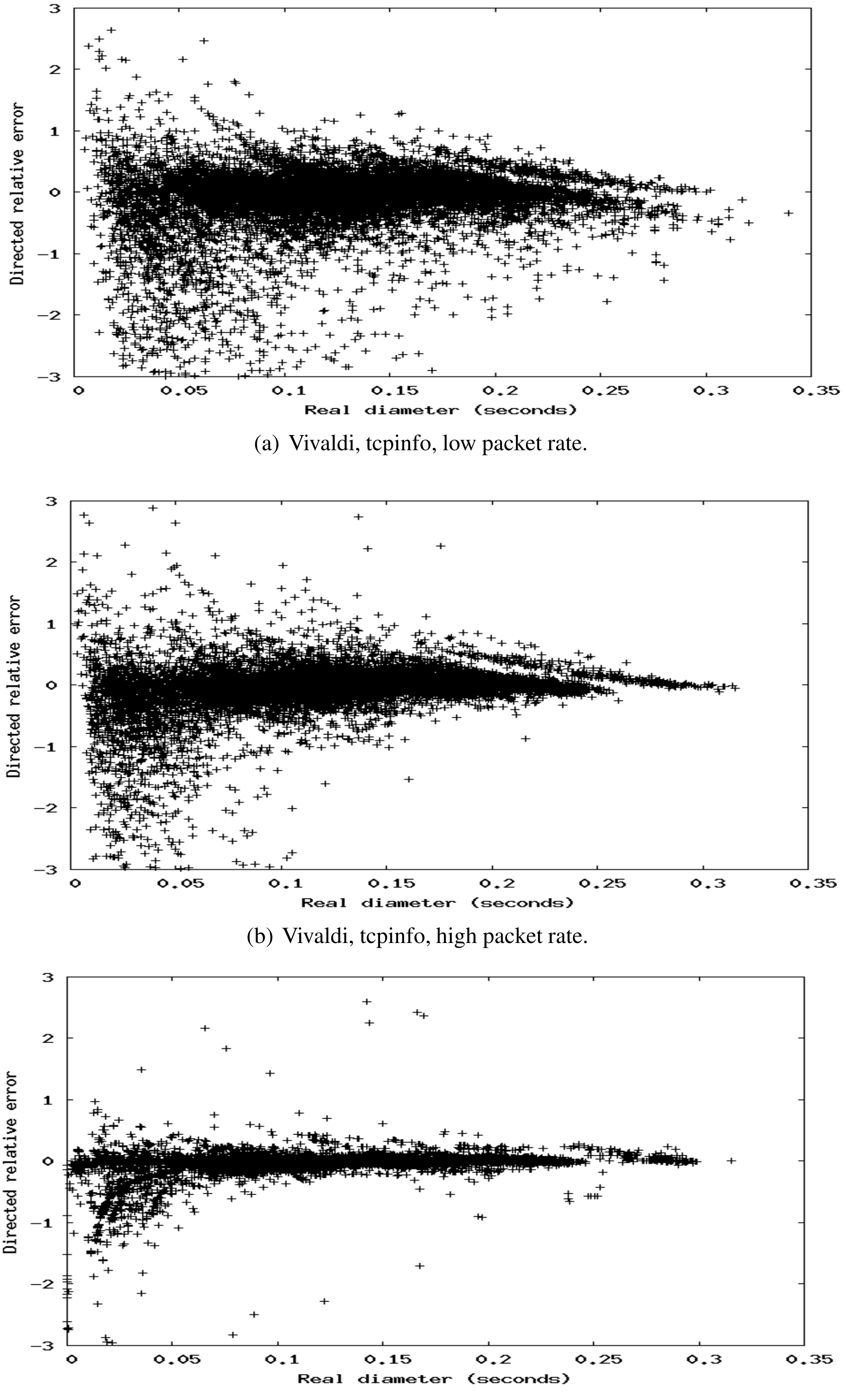


(a) Vivaldi, ping, low packet rate.

(b) Vivaldi, ping, high packet rate.

(c) Netvigator.

Figure 15.4: Reported diameter vs. real diameter in group trees.



(a) Vivaldi, tcpinfo, low packet rate.

(b) Vivaldi, tcpinfo, high packet rate.

(c) Netvigator.

Figure 15.5: Directed relative error of the reported diameter to the real diameter in group trees.

15.3 PlanetLab experiments: Comparison between tree algorithms

The previous results used the dl-SPT algorithm to show the penalties of using latency estimates to construct trees. The following evaluates the penalties when the latency estimates are applied to several tree algorithms. These tree algorithms include spanning-tree algorithms and dynamic tree algorithms that were evaluated in chapter 9 and 12 through simulations.

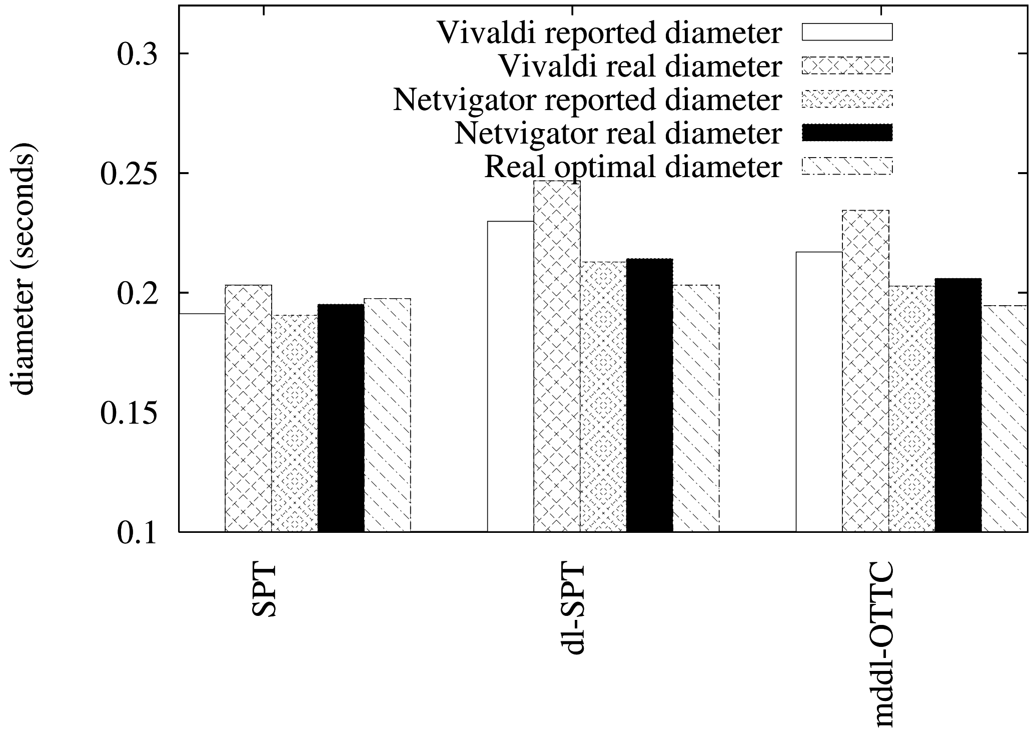
Figure 15.6 and 15.7 shows the tree diameter performance of the dynamic algorithms and the tree algorithms when applied to Netvigator and Vivaldi estimates, as well as the real all-to-all pings. We include Dijkstra's shortest path tree (SPT) [58] results for the sake of comparison. Generally, we see that trees produced from Netvigator estimates have a closer to the real-world reported diameter than those from Vivaldi. The insert and remove strategies that search for well-placed core-nodes to use as Steiner-points (RTR-MDDL, RTR-P and ITR-MDDL) do not perform well when applied to Vivaldi estimates. In particular, RTR-P shows a clear tendency in disfavor of using the Vivaldi estimates, independent of the chosen insert strategy. RTR-P aggressively prunes non-member nodes and searches for new well-placed nodes to use as Steiner-points. We see that the lower quality latency estimates from Vivaldi have an impact on the performance of RTR-P.

It is also important and remarkable that estimates of the diameter are in all cases lower than the diameter that is actually achieved. More specifically, the Vivaldi reported diameter is lower than the Vivaldi real diameter. This is supported by the observations made already in chapter 6, that more latencies are under- than over-estimated by Vivaldi and Netvigator. It forces applications relying on estimation techniques to make conservative assumptions about the diameter of subgroups whenever the delay limits for the application are hard.

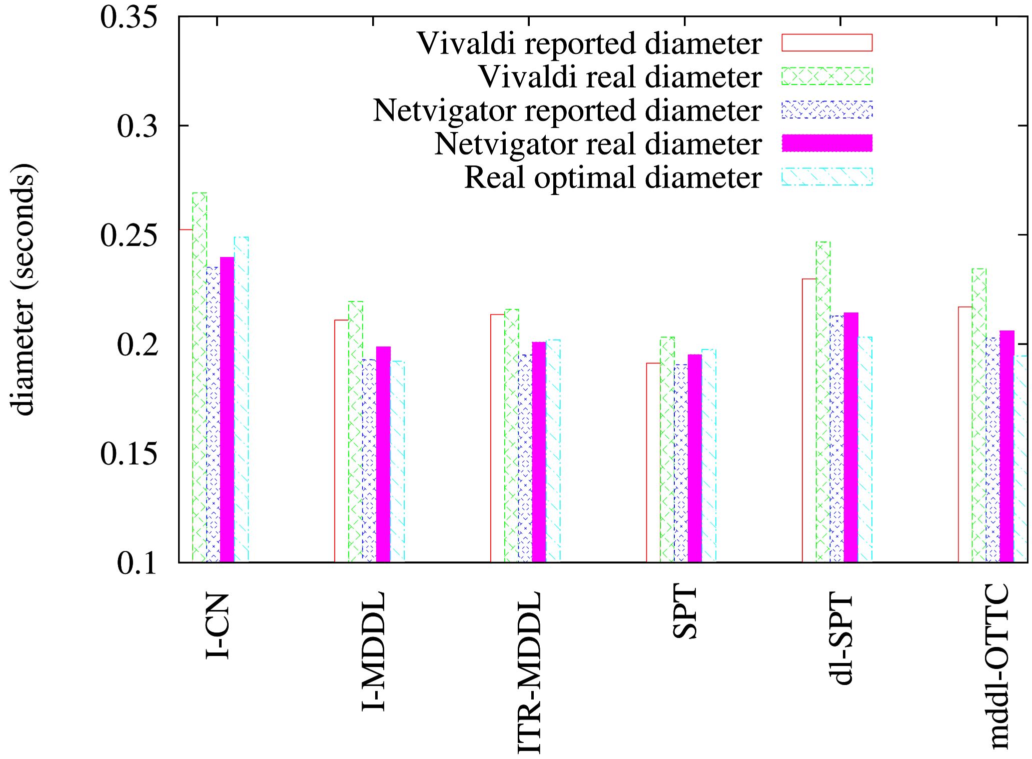
From figure 15.8 we see that ITR-MDDL combined with RK and RTR-MDDL suffers from non-member node degradation, i.e., an increasing number of non-member nodes in the trees. Figure 15.9 shows that the number of *edge changes* is higher with RTR-P. One approach may be to run RTR-P only periodically. dl-SPT and mddl-OTTC both suffer from higher *execution time* and more *edge changes*.

15.4 Summary of the main points

We conducted an experimental analysis of the latency estimation techniques Vivaldi and Netvigator in the context of dynamic group communication for distributed interactive applications. From chapter 6 we know that Netvigator yields latency estimations that are very close to optimal, while the estimations from Vivaldi are not as good. Vivaldi's advantages are that it is very easy to deploy in a peer-to-peer fashion and it handles membership dynamics. Netvigator, on

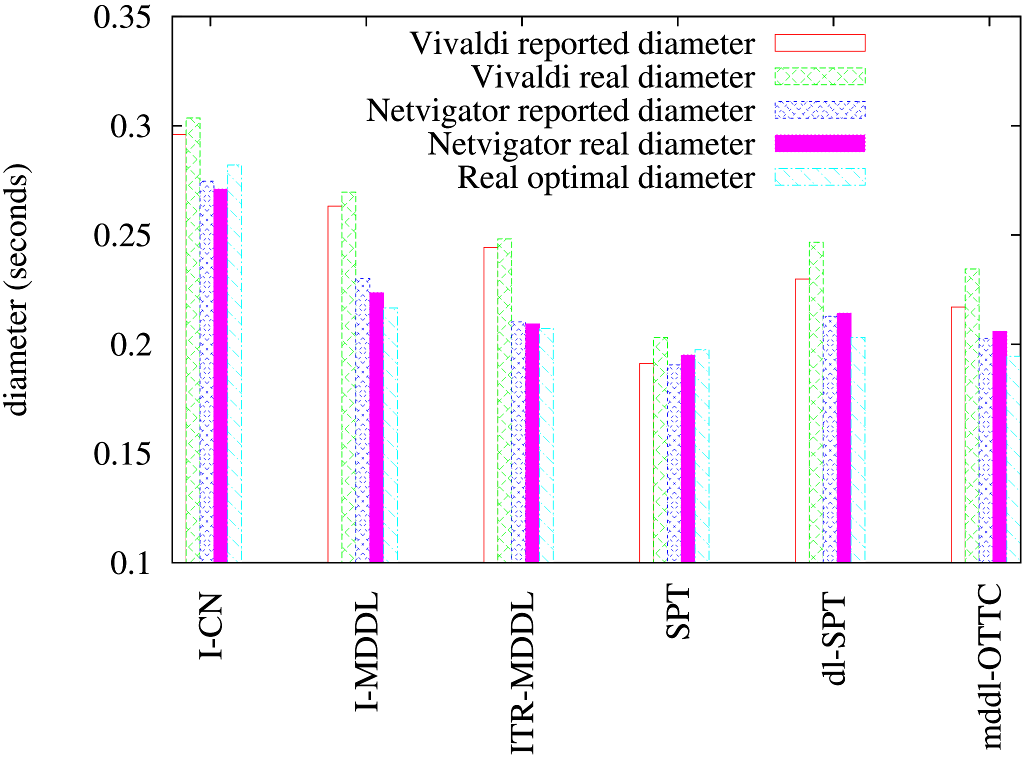


(a) Tree algorithms

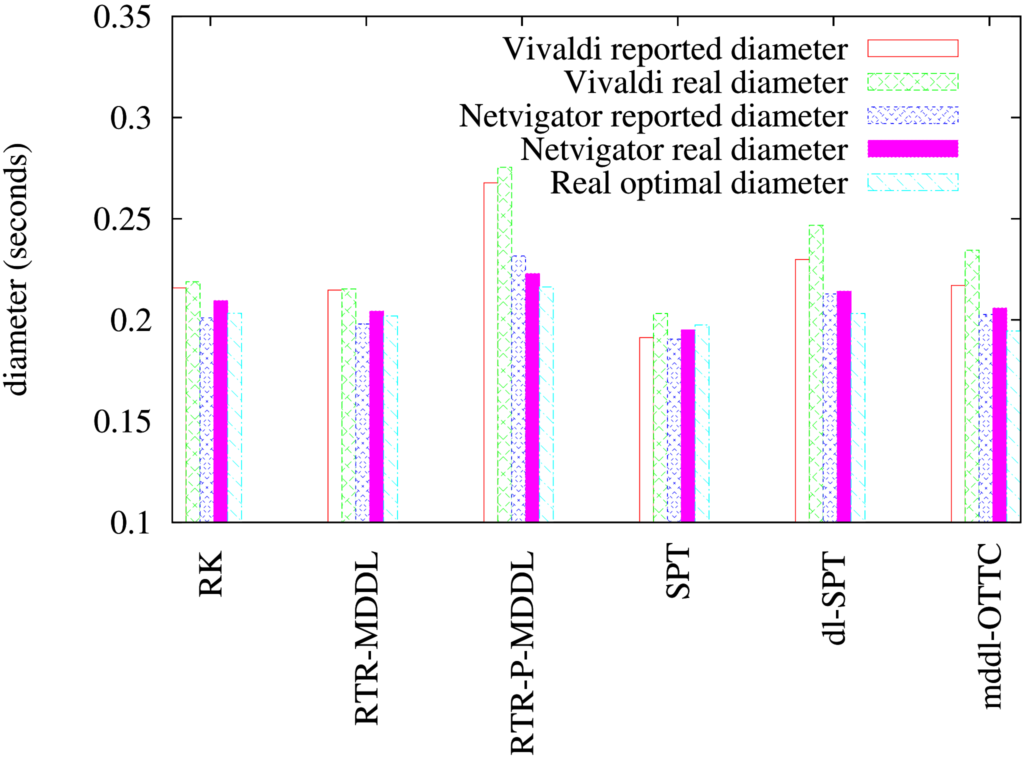


(b) Remove strategy is RTR-MDDL.

Figure 15.6: Comparison of reported, real and optimal diameter (group sizes 20-30).



(a) Remove strategy is RTR-P-MDDL.



(b) Insert strategy is ITR-MDDL.

Figure 15.7: Comparison of reported, real and optimal diameter (group sizes 20-30).

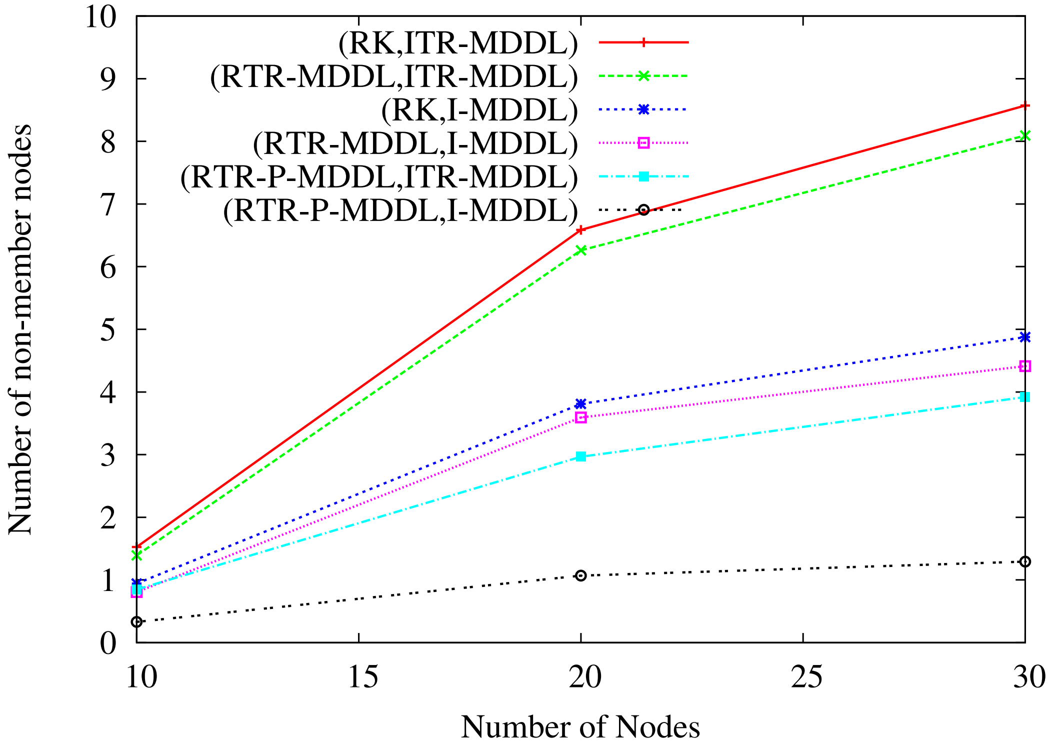


Figure 15.8: Number of non-member nodes in trees.

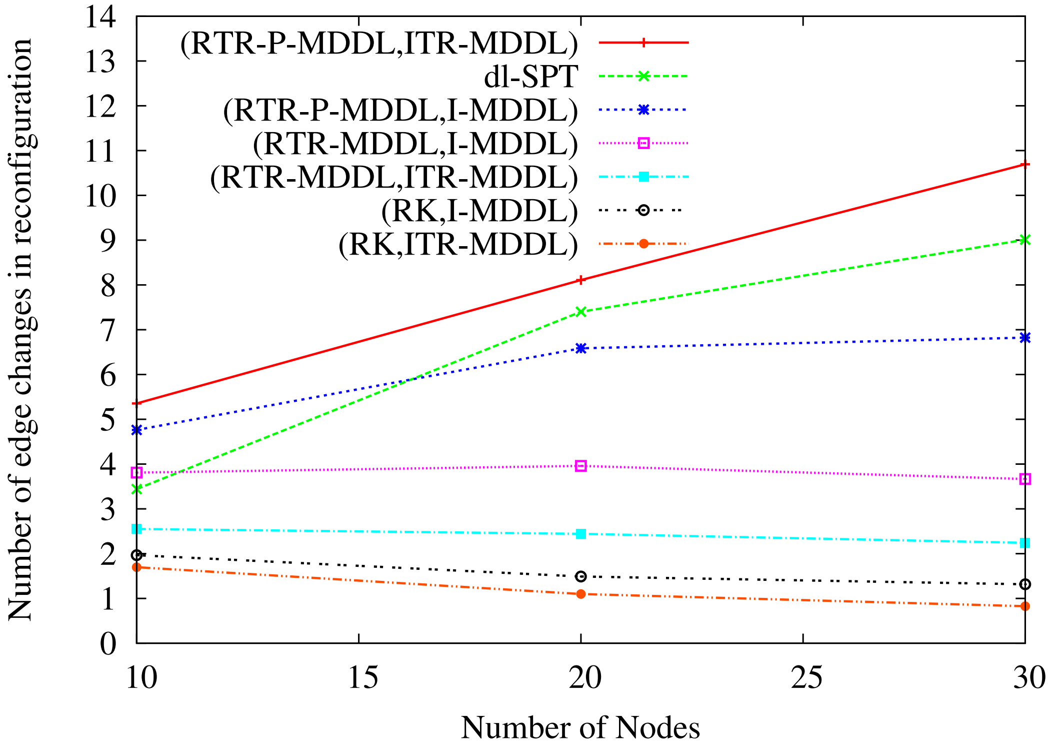


Figure 15.9: Number of edge changes in membership changes.

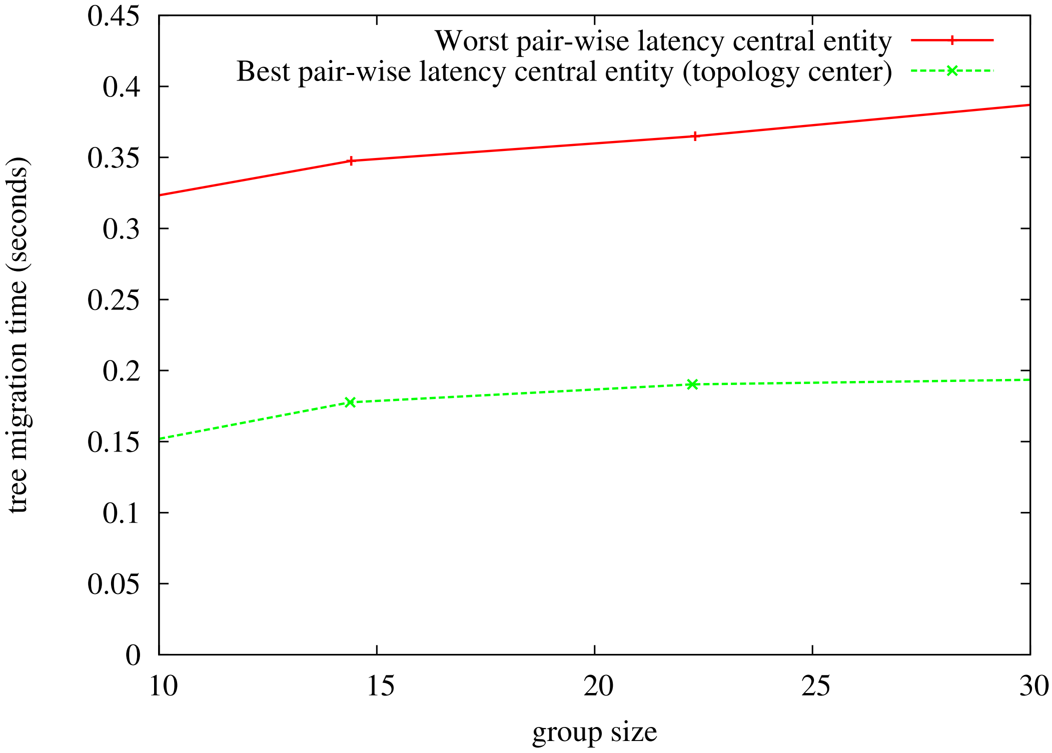


Figure 15.10: Tree migration times for central entity chosen by k-Median core-node selection algorithm, compared to the worst case pair-wise.

the other hand, needs an infrastructure (Landmark nodes).

In this chapter, we experimentally tested a centralized membership management architecture for dynamically changing subgroups in a distributed interactive application. The architecture included latency estimation techniques, core-node selection algorithms and tree algorithms that together made it possible to build multicast trees optimized for the tree diameter. We found that the collection of techniques and algorithms worked very well together.

The advantage of a centralized approach is the consistency it gives, however, the drawback is the administration latencies involved in membership and group updates. Especially the tree migration time is an issue when multicast trees are dynamically updated. Figure 15.10 shows the average tree migration times for different group sizes. It is quite clear that the k-Median core-node selection algorithm ensures that a central entity is chosen that has low pair-wise latencies to the clients in the member network. The worst case central entity placement may be a valid situation in cases where a game provider has a static central entity that serves the entire world. A static central entity cannot be in the current topological center at all times, because as the night and day passes the clients in a game dynamically shifts from continent to continent.

Chapter 16

Distributed interactive application scenarios: Applying the research

The studies performed from chapter 5 through 15 included many different group communication techniques and algorithms that were all evaluated towards their applicability to distributed interactive applications. The evaluations revealed several algorithms that are suitable to address the thesis goals (described in section 2.5).

Although the main goals of the thesis focused on enabling a specific type of distributed application, the research is applicable to very many application scenarios. It is clear that we cannot address every possible scenario, but in this chapter, we do present a range of basic and more advanced choices an application designer must make during development. Table 16.1 presents some of the parameters and options that are possible. In the table we highlight what we conceive to be very important variables in terms of research categories, metrics, etc, and then give a wide range of possible parameters and values for each of them. These are later on used in table 16.2 to describe specific distributed interactive applications.

The discussions in this chapter are based on the results obtained through simulations and experiments. For each discussion we suggest algorithms that are usable for selected application scenarios. More specifically, the discussions include questions related to which architecture, management approaches and algorithms to choose for distributed interactive applications. In short, we found that centralized solutions are desirable because they yield lower administration latencies than distributed options. However, we did find that the data paths should be among interacting clients to avoid a more costly and less scalable centralized client/server communication model. Generally, the requirements in distributed interactive applications are very much tied to link latency and bandwidth. Chapter 2 introduced several distributed interactive applications, for which one important requirement was:

Variable	Parameters/values
Group update	static = no-updates, dynamic = updates
Application members	low = $[0,100]$, medium = $[100,1000]$, large = $[1000,\infty]$
Group size (nodes)	low = $[0,100]$, medium = $[100,1000]$, large = $[1000,\infty]$
Super nodes	yes (use as Steiner-points), no
Architectures	Client/server, peer-to-peer, hybrid centralized/distributed
Optimization	Diameter, total-cost, reconfiguration-time, edge-changes (stability)
Constraints	Bounded diameter, degree-limits, total-cost
Latency (ms)	low ≤ 100, medium ≤ 500, large ≤ 1000
Bandwidth (kbps)	low ≤ 10, medium ≤ 100, large ≤ 500

Table 16.1: Variables and values

To achieve real-time interaction across the Internet requires sufficiently low latencies between the interacting participants.

This is the most basic requirement for any application that aims to support real-time interaction across the Internet. Section 2.3.5 presented specific latency requirements that were based on user satisfaction studies. The studies revealed that the latency requirements vary depending on the application type:

From 100 *milliseconds for first-person shooter games and* 150-200 *milliseconds in audio/video conferencing, to* 500 *milliseconds for role-playing games and* 1000 *milliseconds for real-time strategy games [70, 27].*

We observe that it is harder to support real-time interaction between multiple parties when the latency requirement is 100 milliseconds, than if it is closer to 500 milliseconds. For this reason, the first-person shooter games that exist today support rather few concurrent players. Furthermore, the audio/video conferencing applications that exist do all require extra equipment to work when the number of participants start to exceed in numbers. This is due to the strict latency requirements combined with bandwidth-demanding audio/video streams. The bandwidth capacity on average clients in the Internet is increasing, but it is still too low to be able to multicast many bandwidth intensive audio/video streams in an overlay network.

Table 16.2 summarizes some of the characteristics and requirements that a few existing distributed interactive applications have. In the table, we use a few of the variables and values from table 16.1 to describe the applications. We see that most of the applications have latency requirements that are fairly low (≤ 200). Moreover, we observe that it is expected that the applications support membership dynamics to some degree. Many of the applications, especially the games (FPS, RTS and RPG), also divide the application members into sub-groups. This is done mainly to achieve robustness, consistency and consequently better scalability.

The rest of the chapter is organized in the following manner. Section 16.1 briefly discusses possible architectures of distributed interactive applications. Section 16.2 summarizes how the membership management in application scenarios with one or multiple groups should be approached. Section 16.3 compares some basic types of overlay networks: source overlays and shared overlays. Section 16.4 introduces and compares static and dynamic groups. Section 16.5 and 16.6 go into more details regarding how static and dynamic groups can be treated, and how their properties and requirements can be addressed using overlay construction algorithms. Section 16.7 discusses what influences the performance of an overlay construction algorithm. Finally, section 16.8 provides brief summaries of the main points.

16.1 Architecture: centralized vs. distributed

Current distributed interactive applications often employ a centralized architecture. Section 2.2 described MMOG games and how the game companies spend millions of dollars in development, but still most of them employ a rather simplistic client/server architecture. This is likely due to the immense complexity of the MMOGs, and the fact that large centralized server-parks are easier to manage and control. In the thesis, we also chose a centralized architecture because of the consistency and control it yields compared to distributed architectures, but our approach was different from a client/server architecture.

We proposed a group management architecture that a central entity executes (chapter 5). The architecture consists of a centralized membership management, a resource management and an overlay management with centralized graph algorithms, and finally a network information management to retrieve all-to-all path latencies.

We used centralized graph algorithms because distributed graph algorithms currently suffer due to large end-to-end path latencies in the Internet. Centralized graph algorithms need link information if they are to address latency requirements. Therefore, we used latency estimation techniques to measure and estimate link latencies. Moreover, the client interactions should be multicast among the clients, and we considered application layer overlay multicast as a way of achieving group communication. An overlay network is used by interacting clients to distribute events through it. Generally, to multicast application data through overlay networks is efficient in terms of bandwidth consumption, but may increase the end-to-end latency compared to a centralized server solution. However, if the centralized server is mis-placed, for example, located far away from the interacting clients, this approach will also increase the end-to-end latencies.

16.2 Membership: One group vs. multiple sub-groups

Section 2.3.2 discussed the difference between managing one group of clients, to having multiple sub-groups of clients. We identified that when an application manages one group of clients it suffices to form low-latency event-distribution paths in which all clients are reachable, and

Application	*Membership change*	*Application members*	*Subgroup sizes*	*Consistency requirements*	*Latency bound*	*Bandwidth requirements*
FPS	high	low	low	high	100 ms	low
RTS	high	medium	low	high	1000 ms	low
RPG	high	large	low/medium	high	500 ms	low
Audio Conference	low	low	-	medium	200 ms	medium
Video Conference	low	low	-	medium	200 ms	large
Remote Operation	-	low	-	critical	100 ms	medium
Military CS	high	large	low/medium	critical	100 ms	medium
3D Conference	low	medium	low/medium	high	200 ms	low

Table 16.2: Application table.

a membership management system is enabled by distributed mechanisms for handling client churn.

When multiple sub-groups are used to distribute events in an application, the approach poses membership management challenges related to how the groups are updated.

We proposed a centralized approach to the membership management for such scenarios of multiple sub-groups of clients (section 5.2). One issue with a centralized approach is the potential large latencies from some group members to the node executing the membership management. Therefore, a core-node selection algorithm should be used to select a well-placed node that may execute the membership management. In the evaluations of chapter 7 we found that the core-node selection algorithm k-Median is able to find core-nodes that yield low maximum one-way latencies to groups of clients. Generally, a centralized membership management approach is applicable to all the application examples in table 16.2.

16.3 Group communication: Source-overlays or shared-overlays

In the thesis, we have evaluated and discussed a wide range of overlay construction algorithms. These algorithms construct two main types of overlay networks: source-trees and shared-overlay networks.

The minimum-latency approach is to build a *source-tree* for each client that is interacting and distribute the events through them. However, this is not scalable for large numbers of interacting clients, mainly because application layer overlay networks are inherently complete networks (full meshes) of shortest paths, such that a source-tree has the shape of a star. Consequently, each source has to send each client one individual copy of its application events. In the thesis:

We mainly consider shared-overlays that are used by all the members of a single group to multicast the group's application events. There are two main types of shared-overlays: shared-trees and shared-subgraphs.

A *shared-tree* is a connected subgraph that is acyclic (definition 21). The advantage of a shared-tree is the reduced average stress-level on the clients that are multicasting events. However, the drawback is that a shared-tree cannot minimize the pair-wise latencies between the clients. Despite this drawback, a shared-tree is much more desirable than a source-tree because using one shared-tree leads to less resources spent when the number of clients is large.

A *shared-subgraph* is a connected subgraph that may contain cycles (definition 20). The assumption is that a client does not multicast previously seen events from the same source. Such shared-subgraphs are also used to multicast events, but since they are likely to contain loops there is an increase in the bandwidth consumption. On the other hand, a shared-subgraph has the advantage that it decreases the average pair-wise latencies, and also provides a level of data delivery resilience. However, in our evaluations, we found that shared-trees are able to compete with shared-subgraphs in terms of the overlay diameter.

16.4 Group updates: Static groups vs. dynamic groups

One major difference that affects which overlay construction algorithm to choose, is whether the client groups in the application are static or dynamic:

- ***Static groups*** are created at some point, and remain unchanged until their group-sessions are over. Generally, a static group's overlay network must be created by overlay construction algorithms that construct overlays from scratch.

- ***Dynamic groups*** are created at some point and may be changed continuously until their group-sessions are over. A dynamic group's overlay network should use overlay construction algorithms that update an existing overlay based on incoming join and leave requests.

The following sections introduce issues related to how static groups are created, and how dynamic groups are updated. The issues are tightly linked to latency requirements, bandwidth issues, group-sizes, etc (table 16.1). These issues are discussed and configuration options are suggested to address them. The configurations are specific algorithms from the group communication simulations and experiments in the thesis. In principle, the algorithms used for static groups may be applied for dynamic groups, but we only discuss the most suited algorithms that we identified.

16.5 Static groups of clients

Spanning or Steiner overlay algorithms should be used in scenarios of static groups of clients, because they construct overlay networks from scratch.

This is because spanning and Steiner overlay algorithms are the two groups of algorithms that construct overlays from scratch the best (chapter 9 , 10 , 11). Dynamic overlay algorithms are only discussed for dynamic groups because they are not designed to create overlay networks from scratch, but rather include or remove single nodes based on group membership.

The basic requirement to the overlay construction algorithm is to construct an overlay with a diameter that is inside the application's user satisfaction bounds. However, a user satisfaction bound may or may not exist, depending on the appliction type. This and other issues are addressed in the following sections. The questions that we try to address are:

- Are there hard-to-meet low-latency requirements?
- Is there a latency bound based on user satisfaction studies?
- Is there a bandwidth-intensive data-flow?
- Is there a large client-base (with super-nodes)?

For each question, we give recommendations to what algorithms should be used.

16.5.1 Hard-to-meet low-latency requirements

When there are hard-to-meet low-latency requirements, a minimum-diameter overlay should be used to distribute the events.

As discussed in section 16.3, the minimum-latency approach is to build a source-tree for each node. However, each source tree is now shaped as a star because application layer overlay networks are inherently complete networks (full meshes) of shortest paths.

A minimum-diameter tree algorithm (definition 32 and 46), on the other hand, ensures that the clients in the group are within the least possible maximum pair-wise latency possible. A minimum-diameter tree in a complete graph made of shortest paths has the shape of a star, where one or two nodes connect to the remaining nodes. This approach puts extra stress on the nodes in the middle of the star. Therefore, it is only suitable if these nodes have enough available bandwidth and computational power to multicast all the data-streams. A minimum-diameter tree can be approximated by using Dijkstra's SPT algorithm from the source node that has the minimum eccentricity.

Chapter 9 also identified the close-to-optimal $O(n^3)$ mddl-OTTC and md-OTTC to be suited spanning-tree algorithms for minimizing the diameter. mddl-OTTC has degree-limits, but is otherwise equal to md-OTTC. mddl-OTTC constructs a spanning-tree on a complete graph in less than 100 milliseconds when the group size is less than 170.

The drawback of a minimum-diameter tree is that the pair-wise latencies are relatively high, due to a star-shaped tree with long low-latency links to the middle. A minimum-diameter subgraph algorithm is another option (definition 83). It aims to create a subgraph that has a diameter equal to the input graph, but has to obey a total cost bound. In the thesis, we evaluated several configurable subgraph algorithms that approximated minimum-diameter subgraphs. For example, kIT(mddl-OTTC,$k = 2$) and kDL(mddl-OTTC,$k = 80$) created subgraphs of a very low diameter, which also yielded a lower average pair-wise latency.

16.5.2 User-satisfaction latency bounds

If there exist a latency bound for the application type based on user satisfaction studies, it is possible to use a diameter bounded overlay construction algorithm.

In chapter 9, we identified the close-to-optimal $O(n^3)$ dl-OTTC and OTTC to be suited spanning-tree algorithms that include a diameter bound. dl-OTTC is degree-limited, but when the degree-limit is set to infinite, the dl-OTTC is equal to the OTTC algorithm. Although they are both $O(n^3)$, they are relatively fast, and in our experiments, they constructed spanning-trees on a complete graph in less than 50 milliseconds when the group size was less than 170. As we can see, dl-OTTC and OTTC are slightly faster than md-OTTC and mddl-OTTC, but construct spanning-trees of a slightly higher diameter.

A drawback with diameter bounded algorithms is that there is no guarantee that they find a solution in which all group nodes are spanned within the bound. In our simulations, we applied dynamic relaxation of the diameter bounds, which proved to work very well.

16.5.3 Bandwidth-intensive data-flows

The overlay construction algorithm should be degree-limited when a group is used to distribute bandwidth-intensive flows.

The degree-limit on a given node ensures that the number of edges connected to it in the overlay, does not increase beyond its degree-limit. The degree-limits should be calculated based on the available bandwidths on the group nodes and the bandwidth in the flows the overlay network is intended to distribute.

In our evaluations, we found a number of degree-limited overlay construction algorithms that were suitable for constructing low-latency overlays. The drawback of having degree-limits is that they lead to an increased overlay diameter, because overlay construction algorithms are forced to use non-optimal edges during the construction. However, in our experiments, we found that that dl-OTTC and mddl-OTTC constructed trees that could compete with degree unlimited algorithms when the degree-limit was uniformly set to 10 for all nodes. A degree-limit of 5 resulted in higher overlay diameters, and made it more difficult to meet the diameter bounds. In cases of non-uniform degree-limits, it is important that nodes located centrally among the group members have a higher degree-limit. This is because central nodes are more likely to be inner nodes that utilize the entire degree capacity.

16.5.4 Large client-base with super-nodes

In a scenario with a large client-base, it is advantageous to search for well-placed super-nodes among the clients.

Super-nodes can be used as Steiner-points for Steiner-tree and -subgraph algorithms that yield overlays of lower diameters than spanning-tree and -subgraph algorithms. In chapter 8, we identified that we can apply the k-Median core-node selection algorithm to find super-nodes (chapter 7). The super-nodes can be regular clients participating in the interaction, or servers and proxies made available by the application provider. The application for super-nodes is to add them as Steiner-points to complete group graphs, and then let Steiner-tree or -subgraph algorithms construct overlays on the group graphs. The thesis proposed that the number of Steiner-points to add should be determined by equation 8.1. However, the general observation is that more is better, but the execution time increases.

In chapter 10, we identified a number of close-to-optimal Steiner-tree spanning-heuristics. These heuristics construct a spanning-tree on the input graph, and then prune leaf Steiner-points. In the investigations, we found that sdl-OTTC and smddl-OTTC are excellent options for constructing close-to-optimal Steiner-trees. sdl-OTTC is identical to dl-OTTC except for the pruning of Steiner-points (similarly for smddl-OTTC and mddl-OTTC). In addition, we found that their execution times are only slightly higher due to the added Steiner-points from equation 8.1.

16.6 Dynamic groups of clients

Dynamic overlay algorithms should be used in scenarios of dynamic groups of clients, because they are designed to insert and remove nodes from existing overlays.

In dynamic groups, clients may freely join and leave in the course of the group's life-time. Hybrid cases between static and dynamic groups may also occur. For example, if a group has an initial number of members at the time it is created (greater than one), it should be created as a static group. Then, clients may join and leave the group, and as such be treated as a dynamic group in the rest of the group's life-time.

For dynamic groups, the group reconfiguration time is very important, and in that respect, dynamic overlay algorithms are suitable because they are generally very fast and simplistic.

However, this may lead to sub-optimal overlays that have too high diameter. In the following sections we try to address questions regarding the performance of dynamic overlay algorithms. Speceficially, we try to answer these questions:

- Is there a small dynamic group of clients?
- Is there a large dynamic group of clients?
- What are the effects of reconfiguring large groups?

For each question, we give recommendations to what algorithms should be used.

16.6.1 Small dynamic groups

In application scenarios in which there is a small dynamic group of clients, a dynamic overlay algorithm should be used to create the group's overlay network.

Chapter 12 evaluated dynamic tree algorithms, and the results (figure 12.16) showed that when the degree limit was high (= 10), the algorithm DA(RTR-MDDL, I-MDDL) was able to compete with the close-to-optimal smddl-OTTC for group sizes smaller than 80. For larger group sizes, the dynamic-tree algorithm yielded tree-diameters that were about 30 % higher. Furthermore, when the degree-limit was decreased (= 5) they only yielded satisfactory tree-diameters for group sizes smaller than 40. From this, we see that the evaluated dynamic-tree algorithms are unfit when the degree-limit is low and the group size is large. However, they do work very well with a higher degree-limit and smaller group sizes.

Chapter 13 evaluated dynamic subgraph algorithms, and we identified in figure 13.5 that kDA(RTR-P-MC, I-MC,$k = 3$) yielded subgraphs of a consistently low diameter, throughout the group-size range (< 170). The drawback is that cyclic subgraphs add more network traffic, thus increasing the network-cost in terms of the bandwidth consumption. Figure 13.11 showed that the added total cost was around three-fold from the close-to-optimal minimum-cost tree.

16.6.2 Large dynamic groups

In application scenarios in which there is a large dynamic group of clients, a reconfiguration dynamic overlay algorithm should be used to create the group's overlay network.

What a large group size is, is a source for discussion, but group sizes of more than 1000 is quite large. Applications that have such large dynamic groups of clients are harder to achieve, because it is hard for a dynamic overlay algorithm to maintain an overlay that yield a sufficiently low overlay-diameter. On the one hand, the algorithms should be fast enough to enable real-time group changes, but on the other hand, they should also have enough logic to keep the overlay-diameter low.

Many of the proposed dynamic-tree and subgraph algorithms are very fast because they only consider smaller fractions of the existing group-overlay when they insert and remove nodes from it. However, we did see that many dynamic overlay algorithms cannot maintain overlays of a consistently low diameter when the group sizes increase.

Chapter 14 proposed two reconfiguration algorithms, one for trees (RDA) and one for subgraphs (kRDA). The dynamic reconfiguration tree algorithm applies a dynamic-tree algorithm until the tree-diameter has increased beyond a user-set bound. Then, a diameter-reducing Steiner-tree algorithm reconfigures the entire tree. In the evaluation, we showed that a total reconfiguration is often necessary to keep a consistently low diameter when the group size increases. Hence, for applications with dynamically changing larger groups, reconfiguration algorithms are most likely necessary. Similar results were shown for the subgraph reconfiguration algorithm.

16.6.3 Effects of reconfiguring larger groups

When very large groups are reconfigured from scratch due to broken diameter bounds, it means that many of the existing communication paths are changed.

In the thesis, we defined the stability of the overlay network as the number of edges that change across reconfigurations. It is not desirable to have a highly unstable overlay network in which the communication paths change rapidly. However, it is also not desirable to have an overlay network with a diameter larger than what is acceptable for a sufficient user satisfaction.

A valid option is to migrate the overlay network updates slowly, such that the overlay keeps the "old" communication paths until the "new" paths are setup. Moreover, when overlay networks are highly unstable, it is likely that highly connected cyclic subgraphs are more suitable to multicast the data. This is because cyclic subgraphs have multiple paths to the clients and therefore create data redundancy, this way the reconfigurations are less likely to disrupt all the active paths.

16.7 Influences on the overlay construction

The performance of an overlay construction algorithm is influenced by many factors. The following sections discuss a few of the more important influences and how they may be addressed. The questions we aim to answer are these:

- How is the diameter influenced by algorithms and parameters?
- How is the overlay construction time influenced by algorithms and parameters?
- What about the relay penalty in overlay networks?

Each of the questions are addressed in separate sections.

16.7.1 Influences on the diameter

How successful an overlay construction algorithm is that opts for a low diameter overlay network, is influenced by:
1) the degree-limits on the centrally located nodes,
2) whether there are available Steiner-points (super-nodes),
3) the maximum number of edges it is allowed to add to the overlay.

Figure 11.9(b) evaluated how the diameter produced by mddl-OTTC and the subgraph algorithm kIT varies depending on the degree-limit, the number of edges, and whether the input graph has Steiner-points. We observed that Steiner-points are more important to obtain a low diameter, than doubling the number of edges from a tree to a cyclic subgraph. However, the most important factor is the degree-limit. A high degree-limit enables the diameter optimizing heuristics to construct overlays of a smaller diameter, because the degree capacity centrally is increased. When well-placed Steiner points are added by k-Median, this further enhances the algorithm's performance, also because the degree capacity centrally is increased.

From these results, we see that it is difficult to obtain a low tree-diameter when the degree-limits are low. What a low degree-limit is, can be discussed, but in our tests we consider a degree-limit of 5 or less, to be low. If the degree-limits have to be low, then well-placed Steiner-points should be added to increase the degree capacity. If super-nodes are unavailable, the last option is to add more edges and create a highly connected cyclic subgraph. However, this may also increase the bandwidth consumption on some nodes.

16.7.2 Influences on the overlay construction time

The overlay construction time is influenced by:
1) whether the overlay construction algorithm is centralized or distributed,
2) if the network information is available at the time of construction,
3) the overlay construction algorithm's time complexity,
4) the size of the input graph,
5) the constraints applied to the construction.

In the thesis, we evaluated the overlay construction time for each of the overlay construction algorithms we tested. They were tested in a group communication scenario where they constructed or updated a group's overlay network whenever it was changed.

We only evaluated centralized algorithms because distributed algorithms are too slow to handle our latency bounds. Distributed algorithms suffer mostly because the end-to-end latencies in the Internet currently are very high. Moreover, the network information must be present when the overlay construction starts. If the network information has to be gathered in parallel or before the overlay construction, it will increase the construction time greatly.

The overlay consetruction time is heavily influenced by the time complexity of the algorithm executing it. In our tests we did not apply any algorithm with a time complexity of more than $O(n^3)$. Most of these algorithms were found to be sufficiently fast in our evaluated group size range $(0-170)$. However, the size of the input graph also influences the overlay construction time. In our scenario, we use application layer graphs that are complete graphs. In such complete graphs the number of edges quickly grows to be very many. We addressed this issue by using edge-pruning algorithms that reduce the number of edges by carefully removing the bad edges. Edge-pruning mainly has an effect on an algorithm that has a time complexity which depends on the edge set size (chapter 8).

Constrained overlay construction algorithms may face situations where they are unable to continue the construction, due to very strict constraints. In our studies, we used a dynamic relaxation procedure, which relaxed the constraints whenever an algorithm faced such situations. Generally, this is a better approach than abandoning the constraints altogether, but the overlay construction time may increase.

16.7.3 Influences of the relay penalty in overlay networks

When nodes in the network are used to relay data, it adds to the path latency.

This is especially the case if overlay multicast is used to distribute audio, video and 3D streaming. These rich-media flows often require the streams to be processed on each node, for example, synchronization and mixing of audio/video. If this processing occurs on each node in the overlay network, it may severly increase the pair-wise latencies among the nodes. Such la-

Application configuration					Algorithm configuration			
Application members[1]	Membership change	Subgroup sizes[1]	Latency bound	Bandwidth requirements	Overlay construction algorithm	Degree-limit	Diameter bound	Steiner-points
–	static	–	low	low	smddl-OTTC	high	–	eq 8.1
–	static	–	low	high	smddl-OTTC	low	–	eq 8.1
–	dynamic	small	low	low	DA(RTR-MDDL,I-MDDL)	high	–	eq 8.1
–	dynamic	small	low	high	kRDA(RTR-P-MC,I-MC,smddl-OTTC,$k = 2$,low)	low	low	eq 8.1
–	dynamic	medium	low	low	RDA(RTR-MDDL,I-MDDL,smddl-OTTC,low)	high	low	eq 8.1
–	dynamic	medium	low	high	kRDA(RTR-P-MC,I-MC,smddl-OTTC,$k = 2$,low)	low	low	eq 8.1
–	dynamic	large	low	low	kRDA(RTR-P-MC,I-MC,smddl-OTTC,$k = 3$,low)	high	low	eq 8.1
–	dynamic	large	low	high	kRDA(RTR-P-MC,I-MC,smddl-OTTC,$k = 3$,low)	low	low	eq 8.1

[1] "–" indicates that the application configuration is independent of the algorithm configuration.

Table 16.3: Application configuration and corresponding proposed algorithm configuration.

tencies must therefore be taken into account when overlay networks are built, by, for example, adding a hop-based relay latency or penalty.

If each hop in an overlay network has a relay penalty, it makes the hop-diameter of an overlay network an important metric. Diameter optimizing overlay construction algorithms that enforce degree-limits, are likely to have a larger hop-diameter than degree unlimited algorithms. The lower the degree-limit is, the larger the hop-diameter is. For example, if the degree-limit is 2, the only possible shape of an overlay network is a line (snake), which has a hop-diameter equal to the number of nodes minus one.

16.8 Summary of the main points

There are many possible application scenarios for the research done in the thesis. In this chapter, we tried to highlight some of the options that application developers have when they design distributed interactive applications.

The choices are more than what is covered in the thesis, but we limited the discussions to this context. Some of the choices we discussed are related to: Centralized vs. distributed architectures and algorithms, membership management of one or multiple groups, group communication using source-overlays or shared-overlays, static or dynamic groups and which algorithms to apply, and finally algorithmic performance and influences.

Table 16.3 summarizes a few of the configurations that we discussed, with a specific overlay construction algorithm for each application configuration. We observe that the better algorithm choices vary more when the membership is dynamic. Furthermore, with dynamic groups the algorithm configurations quickly become important as the application configuration is more demanding. We do believe that the proposed algorithm configurations are good options for supporting applications with the given set of requirements and configurations.

Chapter 17

Conclusions and future work

Through the research in the thesis, we wanted to address many of the unsolved issues that are found in the application-area distributed interactive applications. These applications aim to support real-time interaction across the Internet, where the main issue is related to distributing the interactions through Internet paths with sufficiently low latencies. The studies we performed included many different group communication techniques that were all evaluated towards their applicability to distributed interactive applications, and the evaluations revealed several algorithms that are suitable.

In the following sections, we provide brief summaries of our findings in the thesis. For each section, we repeat the initial goals from section 1.4 and discuss what we achieved. First, section 17.1 gives a brief summary of the thesis. Section 17.2 summarizes the contributions within each of the four goals we presented in section 2.5. Section 17.3 contains critical assessments of the research conducted in the thesis. Section 17.4 then discusses a few open issues and proposals for future work. Finally, section 17.5 concludes the thesis with brief final remarks.

17.1 Summary of the thesis

Applications that aim to support real-time interactions between multiple clients across the Internet have strict latency requirements related to distributing these client interactions. Such distributed interactive applications are currently hard to achieve mainly due to the limitations that the current Internet yields. The limitations are related to high end-to-end latencies and limited bandwidth capacities. Therefore, there is a need to handle these limitations by applying techniques that aim to identifiy Internet paths with sufficiently low latencies and sufficiently high bandwidths such that distributed interactive applications are enabled.

A focal point of the thesis was to enable distributed interactive applications to support group dynamics, where clients may join and leave ongoing sessions of real-time interaction. From this basic point, we described more application specific requirements:

A client must be able to join a group of clients and in a timely manner start the real-time interaction. Joining and removing clients from groups should be done sufficiently fast to enable continuous real-time interaction. Furthermore, application events must be distributed to all clients such that their latency bounds are met. Finally, Internet resources are scarce, therefore, the events must also be distributed efficiently and cost-effective.

From these requirements, we identified four main research areas that are important to address in order to support the requirements of real-time interactions across the Internet:

- A ***membership management*** must ensure that clients are able to join and leave ongoing sessions of real-time interaction, in a timely fashion (section 5.2).
- A ***resource management*** must ensure that well-placed nodes are found that yield low latencies to groups of clients, such that they are available for management tasks (chapter 7).
- An ***overlay network management*** must ensure that clients are configured in overlay networks that yield sufficiently low-latencies for real-time interaction (chapter 8 to 14).
- A ***network information management*** must ensure that Internet path latencies between the interacting clients are available and sufficiently accurate (chapter 6, section 7.7 and chapter 15).

For each of the research areas, we evaluated different techniques that in varying degrees addressed the problems at hand. For the evaluations, we first implemented the techniques in a group communication simulator and also a real system for group communication that we ran on PlanetLab. Then, through exstensive simulations and real-world experiments, we were able to identify efficient techniques that addressed the requirements very well.

For the membership management, we identified that a centralized approach is most fitting to scenarios of multiple dynamic sub-groups of clients. We also deduced that centralized graph algorithms have the quickness that is desirable to identify nodes and configure Internet paths. For the resource management, we identified suited core-node selection algorithms that find well-placed nodes that yield low latencies to groups of clients. Moreover, quite a few overlay construction algorithms were identified as suitable for an overlay network management. In particular, dynamic overlay algorithms that insert and remove single nodes from overlay networks. Finally, we evaluated latency estimation techniques and found them to be accurate enough to be used by centralized graph algorithms.

The main conclusion is that by applying the techniques we identified as suitable, they will enable distributed interactive applications to support group dynamics. In addition, the techniques we identified are able to handle the Internet path latencies in a satisfactory manner and enable time-dependent event-distribution.

17.2 Contributions and the four goals

The following sections provide brief summaries of our findings and contributions for each of the goals that are summarized in section 2.7.

17.2.1 Membership management

The initial goal for the membership management in distributed interactive applications was:

1) Identify techniques that enable an efficient and timely membership management of multiple dynamic subgroups of clients.

The need for a membership management arises when a distributed interactive application desires to manage many dynamic groups of clients in the same application.

In section 5.2, 3 membership managment variations were evaluated towards expected latency in a membership change request, and the expected consistency of the variation. We deduced that a centralized membership management approach is most fitting to a scenario in which there are multiple dynamic subgroups of clients. This is mainly because a membership management that supports real-time joining and removing of clients is hard to achieve if membership decisions have to be made distributedly through Internet paths with rather high latencies. However, the efficiency of centralized approaches are heavily influenced by the location of the central entity that executes the management tasks. Therefore, we found that a centralized membership management should be executed on a node that yield low latencies to the group of clients that is managed.

17.2.2 Resource management

The initial goal for the resource managment in distributed interactive applications was:

2) Identify techniques that enable a resource management to identify nodes in the (application) network that yield low pair-wise latencies to groups of clients.

The need to identify well-placed core-nodes that yield low pair-wise latencies to groups of clients is motivated by the desire to execute management tasks on them, or use them as (passive) relay nodes that forward application events. For example, a core-node may be used to execute the centralized membership management (section 17.2.1), which is able to manage multiple dynamic sub-groups of clients in a timely fashion when the core-node is centrally located.

Chapter 7 introduced and evaluated several core-node selection algorithms that used latency to search for well-placed core-nodes. We conducted both simulation studies and experiments on PlanetLab to evaluate the performance of the algorithms. In the evaluations, we found that the k-Center algorithm was the better among the tested algorithms, and is appropriate to identify multiple core-nodes in a network. These core-nodes are made available for single core-node selection algorithms that select a core-node to be the membership manager for a group of clients. For single core-node selection, it was found that k-Median was the algorithm that found core-nodes that yielded the lowest maximum one-way latencies to clients in its group.

17.2.3 Overlay network management

The initial goal for the overlay network managment in distributed interactive applications was:

3) Identify techniques that find Internet paths with low pair-wise latencies among clients in a group, and configure them for event-distribution of real-time client interaction.

In the thesis, we introduced the overlay network management to include two main types of techniques: graph manipulation techniques and overlay construction techniques. These techniques must cooperate such that an overlay is created for a group of clients that yield sufficiently low pair-wise latencies. This overlay may then be used by the clients to multicast time-dependent application-events.

Graph manipulation algorithms

Chapter 8 evaluated graph manipulation algorithms, and the main goal was to:

Identify graph manipulation algorithms that manipulate a group's complete graph such that it enables the overlay construction algorithm to execute fast and build desirable low-latency overlay networks.

An important observation is that application layer overlay networks are complete graphs of shortest paths. Therefore, a group of clients, defined by the membership management, is also a complete graph. Running overlay construction algorithms on such complete graphs is potentially time consuming, and very often due to the large number of edges.

In our research, we focused on two types of graph manipulation algorithms. One was to reduce the complete group graphs to only include the better links such that the overlay construction executes faster. The other was to identify well-placed core-nodes in the application network that are used as Steiner-points (non-member-nodes) in the group graphs for Steiner-tree or Steiner-subgraph algorithms.

For the graph reduction, we proposed two edge-pruning algorithms that take as input a complete group graph and then constructs a new group graph with a smaller edge set that still yield low pair-wise latencies. This reduced group graph is used as input to an overlay construction algorithm. The results showed that edge-pruning significantly reduced the execution time of tree-algorithms that have time complexities that are dependent on the edge set size (figure 8.2(a)). The penalty lies in the quality of the overlay, because the tree-algorithms produce the lowest diameter when using a fully meshed input graph (figure 9.5). For example, the diameter suffered on average less than 20 % when edge-pruning algorithms reduced the edge set up to 80 %, compared to a fully meshed graph (see figure 9.5).

To take advantage of Steiner overlay algorithms, we evaluated how to apply core-node selection algorithms to identify well-placed super-nodes that yield low pair-wise latencies to the clients in the application network. The evaluations in chapter 8 and also later chapters, showed that when such super-nodes are added to input graphs as Steiner-points, they enable Steiner-tree and -subgraph algorithms to construct overlay networks that yield a lower diameter (figure 11.15), with no particular increase in their execution time.

Overlay construction algorithms

Chapter 9 through 14 evaluated a wide range of overlay construction algorithms. The algorithms in these chapters are very different, and they are often applied in varying application scenarios to address specific problems. However, in an application scenario in which clients may join and leave groups of clients that are engaging in real-time interaction, we aimed to:

Identify techniques that in a timely fashion can configure overlay networks for event-distribution based on incoming join and leave requests, such that the overlay:
1) yields a low maximum pair-wise latency,
2) yields a low average pair-wise latency,
3) does not add unreasonable cost to the network,
4) is reconfigured such that its stability is acceptable,
5) does not contain nodes with an unreasonable high stress level.

We focused on two types of overlay construction algorithms; algorithms that construct connected trees, and algorithms that construct connected subgraphs (definition 21 and 22). A tree does not contain cycles, while a subgraph may contain cycles. We evaluated all the algorithms using a simulator that mimiced group communication in which clients join and leave groups of clients. For each join and leave request, a well-placed node (central entity) reconfigured the overlays for the affected groups using an overlay construction algorithm. The overlay algorithms we evaluated were spanning-tree and -subgraph, Steiner-tree and -subgraph and dynamic-tree and -subgraph algorithms. Dynamic algorithms insert and remove single nodes from overlays, while the spanning and Steiner algorithms always reconfigure the entire overlay from scratch. For each of the algorithms, we found alternatives that regulated the stress levels on the nodes in the overlays. In addition, it was only the highly connected subgraphs that added a high network cost.

During the evaluations of the tree algorithms, we identified that dynamic-tree algorithms are very fast and keep the trees stable in terms of the number of edge-changes in a reconfiguration. However, they only produce low diameter trees for smaller group sizes. For larger group sizes, they are not able to maintain a tree that yields a low diameter. Moreover, we found that the Steiner-tree algorithms yield close-to-optimal diameter trees throughout the group range $(0-170)$ that are better than spanning-tree algorithms due to the available Steiner-points. Therefore, we devised an algorithm that took as input a dynamic-tree algorithm, a Steiner-tree algorithm, and a configurable upper diameter bound. If this upper diameter bound is violated, a total reconfiguration using the Steiner-tree algorithm is initiated, otherwise it is the dynamic-tree algorithm that inserts and removes nodes. The goal was to preserve the high stability and low execution time of the dynamic tree algorithms, and still manage to create group-trees with a consistently low diameter. The results showed that with a well-set diameter bound, we were able to achieve the target metrics.

In the evaluations of the subgraph algorithms, we found that highly connected subgraphs yield on average a lower average pair-wise latency than a connected tree. This is to be expected, because there are several paths between the nodes in a subgraph, while in a tree there is only one path. The diameter of the subgraphs were not reduced as much compared to a connected tree. Among the subgraph algorithms, we found that the dynamic-subgraph algorithms are fast and yield stable overlays (small number of edge-changes). The spanning- and Steiner-subgraph algorithms are slower and result in more unstable overlays (high number of edge-changes), but they yield a lower diameter. Therefore, we devised a similar algorithm for reconfiguration of subgraphs that included a configurable integer k, which described the connectivity of the subgraphs. The results showed that we were able to achieve a higher stability and lower execution time, and also keep the diameter and the average pair-wise latency very low. See chapter 9 through 14 for further details and algorithm names.

17.2.4 Network information managment

The initial goal for the network information management in distributed interactive applications was:

4) Identify techniques that are able to obtain accurate all-to-all Internet path latencies.

We give brief summaries of our findings when we evaluated the latency estimation techniques Vivaldi [34] and Netvigator [113]. The first evaluation focused on the accuracy of the latency estimates compared to all-to-all ping measurements. The second evaluation investigated the penalty of applying latency estimates to core-node selection algorithms compared to real latencies. Finally, the third evaluation investigated the penalty when latency estimates are used to construct overlay networks. All the experiments were performed on PlanetLab.

Chapter 6 evaluated the latency estimation accuracy and found that Netvigator yields estimations that are very close to the real measured values. The estimations from Vivaldi were not as good, but we still found them to be usable in our application scenario. Vivaldi's advantages are that it is very easy to deploy in a peer-to-peer fashion, and it handles membership dynamics. Netvigator, on the other hand, needs an infrastructure (landmark nodes). However, a game provider that controls a number of proxies could use Netvigator as it is the better alternative.

Section 7.7 measured the penalties involved when latency estimates are used by centralized core-node selection algorithms to find well-placed core-nodes. The results showed that when Netvigator's latency estimates are used, the k-Median algorithm finds close-to-optimal core-nodes. Vivaldi's latency estimates did not enable k-Median to find the optimal core-nodes, but we did find the core-nodes to be sufficiently well-placed. The main conclusion was that both Vivaldi and Netvigator provide good enough latency estimates to be used for core-node search.

Chapter 15 evaluated a centralized membership management architecture for dynamically changing subgroups in a distributed interactive application. The architecture included latency estimation techniques, core-node selection algorithms and tree algorithms that together made it possible to build multicast trees optimized for the tree diameter. We found that when latency estimates are used to create trees, this estimated diameter is in all cases lower than the diameter that is actually achieved from all-to-all ping measurements. Applications that rely on latency estimation techniques are forced to make conservative assumptions about a group-overlay's diameter whenever the diameter bounds for the application are hard.

All in all, we found that the latency estimates enabled the centralized algorithms to find suitable core-nodes and build overlays that were good enough to be used by a distributed interactive application (see chapter 2 for detailed application requirements).

17.3 Critical assessments of the research

The evaluations of the techniques and algorithms in the thesis were conducted based on results from a group communication simulator and a real-world system for group communication executed on PlanetLab. However, although these experiments and simulations provide a good foundation for analysis and reaching conclusions, it is clear that there are several other ways to perform them. In addition, it is possible to perform similar research and experiments. In this section, we therefore critially asses our research.

17.3.1 Limitations of the PlanetLab tests

The PlanetLab tests were executed on 100 nodes. The limited number was due to end-to-end reachability problems among the PlanetLab nodes. A network with more nodes would enhance the credibility of the latency estimation evaluations. For example, it is unclear how well the latency estimation techniques perform when the number of nodes increases to thousands of nodes. Moreover, due to the small number of PlanetLab nodes, the group sizes were too small to come to conclusive evaluations on how the latency estimates influence the core-node selection algorithms and the overlay construction algorithms.

We think it is likely that the small number of PlanetLab nodes enabled the latency estimation techniques to perform better than they would with thousands of nodes.

17.3.2 Limitations of the group communication simulations

The group communication simulator used flat BRITE graphs with 1000 nodes for the tests. These graphs were transformed to complete application layer graphs made from shortest paths. Hierarchical graphs should also be tested to see their influence on the algorithms. The BRITE graphs are independent of real-world size, but in our tests, we fit the network into an area of 100 × 100 milliseconds. This was an attempt to simulate an area on earth that had day-light at the same time. For example, it is likely that the size fits for Europe and North-America.

The groups in distributed interactive applications are normally defined by an area-of-interest management. In the evaluations, the number of nodes in the application network is 1000, and the group sizes grow to around 170. However, large-scale MMOGs today often have thousands of clients playing their games at the same time. Therefore, the group sizes in these games are likely to grow beyond our maximum simulated group size of 170. Larger group sizes may influence the performance of the overlay construction algorithms. In particular, the diameter of the overlays may increase beyond the latency bounds when the group sizes increase and sub-optimal algorithms have to obey pre-set degree-limitations.

In chapter 8, we identified super-nodes in the application network that were made available as Steiner-points for Steiner overlay algorithms. When Steiner-points are included to group graphs, we do not consider the extra work-load on each of these Steiner-points, such that Steiner points may be in several group-overlays at the same time. In a real-world system, the load on each node should be monitored, and it is likely that more super-nodes need to be found than what was identified in our evaluations.

17.3.3 Limitations of the group dynamics model

The group dynamic in both the PlanetLab tests and simulations was based on a Zipfian distribution (section 2.6.3) and made the nodes join and leave groups of clients continuously. Although this is a well-known model, it is clear that other group dynamics models may influence the performance of some of the algorithms. For example, a group dynamic model that may be more fitting is one that captures the real-world passing of day and night with the consequent shift in client mass.

We were unable to obtain a group dynamics trace from a real-world system, where one valid system would be a large-scale MMOG. Such a trace may have influenced the performance of the dynamic-tree and dynamic-subgraph algorithms in chapter 12 and 13.

17.3.4 Limitations of the algorithms

All the graph algorithms were implemented in a sequential approach using C++, parts of the standard template library (STL) [112] for C++, and the Boost graph library [117]. It is unlikely that our implementation of the algorithms is all the way optimal, such that the execution times are to be considered as guides and not absolute facts. In addition, it is believed that a parallel implementation of many of the algorithms is possible and would further decrease their execution times.

We tested a wide range of graph algorithms and came to a conclusion for each addressed problem to which of them were the better ones. However, we do not claim that we were able to test and identify the best possible algorithms for each problem addressed. What we do believe is that the algorithms we identified as suitable are able to compete, to a large extent, with other better unidentified algorithms.

17.4 Open issues and future work

Group communication research for real-time interaction is a relatively new research-area. In our work, we addressed many different problems and identified a wide range of suitable techniques. Nevertheless, there are still many unaddressed problems and techniques that may be better than the ones we have identified.

17.4.1 The four research areas

In our work, we addressed 4 main research areas, and these are now discussed towards their open issues and future work.

For the *membership management*, it is still not entirely clear how a centralized membership management behaves in a real-world situation where possibly multiple nodes are membership managers for specific groups. Additional testing on PlanetLab is likely to shed light on this behavior, and how it affects the performance in terms of the membership execution latency and average load.

For the *resource management*, there are issues related to how many core-nodes that is a suitable number in a given network. In our evaluations, we found that a limited number in the fifties is sufficient with a Zipfian group dynamics model, a flat BRITE network of 1000 nodes and size 100 × 100 milliseconds. A different experiment setup in terms of group dynamics model and network size may influence this number, and also the performance of the core-node selection algorithms.

In the *overlay network management*, we tested graph manipulation algorithms and overlay construction algorithms. One issue that remains for future work is how the graph manipulation algorithms and the overlay construction algorithms should cooperate, in sequence or in parallel. The overlay construction algorithms should also be verified for performance through theoretical studies. In addition, the execution times of the algorithms are a matter of discussion and future work, due to possible sub-optimal implementations. Many of the overlay construction algorithms are also likely to benefit from caching the node-eccentricities, diameter, etc, of the subgraphs between reconfigurations. We expect this to reduce the execution times of several algorithms.

In the *network information management*, we evaluated the impact of possibly in-accurate latency estimates on centralized graph algorithms. One issue that is not addressed is the influence of unstable or fluctuating link latencies. Moreover, it is not clear how the overlay networks built from latency estimates perform when they are used to multicast application events.

17.4.2 Additional testing and research

We addressed 4 specific goals and identified suitable techniques for each of the goals. Although we performed both group communication simulations and group communication experiments on PlanetLab, it is still unclear how our identified techniques will perform in real-world applications. Therefore, additional testing of selected techniques should be conducted in the Internet to measure their performance.

Moreover, it is possible to use the better performing techniques and create specialized overlay multicast protocols that are usable in more specific application settings. Investigating specific application types and their demands, combined with specific algorithms in an overlay multicast protocol may prove to work well.

For the research done in the thesis, the prime concern was achieving a sufficiently low latency between the interacting parties. However, the load in a real-world system is likely to influence the latency due to varying performance of many of the investigated techniques. Varying the load on each node, the link bandwidth, etc, is necessary to understand how different techniques cope with these challenges.

17.5 Final remarks

In the course of the thesis, we were able to identify suitable techniques to enable distributed interactive application architectures that have multiple dynamic sub-groups of clients. We found that centralized solutions are often desirable due to rather high end-to-end latencies in the Internet. A sufficiently low latency is the paramount requirement if real-time client interactions and group dynamics are to be supported across the Internet.

Group communication research is a relatively new research area. Therefore, the work done in the thesis is to be considered as a starting point for additional testing, reviews and studies. The critical assessments and future work showed that there are still unsolved issues related to both identifying better techniques and verifying the results through studies with larger networks and real-world applications. Consequently, there is room for further improvements, but our framework is already valuable for a number of areas and provides suitable starting points for further studies.

Bibliography

[1] A. Abdalla, N. Deo, and P. Gupta. Random-tree diameter and the diameter-constrained MST. Technical Report CS-TR-00-02, University of Central Florida, Orlando, FL, USA, 2000.

[2] A. Adams, J. Nicholas, and W. Siadak. Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised). RFC 3973 (Experimental), January 2005.

[3] Sudhir Aggarwal, Madhura Limaye, Arun Netravali, and Krishan Sabnani. Constrained diameter steiner trees for multicast conferences in overlay networks. In *Proceedings of the First International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks (QSHINE'04)*, pages 262–271, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Ehud Aharoni and Reuven Cohen. Restricted dynamic Steiner trees for scalable multicast in datagram networks. *IEEE/ACM Transactions on Networking*, 6(3):286–297, June 1998.

[5] Dewan Tanvir Ahmed, Shervin Shirmohammadi, and Jauvane Oliveira. Improving gaming experience in zonal mmogs. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 581–584, New York, NY, USA, 2007. ACM.

[6] Tawfig Alrabiah and Taieb Znati. SELDOM: A simple and efficient low-cost, delay-bounded, online multicasting. In *IFIP Conference on High Performance Networking (HPN)*, pages 95–113, 1998.

[7] Tawfig Alrabiah and Taieb Znati. Delay-constrained, low-cost multicast routing in multimedia networks. *J. Parallel Distrib. Comput.*, 61(9):1307–1336, 2001.

[8] Tawfig Alrabiah and Taieb F. Znati. Low-cost, delay-bounded multicasting. In *International Conference on Telecommunications (onTEL)*, June 1999.

[9] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *ACM Symposium of Operating Systems Principles (SOSP)*, pages 131 – 145, 2001.

[10] Ramnik Bajaj, C. P. Ravikumar, and Suresh Chandra. Distributed delay constrained multicast path setup algorithm for high speed networks. In *Proceedings of International Conference on High Performance Computing*, pages 438–442, December 1997.

[11] A. Ballardie. Core Based Trees (CBT version 2) Multicast Routing – Protocol Specification –. RFC 2189 (Experimental), September 1997.

[12] Tony Ballardie, Paul Francis, and Jon Crowcroft. Core based trees (CBT). In *ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 85–95, 1993.

[13] Suman Banerjee and Bobby Bhattacharjee. Analysis of the NICE application layer multicast protocol. Technical Report UMIACSTR 2002-60 and CS-TR 4380, Department of Computer Science, University of Maryland, College Park, June 2002.

[14] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 205–217, 2002.

[15] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[16] Fred Bauer and Anujan Varma. ARIES: A rearrangeable inexpensive edge-based online Steiner algorithm. In *Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 361–368, 1996.

[17] Paul Beskow, Knut-Helge Vik, Carsten Griwodz, and P. Halvorsen. Latency reduction by dynamic core selection and partial migration of game state. *Proceedings of NetGames'08, Worcester, MA, USA*, oct 2008.

[18] Kadaba Bharath-Kumar and Jeffrey M. Jaffe. Routing to multiple destinations in computer networks. *Communications, IEEE Transactions on [legacy, pre - 1988]*, 31(3):343–351, March 1983.

[19] R. Bless and K. Wehrle. IP Multicast in Differentiated Services (DS) Networks. RFC 3754 (Informational), April 2004.

[20] B.C. Brookes. The derivation and application of the Bradford-Zipf distribution. *Journal of Documentation*, 24(4):247–265, 1968.

[21] E. Brosh. Approximation and heuristic algorithms for minimum delay application-layer multicast trees, 2003.

[22] Marc Bui, Franck Butelle, and Christian Lavault. A distributed algorithm for constructing a minimum diameter spanning tree. *J. Parallel Distrib. Comput.*, 64(5):571–577, 2004.

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

[24] Chris Chambers, Wu chang Feng, Sambit Sahu, and Debanjan Saha. Measurement-based characterization of a collection of on-line games. *In the Proceedings of the 5th ACM SIGCOMM Workshop on Internet measurement, Berkeley, CA, USA*, pages 1–14, October 2005.

[25] Yatin Chawathe. Scattercast: an adaptable broadcast distribution framework. *ACM/Springer Multimedia Systems*, 9(1):104–118, 2003.

[26] Yang-Hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 1–12, 2000.

[27] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, November 2005.

[28] Clip2. The bitterrent protocol specification. http://www.bittorrent.org/beps/-bep_0003.html.

[29] Clip2. The gnutella protocol specification v0.4. http://www9.limewire.com/developer/-gnutella_protocol_0.4.pdf.

[30] Doru Constantinescu. *Overlay multicast networks: Elements, architectures and performance*. PhD thesis, Karlskrona. Blekinge Institute of Technology, 2007.

[31] Manuel Costa, Miguel Castro, Antony Rowstron, and Peter Key. Pic: Practical internet coordinates for distance estimation. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 178–187, Washington, DC, USA, 2004. IEEE Computer Society.

[32] Yi Cui, Baochun Li, and Klara Nahrstedt. On achieving optimized capacity utilization in application overlay networks with multiple competing sessions. In *Proceedings of the Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 160–169, June 2004.

[33] Yi Cui and Klara Nahrstedt. High-bandwidth routing in dynamic peer-to-peer streaming. In *P2PMMS'05: Proceedings of the ACM workshop on Advances in peer-to-peer multimedia streaming*, pages 79–88, New York, NY, USA, 2005. ACM Press.

[34] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 15–26, 2004.

[35] Vasilios Darlagiannis, Andreas Mauthe, and Ralf Steinmetz. Sampling cluster endurance for peer-to-peer based content distribution networks. In *SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, January 2006.

[36] Marcus Poggi de Aragão and Renato Fonseca F. Werneck. On the implementation of MST-based heuristics for the Steiner problem in graphs. In *ALENEX*, pages 1–15, 2002.

[37] Stephen Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei. An architecture for wide-area multicast routing. In *ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 126–135, 1994.

[38] Narsingh Deo and Ayman Abdalla. Computing a diameter-constrained minimum spanning tree in parallel. *Lecture Notes in Computer Science*, 1767:17–??, 2000.

[39] Hrishikesh Deshpande, Mayank Bawa, and Hector Garcia-Molina. Streaming live media over a peers. Technical Report 2002-21, Stanford University Database Group, August 2002.

[40] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, / 2000.

[41] A. Elmokashfi. Scalable, decentral qos verification based on prediction techniques and active measurements. Master's thesis, Blekinge Institute of Technology, Blekinge, Sweden, January 2007.

[42] A. Elmokashfi, M. kleis, and A. Popescu. Netforecast: A delay prediction scheme for provider controlled networks. In *IEEE Globecom*, November 2007.

[43] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification. RFC 2362 (Experimental), June 1998.

[44] Michalis Faloutsos. *The Greedy, the Naive, and the optimal multicast routing: from Theory to Internet Protocols*. PhD thesis, University of Toronoto, Graduate Department of Computer Science, 1999.

[45] Gang Feng, Tak-Shing, and Peter Yum. Efficient multicast routing with delay constraints. *Int. J. Commun. Sys.*, 12:181–195, January 1999.

[46] B. Fenner. IANA Considerations for IPv4 Internet Group Management Protocol (IGMP). RFC 3228 (Best Current Practice), February 2002.

[47] M. Fleury, A. C. Downton, and A. F. Clark. Performance metrics for embedded parallel pipelines. *IEEE Transactions on Parallel and Distributed Systems*, 11(11):164+, 2000.

[48] Sally Floyd. Tools for bandwidth estimation, 2008.

[49] Geoffrey Fox and Shrideep Pallickara. The Narada event brokering system: Overview and extensions. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 353–359, 2002.

[50] P. Francis, S. Ratnasamy, R. Govindan, and C. Alaettinoglu. Yoid project, 2000.

[51] Wikipedia: The free encyclopedia. First-person shooter games, 2008.

[52] Wikipedia: The free encyclopedia. History of graph theory, 2008.

[53] Wikipedia: The free encyclopedia. Real-time-strategy games, 2008.

[54] Wikipedia: The free encyclopedia. Role-playing games, 2008.

[55] Wikipedia: The free encyclopedia. Zipf's law, 2008.

[56] Funcom. Age of conan. http://www.ageofconan.com/, May 2008.

[57] A. Ganesh, A. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols, 2003.

[58] M.T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. John Wiley and Sons, 2002.

[59] Carsten Griwodz and Pål Halvorsen. The fun of using TCP for an MMORPG. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 1–7. ACM Press, May 2006.

[60] Carsten Griwodz, Knut-Helge Vik, and Pål Halvorsen. Multicast tree reconfiguration in distributed interactive applications. In *International Conference (NIME)*, pages 1219 – 1223, January 2006.

[61] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *In Proceedings of the SIGCOMM Internet Measurement Workshop (IMW 2002)*, Marseille, France, November 2002.

[62] Anupam Gupta. Steiner points in tree metrics don't (really) help. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 220–227, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[63] Nicholas Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *In proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[64] Refael Hassin. Approximation schemes for the restricted shortest path problem. *Math. Oper. Res.*, 17(1):36–42, 1992.

[65] Refael Hassin and Asaf Levin. Minimum restricted diameter spanning trees. *Discrete Appl. Math.*, 137(3):343–357, 2004.

[66] Oliver Heckmann, Michael Piringer, Jens Schmitt, and Ralf Steinmetz. On realistic network topologies for simulation. In *International workshop on models, methods and tools for reproducible network research*, pages 28–32, 2003.

[67] David A. Helder and Sugih Jamin. Banana tree protocol, an end-host multicast protocol. Technical Report CSE-TR-429-00, Department of Electrical Engineering and Computer Science, University of Michigan, 2000.

[68] J. M. Ho, D. T. Lee, C. H. Chang, and C. K. Wong. Bounded diameter minimum spanning trees and related problems. In *SCG '89: Proceedings of the fifth annual symposium on Computational geometry*, pages 276–282, New York, NY, USA, 1989. ACM Press.

[69] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. North-Holland, Amsterdam, Netherlands, 1992.

[70] International Telecommunication Union (ITU-T). One-way Transmission Time, ITU-T Recommendation G.114, 2003.

[71] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James O'Toole Jr. Overcast: Reliable multicasting with an overlay network. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 197–212, 2000.

[72] Xiaohua Jia. A distributed algorithm of delay-bounded multicast routing for multimedia applications in wide area networks. *IEEE/ACM Transactions on Networking*, 6(6):828–837, December 1998.

[73] Cheng Jin, Qian Chen, and Sugih Jamin. Inet: Internet topology generator. Technical Report CSE-TR-433-00, Dept. of EECS, September 2000.

[74] Dulal C. Kar. Internet path characterization using common internet tools. *J. Comput. Small Coll.*, 18(4):124–132, 2003.

[75] Ayse Karaman and Hossam S. Hassanein. Core-selection algorithms in multicast routing - comparative and complexity analysis. *Computer Communications*, 29(8):998–1014, 2006.

[76] Maleq Khan, Gopal Pandurangan, and V. S. Anil Kumar. A simple randomized scheme for constructing low-weight k-connected spanning subgraphs with applications to distributed algorithms. *Theor. Comput. Sci.*, 385(1-3):101–114, 2007.

[77] Samir Khuller, Balaji Raghavachari, and Neal Young. Balancing minimum spanning and shortest path trees. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 243–250, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[78] Samir Khuller and Uzi Vishkin. Biconnectivity approximations and graph carvings. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 759–770, New York, NY, USA, 1992. ACM.

[79] Joshua D. Knowles and David W. Corne. A comparison of encodings and algorithms for multiobjective minimum spanning tree problems. In *In Proceedings of the 2001 Congress on Evolutionary Computation (CEC'01)*, pages 544–551. IEEE Press, 2001.

[80] Guy Kortsarz and David Peleg. Approximating shallow-light trees. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 103–110, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[81] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 282–297, New York, NY, USA, 2003. ACM.

[82] L. Kou, George Markowsky, and Leonard Berman. A fast algorithm for Steiner trees. *Acta Informatica*, 15:141–145, June 1981.

[83] M. Kwon and S. Fahmy. Topology-aware overlay networks for group communication, 2002.

[84] D. Leonard and D. Loguinov. Turbo king: Framework for large-scale internet delay measurements. In *In Proceedings of IEEE INFOCOM*, April 2008.

[85] Jin Liang and Klara Nahrstedt. Dagstream: Locality aware and failure resilient peer-to-peer streaming. In *SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, January 2006.

[86] Leslie S. Liu and Roger Zimmermann. ACTIVE: A low latency P2P live streaming architecture. In *SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, January 2005.

[87] Leslie S. Liu and Roger Zimmermann. Immersive peer-to-peer audio streaming platform for massive online games. In *International Conference (NIME)*, 2006.

[88] Chor Ping Low and Xueyan Song. On finding feasible solutions for the delay constrained group multicast routing problem. *IEEE Transactions on Computers*, 51(5):581–588, 2002.

[89] Eng Keong Lua, Timothy Griffin, Marcelo Pias, Han Zheng, and Jon Crowcroft. On the accuracy of embeddings for Internet coordinate systems. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet measurement*, pages 1–14, New York, NY, USA, 2005. ACM.

[90] Emily Larson Luebke. *k-Connected Steiner Network Problems*. PhD thesis, University of North-Carolina at Chapel Hill, Department of Operations Research, 2002.

[91] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: Universal topology generation from a user's perspective. Technical Report BUCS-TR-2001-003, Computer Science Department, Boston University, April 2001.

[92] Eva Milkova. The minimum spanning tree problem: Jarnik's solution in historical and present context. *Electronic Notes in Discrete Mathematics*, 28:309–316, March 2007.

[93] B. Moret and H. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree, 1994.

[94] J. Moy. MOSPF: Analysis and Experience. RFC 1585 (Informational), March 1994.

[95] S. C. Narula and C. A. Ho. Degree-constrained minimum spanning trees. *Computers and Operations Research*, 7:239–249, 1980.

[96] Katia Obraczka and Peter Danzig. Finding low-diameter, low edge-cost, networks, 1997.

[97] Wei Tsang Ooi. Dagster: contributor-aware end-host multicast for media streaming in heterogeneous environment. In *SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, pages 77–90, January 2005.

[98] P2PSim. P2PSim King Data Set, 2008.

[99] Wladimir Palant, Carsten Griwodz, and Pål Halvorsen. Evaluating dead reckoning variations with a multi-player game simulator. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 20–25, May 2006.

[100] Mehrdad Parsa, Qing Zhu, and J. J. Garcia-Luna-Aceves. An iterative algorithm for delay-constrained minimum-cost multicasting. *IEEE/ACM Transactions on Networking*, 6(4):461–474, 1998.

[101] Dimitrios Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An application level multicast infrastructure. In *In Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 49–60, 2001.

[102] Sriram Raghavan, G. Manimaran, and C. Siva Ram Murthy. Algorithm for delay-constraint low-cost multicast tree construction. *Computer Communications*, 21(18):1693–1706, December 1998.

[103] Sriram Raghavan, G. Manimaran, and C. Siva Ram Murthy. A rearrangeable algorithm for the construction delay-constrained dynamic multicast trees. *IEEE/ACM Transactions on Networking*, 7(4):514–529, 1999.

[104] Günther R. Raidl and Bryant A. Julstrom. Greedy heuristics and an evolutionary algorithm for the bounded-diameter minimum spanning tree problem. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 747–752, 2003.

[105] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. CAN: A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[106] V.J. Rayward-Smith and A.R. Clare. The computation of nearly minimal Steiner trees in graphs. *International Journal of Mathematical Education in Science and Technology*, 14(1):8pp, 1983.

[107] Reza Rejaie and Antonio Ortega. PALS: peer-to-peer adaptive layered streaming. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 153–161, 2003.

[108] Wilson Rivera. Scalable parallel genetic algorithms. *Artif. Intell. Rev.*, 16(2):153–168, 2001.

[109] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[110] H. Salama, Y. Viniotis, and D. Reeves. An efficient delay constrained minimum spanning tree heuristic. In *Proceedings of the 5th International Conference on Computer Communications and Networks (ICCCN)*, October 1996.

[111] Hussein F. Salama, Douglas S. Reeves, and Yannis Viniotis. Evaluation of multicast routing algorithms for real-time communication on high-speed networks. *IEEE Journal on Selected Areas in Communications*, 15:332–345, 1997.

[112] SGI. C++ Standard Template Library, 2008.

[113] Puneet Sharma, Zhichen Xu, Sujata Banerjee, and Sung-Ju Lee. Estimating network proximity and latency. *SIGCOMM Comput. Commun. Rev.*, 36(3):39–50, 2006.

[114] S. Shi and J. Turner. Routing in overlay multicast networks. In *Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002.

[115] Sherlia Y. Shi, Jon Turner, and Marcel Waldvogel. Dimensioning server access bandwidth and multicast routing in overlay networks. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 83–92, June 2001.

[116] Yunxi Sherlia Shi. *Design of overlay networks for Internet multicast*. PhD thesis, Washington University, Sever Institute of Technology, Department of Computer Science, Sain Louis, USA, August 2002.

[117] Jeremy G. Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[118] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, 2001.

[119] D. Stopp and B. Hickman. Methodology for IP Multicast Benchmarking. RFC 3918 (Informational), October 2004.

[120] H. Takahashi and A. Matsuyama. An approximate solution for the steiner trees in graphs. *Intl. J. Math. Japonica*, 6(1):573–577, 1980.

[121] Tandberg. Tandberg video-conferencing systems, 2008.

[122] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems Principles and Paradigms*. Prentice Hall, 2002. ISBN 0-13-088893-1.

[123] David Thaler and Chinya V. Ravishankar. Distributed center-location algorithms. *IEEE Journal of Selected Areas in Communications*, 15(3):291–303, 1997.

[124] Duc A. Tran, Kien A. Hua, and Tai T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming. In *Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2003.

[125] Knut-H. Vik, C. Griwodz, and P. Halvorsen. Dynamic group membership management for distributed interactive applications. In *IEEE Conference on Local Computer Networks (LCN)*, pages 141–148, 2007.

[126] Knut-H. Vik, C. Griwodz, and P. Halvorsen. On the influence of latency estimation on dynamic group communication using overlays. In *Multimedia Computing and Networking (MMCN), San Jose, CA, USA*, January 2009.

[127] Knut-H. Vik, P. Halvorsen, and C. Griwodz. Constructing low-latency overlay networks: Tree vs. mesh algorithms. In *Local Computer Networks (LCN), Montreal, Canada*, November 2008.

[128] Knut-Helge Vik. Game state and event distribution using proxy technology and application layer multicast. In *ACM International Multimedia Conference (ACM MM)*, pages 1041–1042, 2005.

[129] Knut-Helge Vik, C. Griwodz, and P. Halvorsen. Applicability of group communication for increased scalability in mmogs. *Proceedings of NetGames'06, Singapore*, oct 2006.

[130] Knut-Helge Vik, Paal Halvorsen, and Carsten Griwodz. Evaluating steiner tree heuristics and diameter variations for application layer multicast. *Accepted for publication in Computer Networks on Complex Computer and Communication Networks*, November 2008.

[131] Knut-Helge Vik, Paal Halvorsen, and Carsten Griwodz. Multicast tree diameter for dynamic distributed interactive applications. In *Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1597–1605, April 2008.

[132] Knut-Helge Vik and Sirisha R. Medidi. QuaSAR: Quality of Service aware Source initiated Ad-hoc Routing. In *1st IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, October 2004.

[133] Jürgen Vogel, Martin Mauve, Volker Hilt, and Wolfgang Effelsberg. Late join algorithms for distributed interactive applications, 2004.

[134] Stefan Voss. Steiner's problem in graphs: heuristic methods. *Discrete Appl. Math.*, 40(1):45–72, 1992.

[135] D. Waitzman, C. Partridge, and S.E. Deering. Distance Vector Multicast Routing Protocol. RFC 1075 (Experimental), November 1988.

[136] I-Lin Wang. *Shortest paths and multicommodity network flows*. PhD thesis, Georgia Institute of Technology, May 2003.

[137] Wenjie Wang, David A. Helder, Sugih Jamin, and Lixia Zhang. Overlay optimizations for end-host multicast, October 2002.

[138] Zheng Wang and Jon Crowcroft. Bandwidth-delay based routing algorithms. In *In Proceedings of IEEE GlobeCom*, pages 2129–2133, 1995.

[139] Bernard M. Waxman. Routing of Multipoint Connections. *IEEE Journal of Selected Areas in Communications*, 6, December 1988.

[140] Bernard M. Waxman. Dynamic Steiner tree problem. *SIAM J. Discrete Math.*, 4:364–384, 1991.

[141] Clay Wilson. Avatars, virtual reality technology, and the u.s. military: Emerging policy issues, April 2008.

[142] P. Winter. Steiner problem in networks: a survey. *Netw.*, 17(2):129–167, 1987.

[143] B. Wong, A. Slivkins, and E. Sirer. Meridian: A lightweight network location service without virtual coordinates, 2005.

[144] Bang Ye Wu and Knu-Mao Chao. *Spanning Trees and Opitmization Problems*. Chapman and Hall/CRC, 2004.

[145] Lih-Chyau Wuu, Long Song Lin, and Shing-Chyi Shiao. Constructing delay-bounded multicast trees in computer networks. *Journal of Information Science and Enfineering*, 17(3):507–524, 2001.

[146] Beverly Yang and Hector Garcia-Molina. Comparing hybrid peer-to-peer systems. In *International Conference on Very Large Databases (VLDB)*, pages 561–570, 2001.

[147] A. Young, C. Jiang, M. Zheng, A. Krishnamurthy, L. Peterson, and R. Wang. Overlay mesh construction using interleaved spanning trees, March 2004.

[148] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *In Proceedings of IEEE INFOCOM*, pages 594–602, 1996.

[149] Ben Y. Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, 1996.

[150] Q. Zhu and W. Dai. Delay bounded minimum Steiner tree algorithms for performance-driven routing, 1993.

[151] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA, 2001. ACM.

Appendix A

List of Publications

Much of the research done in the thesis has been published in proceedings of various international conferences and journals. The list of publications from the thesis is presented below, along with a short description of the author contributions.

Authors: Knut-Helge Vik.
Title: Game State and Event Distribution using Proxy Technology and Application Layer Multicast.
Published: In Proceedings of ACM Multimedia 2005, Doctoral Symposium Singapore, November 2005.
Author Contributions: Knut-Helge Vik was the main contributor to the paper.

Authors: Carsten Griwodz, Knut-Helge Vik and Pål Halvorsen.
Title: Multicast Tree Reconfiguration in Distributed Interactive Applications.
Published: 2nd IEEE International Workshop on Networking Issues in Multimedia Entertainment (NIME'06), Short Paper, Las Vegas, NV, USA, January 2006.
Author Contributions: Carsten Griwodz was the main contributor to this paper. Knut-Helge Vik contributed parts of the related work and discussions.

Authors: Knut-Helge Vik, Carsten Griwodz and Pål Halvorsen.
Title: Applicability of Group Communication for Increased Scalability in MMOGs.
Published: 5th Workshop on Network and System Support for Games (NetGames), Full Paper, Singapore, October 2006.
Author Contributions: Knut-Helge Vik developed the simulator, algorithms and performed all simulations. In addition, Knut-Helge Vik was the main contributor to the writing. Carsten Griwodz and Pål Halvorsen mainly contributed through discussions.

Authors: Knut-Helge Vik, Carsten Griwodz and Pål Halvorsen.
Title: Dynamic Group Membership Management for Distributed Interactive Applications.
Published: 32nd Annual IEEE Conference on Local Computer Networks (LCN), Regular Paper, Dublin, Ireland, October 2007.
Author Contributions: Knut-Helge Vik developed the simulator, algorithms and performed all simulations. In addition, Knut-Helge Vik and Carsten Griwodz were the main contributors to the writing. Pål Halvorsen mainly contributed through discussions.

Authors: Knut-Helge Vik, Pål Halvorsen and Carsten Griwodz.
Title: Multicast Tree Diameter for Dynamic Distributed Interactive Applications.
Published: 27th International IEEE Conference on Computer Communications (INFOCOM), Phoenix (AZ), USA, April 2008.
Author Contributions: Knut-Helge Vik developed the simulator, algorithms and performed all simulations. In addition, Knut-Helge Vik was the main contributor to the writing. Pål Halvorsen and Carsten Griwodz mainly contributed through discussions.

Authors: Knut-Helge Vik, Pål Halvorsen and Carsten Griwodz.
Title: Constructing Low-Latency Overlay Networks: Tree vs. Mesh Algorithms.
Published: 33rd Annual IEEE Conference on Local Computer Networks (LCN), Regular Paper, Montreal, Quebec, Canada, October 2008.
Author Contributions: Knut-Helge Vik developed the simulator, algorithms and performed all simulations. In addition, Knut-Helge Vik was the main contributor to the writing. Pål Halvorsen and Carsten Griwodz mainly contributed through discussions.

Authors: Knut-Helge Vik, Pål Halvorsen and Carsten Griwodz.
Title: Evaluating Steiner tree heuristics and diameter variations for Application Layer Multicast.
Published: Accepted for publication in the special issue of Computer Networks: The International Journal of Computer and Telecommunications Networking on Complex Computer and Communication Networks, Elsevier, November 2008.
Author Contributions: Knut-Helge Vik developed the simulator, algorithms and performed all simulations. In addition, Knut-Helge Vik was the main contributor to the writing. Pål Halvorsen and Carsten Griwodz mainly contributed through discussions.

Authors: Knut-Helge Vik, Carsten Griwodz and Pål Halvorsen.
Title: On The Influence Of Latency Estimation On Dynamic Group Communication Using Overlays.
Published: 16th Annual Multimedia Computing and Networking (MMCN), San Jose, California, USA, January 2009.
Author Contributions: Knut-Helge Vik developed the simulator, algorithms and performed all simulations. In addition, Knut-Helge Vik was the main contributor to the writing. Pål Halvorsen and Carsten Griwodz mainly contributed through discussions.

Authors: Paul Beskow, Knut-Helge Vik, Carsten Griwodz and Pål Halvorsen

Title: Latency Reduction by Dynamic Core Selection and Partial Migration of Game State.

Published: 7th Workshop on Network and System Support for Games (NetGames), Full Paper, Worcester, Massachusets, USA, October 2008.

Author Contributions: Paul Beskow and Knut-Helge Vik were the main contributors to the paper. Paul Beskow was responsible for the writing. Carsten Griwodz and Pål Halvorsen contributed through discussons.

Authors: Paul Beskow, Knut-Helge Vik, Carsten Griwodz and Pål Halvorsen

Title: The Partial Migration of Game State and Dynamic Server Selection to Reduce Latency.

Published: Accepted for publication in the special issue of Multimedia Tools and Applications (MTAP): Massively Multiuser Online Gaming Systems and Applications, Springer, 2009.

Author Contributions: Knut-Helge Vik and Paul Beskow were the main contributors to the paper. Paul Beskow and Knut-Helge Vik were responsible for the writing. Carsten Griwodz and Pål Halvorsen contributed through discussons.

Appendix B

List of Abbreviations

This is a list of abbreviations that are used in the thesis. The list contains the investigated algorithms and problem definitions. It also contains a range of metric-, application- and protocol-abbreviations.

aCL	add-Core-links
aCLO	add-Core-Links-Optimized
ADH	Average distance heuristic
ALM	Application layer multicast
BCT	Bounded compact-tree heuristic
BDDLMST	Bounded diameter degree-limited minimum-cost spanning-tree problem
bddlo-DNH	Bounded diameter degree-limited optimized distance network heuristic
bddlo-SPH	Bounded diameter degree-limited optimized shortest path heuristic
bddlr-DNH	Bounded diameter degree-limited randomized distance network heuristic
bddlr-SPH	Bounded diameter degree-limited randomized shortest path heuristic
BDDLSMT	Bounded diameter degree-limited Steiner minimum-cost tree problem
BDMST	Bounded diameter minimum-cost spanning-tree problem
bdo-DNH	Bounded diameter optimized distance network heuristic
bdo-SPH	Bounded diameter optimized shortest path heuristic
bdr-DNH	Bounded diameter randomized distance network heuristic

bdr-SPH	Bounded diameter randomized shortest path heuristic
BDRB	Bounded diameter residual balanced tree problem
BDSMT	Bounded diameter Steiner minimum-cost tree problem
BFS	Breadth-first search
br-DNH	Bounded radius distance network heuristic
br-MST	Bounded radius minimum-cost spanning tree
br-SPH	Bounded radius shortest path heuristic
brdl-DNH	Bounded radius degree-limited distance network heuristic
brdl-MST	Bounded radius degree-limited minimum-cst spanning tree
brdl-SPH	Bounded radius degree-limited shortest path heuristic
BRDLMST	Bounded radius degree-limited minimum-cost spanning-tree problem
BRDLSMT	Bounded radius degree-limited Steiner minimum-cost tree problem
BRITE	Boston university representative Internet topology generator
BRMST	Bounded radius minimum-cost spanning-tree problem
BRSMT	Bounded radius Steiner minimum-cost tree problem
CDF	Cumulative distribution function
CNLS	Closest neighbor loss significance
CT	Compact-tree heuristic
d-MST	Degree-limited minimum-cost spanning tree problem
d-SMT	Degree-limited Steiner minimum-cost tree
d-SPT	Degree-limited shortest path tree problem
DA	Dynamic-tree algorithm
db	Diameter bound
DFS	Depth-first search
DHT	Distributed hash table

dl-aCLO	degree-limited add-Core-links-optimized
dl-BL	degree-limited Best-links
dl-DNH	Degree-limited distance network heuristic
dl-MST	Degree-limited minimum-cost spanning tree
dl-OTTC	Degree-limited one-time tree construction
dl-RGH	Degree-limited randomized greedy heuristic
dl-SPH	Degree-limited shortest path seuristic
dl-SPT	Degree-limited shortest path tree
dl	Degree-limit
DNH	Distance network heuristic
DST	Dynamic Steiner-tree problem
ecc	Eccentricity
FPS	First-person shooter game
I-CN	Insert center node
I-MC-MDDL	Insert miminum cost and minimum diameter degree-limited edge
I-MC	Insert miminum cost degree-limited edge
I-MDDL	Insert miminum diameter degree-limited edge
I-MRDL	Insert miminum radius degree-limited edge
IBD-MC	Insert bounded diameter minimum-cost edge
IBR-MC	Insert bounded radius minimum-cost edge
ICMP	Internet control message protocol
IP	Internet protocol
ITR-MC	Insert try reconfiguration and minimum-cost edge
ITR-MDDL	Insert try reconfiguration and minimum diameter degree-limited edge
kBL	k-Best links

kBL	k-Best-links
kCL	k-Core links
kCT	k-Combined tree constructions
*k*DA	*k*-Dynamic-subgraph algorithm
kDL	k-Diameter links
kHV	Knut-Helge Vik
kIT	k-Iterative tree constructions
kLL	k-Long links
kPT	k-Parallel tree constructions
*k*RDA	*k*-Reconfigure-subgraph dynamic algorithm
md-DNH	Minimum diameter distance network heuristic
md-OTTC	Minimum diameter one-time tree construction
md-SPH	Minimum diameter shortest path heuristic
mddl-DNH	Minimum diameter degree-limited distance network heuristic
mddl-OTTC	Minimum diameter degree-limited one-time tree construction
mddl-SPH	Minimum diameter degree-limited shortest path heuristic
MDDL	Minimum diameter degree-limited spanning-tree problem
MDST	Minimum diameter spanning-tree problem
MMOG	Massively multiplayer online game
MRDL	Minimum radius degree-limited spanning-tree problem
MRST	Minimum radius spanning-tree problem
MST	Minimum-cost spanning tree
N-DST	Non-reconfigurable dynamic Steiner-tree problem
NP	Nondeterministic polynomial time
OSI	Open systems interconnection

OTTC	One-time tree construction
P2P	Peer-to-peer
R-DST	Reconfigurable dynamic Steiner-tree problem
R-MC	Remove miminum cost degree-limited edge
R-MDDL	Remove miminum diameter degree-limited edge
RDA	Reconfigure-tree dynamic algorithm
RGH	Randomized greedy heuristic
RK	Remove keep as non-member node
RPG	Role-playing game
RRL	Relative rank loss
RS-MC	Remove search miminum cost degree-limited edge
RS-MDDL	Remove search miminum diameter degree-limited edge
RTR-MC	Remove try reconfiguration and minimum-cost edge
RTR-MDDL	Remove try reconfiguration and minimum diameter degree-limited edge
RTR-P-MC	Remove try reconfiguration, prune and minimum-cost edge
RTR-P-MDDL	Remove try reconfiguration, prune and search for minimum diameter degree-limited edge
RTR-PS-MC	Remove try reconfiguration, prune and search for minimum-cost edge
RTR-PS-MDDL	Remove try reconfiguration, prune and search for minimum diameter degree-limited edge
RTS	Real-time strategy game
RTT	Round-trip time
s-SPT	Steiner shortest path tree
sdl-OTTC	Steiner degree-limited one-time tree construction
sdl-RGH	Steiner degree-limited randomized greedy heuristic
sdl-SPT	Steiner degree-limited shortest path tree

smddl-OTTC	Steiner minimum diameter degree-limited one-time tree construction
SMT	Steiner minimum-cost tree
SPH	Shortest path heuristic
SPT	Shortest path tree
Steiner-MDDL	Steiner minimum diameter degree-limited spanning-tree problem
Steiner-MDST	Steiner minimum diameter spanning-tree problem
Steiner-MRDL	Steiner minimum radius degree-limited spanning-tree problem
Steiner-MRST	Steiner minimum radius spanning-tree problem
TCP	Transmission control protocol
TTL	Time-to-live