Stefan Wieser, MComp. Bakk.techn.

FLOCKS: A DYNAMIC, SELF-ORGANIZING OVERLAY NETWORK FOR MULTIMEDIA DISTRIBUTION

DISSERTATION

zur Erlangung des akademischen Grades
Doktor der technischen Wissenschaften

Studium: Informatik

Alpen-Adria-Universität Klagenfurt
Fakultät für Technische Wissenschaften

1. Begutachter:	Univ.-Prof. Dipl.-Ing. Dr. Laszlo Böszörmenyi
	Institut für Informationstechnologie
	Alpen-Adria-Universität Klagenfurt, Austria
2. Begutachter:	Dr. Eng. Pier Luca Montessoro
	Department of Electrical, Managerial and Mechanical Engineering
	University of Udine, Italy

November 2012

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende wissenschaftliche Arbeit selbstständig angefertigt und die mit ihr unmittelbar verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für wissenschaftliche Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die während des Arbeitsvorganges gewährte Unterstützung einschließlich signifikanter Betreuungshinweise ist vollständig angegeben.

Die wissenschaftliche Arbeit ist noch keiner anderen Prüfungsbehörde vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Declaration of honour

I hereby confirm on my honour that I personally prepared the present academic work and carried out myself the activities directly involved with it. I also confirm that I have used no resources other than those declared. All formulations and concepts adopted literally or in their essential content from printed, unprinted or Internet sources have been cited according to the rules for academic work and identified by means of footnotes or other precise indications of source.

The support provided during the work, including significant assistance from my supervisor has been indicated in full.

The academic work has not been submitted to any other examination authority. The work is submitted in printed and electronic form. I confirm that the content of the digital version is completely identical to that of the printed version.

I am aware that a false declaration will have legal consequences.

Unterschrift/Signature:

Klagenfurt, 19. November 2012

Contents

List of Tables

List of Figures

Acknowledgements

I am deeply grateful to my advisor Lazslo Böszörmenyi for his tireless efforts during the past years, which are too countless to list. From providing me anything necessary to perform my research, over his constant availability and readiness for discussion, no matter the date and time, to his feedback, criticism and support. The liberty of research direction given at the beginning, the weekly meetings during the course of the thesis, and the insistence on finishing it in a timely fashion - he went far beyond anything expected, and I do appreciate that greatly. Without his help, none of this would have been possible.

I am also grateful to Pier Luca Montessoro, who went out of his way to provide me with input on my work, interesting discussions and valuable feedback. I would also like to thank him for his readiness to peer review my thesis.

In addition, I would like to thank my colleague, Mario Taschwer, for his input throughout my research, and Manfred del Fabro for his help and motivation, especially during the final stages of my thesis.

Finally, I am thankful to my family, especially my parents, for their continuous support that has made my studies possible.

Abstract

This thesis investigates flexible self-organizing overlay networks for multimedia delivery - networks that are dynamically built on existing infrastructure to support the preferences of applications using them. For example, consider a large sports event with several concurrent competitions, in which hundreds of thousands visitors produce and consume videos with their mobile devices. These devices connect to each other to exchange videos. However, as the interests of their owners change, the overlay must adapt as well: existing connections must be replaced with connections to other relevant devices, such as devices of other visitors with similar interests. Deciding which devices should connect to each other, while also optimizing network performance is not trivial. This thesis attempts to solve that issue by using a flexible and scalable self-organizing overlay located between an application and its underlying network, which optimizes itself based the application's interest in content and network quality. A special focus is placed on including metrics relevant to multimedia delivery, such as jitter and bottleneck bandwidth, in the optimization. As these metrics cannot be predicted by a third node, established approaches such as matchmaking cannot be used. Therefore, a new approach based on a novel interest-property concept is introduced, which provides the flexibility of traditional matchmaking approaches, but also supports optimization based on parameters that can only be determined once two potential neighbours connect. In addition, a distributed way for nodes to estimate the level of service they are likely to receive to another node is introduced. A "Flocks" prototype overlay that implements the concepts and techniques described in this thesis is developed and evaluated. Results show that Flocks optimize even large overlays based on content and network metrics quickly and with modest overhead.

CHAPTER 1 Introduction

This chapter motivates the topic of this thesis, provides practical and theoretical views on the issues it addresses, and places it in context of related research works. Finally, it outlines the main contributions of this thesis, and concludes with a brief description of its structure.

1.1 Motivation

In this thesis, techniques for flexible and scalable self-organizing overlay networks are discussed. A special emphasis is placed on suitability for multimedia delivery as required, for example, by information services for social events.

Such large social events present a challenge for multimedia delivery. Consider that during such events many visitors and participants carry hand-held devices with them at all times. Common devices of today are often able to produce and display high quality video content, which brings exciting opportunities: Videos can be taken by anyone, and for anyone. With that, it is no longer necessary to rely only on general, high level news reports made by professional teams. Instead, visitors who record videos become news reporters themselves, and can supply large amounts of raw footage to anyone interested.

Recent advances in video summarisation have enabled the automated creation of news, tailored to the constraints and preferences of each individual interested in them. As a result, anyone visiting such an event has a wealth of information at her fingertips: By watching videos taken by other visitors, users can keep track of their favourite athlete during a race, or of their favourite actor at an awarding event, etc. In a large event with many concurrent activities, visitors can keep track of other activities with timely delivery of short, summarized videos of recent interesting situations.

Supporting social events by rich content puts high requirements on the underlying network: Large amounts of data may be produced by any device, and must simultaneously be made available to all users that need to receive it (both of which are referred to as network nodes, or simply nodes from here on). Therefore, the scalability and adaptability of the underlying network play key roles in improving the overall quality of service provided to applications and - finally - the end users.

Overlay networks (ONs) - virtual networks built on top of existing, hard wired connections - offer strategic distribution of such data. However, while networks for efficient data distribution have been built, it remains a difficult task to construct overlay networks for situations in which high flexibility, scalability and performance are required. Large social events, for example, give rise to several questions, to which finding a good answer is not trivial:

- How can we keep an overlay network scalable to a large number of nodes simultaneously producing and consuming video?
- How can we organize the overlay network to support several different Quality of Service (QoS) metrics concurrently?
- How can we incorporate devices with different operational constraints?
- How can we combine constraints imposed by the user with network constraints?

Creating and maintaining an overlay network that fulfils all of the aforementioned requirements is a challenge by itself. Supporting a large number of nodes with individual requirements and goals, which differ from node to node makes this problem even more complex.

Self-organization offers an interesting approach to this issue, as it avoids centralized control and with it the scalability and reliability implications inherent to utilizing centralized components. Nodes in self-organizing systems arrange themselves, using only locally available information to guide their decisions. For example, a node can simply pick neighbours to which it has the lowest delay. The resulting optimization of the entire network is an emergent property that every node has contributed to - without requiring a global view or even having explicit knowledge of the bigger picture they achieved. However, as finding the "correct" local decision for a well-performing self-organizing system can be a difficult task, such systems are also time-consuming and error-prone to develop.

Therefore, this thesis introduces a robust concept for creating and maintaining self-organizing overlay networks, which are scalable, flexible and general enough to be applicable for a wide range of different use cases. A prototype, called the "Flocks overlay" is developed, which implements the techniques discussed in this work. During the evaluations, the need for estimating the quality of service received on a path within the overlay is identified, prompting the design of "QoS maps", a distributed data structure that permits scalable estimations for this purpose.

1.2 Context

This thesis is concerned with the topic of self-organizing overlay networks, and their use for multimedia delivery.

An overlay network is defined as a virtual network, which is built on top of an existing one [7]. Overlay networks are usually built in order to offer services that are not provided by the existing network. The most well-known overlay network is the Internet, which is built on top of the existing telephone network [8], and which itself has a large number of overlay networks that build upon it.

Overlay networks, and the idea that each member *node* of an overlay network can not only receive but also share data (*Peer to Peer*, or P2P) are not new and similar concepts have been used in many earlier systems [9]. However, they gained a large amount of public attention with the advent of the Napster system in 1999. The P2P approach posed a contrast to the traditional idea of distinguishing between dedicated *server* (which are responsible for providing services) and *client* nodes (which consume the aforementioned services). In a P2P network, most nodes are *peers*, and take the role of both server and client: a peer may consume data from a peer, and simultaneously provide it to another. The key advantage of P2P is that the work load is distributed now among all peers, instead of being concentrated on a single server.

For example, assume a large HD video needs to be distributed to fifty nodes. In a traditional client-server based Video on Demand (VoD) implementation, the server needs to send this video to every node. As, despite its age, multicast is not widely supported, unicast is used in practice. If the number of client nodes increases, the load, and the bandwidth requirements at the server increase as well. In a P2P network, the server ideally only needs to send out the video file a single time, as peers then disseminate the file among themselves. When more peer nodes join, the available resources in the network also increase (usually allowing peer nodes to receive a better *quality of service* due to the higher number of content sources), while ideally the work load on the server stays the same. The bandwidth costs and the load on the server are reduced, as they are distributed among the peers.

As seen, Quality of Service (QoS) is an important concept in multimedia delivery. QoS is defined by the International Telecommunication Union as the "totality of characteristics of a telecommunications service that bear on its ability to satisfy stated and implied needs of the user of the service" [10]. A number of QoS related metrics, such as delay, inter-packet delay variation ("jitter"), bottleneck bandwidth and loss rate, are relevant to networks - and with that, overlays. Generally, QoS is associated with guaranteed resources: For example, a service provider may introduce a paid "Gold QoS" level, in which its user is guaranteed a bottleneck bandwidth of no less than $2000 KBit/s$. Such guarantees are traditionally made through reservation and admission control [11]. Most service on the Internet is delivered on a "best effort" basis (thus, no QoS is guaranteed). Even though overlays building up on the Internet can therefore not guarantee QoS, they can still be *QoS-aware*. A QoS-aware overlay actively seeks to maximize the QoS it provides to its users, for example, by avoiding to route over network segments with poor bandwidth or high loss rate.

However, initial overlay networks, such as Napster, were not so much concerned with efficient multimedia delivery, but much rather with locating files that the user wanted. Once a file is found, it is simply transferred from that peer via *simple* or *progressive download* over the Hypertext Transfer protocol (HTTP). This approach works well enough for small files, but is not particularly suitable for delivering large continuous (video) data due to the lack of QoS awareness [12]. In addition, while Napster leveraged the cost- and load-saving properties of P2P to download the files, it used central servers to search for them. While the bandwidth costs were distributed among the peers, the central servers still presented bottlenecks and single-points of failure.

Since then, *structured overlay networks* have emerged that provide efficient and scalable distributed search and routing [13, 14, 15, 16]. None of these overlay networks are concerned with continuous multimedia delivery and QoS, however. Efforts have been made to add QoS awareness to these networks. However, they are unable to deal with additive QoS metrics (such as delay, an important metric in Internet telephony), or a combination of different QoS metrics [17].

BitTorrent, introduced in 2001, is a popular protocol for an *unstructured overlay network* for disseminating large files. Following the protocol results in an emergent behaviour that clusters nodes with similar bandwidth together [18], even though BitTorrent itself does not explicitly support QoS. While BitTorrent is very efficient in disseminating large files, it does not account for the playback deadline and so is poorly suited for multimedia dissemination [19]. Several work has modified the algorithms behind the protocol to support on-time delivery of multimedia data with BitTorrent [20, 19]. However, despite this, a node's ability to continuously stream multimedia data is highly dependent on the region it resides in. As a result, BitTorrent-based streaming is only reliable for regions where resources are abundant; in resource critical regions fall-back services (such as dedicated servers) are needed to retain satisfactory performance [21].

BitTorrent-based dissemination with fall-back services is followed by popular *P2P Streaming* services such as UUSee and P2PView, which provide live multimedia streaming to their users. While both networks are closed, proprietary systems, it is known that, similar to BitTorrent, they split files into pieces and run a *peer selection algorithm* that selects high-bandwidth neighbours. Peers also exchange neighbour lists to "recommend" better neighbours [22], a method not present in BitTorrent. However, even though both approaches consider bandwidth, neither is location aware in the sense that they would prefer nearby nodes for better network performance and behaviour [23]. As a result, a path with several adjacent overlay nodes may have a high amount of bandwidth, but route back and forth across large distances in the underlying network. As they frequently share common links in the underlying network, they impacting the overall performance negatively [23]. Furthermore, less populated channels suffer from the aforementioned problems of BitTorrent-based streaming, and a substantial amount of peers cannot maintain a satisfactory performance level [22].

Ant Colony Optimization (ACO) is a method to optimize networks in a self-organized manner. It is inspired by the behaviour of ants: An ant that finds food leaves pheromones behind on its way back to the nest, which other ants use to locate the food source. As the other ants return, they leave pheromones themselves, reinforcing the scent. Shorter paths have more ants travelling on it, and thus have a stronger scent, and as such ants find the shortest path over time. Pheromones also evaporate over time, so once the food is harvested, the path expires over time.

ACO uses the same concept as an adaptive approach to find the current best paths: Ant-Packets are released periodically that travel across the network to a target node. They record the quality of the path they travelled, and use that information to reinforce the path with a virtual pheromone as they return to their origin [24]. Given alternative paths, an ant-packet will chose the path with more pheromones with a higher probability. Over time, if the network remains reasonably stable, ants will find the path with the highest QoS. Similar to real pheromones, the virtual pheromones expire over time, and as a result this approach is responsive to changes in QoS.

As more and more ants packets are introduced to the network, sophisticated optimization is necessary to retain scalability [25]. In addition, ACO relies on existing connections, and does not consider building connections between two previously unconnected nodes. As such, it can only find the optimal path of an existing network graph, but cannot improve the network graph itself. Given any number of overlay network nodes, ACO is not guaranteed to find the optimal path between two nodes, unless the overlay network is *logically* fully connected[1]. If we consider that P2P streaming overlay networks may have many thousands of nodes, it becomes clear that a fully connected network is not an option. For example, to fully connect an overlay with 10 000 nodes, 49 995 000 links are needed. Last but not least, organizational measures, such as clustering of nodes, are not straightforward to implement with ACO.

An interesting self-organizing approach is self-aggregation which, unlike ACO,

[1]Not all overlay nodes need to be connected in the underlying network, however ACO only finds the optimal path if a logical path exists between every pair of nodes in the overlay.

builds new connections between previously unconnected nodes [26]. This is accomplished through *matchmaking*: A node, the *initiator*, asks one of its neighbours to perform the role of a *matchmaker*. The matchmaker then searches for a node in its immediate neighbourhood that offers a good[2] match for the initiator. If such a match is found, it is returned to the initiator, and both nodes (the initiator and the node found by the matchmaker) connect. In order to keep the number of links in the overlay constant, the matchmaker then severs the connection between itself and the matched node. Noise in the form of random requests that do not improve the aggregation is used to escape local optima [27]. This algorithm has been shown to result in fast aggregation and clustering [26]. As the matchmaker is not aware of the QoS between the initiator and its other neighbours, matchmaking lacks QoS awareness and is not well suited for optimizing overlay networks that are used for multimedia delivery.

Estimation algorithms that use abstract coordinate systems [28] and landmarks [29, 30] can be used to estimate the "network distance" between two nodes. These algorithms, however, need to run on the nodes being matched instead of the matchmaker due to the large amount of additional information they require for accurate predictions [31].

The context of this thesis lies in combining this self-aggregation and network awareness into a flexible and scalable self-organizing overlay network, to support multimedia data dissemination in situations where many nodes are consumers and producers in a P2P like fashion.

1.3 Contributions

This thesis investigates several approaches to supporting multimedia data dissemination through self-organizing overlay networks. In this context, several contributions are made:

[2]The definition of a good match is dependent on the implementation of the matchmaker. In most algorithms, the initiator sometimes requests a match between nodes of a different class in order to escape local optima. However, it is still necessary that initiator and matchmaker agree on the definition.

1.3.1 The Flocks Overlay Network

Existing self-organizing overlay networks frequently focus only on content-awareness, and only offer rudimentary support for network awareness. Networks that do consider QoS concentrate on a single predefined metric, and optimize their structures against it. The success of BitTorrent and UUSee shows that this approach works well for fast dissemination of data. However, reasonable concerns of negative implications on the underlying network, caused by the lack of network awareness, have been brought up. In addition, consider that several QoS metrics are critical to the dissemination of multimedia data.

Assume, for example, that a path with bottleneck bandwidth is selected by the overlay, which also has a high delay variation ("jitter"). As a result, end-devices require a larger buffer so that they can compensate for this variation in delay. Such paths are not applicable to all end-devices:

- Mobile devices equipped only with limited memory (for example, an older iPhone with 128 MB of memory), find a path with low delay variation is more suitable, even if it has a lower bottleneck bandwidth.
- Powerful end-devices (for example, a Samsung Galaxy S II with 1024 MB of memory) that are equipped with sufficient memory find the first path more suitable, as they can compensate for the jitter and benefit from the large bottleneck-bandwidth.

An overlay that only considers a single QoS metric can not provide suitable paths for both the aforementioned end-devices. Therefore, it is important to include these additional QoS-relevant metrics in the routing and neighbour selection process as well.

Finding the optimal path with multiple QoS constraints is a difficult problem. It has been suggested that a single weighted metric may be used, however combining several QoS metrics into a single, new metric discards significant information. As a result, using a single weighted metric is not guaranteed to find the optimal (or even a feasible) path, even if one exists.

Therefore, this work investigates the feasibility of an overlay network that we named “Flocks”, which combines network- and content-awareness. In particular, it allows a choice of whether, and to what degree, nodes should be involved in routing multimedia data that is not relevant to them, if doing so improves network awareness. Additionally, a Flocks overlay network can be optimized against numerous QoS metrics, as well as several QoS metrics at once. Flocks are built in a self-organizing manner, all decisions rely solely on local information of the participating nodes.

To express a node’s interest in content and QoS in a flexible manner, a novel interest-property concept is introduced. Its flexibility and expressiveness are shown by creating Flocks overlays that behave similar to common protocols using only the aforementioned interest-property concept. For example, just by setting appropriate interests and properties, a Flocks overlay can behave similar to specific overlays (such as BitTorrent overlays), or create more general QoS-aware overlays that support several scenarios frequently encountered with multimedia delivery. This support includes:

- Overlays for single- and multiple-source multimedia data dissemination,
- overlays clustered based on several *content classes* (different content that nodes may be interested in; for example, two separate multimedia streams for the swimming and the running competition in a large sports event), where within each cluster, the overlay is optimized against QoS metrics such as delay, or bottleneck bandwidth,
- nodes that are interested in several content (for example, a node may wish to obtain both the stream for the swimming and the stream for the running competition), and
- nodes that change their interest over time (for example, a node that was previously interested in the stream for the swimming competition now wishes to obtain the stream for the running competition instead).

Each scenario is evaluated and the limitations of the Flocks overlay network are determined.

In addition, it is investigated how quickly Flocks settle on a stable network configuration. Their scalability is evaluated on large overlay networks that vary from hundreds to thousands of nodes, and measuring the amount of traffic such a self-organizing overlay network produces in relation to the number of nodes in the overlay network. Furthermore, it is investigated how quickly interests and properties may change to still achieve a stable configuration with a given traffic limit. The network-awareness reached by the Flock nodes using their local view is evaluated by comparing the effective QoS available in the overlay network with an optimal solution computed by a centralized component with a global view. A prototype of the Flocks overlay network is developed and evaluated in simulation and under "real" Internet conditions on the PlanetLab.

1.3.2 Scalable Self-Organizing Hierarchical Aggregation

The lack of scalability is a frequent problem in many decentralized, unstructured overlay networks. If a global view is no longer an option due to the size and the dynamics of the overlay network, nodes have to resort to limited, local information in order to locate a node. Approaches that use random walk or limited flooding to locate the target node work well for smaller overlay networks, but do not scale well. Recently, many overlay networks have resorted to a two-level organization in order to improve scalability: In this approach, a second overlay is used that consists only of "super-nodes"[3], each of which manage many dozens of nodes in the underlying overlay network. However, in overlay networks that disseminate multimedia data, nodes may not be able to handle more than a few neighbours. As such, scalability is limited even with a two-level approach.

To increase scalability for such networks, I used the Basic Partition [33] algorithm to allow any overlay network to organize itself in a hierarchical manner, so that the number of neighbours is limited, and the number of hierarchy levels remains

[3]Super-nodes are participants in the overlay that are reside more "powerful" systems (run on faster hardware, have a large amount of memory, ...), and are therefore delegated additional tasks by the overlay. Frequently, these nodes are also ordinary participants. For example, in the Skype overlay, any node running Skype may become a "super-node", if it determines that it disposes over a large amount of bandwidth. Skype super-nodes, among other tasks, route information between other Skype nodes that can not communicate directly (for example, due to firewall constraints) [32].

flexible. As a result, the overlay network is partitioned into distinct groups that can each be represented by a single, virtual "super-node" instead. However, the Basic partition does not deal well with hubs[4], which frequently occur in large overlay network and hierarchical topology aggregations. As such, I improved Basic Partition algorithm so that it can be used for hierarchical aggregation of scale-free networks[5] as well. A through evaluation is performed to evaluate the performance of the resulting algorithm in comparison to the original Basic Partition. Finally, a routing algorithm is developed and evaluated, which utilizes the hierarchical organization providing scalable routing with a computational complexity of $O(\log_2(n))$.

1.3.3 Scalable Decentralized QoS estimation

An important contribution of this work is the introduction of QoS maps. Frequently, it is necessary to know the QoS on a specific path to a target node: For example, overlay networks often need to make a routing decision to select between multiple available paths. In addition, Flocks themselves often need to optimize the QoS to a specific node in the overlay network, such as to the node the required multimedia data is located on. It is clear that sharing accurate QoS information at all times does not scale in large overlay networks. Therefore, it becomes necessary to enable nodes to quickly estimate the QoS to any given node, preferably on multiple alternative paths, with little overhead. Algorithms that estimate the QoS using abstract coordinate systems or landmarks estimate the QoS of the underlying network, and their estimates differ from the QoS achieved in the overlay network unless the overlay network matches the underlying structure perfectly.

As a result, QoS maps are introduced, which give each node a view of the QoS "landscape" of the *overlay network*. A node's QoS map contains detailed and up-to-date information of its immediate neighbourhood, and becomes more and more coarse for distant nodes. It is shown that due to their hierarchical structure, QoS

[4]Hubs are nodes with a large number of neighbours.

[5]Scale-free networks are networks in which the distribution of the nodes' degree follows a power-law. Such networks exhibit a small number of very large hubs. Scale-free networks are frequently caused by preferential attachment, that is, a joining node is more likely to connect to a node with an already high degree [34]. Scale-free properties can be observed in numerous domains, including websites found on the Internet, and paper citations [34].

maps seamlessly scale to very large overlay networks. Furthermore, it is shown that nodes are able to obtain a rough estimate of the QoS experienced on any given path through the Flocks overlay, using only local information. Finally, a multi-level QoS estimation algorithm that allows refining this estimate to an arbitrary accuracy by using local information from other nodes, is developed and evaluated.

1.4 Structure

The remainder of this thesis is structured into six chapters. Chapter 2 provides a brief introduction into each of the relevant research domains, reviews related work and discusses it in the context of this thesis. Chapter 3 identifies several important properties that a self-organizing overlay must have in order to be suitable for multimedia delivery. Chapter 4 defines the Flocks overlay, and describes the key ideas behind it and its operations. A prototype is developed and evaluated in Chapter 5 through simulation and under "real" Internet conditions. During the evaluations, several areas in need of improvement are identified, which are explored in Chapter 6. Finally, Chapter 7 concludes the thesis and outlines future work.

CHAPTER

2 Related Work

2.1 Background

The term "overlay network" describes a very broad concept and includes a large number of today's networks. A good definition - which this thesis follows - is that an overlay network is a "virtual network of nodes and logical links that is built on top of an existing network with the purpose to implement a network service that is not available in the existing network", found in [35]. What encompasses this "existing network", however, is just as broad as the term "overlay network" itself: The different layers of the OSI model [36], for example, are overlays[1], even though they operate on a much lower level than what the bulk of current research is focused on.

This thesis is concerned with application level overlays that build up on the transport layer of the OSI model: They are therefore applicable in local area networks, as well as the inarguably most popular overlay network - the Internet [37].

In addition to providing an abstraction layer, overlays are an important method to implement new network services, as directly modifying the underlying network (such as the Internet) is usually not an option: Typically, the user has no control over the individual network components that are traversed from one endpoint to another. Therefore, this chapter will review relevant overlay networks. It will give a brief introduction into the different types of overlay networks found in current literature, and provide specific examples of overlays and their differences to the Flocks overlay. It will then focus on different issues relevant to QoS aware routing in overlay networks, detail the solutions employed in similar networks, and how they differ to the approach followed by Flocks.

[1]For example, the data link layer overlays the physical layer, providing physical addressing, which the physical layer does not offer.

2.2 Content Dissemination in Overlay Networks

The focus of this thesis is on overlay networks for a specific purpose: data dissemination where a great degree of flexibility is necessary (such as in large social events), with a special emphasis on continuous data. [38] lists a number of characteristics that an overlay network must have to enable efficient data dissemination to one, multiple or all nodes: Quality of Service, robustness and scalability. Additionally, an overlay network must be adaptive to changes in the underlying network to provide good quality of service [39].

As explained at the beginning of the chapter, overlay networks can be very generic. At the very minimum, we assume that each node in the network can be addressed individually. Therefore, an unique identifier is required. Without loss of generality, this unique identifier could be the Internet protocol (IP) address of the node and a port number[2], as commonly used by TCP or UDP, but any unique identifier may be used instead. This thesis assumes that broadcast and multicast do not exist in the underlying network, or are not adequate, and so the use of an application level overlay network is necessary to disseminate data. This is a reasonable assumption for Internet applications: IP multicast, while existent, is not commonly supported by Internet service providers [40]. In addition, overlay networks exist that outperform IP multicast [41].

Two main approaches are used in overlay networks to disseminate content: The traditional Client-Server approach, and Peer-to-Peer dissemination. This following section will briefly describe both of them.

In the traditional client-server approach, two classes of entities are used: Clients, and servers. Servers are the only providers of content and service. They are comparatively few, powerful systems, that are contacted by many, less powerful clients which request "content or the execution of services [from the server], without sharing any of its own resources" [42]. The main drawback of the client-server approach is that servers frequently become bottlenecks and single points of failures. Maintaining ample resources (such as bandwidth, processing power) to handle and respond to requests is expensive in terms of hardware and maintenance costs, and does not scale

[2]As a result, it is possible that one physical node is host of multiple overlay nodes.

to a large number of clients [43]. In addition, sudden surges in clients requesting content, so called “flash events” that cause “Flash Crowds” [44], are only handled by over-provisioning resources in anticipation of a “flash event”. Because of that, the client-server will not be further investigated in this thesis.

A node that follows the peer-to-peer approach acts as both client and server. Similar to a client, it requests content from other nodes, but it also provides this content to nodes in the network, like a server would do. In this way, the peer-to-peer approach distributes load among all peers. Increasing the number of peers also increases the amount of resources the network has. As a result, peer-to-peer networks are able to offer high scalability, and resistance to Flash Crowds. It should be noted that many peer-to-peer applications consider a server from which data enters the network, or a server that operates as a backup[3]. However, as peers begin distributing data they received to other nodes, the load on such a server is reduced significantly [41], as opposed to a pure client-server architecture.

2.3 Peer-to-Peer Overlays

Peer-to-peer based approaches can be classified in multiple dimensions, based on their reliance of centralized components, their overlay structure and their means of communication. A peer-to-peer overlay may be *hybrid*, in case it requires a central component to provide part of the overlay’s service, or *pure* in case it does not. An overlay may be structured, in which case all nodes maintain routing information to deterministically reach any other node in the overlay, or unstructured, if this is not the case. Communication may also occur in a structured way, such as building a multicast tree to disseminate data from a source node to all other nodes in the overlay, or unstructured, such as using gossip protocols, in which a message is forwarded to a random subset of neighbours. This section will discuss each of the dimensions in order.

[3]For example, consider an overlay in which peers exchange video data, and also play back the video as they receive it (the video is “streamed”). In this situation, a server may be used as a backup, to provide nodes with data that they did not receive from any other peer in time for playback.

2.3.1 Hybrid Overlays

In hybrid overlays, centralized components are used to provide part of the network's service [42]. They most commonly act as rendezvous points that for assigning neighbours to joining nodes, and are responsible for failure recovery by reassigning neighbours when nodes leave the network. BitTorrent [45] is a prominent example for a peer-to-peer protocol that uses a centralized rendezvous point, the "BitTorrent Tracker". Centralized components may also act as a central directory used to locate resources in the network. The original Napster network, despite using the peer-to-peer approach to exchange files, required a centralized server for locating them [46]. BiToS, a protocol that extends BitTorrent to support streaming, uses a centralized backup server to contact for pieces that will not be received in time for playback [20].

NICE [47] uses a hierarchical layout to maintain scalability, similar to the hierarchical aggregation used by the QoS estimation of Flocks. A similar hierarchical layout is also used by ZIGZAG [48], which uses different overlay creation and maintenance strategies to outperform NICE. While several concepts, such as splitting and merging, are applicable to Flocks, both NICE and ZIGZAG require a hierarchy root that receives all join requests and assigns joining nodes to their correct neighbourhood in the hierarchy. Both overlays are QoS-, but not content-aware, and thus differ substantially from Flocks. They also differ from the hierarchical aggregation introduced in this thesis, which aggregates an existing topology for scalable QoS estimation - while NICE and ZIGTAG both create a *new* topology that minimizes some QoS metric to a specific node.

In conclusion, centralized components pose a problem to scalability. They either tend to maintain a global view of some sorts, or are prone to receiving large amounts of requests and thus inherit the previously mentioned disadvantages of the client-server approach. As their capacity limits the maximum amount of nodes their overlay network may support, they present bottlenecks. In addition, they are a single points of failure, as their removal impairs the network, or makes it outright unusable.

2.3.2 Pure Peer-to-Peer Overlays

Pure peer-to-peer overlays are more resilient by definition, as the removal of any single component must have no effect on the network service, beyond the loss of whatever resources were located on that node [42]. They promise greater scalability than hybrid overlays, but their achieved scalability depends heavily on the structure and the implementation of the overlay. The lack of a centralized component makes pure peer-to-peer overlays attractive solutions for large, dynamic events, in which a centralized component would pose a bottleneck and single point-of-failure. As a result, the remainder of this chapter will focus on pure peer-to-peer overlays, and make note if any centralized components exist.

A prominent example for a pure peer-to-peer overlay is the now-discontinued Gnutella protocol, which initially faced scalability issues from using localized flooding to locate a file in the overlay: Queries for files are always forwarded a predefined distance from the node they originate from, putting a significant amount of load on the overlay and its nodes. Additionally, rare files can only be found if they were located inside the (localized) search radius of the node trying to find them [49]. Later versions of Gnutella introduced a two-tier architecture that alleviated most of these problems and improved scalability [50].

Structure - or the lack thereof - is an important dimension for classifying overlay networks. Specifically, overlay networks can be classified into two broad classes: Structured overlay networks, in which a node's neighbourhood is well defined (for example, a node's neighbours may be determined based on their unique identifiers), and unstructured networks, where this is not the case.

2.3.3 Structured Overlays

In structured overlays, nodes "cooperatively maintain routing information about how to reach all nodes in the overlay" [51]. In contrast to unstructured overlays, any content is located deterministically, using a bounded number of messages and path length. Structured overlay networks have been shown to scale well, offer resilience and low maintenance overhead compared to unstructured overlays [50]. This low maintenance overhead may seem surprising at first: maintaining an overlay under structural constraints appears to be more difficult than simply keeping a unstructured mesh connected. However, structured overlays can exploit their structure to reduce the amount of information that nodes exchange periodically. For example, by partitioning failure detection responsibility, it is possible to reduce the number of keep-alive probes by a factor of 32 [50]. Similar results have been shown for search and constrained flooding [50].

Many well-cited structured overlays, such as Chord [15] and Pastry [13], do not apply directly to video streaming. They are included for the sake of completeness, and will not be considered beyond this section. Both overlays implement Distributed Hashtables (DHT) [52] and therefore aim to distribute load uniformly, which makes them good choices for storing and retrieving small pieces of information. However, I do not consider them applicable for managing the large amounts of data typical to multimedia applications, because they offer no control over data placement [53, 54] (therefore, data may be "stored far from its users" [53]) or node organization[4]. In addition, they do not consider QoS. As a result, nodes that are neighbours in the overlay may be very distant in the underlying network. This so-called Topology Mismatch Problem [55, 56] results in unnecessary traffic and poor performance, as messages are routed over long paths on seemingly adjacent nodes [54]. [17] alleviates this issue by deriving node identifiers in a QoS-aware manner, however even with this approach they cannot support additive QoS metrics, such as delay [17].

Narada [57] is a self-organized protocol that builds structured overlays in two steps: In the first step, nodes are connected randomly in a mesh. This mesh is then subsequently optimized against an application-chosen QoS metric, such as delay, or bottleneck bandwidth. Similar to Flocks, each node only has a limited number of neighbours. In a second step, multi-cast trees rooted at each source node are built on top of the mesh. As a result, efficient multicast data dissemination becomes possible. However, it is not directly compatible to Flocks, because it modifies the overlay itself as well. While protocols like Narada appear similar to Flocks due to their QoS-driven optimization, they are concerned primarily with data dissemination: As a pure multicast protocol they assume that all nodes in the system are interested in all content. This makes it difficult to support nodes that are interested in different

[4]Instead, nodes are organized based on their identifiers.

parts of content. It should also be noted that Narada requires that every node in the overlay maintains a list of all other overlay nodes. As such, while Narada does not have any centralized components and is designed to work in completely self-organized manner [57], it does not scale beyond small to medium sized overlays.

SkipNet [53] is a notable self-organizing structured overlay that offers content- and path locality to address practical shortcomings of other structured overlays. If node identifiers are assigned in a QoS-aware manner, SkipNet can be both content- as well as QoS-aware.

The path-locality of SkipNet is similar to the content-awareness of the Flocks overlay: It guarantees that messages between nodes located in the same organization are routed only inside that organization. This is achieved by giving the node identifiers of nodes that should be placed near to each other similar keys. SkipNet's second property, its content-locality, is not directly comparable to the Flocks overlay, as it is a special DHT feature that allows content to be placed on a specific node. Content-locality is achieved by prefixing the content with a key identical to the identifier of the node it should be placed on[5]. A "Proximity" Table (P-Table) is used to route in a QoS-aware manner, within the limits of the structured overlay. To guarantee the aforementioned locality properties, node identifiers must be assigned in a way that all nodes physically close to each other also have identifiers that reflects this closeness[6]. The authors of SkipNet suggest that the identifiers can be chosen so that nodes in the same organization receive similar identifiers. As a result, the overlay matches the underlying topology well, resulting in high QoS. However, this knowledge is not derived in a QoS-aware manner, and requires an external entity with knowledge about the nodes' locations.

Despite these apparent similarities, SkipNet suffers from two main limitations, many of which are shared by other structures overlays:

First, the structure of the overlay itself puts restrictions on which nodes can be neighbours of each other. As a result, a high-QoS link between two nodes that cannot be neighbours based on structural restrictions of the overlay can never be found.

[5]This also works if a prefix similar to the identifier of the node it should be placed on is used, if no other node is closer. However, in that case placement on a specific node is no longer guaranteed.

[6]This is done by establishing a total order over all identifiers. Two nodes that are physically close to each other also need to have identifiers close to each other in this order.

Second, links between overlay nodes are primarily assigned based on the nodes' identifiers. Many structured overlays, including SkipNet, can be made QoS aware if these identifiers are derived in a QoS-aware manner [17]. However, it is unclear how changes to the QoS in the underlying network should be handled, as it would require the content on the node to be relocated.

Substantial research exists for multimedia delivery in tree-based structured overlays [47, 58, 38, 59, 48]. They directly support multicast delivery by disseminating data from the tree root towards all leaf-nodes. Due to their cycle-free nature, none of the nodes receive multiple packets. It is important that nodes with the highest bandwidth are placed close to the root, as a low-bandwidth node becomes a bottleneck to all its downstream nodes. The main disadvantage of single tree-based structured overlays, such as [47, 48], is their poor resilience to peer churn [38]. If any interior node fails, its children become disconnected from the tree and must re-build that tree segment before any nodes downstream of the failing node receive data again. This results in a noticeable disruption of service for the affected nodes [39]. As the degree of each node is bounded, most nodes are interior nodes, and thus affect the overlay adversely upon departure. To improve performance and resilience, multi-tree based approaches have been proposed [58, 38, 59], in which every non-root node is an interior node in only one tree, and a leaf-node in all the others. As a result, a failing node only affects the distribution path in one tree. While its former children repair the overlay, they continue to receive data from all the other trees. However, this approach is prone to "deadlock events", during which trees becoming saturated with leaf-nodes in the presence of churn [39]: In this situation, at least one tree cannot accommodate any new nodes because it is imbalanced, and the number of children per node (the *degree*) is limited. Research has shown that such deadlock events occur frequently, in particular for trees with low degree nodes [39].

2.3.4 Unstructured Overlays

Unlike their counterparts, unstructured overlays lack a deterministic structure. They offer a greater flexibility than structured overlays would, because any other overlay node is a potentially valid neighbour. As a result, they can exploit routing in the underlying network better than structured overlays, and achieve higher QoS. The disadvantages for unstructured overlays are higher costs for routing and searching. While routing in structured overlays is deterministic and frequently achieves logarithmic and better complexity with a fixed-size routing table, unstructured overlays must resort to hierarchical approaches to maintain scalability with increasing network size.

Even though the overlays built by Flocks can have a structure (for example, it is possible to build rings or cubes using appropriate interests), they are in fact unstructured: Every overlay node is a valid candidate neighbour for every other node.

Many unstructured overlays have been developed that follow similar approaches, or approaches that I have considered during the design of Flocks. This section provides a survey of these unstructured overlays.

An interesting idea is pursued by SORMS [60]. Their approach considers communication between two nodes a sign that both nodes must be interested in each other. Therefore, it organizes its overlay based on how recently two nodes communicated. SORMS also has an explicit notion of interest, however it only implements a subset of the functionality provided by Flocks. For example, unlike Flocks, QoS is not considered. In addition, Flocks can easily be configured to build an identical overlay by using communication recency as interest, and thus mimicking SORMS.

LTM [61] is a protocol that improves the location awareness in unstructured peer-to-peer systems by making nodes disconnect their slowest neighbour. The approach is generic, and can be extended to use additional criteria of what constitutes a "poor" neighbour, such as several QoS metrics, or the content it requires, however this was not investigated in the paper. Flocks pursue a similar approach in that each node disconnects the neighbours in which they have the lowest interest in, however, LTM requires synchronized clocks, and employs flooding to distribute QoS information, which gives raise to scalability concerns in large overlays.

An interesting self-aggregation approach with algorithms from the CASCADAS project [62] is evaluated in [26, 63]. Their goal to re-organize the overlay by creating links between compatible nodes and severing links to incompatible nodes is related to the actions performed by Flocks. For example, in [26], nodes are assigned different classes, and two nodes are compatible if their classes match. In the resulting overlay, nodes of each class are grouped together in a cluster. It is clear that compatibility and interest are very similar. The substantial differences between Flocks are that the approach introduced in [26] uses matchmaker nodes, keeps a constant number of links in the overlay, and does not directly consider QoS.

The matchmaking approach offers an efficient way to bring nodes together that are not directly connected. During the matchmaking process, the matchmaker makes two of its own neighbours initiate a connection to each other. It usually chooses two neighbours that are compatible with each other to improve the overlay, however, with a small probability it may match two incompatible nodes instead. This is done to escape local maxima. The algorithms do not require dedicated matchmakers: Any node periodically elects itself as a matchmaker begins the matchmaking process. In order to keep the number of links in the overlay constant, the matchmaker drops the connection to one of the two nodes after the matchmaking has completed.

In contrast, Flock nodes act as their own matchmakers: They obtain a list of candidate peers from their direct neighbours, along with their properties. Exchanging this information produces additional traffic, however it allows each node to run QoS estimation algorithms to each candidate, connect to it directly and then confirm, or disprove, the estimation. This is an important distinction to the approach in [26]: As the underlying network acts similarly to a black box, dedicated matchmaker nodes can not make informed decisions about what neighbours to match if QoS must be considered. Flocks, on the other hand, use QoS estimation and make their “match” based on properties that are not known before they establish a connection to a partner. As a result, they are able to consider all QoS metrics before making a decision, and avoid dropping a connection with high QoS.

2.3.5 Gossip Protocols

Gossip protocols disseminate messages by randomly selecting several nodes from the overlay, and relaying the message to them. Every node that receives the message for the first time repeats this procedure. As each message spreads in a snowball- or virus-like manner, gossip protocols are also referred to as epidemic protocols [64]. The key properties of gossip protocols are as follows: They have been proven to reach every node in the overlay after a logarithmic amount of steps (dependent on the number of nodes in the overlay) with very high probability [65], while remaining resilient of churn and network failures, due to their intrinsic redundancy. As they select the recipients of their neighbours as at random [66], they also distribute load among all nodes in the overlay.

Many approaches that implement gossip protocols require that each node maintains a list of all other nodes in the system (a global membership table, or global view), that they can select a random sample from. This results in a significant overhead for large overlays, and overlays with high churn, as the tables of every node in the system must be updated on every join and leave. A scalable approach that does not require a global membership table is the use of *partial views*, in which each node only maintains a list containing a subset of nodes in the overlay. The performance of the gossip protocol heavily depends on the quality of this subset: In order for the aforementioned qualities of gossip protocols to hold despite the limited view, the subset must contain an uniform random sample of nodes in the overlay [67]. In addition, using only the partial views, a message must be able to reach every other node in the overlay: the overlay graph established by the partial views of all nodes must not be partitioned. Maintaining this property in presence of churn and large-scale failure is not trivial, and often requires significant time, during which the reliability of all disseminated messages is degraded [66].

HyParView [66] is an interesting approach in which each node maintains two views: A small active view with nodes, to which an open connection is maintained, and a larger passive view containing at least $\log(n)$ nodes, which serves as a list of replacement nodes, should a node from its active view fail. In this case, a node from the passive view is promoted to the active view. Both views have a fixed size. A node (the “originating node”) regularly updates its passive view by sending out a “shuffle” request. This request is forwarded for a specific number of hops, and the final node dealing with the request sends a list of nodes from both its active and passive views back to the originating node. The originating node then combines the list of nodes with its own passive view, removing nodes from its passive view as necessary. This procedure ensures that failed nodes are eventually removed from all passive views.

While HyParView does not use a notion of interest, Flocks are still similar in that they also use an active (nodes they maintain connections to) and a passive view (their candidate list). However, the nodes a Flock node has in its active view are not chosen at random: they are the nodes that it has the highest interest to. HyParView demotes a node from its active to its passive list (essentially severing the connection) only under the following two circumstances: First, if a node leaves or fails, it is removed from every other peer's active list. Second, if a node has no nodes in its active view, it will send a "high priority" join request to a node in its passive view [7]. Upon receiving a high priority join request, a node will always accommodate it. If its active view is full, another node is removed from it to accept the node that sent the high priority join request.

In addition to the two circumstances (that cause a node to move one of its neighbours from its active to its passive list) mentioned above, a Flock node may also disconnect a neighbour if it has too little interest in it. Finally, a Flock node may disconnect one of its neighbours to make room for accepting a neighbour it has higher interest in, even if that neighbour did not send a "high priority" join request.

It should be noted that the Flock overlay supports a similar feature to HyParView's "high priority" join request. A Flock node can be configured to maintain a limited number of "grace connections", which are connections to nodes that it has no interest in, but that have no other neighbours. This allows said nodes to stay connected to the overlay, and thus obtain the identifiers of nodes that may be more fitting neighbours.

A notion similar to the interest used by Flocks is used in T-MAN [68], which is a gossip-based protocol that builds overlay topologies based on a ranking function. Similar to HyParView, T-MAN distinguishes between an active view (nodes it maintains a connection to) and a passive view (a list of nodes provided by a distributed peer sampling service [67]). By choosing the appropriate ranking functions, the topology of the overlay can mimic well known topologies such as the one of a ring, torus or

[7]A node's active view becomes empty if it loses the connection to its last neighbour. For example, consider a small overlay of four nodes, in which each node can have at most two neighbours. Therefore, it will have two nodes in its active list, and one node in its passive list (which will not accept it as a neighbour, because it already has two different neighbours as well). If that node's connections to the two neighbours fail, it will move the nodes from its active to its passive list. As the node's active list is now empty, it will send a high priority join request to the node in the passive list that would not accept it earlier.

trees. Their approach is very similar to Flocks in that nodes exchange arbitrary properties, and rank their neighbours based on these. Flocks can directly use the ranking function by T-MAN, except for the special treatment of the zero-interest value: This is because a zero-interest value instructs a Flock node to disconnect the neighbour it has no interest in, even if it would be able to accept additional neighbours. Even though T-MAN is very similar approach, Flocks are more generic as they permit the propagation of a node's properties farther than one hop, if necessary. This allows, for example, to give a node with an "important" property greater visibility. The protocol description of T-MAN is generic enough to include QoS estimation, as they suggest that any property of neighbour node may be used to rank it. Even so, QoS estimation is not specifically introduced, and the construction of a QoS map to be used for subsequent QoS estimation (and thus, ranking of a neighbour), which is not present in T-MAN, is a major feature of the Flocks overlay.

Another difference to the Flocks protocol is that the interest function used by a Flock node is specific to that node, and that two nodes may use different interest functions. This is particularly important in large social events, where part of the nodes are consumers, and part of the nodes are producers. Both groups have different interests: Producers are interested in nodes having high bandwidth. In addition, they prefer consumer nodes that are willing to relay to other consumer nodes, as they reduce the resources the producer needs to provide. Consumers, on the other hand, are simply interested in the content, and possibly use different QoS metrics, such as jitter. In contrast, T-MAN uses the same ranking function for every node in the overlay, making it difficult to implement this topology intuitively.

X-BOT [69] is a self-organizing protocol that optimizes overlay networks based on link cost. It requires a local oracle to compute the link cost between two nodes, which must be defined by the application. The link costs between two neighbours are compared locally, to decide which of two neighbours should be kept. While the authors only mention network metrics, their descriptions are generic enough that any content-related metric can be used[8], and that content- and network-metrics may be combined. This approach is very similar to the interest function used by Flocks.

[8]This holds true as long as the degree of the match between itself and the neighbour node can be expressed in a number; a higher number indicates a better match.

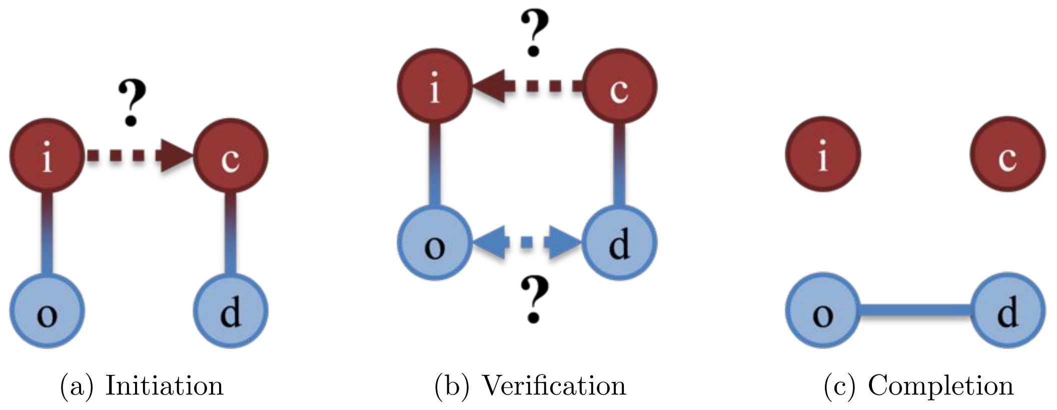


Figure 2.1: The optimization step performed by X-BOT. Nodes that have the same colour are considered good matches.

Indeed, any X-BOT oracle can be used for the Flocks overlay, under the caveat that it need to be changed to produce only non-zero values.

The optimization step of X-BOT is substantially different to Flocks. As illustrated in Figure 2.1, four nodes are involved in an X-BOT optimization step: The initiator node i, the candidate node c, and their two neighbours o (which is the neighbour of i that is disconnected in favour of c), and d (which is the neighbour of c that is disconnected in favour of i). Once node i initiates the optimization step (see Figure 2.1a), all involved nodes verify whether their new neighbours are more fitting matches than their previous neighbours (see Figure 2.1b). Only if all four nodes benefit from the optimization, it actually takes place. In this case, node i connects to c, and o connects to d. The links between c and d, as well as i and o are severed (see Figure 2.1c). As a result, the total number of links in the overlay is preserved. Only two nodes are involved in Flock optimization step: Node o, the neighbour of i that is replaced with c, and node d, the neighbour of c that is replaced with i are not explicitly considered. If node i or node c exceeds their preferred number of neighbours after the optimization, they will disconnect nodes c and d due to a lack of interest, similar to X-BOT. Both nodes can then initiate new connections based on their peer lists, including connections between each other[9]. Still, both protocols are

[9]Even though c and d are no longer direct neighbours, they obtained each other's information prior to their disconnection from i and c, which were their direct neighbours and exchanged node lists. As a result, the addresses of d is in the "passive view" of node c - and vice versa.

not equivalent: Flocks are more likely to partition, because the disconnected links between i and o, as well as c and d are not immediately replaced. This limitation does not pose an issue, because the Flocks protocol guarantees that nodes o and c are aware of each other, and can initiate a connection. In addition, the local view used by Flock nodes is rich enough so that it is unlikely that the network will partition (see considerations in Section 5.2.1).

In addition, the neighbours o and d can veto the optimization in X-BOT. In this case, no optimization will take place, which can prevent reaching the optimal topology if such a step is necessary. Flocks nodes are more aggressive during their optimization by disconnecting nodes if it provides a local benefit.

Similar to Flocks and HyParView, each X-BOT node keeps a number of neighbours to which an active connection is maintained (active view), and a list of candidate nodes that can be used for optimization (passive view). It also sacrifices stability to optimize the overlay, which Flocks nodes do as well. However, due to its rigid optimization process, it is not as flexible as the Flocks overlay. For example, Flocks can dynamically increase and decrease the total number of links in the overlay, which is not possible with X-BOT. In addition, the X-BOT protocol assumes that the link costs are symmetric, which is not necessarily the case in practice, and restricts the use of X-BOT in situations where they are not. Additional steps need to be inserted into the optimization procedure to consider asymmetric link costs.

2.4 Quality of Service Aware Routing

Quality of Service plays an important role when choosing a route to the target node. Traditionally, QoS can be guaranteed by employing end-to-end resource reservation and admission control [11]. In order to distinguish between the set of features and specifications that make up QoS as defined in [11], and the quality of the service that end users experience, this thesis uses QoS to refer to the former, and refers to the latter as the *level of service* (LoS) experienced by users. QoS is recognized to be crucial for multimedia applications, in which the various QoS-related metrics (referred to QoS metrics in the remainder of the thesis for brevity, these include, for example, throughput, delay, jitter and loss), play an important role regarding the level of service (LoS) the users get. Intserv [70] and Diffserv [71] implement QoS, however, they demand support from all routers on a path, and thus require large changes to the existing network infrastructure. As a result, the majority of the Internet still uses the best-effort service model, in which no resources are guaranteed, and packets are delivered on a best-effort basis [72].

2.4.1 Overlay-Specific Issues

Overlay networks are an attractive solution for improving the LoS, as they are cheap to deploy, and require no change to existing infrastructure. It is important to note that overlay networks are subject to the best-effort routing used on the Internet. In particular, they have no direct influence on how or over which Autonomous Systems (AS) their traffic is routed, which is dictated by policy-based inter-AS routing (for example, by the the Border Gateway Protocol BGP [73]). The basic idea of Flocks is that even if they are not implemented in routers, they can *influence* actual routes by sending traffic over specific overlay nodes. For example, consider the fully connected four-node overlay with the nodes A, B, C and D shown in Figure 2.2. This overlay spawns over seven Autonomous Systems. Assume node A needs to send data to node D. As A and D are neighbours in the overlay, A can directly send to D, leaving the decision of how the data is best routed up to the routing protocol in the underlying network (which will likely route from AS1 over AS5 to AS4, which D is in). However, node A could route the data over node B or C, leading to a route consisting of three hops in the overlay network: $A - B - D$ (likely to bypass AS5, 6 and 7) or $A - C - D$ (likely to bypass AS2, 3 and 5). Two four-hop routes are also possible: $A-B-C-D$ (likely to bypass AS7 and AS3) and $A-C-B-D$ (likely bypasses the border routers between AS1 and AS2). By relaying to node B or C first, a different route involving different AS may be chosen in the underlying network. Making the detour over additional overlay nodes may bypass an AS with a poor LoS that would have been chosen otherwise. Thus overlay networks can grant a LoS improvement that would otherwise not be available. This approach is commonly referred to as "detour routing" in literature.

It is a well known fact that introducing a detour while routing over the Internet

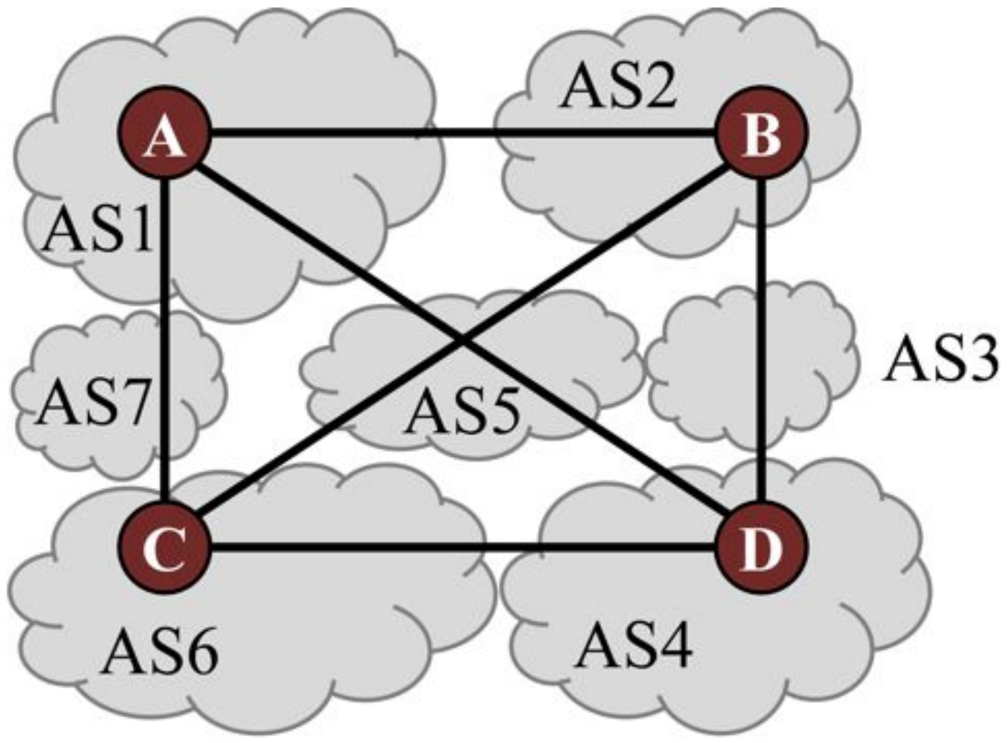


Figure 2.2: A four-node overlay spanning over seven AS that illustrates the effect of *detour routing*

may be shorter than taking the "direct" path. Referred to as Triangle Inequality Violations (TIVs), this phenomenon plays a very important role when it comes to QoS estimation, and will be discussed in greater detail later in this chapter. TIVs are a common occurance, and while most routes only suffer from slight violations, a small fraction of routes are the cause of severe TIVs [31]. This is demonstrated by [55], who construct an overlay network where each node attempts to minimize its delay to all other nodes in a fully decentralized way. The authors base their simulations on end-to-end delay values measured in the PlanetLab. They show that the broadcast delay in their optimized overlay is significantly lower than the broadcast delay in the underlying network, and note that this is due to TIVs [55]. This shows that QoS aware overlay construction is important to reduce TIVs and to improve the LoS end-users receive.

Improving LoS by routing over specific overlay nodes is not a new approach: A similar idea is pursued by QRON [72]. It introduces powerful nodes called overlay brokers (OB) into strategic positions on the Internet, which are then used by end-users. These end-users are simply consumers, and not part of the peer-to-peer overlay itself. An OB can directly route to any AS its own AS is connected to, and so explicitly bypasses an AS with a poor LoS. Therefore, a certain degree of support from the Internet Service Provider (ISP) of the OB is necessary. A drawback of QRON is that each node must know the number of the AS it is in, and what other AS its own AS is connected to. Links between overlay nodes are strictly built based on that information: Nodes in the same AS are always connected to each other, nodes in two different AS are only connected if a link between both AS exists. However, obtaining this information is not straightforward. Even though public BGP routing tables exist [74], is has been shown that analysing these routing tables only provides an incomplete view of the links between AS, with a “substantial” amount of AS-links missing [75]. In contrast, a Flocks node is more likely to make the underlying network use any available AS links, because it considers every other overlay node a potential candidate neighbour node, whether or not it is in a seemingly adjacent AS. The Flocks overlay is, in fact, not aware of AS and the links between them. Even if it were, it can be argued that it would have no practical way of forcing the underlying network to take a specific route. As a result, it treats the underlying network as a black box that cannot be directly influenced except by routing over a specific overlay node, which, as detailed before, is a reasonable assumption. To determine the goodness of a neighbour for this purpose, each node measures various QoS metrics, such as bandwidth, jitter, round-trip-time, and bases its routing decisions on that.

Another approach to improve LoS is presented by OverQoS [76], which achieves better than best-effort services by controlling the traffic on the virtual links between overlay nodes. Flocks differ from OverQoS in that they do not provide performance guarantees, and rather observe the level of service on each link. However, the Controlled Loss Virtual Links (CLVLs) that OverQoS introduces deserve being mentioned and are of particular interest: CLVLs compensate from loss experienced in the underlying network, by adding redundancy to data packets. This has two implications: First, they make high throughput a very desirable metric, as it can compensate for a second QoS metric, loss. Second, their concept can be adapted to other overlays, such as the Flocks overlay network. CLVLs are out of scope of this work and Flocks do not implement them, however they provide an attractive way of further improving the LoS.

An unique issue to overlay networks is caused by correlation of links in the underlying network: Two seemingly independent links in the overlay may share a common link in the underlying network and, as a result, QoS-related metrics such as bottleneck bandwidth. Figure 2.3 motivates an example. Consider the physical network in Figure 2.3a. The four nodes that participate in the overlay are separated by a shared bottleneck. Numbers on the individual links represent their total bandwidth in Mbps. A possible overlay network for this physical network is shown in Figure 2.3b. The numbers on the overlay links (shown as dashed lines) represent the bottleneck bandwidth that each node measures individually. As a result, node A might assume that it has a bandwidth of 3 Mbps to node C, and 3 Mbps to node D, when in fact both connections share a single physical link. As the overlay fails to recognize the shared bottleneck, it is likely to make wrong routing decisions: For example, based on the available data, node A could assume that it has a 3 Mbps path to node B (A-C-B), which routes over the bottleneck link twice. If node B then assumes that it can reach node A in a similar fashion (B-D-A), two flows (from A to B and from B to A) are routed over the bottleneck link, resulting in an effective bandwidth of $1.5 Mbps$ per flow (assuming fair queuing). If C and D make analogous routing decisions, the effective bandwidth is further reduced.

Shared bottlenecks are very common in single-homed networks (such as end-users), which are only connected to one physical neighbour node. In this case, sending data over one overlay link also reduces the available bandwidth of all other overlay links, because they share a common path in the underlying network.

Unfortunately, detecting shared bottlenecks is not trivial, as the topology of the underlying network is usually not available. Shared bottlenecks can be identified by correlating one-way packet delays from different streams [77, 78], under the assumption that packets which cross the same bottlenecked router will experience the same queuing delays. However, obtaining the one-way packet delay from one node to another requires synchronized clocks, or appropriate hardware. Therefore, relying on the availability (or the accuracy) of this metric is not practical in the common case.

[79] introduces a passive approach for detecting flows with common bottlenecks:

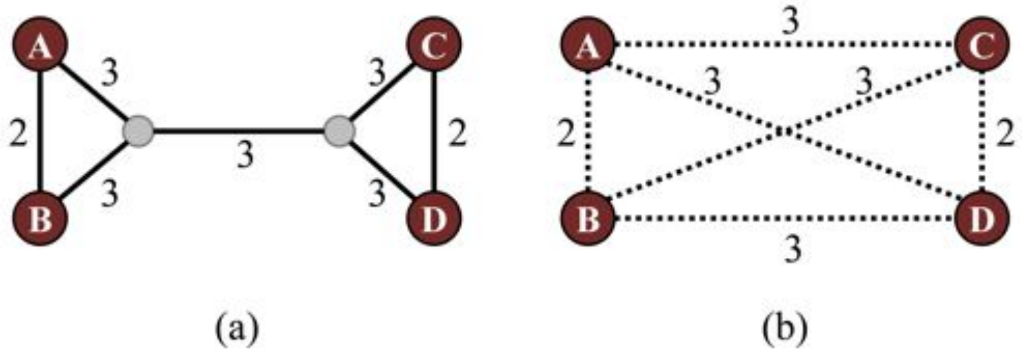


Figure 2.3: A small network that illustrates the cause and effect of *link correlation*. The physical network is shown in (a), whereas the overlay build on this physical network is shown in (b).

The authors observe that the inter-packet spacing of consecutive packets emitted by a bottleneck is largely constant, and therefore, multiple flows sharing the same bottleneck also exhibit this behaviour. Based on this, it is possible to cluster all incoming flows based on their common bottleneck. This approach yields a high accuracy (90% to 99% [79]) if a large proportion of the traffic crosses a common bottleneck, however the accuracy drops sharply if this is not the case [79]. For dividing flows into the correct clusters, the authors compare Sample-Weighted and Cluster-Weighted KMeans clustering against a 2 Dimensional Cluster-Weighted KMeans approach, which is not only based on the current inter-packet spacing, but also on the inter-packet spacing of the packet before that. They note that the 2 Dimensional approach outperforms the other approaches, and suggest that using even higher dimensional features might increase robustness, in case a lot of cross-traffic that does not share a common bottleneck is present. [80] uses the clustering algorithm of [79] to introduce overlays with *linear capacity constraints* (LCC), and present a heuristic to construct LCC overlays. In contrast to regular overlays, LCC overlays feature a dependency matrix, and can thus model how traffic on one link may reduce the available bandwidth on multiple other dependent links. It should be noted that even though obtaining the maximum bandwidth path from one node to another in a regular overlay is computationally simple[10], finding the maximum bandwidth path in a LCC network is in fact proven to

[10]For example, the maximum bandwidth path can be obtained using as a modified version of Dijkstra's algorithm, in which low bandwidth links have very high costs.

be NP-complete [80], as it can be reduced to the Path with Forbidden Pairs problem (PFP, [81]).

Despite this, a heuristic that builds a multicast path in a LCC overlay based on observing link performance when the network is saturated is introduced in [82]. This heuristic can directly be applied to the Flocks overlay, so that shared bottlenecks can be avoided.

2.4.2 Biologically Inspired Routing

Biologically inspired algorithms play a major role in self-organizing routing mechanisms. They make use of agents that travel the overlay and communicate directly or through pheromones, to collaboratively determine the routes providing the highest LoS, which should then be used. Biologically inspired routing mechanisms are flexible and highly robust. As overlay networks may be prone to failure and high churn, they make excellent candidates for overlay routing. This section provides a review of the relevant approaches.

Ant Colony Optimization (ACO) [83] is arguably the most popular method for biologically inspired routing, and a large number of ACO-based routing algorithms have been proposed [84, 24, 85, 25]. They find paths with high QoS, in particular for overlays with high mobility and failure rates. Simulated ants (small, fixed size packets), are periodically introduced into the network, and are targeted at a random destination node. After reaching the destination node, they return to the source, leaving "pheromones" on paths they previously took. Good paths are travelled by more ants at the same time, thus accrue more pheromones. Subsequent ants will prefer paths with high amounts of pheromones, thus further reinforcing them, however they may also pick other paths with low probability. As a result, a certain degree of noise is introduced, that causes ants to explore other paths as well. As pheromones "evaporate" over time, a path that is congested, or that failed altogether is removed quickly, due to the lack of pheromone reinforcement ("bad news travel fast" [84]). It has been shown that following this approach ants will eventually find the route with the lowest cost [84]. However, once a route has been established, it takes significant time before ants converge to a better path. This is because ants initially only have a low probability of selecting, and thus reinforcing the better path ("good news travel slowly" [84]). In addition, the approach outlined above assumes that paths have symmetric cost, which is not usually the case on the Internet, and cannot deal well with cycles [24]. Substantial work exists to address these shortcomings: [84] introduced uniform ants, which pick each path with equal probability, and thus converge to better paths much more quickly. AntNet [24] uses non-uniform ants, and allows ants to record information about the path segments they visited while travelling the network. As a result, an ant travelling back through the network can reinforce paths with high QoS more, and path with poor QoS less. Furthermore, the memory of previously visited nodes allows the ant to detect cycles [24]. Cycles are explicitly removed from an ant's memory. As a result, paths that lead to a circle are not reinforced through pheromones, and are thus less likely to be considered by future ants. The memory, and thus the size of each ant, is dependant on its time-to-live.

Traditional ant-based approaches like these scale poorly with the number of nodes in the overlay: First, each node produces ants targeted at random other nodes in the overlay, meaning that they need to have knowledge of all other nodes. As a result, the routing table on each individual node increases linearly with the overlay size. Second, they produce significant traffic: In an overlay with N nodes, each node periodically generates ants targeted at $N-1$ nodes.

A modified AntNet algorithm that only uses local information has been introduced in [85], however, it only achieves half of the performance of the original algorithm. It successfully reduces routing tables to a size independent of the number of overlay nodes: Each node records only two pheromone reinforcement values for their direct neighbours, one to indicate the pheromones on its link, and the other indicating how good of a node it is to reach other nodes from. It is unclear how this limited routing information is used to route data packets.

The same authors introduced a Distributed Genetic Algorithm (DGA) that applies genetic mutation to routes, to find the optimal path. Each route is represented as a gene, containing a given amount of numbers. The i^{th} number stands for the output link on the i^{th} node that should be chosen by the agent. As a result, the size of the gene determines how much of the overlay around a node is explored. Once the agent has reached the final node listed in the gene, or finds itself located at a node of which

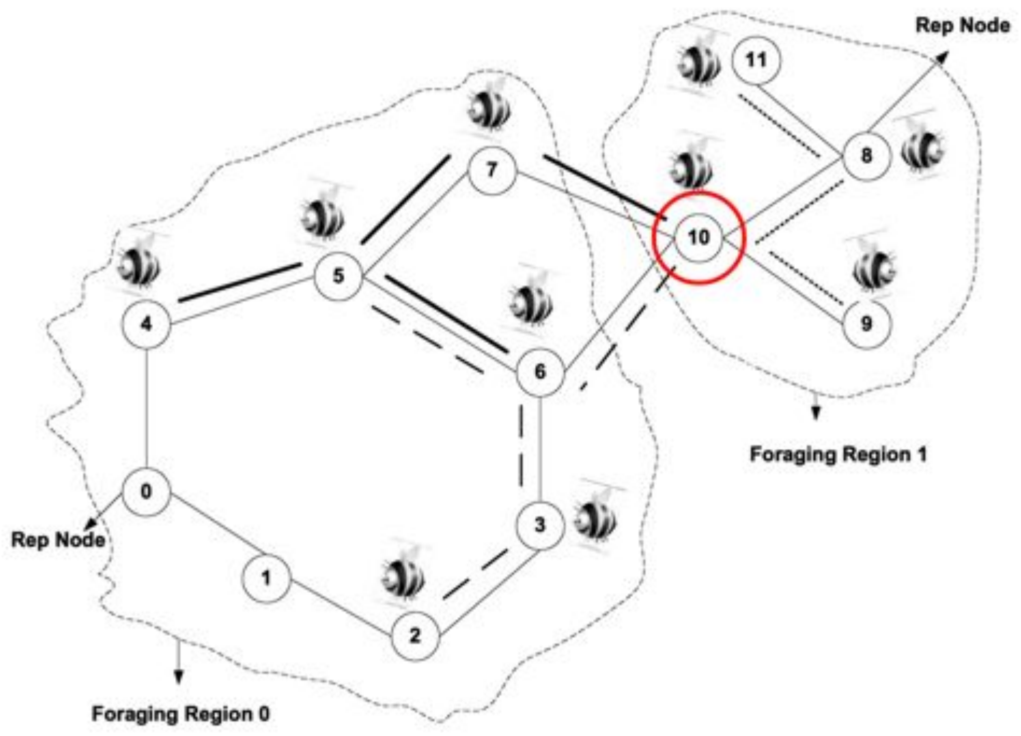


Figure 2.4: An overlay with eleven nodes executing the BeeHive algorithm (adapted from [2])

it has already explored all neighbours, it returns to the node it originated from. This node applies selection, cross-over and mutation between the genes of the agents with the highest fitness to generate two new genes. The genes of the agents with the lowest fitness are destroyed. While DGA only uses local information, it performs significantly worse than AntNet and well-established routing algorithms such as OSPF [86].

BeeHive [2] is an alternative approach that addresses the scalability issues of ant-based algorithm with so-called Bee Colony Optimization (BCO): It divides overlays into "foraging regions", which are then visited by simulated bees. Each foraging region has a representative node. Two types of bees are introduced: Short-distance bees are flooded by non-representative nodes, in a limited radius (measured in hops) around the node they originated from (the foraging *zone*). Long-distance bees are launched by representative nodes, and travel across foraging zones.

Figure 2.4 shows the BeeHive algorithm in an overlay with two foraging regions:

The network is partitioned into two foraging *regions*, each of which has a "representative node"[11], of which long-distance bees are launched (nodes 0 and 8 in the figure). Short-distance bee agents, which travel a distance of up to three nodes away from their originating node, are launched by node 10 (highlighted). The different types of lines next to the edges represent the paths followed by the different agents. Long-distance bee agents are launched by representative nodes, and travel a greater distance.

As BeeHive employs flooding, it does not require knowledge of what other nodes exist in the overlay. [2] states that a radius of 7 hops for short-distance bees and 40 hops for long-distance bees is sufficient to find the optimal path in their overlay. As bees are flooded, the neighbour tables of the nodes they passed are updated with the delay the bees experienced. Subsequent data packets use the information and probabilistically select the neighbour with the highest "goodness". In a way, dividing the overlay foraging region can be seen as hierarchical routing: Short-distance bees operate only within their region, while long-distance bees operate on a higher level, disseminating information from cluster to cluster. This theoretically allows BCO scale to larger overlays than traditional ACO approaches.

The authors of [2] found that the throughput of BeeHive is similar to the original AntNet [24], however with lower average delay. While no explanation is given, it can be assumed that BeeHive finds paths with low delay faster than AntNet due to the limited flooding they employ. Unlike AntNet, BeeHive does not require memory on the agents. Instead, the delay experienced by a bee while crossing a link is sent back to the preceding node, which updates its routing tables. Due to this, and the limited flooding, the overhead of BeeHive is 120% higher than AntNet. While the authors note that traffic overhead is independent of the number of nodes in the overlay, it is dependent on the radius that short- and long distance node explore. If the diameter of the network increases, this radius, and with it the traffic overhead, must be increased to guarantee finding the optimal path in the overlay.

[25] introduced a hierarchical system for ACO, similar to the one used in BeeHive, in which the overlay is split into clusters. The authors mention that clustering

[11]BeeHive uses the node with the lowest identifier in the foraging region as the representative node.

is based on the nodes' traffic patterns: For example, Spanish nodes likely exchange traffic between each other, and should be placed in the same cluster [25]. However, it is unclear what the actual criteria or clustering algorithm is. Regular ants only travel within the cluster they originate from. So-called "Super-Ants" are introduced to find routes between multiple clusters. SuperAntNet shows a trend towards bringing nodes that are interested together: Nodes that communicate frequently are likely interested in each other, and are placed in the same cluster. "Super-Ants" then travel between clusters of interested nodes. The idea of interested clusters may appear similar to Flocks. However, there are several important differences between Flocks and SuperAntNet. Flocks change the overlay structure and bring nodes that are interested in each other closer together. Consider that the Spanish nodes from the earlier example may be spread far apart. As a result, the cluster in SuperAntNet may be very large. Flocks reorganize the overlay so these nodes are close together, and in doing so, can also consider other factors such as QoS metrics. This is particularly important if nodes that are interested in each other benefit from detour routing. Finally, Flocks are more general, as a node's interest can be based not only on communication recency, but on numerous other factors. This will be shown in Chapter 4.

2.4.3 QoS Estimation and Measurement

Benefits from detour routing are very common on the Internet. As a result, the best neighbour is not necessarily the node with the highest LoS, but the node that provides the highest LoS to whichever destination that communication will take place with.

Measuring the benefit of detour routing is easily possible with ACO: If a node releases its agents only towards the overlay nodes it interested in, it will quickly establish good routes to them through the overlay. The aforementioned optimizations can be used to reduce the burden caused by ants, making ACO a robust and efficient approach if the structure of the overlay network has already been established. However, Flocks are unable to benefit from this, as they are concerned also with building the overlay network. As ACO can only use existing paths, it is necessary to use a fully connected overlay to ensure all possible routes are considered. While feasible for smaller overlays, this approach does not scale well to large numbers of nodes. In addition, ACO requires time to establish good routes, making it expensive to evaluate the suitability of a candidate neighbour. As a result, a different approach has to be followed.

Direct measurement is a well researched topic, for which significant literature exists. However, as the routing in overlay networks can differ from the routing in the underlying network, they work only for direct overlay neighbours. Essentially, different methods must be applied to measure different QoS metrics between direct neighbours:

- Packet Delay Variation (Jitter) can be obtained by either neighbour measuring the interarrival times of packets in a data transmission that spans multiple packets. Jitter is an additive QoS metric [87], as the delay variation adds up as data is transmitted from node to node.

- One-Way-Delay (OWD) is non-trivial to measure, as it requires that the clocks of both neighbours are perfectly synchronized. In this case, the sending time can be recorded in the packet, which is then used by the receiving node to compute the delay. Simply taking the half of the round trip time, a trivial to measure metric, does not achieve accurate estimations in networks where forward and backward paths and their delays are asymmetric [88]. Synchronization can be achieved with the Network Time Protocol [89], a GPS receiver that provides the wall clock time of the GPS satellites [90], or the IEEE 1588 Standard [91]. However, none of these methods are universally applicable: The accuracy of NTP in a Wide Area Network (WAN) is in the range of $10 - 20ms$, which may be greater than the transmission time itself [92]. GPS synchronization requires specialized hardware. The GPS receiver must be able to be in line of sight with at least one GPS satellite, which makes this method prohibitive for nodes in closed rooms, whereas the cost of the specialized hardware makes it prohibitive to employ on a large scale. The IEEE 1558 standard is restricted to synchronization within LAN boundaries [92]. [88] suggests estimations very close to the actual measured OWD are possible, however the quality of the results depends strongly on an initial estimation of the OWD. Similarly to jitter, One-Way-Delay is an additive metric.

- Loss rate can be observed by numbering packets that are transmitted to a neighbouring node. The neighbouring node reports back with feedback about any packets that were missed. This approach is followed by the Real-Time Transport Control Protocol (RTCP) [93]. For reliable protocols such as TCP, the number of retransmissions can be used instead. Loss rate is a multiplicative QoS metric [87], as a packet must not be lost on any path segment in order to arrive.

- The available bandwidth of the bottleneck link between two nodes can be estimated with the probe gap model (PGM) [94], in which pairs of packets are sent. A small delay is introduced between the packet pair, so that the second packet will end up in the queue of the bottleneck router before the first packet leaves it. The neighbour node then measures the interpacket delay between the packet pair to deduce how busy the bottleneck router is [94]. While this method achieves more accurate results and introduces less traffic than related packet pair approaches [95], it shares the susceptibility to jitter. Assuming that the propagation delay remains the same, and that the routes do not change, jitter is introduced by variations in queuing delay. These are reasonable assumptions, as it has been shown that Internet routes and their performance remain relatively stable [96, 97]. Bottleneck bandwidth is a restrictive metric (also referred to as a *concave* metric in the literature): The bottleneck bandwidth on a path is the minimum bandwidth experienced on any of its segments.

Substantial research has been done to estimate the QoS to any particular node: Geometric [28] and landmark-based [29, 30] methods estimate QoS metric in the underlying network. These methods can be used in overlays to estimate QoS metrics to a direct neighbour and potential new neighbours, however they suffer from the effects of TIVs [31] as they assume a metric space, in which TIVs can not occur [90]. They cannot, however, be used to estimate the QoS metrics to any node other than a node's direct neighbours. This is because the LoS an overlay node experiences to nodes other than its direct neighbours depends heavily on the overlay structure. As a result, the aforementioned estimation methods are only accurate if the overlay matches the topology of the underlying network exactly. In all other cases, traffic from the overlay network is routed over additional nodes in the underlying network, decreasing the accuracy of the estimation methods.

Topology Aggregation

Therefore, the overlay network itself needs to provide sufficient information on what overlay nodes it will route over, and what the QoS metrics between each pair of nodes on the route are. Providing and updating a complete view of the overlay for this purpose is not a feasible option, as it does not scale well: Every overlay node must be kept up to date when the overlay structure or the QoS of a link change. In addition, significant memory is required from each node to store the entire topology. Topology Aggregation (TA) is an established method to improve scalability [3]. Figure 2.5 demonstrates this concept on a 51-node overlay. Figure 2.5a displays the full hierarchy: The bottom-most level contains all nodes of the overlay, without any aggregation. These nodes are grouped into distinct "clusters", based on some criteria[12]. Each cluster is represented by a single (virtual) node in the next highest hierarchy level. This process can be repeated, in that several clusters are combined into a single super-cluster in the next highest level. The result is a compact, high-level representation of the overlay network that can be used for routing. Each node is only aware of the topology of its own cluster, and the topologies of any parent clusters. Figure 2.5b shows only the part of the hierarchy that nodes in the bottom-left cluster are aware of.

TA is a straightforward process: First, several nodes are grouped into "clusters" based on some criteria. Second, the QoS metrics within the cluster are aggregated to reduce the amount of information that must be advertised. Nodes outside the cluster no longer see its internal structure, and are no longer directly aware how packets within the cluster will be routed. Instead, they see aggregated QoS information, such as what delay to expect if data is routed through the cluster. This process can be repeated hierarchically: several clusters can be grouped into a super-cluster. The QoS metrics of all clusters within are aggregated, and as a result, clusters outside that super-cluster no longer need to be concerned with its structure. Hierarchical

[12]This is explained in more detail in the following Section 2.4.3).

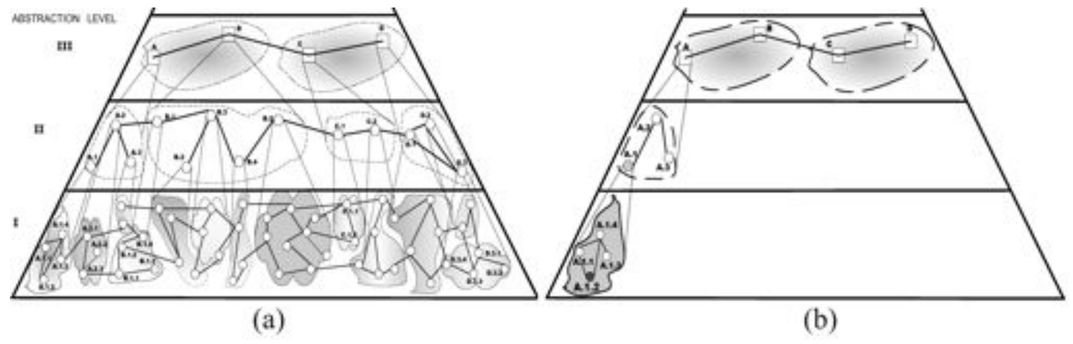


Figure 2.5: An overlay with 51 nodes is aggregated into a three level hierarchy (adapted from [3])

routing is considered to be a critical aspect for maintaining scalability [98].

The Flocks overlay uses hierarchical topology aggregation as shown in Figure 2.5 for routing, and to allow estimating the QoS that can be provided to an arbitrary node. The construction, the structure and the estimation algorithm are explained in Chapter 6.2.

Distributed Partitioning

Partitioning an overlay into clusters of nodes for TA is not trivial, as graph partitioning itself is an NP-complete problem [81]. Several efficient and scalable heuristics for k-way partitions have been proposed [99, 100], however their use requires choosing an appropriate k to ensure that the individual partitions are small enough that routing and full views within each partition are still feasible, and at the same time large enough so TA can reduce the complexity of the overlay in as few iterations as possible. The number of nodes in an overlay is usually not known: Optimal distributed counting algorithms exist [101], however the overhead is significant. As a result, it is difficult to select an appropriate k in advance. It is possible to recursively bisect a graph until the partitions are small enough. However, this approach requires a higher runtime and results in worse partitions than a direct k-way partition [102].

[4] introduces an interesting variation: The authors use a stochastic approach, the influence model, to allow the network to self-partition. The influence model [103] works on directed graphs, in which every graph node has a specific state. Based on influences between graph nodes, each node may transition to a different state with

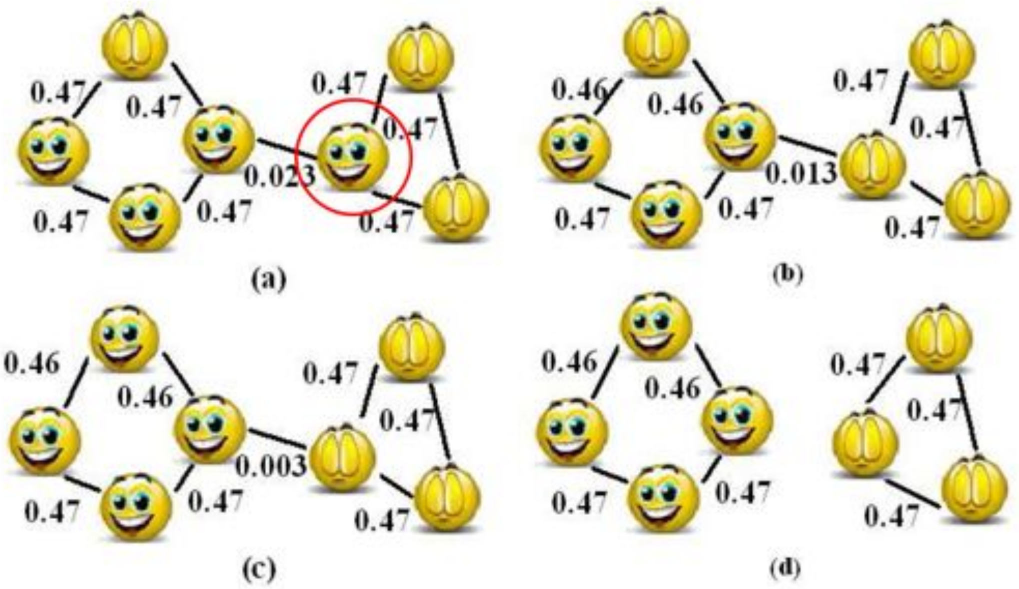


Figure 2.6: A seven-node network partitions itself based on the influence model, which different icons represent different states (adapted from [4])

a certain configurable probability. As a result, the influence model can be used to group the overlay graph into distinct clusters, based on their state, if probabilities are chosen carefully. The algorithm introduced in [4] is flexible with regards to what comprises a state: For overlays, each node could use its AS number. In this case, the overlay would be partitioned based on the AS of the nodes.

The algorithm introduced in [4] uses carefully selected probabilities to ensure that a node with many neighbours, which have a different state than itself[13], will be influenced by them, and eventually assume their state with a very high probability. In Figure 2.6, a seven-node network partitions itself using the influence model as described in [4]. The network consists of two strongly connected clusters, connected by a single link. Each cluster has a node of a different state. It can be seen that the two links from the rightmost nodes in (a) exert a much greater influence on the highlighted node than the link from its only matching neighbour. As a result, the highlighted node changes its state in (b). The same process can be observed on the top-most node of the left cluster between (b) and (c), after which the algorithm is stopped.

[13]For example, a single node belonging to AS 760 that is surrounded by nodes belonging to AS 1111 will assume the state of belonging to AS 1111 instead of forming a separate group.

It should be noted that this approach does not change the overlay structure. In order to build meaningful partitions based on AS number, delay or another metric, appropriate nodes must already be grouped together. After a given number of iterations the algorithm partitions the network based on the state each node is in, and then terminates.

This algorithm has a number of favourable properties. First, it is flexible regarding what metric is used for a node's state. If appropriate metrics are used, a network is partitioned based on "weak links", such as an AS or a domain boundary. Second, it is completely decentralized: A node only needs to communicate with its direct neighbours, in order to convey its current state. As the size of each partition is unbounded, however, it is not directly suitable for TA. For example, if the AS number is used as a metric, and all nodes originate from the same AS, no partitioning would occur. Furthermore, in an overlay every node may well be from a different AS. If all AS numbers are different, only trivial partitions with a single node are created, unless, by chance, one node influences another to assume its state. Everything considered, this means that a suitable metric has to be known *a priori*. Introducing artificial entropy[14] alleviates this problem, but also diminishes the quality of the partitions.

An alternative approach would be using the algorithm at the bottom of the hierarchy, to partition the network at domain boundaries, and use a different partitioning algorithm to build the remaining hierarchy levels. This improves the locality and, as a result, the quality of the aggregation by ensuring that nodes with very different QoS will end up in different clusters for aggregation purposes. This was not pursued further by me due to the complexity of global synchronization (which is required by the algorithm), and because Flocks are able to increase the accuracy of the aggregation on demand, by breaking up parts of the hierarchy again.

In either case, a number of other properties of the algorithm must be considered: As mentioned before, global synchronization is necessary to ensure all nodes have updated their influence probabilities before proceeding to the next "time step". In addition, knowledge of the overlay graph is required to set Δ, a global scaling parameter used to compute the influence probabilities. While this parameter can be based on

[14]For example, a random number between 0 and k can be assigned, resulting in a k-way partition of the overlay.

knowing the maximum connectivity of any single node [4], this information may still be difficult to obtain. Deciding when to stop the algorithm is not trivial either: The authors achieve distributed termination by explicitly reducing the influences between nodes. Therefore, the quality of the resulting partition depends on carefully choosing an appropriately small influence reduction size, which in turn increases the iterations the algorithm requires. The simulation uses a network of seven nodes, and while it outperforms existing algorithms, [4] notes that it is not clear how the runtime scales with the size of the graph.

A very different approach, as opposed to k-way partitions (which means limiting the number of clusters and making the number of nodes variable), is limiting the number of nodes within a cluster (and making the number of clusters variable). This appears to be a more reasonable all-purpose method for aggregating the topology of overlays. A simple algorithm that can easily be adapted to accomplish this goal is first introduced in [5], called "Basic Partition". The Basic Partition algorithm takes two parameters: A graph with n nodes, and a parameter $\kappa \geq 1$, which decides how sparse a cluster may become. The algorithm itself is straightforward. Any particular node begins adding neighbouring nodes in layers of increasing distance to its own partition. Only nodes that are not already part of another partition are considered. In the first step, it adds the first layer, its direct neighbours, to its partition. In the second step, the second layer, the neighbours of those nodes that were added in the first iteration, are added. This process continues until fewer than $n^{1/\kappa}$ nodes have been added in the last iteration ("sparsity condition"). At that point the partition is declared as finished. Finished partitions no longer participate in the algorithm. A new node that is not already part of a finished partition is randomly selected, and the algorithm continues until all overlay nodes are part of a finished partition, at which point the algorithm terminates.

Figure 2.7 illustrates this process. The highlighted node in the centre begins executing the Basic Partition algorithms, and starts adding neighbouring nodes in layers of increasing distance to its own partition, which are numbered marked by the dashed lines. Assume that $\kappa = 3$ and $n = 30$ (only 15 nodes closest to the central node are shown for illustration purposes). In this case the algorithm stops if fewer than $n^{1/\kappa} \approx 3.11$ nodes have been added in the last iteration. As a node running the

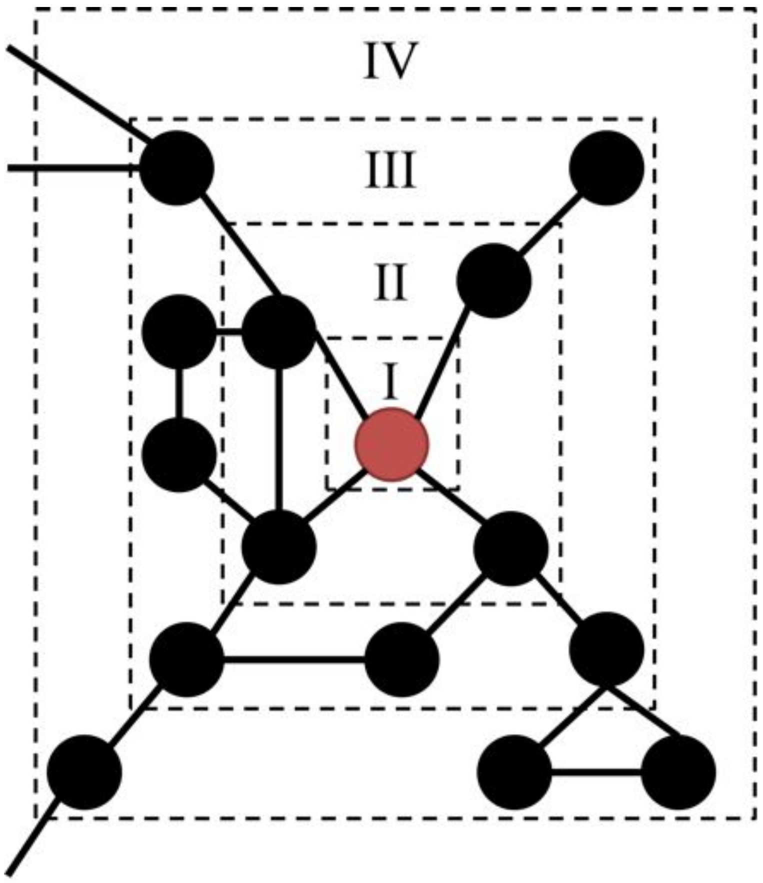


Figure 2.7: The partitioning performed by the basic partition algorithm, as described in [5]

algorithm might not be able to add any nodes at all, it is convenient to see adding itself as the first layer. If none of the other nodes are already part of a different partition, all four nodes in layer II are added in the second iteration. The third iteration adds the seven nodes of layer III. Layer IV only adds three nodes, and as a result, the partition is finished. The algorithm now selects a random node that is not yet part of a finished partition and repeats this process. If no such node can be found, the algorithm terminates.

The Basic Partition algorithm, as introduced in [5] has several disadvantages. First, it is centralized and fully synchronized. A partition must be finished before a new partition starts growing around a new, random node. Second, it also assumes the existence of a global membership table, listing all nodes in the overlay.

A highly interesting, fully distributed variant of the Basic Partition algorithm, in which clusters grow in parallel, has been presented by [33]. The authors introduce an identifier for every partition, which is set to the identifier of the node that the partition starts growing at. All nodes inside the partition are assigned the identifier of the partition. As a partition grows, the partition with the highest identifier prevails against other partitions and “wins” their nodes.

It should be noted that a similar approach is followed by [104]: In the Basagni Algorithm, a node can either be a “clusterhead” or a follower, and its identifier decides what role it assumes: If it has the highest identifier of all its non-follower neighbours, it becomes a clusterhead, otherwise it becomes a follower of the cluster the neighbour with the highest identifier belongs to. [105] uses the same concept in order to build a leaderless hierarchy, arguing that acting as a clusterhead does not impose additional responsibilities on a node, as it simply determines what identifier a partition has. While this idea allows simple and scalable partitioning, it requires that every follower node is a direct neighbour of its clusterhead, and so is limited to partitions with a diameter of at most three hops. As a result, hierarchies may become very high, in particular if the overlay is spare. In contrast, the Basic Partition algorithm supports partitions of arbitrary size.

An overlay is partitioned based on the adapted Basic Partition algorithm [5] in two steps: Initially, all nodes simultaneously start out as a *singleton partition*, which contains only the node itself. They then individually go through three phases: An exploration phase, which is followed by either a growth phase, or a battle phase.

- During the *exploration phase*, every partition attempts to add one layer of nodes that neighbour it. If its own identifier is greater than the identifiers of all nodes it attempts to add, it succeeds, and enters the growth phase. Otherwise, it enters the battle phase.
- During the *growth phase*, the nodes in the partition verify whether the sparsity condition $addednodes \geq n^{1/\kappa}$ is still satisfied, in which case the partition switches to the exploration phase again. If the sparsity condition is violated (that is, $addednodes < n^{1/\kappa}$), the outmost layer is dropped again, and the partition is declared as finished. The nodes that were dropped assume singleton partitions again, with their original identifiers.
- A partition executes the *battle phase* if at least one node it attempts to add has a greater identifier than its own. In this case, the partition loses its outmost layer. Similar to the layers dropped in the growth phase, all nodes in the layer that was lost assume a singleton partition again, with their original identifiers.

Compared to the Basic Partition this approach has a number of helpful properties: First, it runs in a fully distributed fashion. Every node executes the algorithm in parallel, and no central coordination is required. In particular, the algorithm does not require synchronized clocks or leader election. Even though the protocol introduced in [33] is implicitly synchronized, clusters can still grow simultaneously in most cases[15]. Second, it has low runtime in the average case, and functions by only exchanging messages of a small, constant size. Finally, the sparsity condition produces partitions with bounded radius [33], and can easily be replaced by a more general stopping condition. In particular, a stopping condition that only allows a certain partition size (such as, for example, ten nodes) is trivial to implement, as the current partition size is already computed during the growth phase.

[15]If nodes are arranged in a line with ascending identifiers, all nodes but the last node with the highest identifier will fail to add a new layer. As a result, none of the other nodes can proceed until the cluster originating from the last node is declared as finished. At this point, the process repeats, and only the node with the highest identifier that is not part of the finished cluster will succeed. This represents the worst case, yielding the highest time complexity of $O(\kappa^2 n^{1-1/\kappa})$.

Applying the Basic Partition to self-organizing overlays, however, is not straightforward. Recall that the original sparsity condition requires two parameters, n, the number of nodes in the overlay, and κ, which decides how sparse a cluster may become. If κ is chosen to be too large, sparse regions result in very small partitions. For example, refer to Figure 2.7 and recall that $\kappa = 3$. If $\kappa = 4$, $n^{1/\kappa} \approx 2.34$ and only one layer would be added. One of the remaining nodes would begin growing the next partition, however as the network (excluding the now finished partition) is very sparse, none of the nodes would add more than one layer. Therefore, many partitions are created, and in consequence a hierarchical aggregation results in higher hierarchies than necessary. On the other hand, if κ is too small, partitions become larger, and may even result in covering the entire overlay. For example, setting $\kappa = 2$ in Figure 2.7 causes the entire visible portion of the overlay to form a single partition. Following this example, it is clear that the best value for κ not only depends on n, but also the overlay structure (specifically the degree of nodes), and the placement of nodes with high identifiers, none of which are usually not known for dynamic, unstructured overlays.

Replacing the sparsity condition with a more general stopping condition, which stops growing a partition once it exceeds a certain size, gives rise to other issues: Aggregated representations in higher hierarchy levels are frequently very dense, and contain large hubs (nodes that are surrounded by many neighbours). Consequently, if the aforementioned stopping condition is implemented, adding a single layer of neighbours already results in very large partitions, which exceed the desired partition size by far. Dropping the last added layer after a partition exceeds its desired size is counter-productive as well: In this case, hubs are not aggregated at all, and the hub centre (the node that is surrounded by many other nodes) becomes a singleton partition that contains only itself. Once the entire overlay representation consists of such hubs, no aggregation is possible any more. Despite this, the flexibility and the scalability of the Basic Partition makes it an excellent candidate to partition an overlay for TA. Overcoming the issues laid out is the focus of Chapter 6.2, which also presents an adapted version of the algorithm introduced in [5] that creates hierarchies with low height and strictly bounded cluster size.

Aggregation of QoS-related metrics

Once an overlay has be partitioned into clusters of nodes, various methods can be used to aggregate QoS metrics within a cluster. QoS aggregation methods aim to reduce the size of advertisements that need to be sent to other clusters, while attempting to keep the quality and the amount of *information* the same. This is done by transforming the topology of nodes within each cluster into a simpler one, a process referred to as “Topology Transformation”. The QoS experienced for traversing this reduced topology is then advertised to other clusters in a compact representation called “QoS epitome”.

A large number of different topology transformation methods exist. An survey comparing the different methods is found in [3]. An extended version of this survey [1] also gives a detailed comparison of related work in this field that applies and evaluates one or more of these topology transformation methods.

The QoS aggregation presented in this thesis does not limit itself to a specific topology transformation method, and any transformation method described in [3] can be used. As a result, I selected the Full Mesh topology transformation method for its ease of implementation. The Full Mesh topology transformation advertises the QoS experienced by traversing the cluster for every pair of border nodes. The best representation of this advertisement depends on the number, and also what QoS metrics are used.

For single QoS metrics, the best path (lowest delay, highest bandwidth) through each cluster should be used [106], which reduces the size of the advertisement to a single value per entrance and exit points in the worst case, while providing all relevant information to other clusters. For multiple QoS metrics, aggregating QoS is significantly more difficult. [107] introduces a line segment approximation scheme, in which an additive and a restrictive QoS metric (such as delay, and bottleneck bandwidth) for each path are plotted on a 2-dimensional coordinate system. By connecting the points representing the various path, one obtains a staircase that separates the coordinate system into two regions: A feasible one, which contains all delay-bandwidth combinations that can be fulfilled, and an infeasible one. This staircase is then approximated with one or more lines, pursuant of two conflicting goals: For one, the approximation should include as much of the feasible region, and as little of the infeasible region as possible. This reduces the amount of incorrectly rejected requests (requests that would be feasible in the original graph, but fall outside the feasible region in the approximation) and the amount of "crankbacked" requests (requests that are infeasible in the original graph, but fall inside the feasible region in the approximation). At the same time, however, the approximation should use as few lines as possible, reducing the amount of data that needs to be exchanged.

Figure 2.8 illustrates this concept. The shaded area represents the "feasible region" of delay-bandwidth combinations that can be fulfilled by the network, the unshaded area represents the "infeasible region" of delay-bandwidth combinations that cannot. The dashed line represents a possible line-segment approximation of the original graph that is advertised to other networks. It also introduces two more regions: The "crankback region" of requests that are admitted based on the approximation, but are later rejected as they are not feasible. The shaded regions left of the dashed line represent regions in which requests are "incorrectly rejected" based on the approximation, even though they would, in reality, be feasible. Approximation techniques aim to reduce these two regions, while also approximating the original area with as little data as possible.

The amount of data can further be reduced by different representations, such as line-fitting [108], polynomial curves, splines and poly-lines [109]. They also increase the accuracy of the approximation over the line-segment approximation scheme. An excellent survey is given in [3].

I found it reasonable to not restrict the Flocks overlay to a specific aggregation method. Instead, Flocks permit an application to select the aggregation method used, and also advertise the error of the approximation. This is important because the Flocks overlay permits breaking up aggregated information during its multi-level QoS estimation. Based on the error, nodes can decide how useful it would be to break up the aggregated information, and consider a less aggregated (and therefore more accurate) hierarchy level instead.

The topology transformation method used in this thesis includes the best-case QoS between any pair of border nodes in the epitome. This can intuitively be represented with the line segmentation approach. However, it should be noted that the more

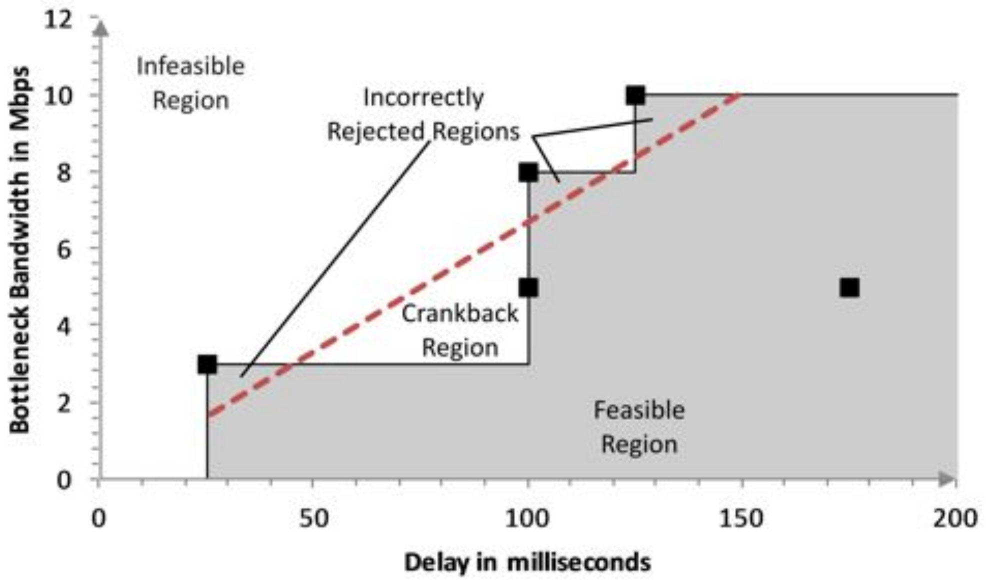


Figure 2.8: QoS parameters of five paths through a small network plotted on a delay-bandwidth plane.

complex methods described before can be used for Flocks as well, as long as their output is compatible: The only requirement Flocks have is that the data can be used to obtain an estimation from each entry to each exit point of a cluster, as well as the error of the approximation.

Topology Aggregation in QRON

Similar to Flocks, QRON also implements TA [72], however there are several differences in the hierarchical aggregation: First, QRON bases its clustering on physical closeness and AS related-information. While this promises cluster boundaries that closely match the structure of the Internet, the information this clustering required is, as mentioned above, difficult to obtain. In contrast, Flocks work on any topology, and create bounded-size clusters based on the nodes' location in the overlay. They require no additional information either, except that each node must have an unique identifier, such as an IP address. As the cluster boundaries do not necessarily match the AS boundaries, multiple AS may be included in a cluster. As a result, the QoS within a cluster can vary greatly, which is a disadvantage of Flocks. However, Flocks allow multi-level QoS estimation by breaking up aggregated information, alleviating this issue.

Second, QRON adopts a top-down approach to build the hierarchy [110], requiring a dedicated root node that is known to all other nodes in the overlay. This dedicated node receives a substantial amount of load, which makes this approach difficult to scale. Flocks introduce a bottom-up approach based on distributed partitioning that is completely decentralized, does not require dedicated nodes, and in which nodes at the top of the hierarchy do not receive more load than nodes near the bottom.

Flocks and QRON [72] both provide algorithms for QoS aware routing through the hierarchy. In contrast to Flocks, QRON requires that the entire path is made known to the source node, meaning that the hierarchy must be completely broken up before data can be sent. Flocks allow the source node to specify high-level routes that are only broken up into lower-level routes when it becomes necessary. The information required to break up high-level routes into low-level ones is placed strategically throughout the network, so the nodes that need to break up routes to continue routing already have the necessary information. In particular, no additional information needs to be sent over the network to facilitate routing, and every node can reach every other node using the information it has locally available. This has a particular importance for QoS estimation: Every node can obtain a high-level QoS estimate to any other part of the overlay using only local information, allowing it to quickly and cheaply estimate the benefit a new neighbour may bring. Flocks still allow breaking up specific or even all clusters, to obtain less-aggregated information about specific sections of a route. For example, a source node may want to break up part of a route that has a large variation of aggregated QoS information. However, this is optional and not required for routing. It is unclear how QRON deals with stale QoS information.

CHAPTER 3 Requirements

The previous chapter briefly listed the key requirements for a multimedia overlay: Scalability, robustness, flexibility, and support for quality of service related metrics. This chapter describes each of the requirements, their importance for overlay networks, and discusses how they have been considered in this thesis.

3.1 Performance

In order to remain *scalable*, the performance of an overlay must not degrade unreasonably as more nodes join the network. This thesis evaluates an overlay's *performance* at its lowest level:

- First, it is concerned with the management overhead for building, improving and maintaining the overlay. Simply considering the number of messages exchanged with neighbouring nodes is not sufficient, as each message may then be infinitely large.

  Research commonly considers the number of messages exchanged for their evaluations. Indeed, one could suggest that considering the number of messages is important, as each message incurs a size overhead (due to packet and protocol headers), and processing overhead (the system calls needed to send and receive the packets). However, I argue that the number of messages alone is a poor metric for the Flocks overlay, as an application layer overlay can employ buffering to alleviate both of the aforementioned problems. Furthermore, the number of packets alone does not give any indication of how fast or slow an overlay is. In fact, an overlay may easily be faster than another, even if it exchanges a greater amount of messages. Assume a protocol in which two nodes exchange six packets, but have to wait for an answer of the other node after each packet.

Due to the propagation and queuing delay in the network, the time required for each of the three packet pairs is equal to the round-trip (RTT) of the network, including a processing delay that will not be considered further. The entire exchange would therefore complete in $3 \cdot RTT$ milliseconds. A second protocol, in which two nodes exchange eight packets but do not have to wait for an answer, completes in RTT milliseconds. As seen, the second protocol performs faster despite using a greater number of packets.

Therefore, I conclude that it is more realistic to measure the management overhead by considering how much *management traffic* is produced expressed, for example, as a measure in bandwidth. Sending messages produces protocol and packet overhead, and is therefore considered by this metric as well. A multimedia overlay must to strive to keep this metric as low as possible, because management traffic reduces the bandwidth available for multimedia delivery. Management traffic in this thesis is measured in kilobit per second ($kbit/s$) in order to relate to video streams, which use the same metric.

- Second, the overlay must, given enough time and stability, eventually reach the optimal configuration. For example, an overlay configured to maximize the bandwidth to its video sources needs to discover alternative routes, replacing bottlenecks with higher bandwidth links. Routing in the underlying network is generally not transparent to the overlay. Connecting two nodes in the overlay forces the underlying network to route between them, and may therefore discover new, better paths. As a result, an overlay must create new virtual links between nodes in search for a better LoS, and not only evaluate its current links, or rely on informations provided by other nodes. Evaluating how well an overlay performs in this respect is not trivial. While improvement (for example, based on the average delay in the overlay) can be measured, it does not give any indication as to how close to the optimum the overlay is. If the optimal configuration of the overlay is known, however, metrics relevant to the QoS between the optimal and the current overlay configuration can be compared.

- Third, the time to reach a stable configuration must be kept as low as possible. Social events have a relatively short lifespan in which they happen: For example, Austria's "Long Night of Research" event, in which universities offer the public insight into current research work, was confined to a six and a half hour relevant time frame. If the overlay configuration changes frequently, existing data flows experience delays, jitter, and possibly even packet loss. As a result, it is important to consider the time an overlay takes to reach a stable configuration, in which (in absence of factors outside the overlay's control, such as churn or failure) large-scale optimizations are no longer necessary. Unnecessary delays in achieving this configuration, and oscillation, in which the overlay continuously alternates between the same configurations, must be avoided to keep this time low. Furthermore, if churn or failure occur, it is important that the overlay quickly reaches a stable configuration again.

3.2 Scalability

Overlays for social events must expect a large number of participants. In small overlays with only a dozen of nodes, a complete view can quickly be obtained, and the overlay configuration can simply be computed on a dedicated node. However, as the number of nodes in the overlay increases, solutions like these are no longer feasible. The "Long Night of Research" event counted 7,000 visitors in Klagenfurt alone, so it is necessary that the performance of the overlay scales from hundreds to thousands of nodes.

Formal proof of scalability for the Flocks overlay is very complex, as the operations of each node involve a certain degree of randomness, in order to escape local optima. Furthermore, nodes may run at different speeds, and at different load. Therefore, I pursued a practical approach to evaluate the scalability of the Flocks overlay. Specifically, I measure the previously mentioned performance metrics (management overhead, time required to find a stable configuration, closeness to the optimal overlay configuration) while adding increasingly many nodes to the overlay. By comparing the measured values for overlays with 100, 200 and 500 nodes, it can be seen whether, for example, the values increase logarithmically, linearly, or exponentially.

3.3 Robustness

Even once an overlay network has reached a stable configuration, nodes may join and leave the overlay, and reconfiguration may be necessary. Therefore, different scenarios must be considered, all of which determine how *robust* an overlay is:

- Flash crowds: Flash crowds is said to occur if a large number of nodes joins the overlay in a very short time span. They are commonly seen on content that suddenly gains significant popularity because, for example, something interesting, unusual or funny happened. For example, if an unusual event happens during the Long Night of Research, many visitors would direct their devices to the content in question, thus joining the overlay and waiting for video of the event to stream. After the initial surge, the number of nodes declines again. In this thesis, flash crowds are considered by the initial bootstrapping phase, in which a large number of nodes joins the network instantaneously.

- Churn: Overlay networks are rarely stable. The continuous process of nodes joining and leaving the overlay is referred to as churn. A joining node must be placed in an appropriate section of the overlay quickly, so that it can start participating. In addition, powerful nodes that join should be placed in key positions, so that they contribute their resources to the overlay and improve the level of service other participants receive. Conversely, the overlay must quickly be repaired as nodes leave, so that data can be accessed without interruption. As churn increases (nodes join and leave faster), it becomes more and more difficult for the overlay to keep up with the changes. Therefore, it is important to consider how much churn an overlay can handle. In my evaluations, I simulate heavy churn as follows: In a 100 second period, every node leaves the overlay. A node that leaves the overlay loses its entire state: It forgets the neighbours it had, and rejoins the overlay with a random list of neighbours fifteen seconds later. This process is repeated three times. Therefore, in a 300 second period, every node leaves and rejoins the overlay thrice.

- Failure: In contrast to churn, in which single, equally important nodes continuously join and leave the overlay, a failure occurs if many nodes leave the overlay at once (for example, due to an outage or a software problem), or if a critical node leaves the overlay (such as one of the two only video sources). It is important that the overlay quickly adapts to the situation, in order to restore service. In addition, it must be avoided that the overlay becomes partitioned, that is, the overlay becomes divided into two or more unconnected components. This thesis evaluates a large scale failure, in which 40% of the overlay nodes fail at the same time, which matches the scale of failures considered in related work.

3.4 Flexibility

The aforementioned requirements are not trivial to fulfil, and they often conflict with each other. For example, it is possible to find an optimal configuration quickly by building a centralized view of the overlay, computing the solution, and reorganizing the overlay. If nodes join or leave, the process is repeated. While this approach will undoubtedly find a good configuration, it produces a high amount of traffic, assumes that the overlay remains stable during the computation, and therefore does not scale. A fully decentralized approach scales better, but requires more time to discover the optimal configuration. This time can be reduced by faster and broader searches, however at the cost of an additional bandwidth requirement.

As can be seen, many of these requirements require a compromise: An application may focus on faster optimization if the additional bandwidth requirement is not an issue. Alternatively, it may accept a slower optimization speed if less traffic is produced by the overlay in turn. Overlays should be *flexible* and adaptive to seamlessly support either approach, as requirements may be dynamic even for a single application: For example, if an overlay is lightly loaded, using additional bandwidth for faster optimization may be acceptable, however as user traffic picks up, slower optimization may be desired instead.

Flexibility does not only concern parameters such as how quickly an overlay improves, but also the criteria used for optimization. For example, a node of a certain overlay may be interested in other nodes that receive the same video stream as itself, so that it can exchange data in an approach similar to BitTorrent. Another overlay could be built for a sports event: Videos are recorded by mounted surveillance cameras and automatically tagged with whatever athletes were detected. In this case, nodes may be interested in very specific athletes, or they may be interested in whichever athletes are currently in the lead. Supporting such different scenarios requires a great deal of flexibility.

In order to offer this degree of flexibility, overlay networks generally use an oracle [69, 68]. This oracle is provided with information about the nodes that should be considered, and returns a number (usually between 0.0 and 1.0) that determines how suitable the nodes are as neighbours. It is clear that the oracle must be able to access as much information as possible, to retain its flexibility. For example, in the matchmaker approach [26], a node ("the matchmaker") decides whether two of its neighbours should connect to each other, instead of itself. However, while this approach works well for content- and class-based overlays, it can not make any decisions based on QoS metrics, because these metrics are usually not known[1] until both neighbours connect. As a result, matchmakers cannot make a well founded decision in this case. Therefore, in order to consider both, QoS-related metrics, as well as content, a new approach, not less flexible than the oracle-based method, must be developed.

[1]Recall that this is due to the fact that the underlying network is considered a blackbox and may use a different path to route between the two neighbours, if they communicate directly.

CHAPTER

4 The Flocks Overlay Network

Based on the requirements described in the previous chapter, I designed a self-organizing, QoS-aware overlay, named “Flocks”. This chapter is divided into four major sections: In the first section, I describe the concept of Flocks and define the terms used throughout the thesis, and introduce the Interest-Property model, which is a fundamental concept that determine how Flock nodes operate. The second section describes the different modes of operation that each node goes through: Bootstrapping, Participation and Termination. It explains the actions a node must implement in each of the states. Section 4.3 reviews the requirements introduced in the previous chapter, and how they are achieved by the Flocks overlay. Finally, Section 4.4 lists three common distribution models, and shows how they can be implemented by Flocks using the Interest-Property concept introduced previously.

4.1 Definitions

Several new terms and concepts related to the Flocks overlay are used throughout this thesis. This section defines several important terms, together with the concepts they relate to.

4.1.1 What is a Flock?

The best way I found to define a Flock is to start with the smallest entity within a Flock - the *Flock node*. A Flock node is an application which uses the protocol described in this chapter (the *Flocks protocol*) in order to communicate with other Flock nodes. The union of all Flock nodes is called the *Flocks overlay*, or simply a *Flock* (of nodes).

The Flocks protocol consists of the algorithms described in this thesis. Any device that runs the Flocks protocol is a Flock node. The protocol is concerned with neighbour selection, overlay construction and overlay maintenance. It also provides functionality to send data to any direct neighbour. Routing and multicast is left to other protocols that build up on the Flocks protocol. For example, any tree-based protocol such as SplitStream [58] or OverCast [111] can build up on the overlay created by Flocks, in order to employ multicast. These protocols benefit from the optimizations performed by the Flocks underlay.

Without limitation, the devices the Flocks protocol is used on may range from small devices with very limited resources (such as a hand-held phone, or surveillance cameras), to powerful, dedicated servers that provide ample resources to the overlay network. A device may host multiple Flock nodes. For example, assume that an application uses the Flocks protocol to watch a single video stream[1]. A user may launch that application twice and watch two video streams at the same time, in which case the device is considered to host two Flock nodes.

A Flock node is addressed by an unique identifier. This unique identifier can be provided by the underlying protocol, but can be obtained by other means. While Flocks are not limited to any specific protocol, the simulations were performed with TCPv4/IPv4. For the IPv4 protocol, which is still used in today's Internet, the combination of IP address and port can uniquely identify a Flock node. The same holds true for the IPv6 protocol.

4.1.2 Property Sets

Every Flock node has its own *Property Set*, which consists of a set of sextuples that specify its assets (for example, access to a certain file, or stream) and general properties (for example, GPS location, installed memory, storage and computing capacities). Each sextuple consists of the following elements:

- Name: The name, through which the property is identified. As a trivial example, assume a node's IP address is stored in a property. In this case, the

[1]Note this is a limitation of the fictional application: If properly configured, a Flock node will position itself within the overlay so that it receives multiple video streams. This is described later in this chapter.

property's name could be "ip_address". The Flocks protocol reserves several property names that refer to QoS-related values. While an application can, except for reserved property names, use any property name it desires, guidelines to how a property name should be chosen are given in Section 4.1.2.

- Value: The value of this particular property instance. In the previous example, the value of the property could be "192.168.1.1". The format and the meaning of a property's value is application dependent.
- Time To Live (TTL): Determines for how many hops this property will be propagated. Every time a property is sent to a neighbour, that neighbour reduces its TTL by one before further processing it. A property with a TTL of zero will not be sent to any nodes. If a TTL of one is used, the property is sent to a node's direct neighbours (at this point the TTL is reduced to zero, and the property will not be sent any further). With a TTL of two, the property is sent to a node's direct neighbours (at which point the TTL is reduced to 1), which then send it to all their neighbours. By default, the TTL of a node's property is two. It can be further increased to increase the visibility of "important" nodes or properties. For example, if the node with the IP address "192.168.1.1" is a critical node, it makes sense to increase the TTL of its IP property.
- Confidence: A floating point value ranging from 0.0 to to 1.0, which represents the confidence in value's accuracy of the node that created the property. A value of 1.0 is used to assert that the node is certain that the value is accurate. Consider again the "ip_address" property: The confidence of the node in that property should be set to 1.0, as it obtained deterministically from a trustworthy source (such as the network layer).
- Originated From: The unique identifier of the node that created the property.
- Propagated From: When a node receives a property from one of its neighbours, it records what neighbour it received it from by storing that neighbour's identifier in this element. A node that creates a property leaves the "Propagated From" element empty.

In order to understand what properties can be used for, consider the following example: Your town hosts a large triathlon event, such as IronMan[2], during which athletes participate in swimming, running, and riding bicycles. A network of stationary cameras is set up, and each camera uses the Flocks protocol and provides a live video stream. Visitors that attend the event have access to devices that also use the Flocks protocol, and can use them watch videos.

Assume that stationary camera records part of the bicycling track. This camera has a "Position" property that contain its GPS coordinates as values, and a "Type" property that contains the string "stream", to indicate that it provides a video stream. In addition, since it records bicycling athletes, it also has an "Activity" property, with "bicycling" as its value. Finally, pretend that the camera can also identify any athlete that it records[3]. Therefore, it has an "Athletes" property, which contains the numbers of all nearby athletes in a comma-separated list.

A Flock node uses the properties of other nodes to consider whether they should be neighbour nodes. For example, a node that is interested in "bicycling" will look for other nodes that have this property. The Flocks protocol does not impose any restriction on what information is contained in these name-value pairs, nor the format that is used. However, since Flock nodes frequently exchange their property sets, it is suggested that the property set, and the properties themselves are kept small. It is also important that every Flock node in the overlay has the same understanding on which property names are used, and how to interpret their values. For example, a Flock node that does not know that the "Position" property exists (or that it contains a GPS coordinate) can not benefit from it. This "common understanding" must be ensured by the application by using consistent property names, and consistent interpretation of their values throughout the overlay.

Interoperability and Co-Existence

This previous example makes it clear that properties are highly domain specific, and application dependant. A node that wishes to observe a specific athlete must know

[2]http://ironman.com/

[3]For example, athletes could be equipped with a RFID tag in their jersey, which is read by an RFID reader as approach the camera.

that:

- A property called "Athletes" may exist on other nodes.
- If it exists, it contains a comma-separated list of numbers.
- Each number corresponds to the starting numbers of nearby athletes that were detected.

This thesis assumes that all nodes in a Flock know in which properties the information they need is stored in. In addition, it also assumes that they know how to interpret the property value. This is a reasonable assumption: The Flocks protocol runs as an underlay - a *substrate network* - of a specific application. Interoperability is the responsibility of that application, and therefore the application developer should publish what property names they use, and what their values signify.

Despite the above, multiple applications can co-exist in a Flocks overlay if none of the property names conflict. A conflict occurs if two different properties use the same name. As a result, it is recommended that applications prefix their own properties with an identifier globally unique to that application. For readability reasons, however, properties in this thesis are not prefixed with globally unique identifiers.

A Flock node is not required to know all properties it sees on other nodes: An unknown property is ignored. Other nodes should assume that a node that lacks a certain property is not aware of it, and must make reasonable default assumptions. For example, in a video delivery network, a property "will_relay" could exist that contains a boolean, which declares whether or not the node in question will relay video content to other nodes. If this property is not present on a node, its neighbours must assume that this node is not aware of relying content, and will, in fact, not relay video content.

Virtual Properties

Quality of Service related metrics are accessed as properties as well. However, they require special consideration: As the routes in the underlying network may be asymmetric, their QoS related metrics may be asymmetric as well. For example, consider the network in Figure 4.1. The labelled nodes are Flock nodes, the unlabelled node

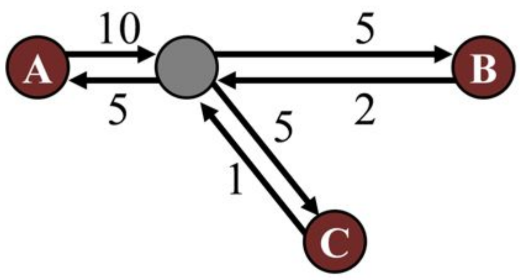


Figure 4.1: A simple network with four nodes to illustrate virtual properties.

a router unaware of the Flocks protocol. The numbers on the links represent its bandwidth in Mbps. Even though node A is connected to the router with a 10 Mbps link, the bottleneck bandwidth available to the application changes depending on what neighbour data is transferred to, and what direction ("from A" or "to A") it is transferred. Specifically, the bottleneck bandwidth from node A to B is 5 Mbps, whereas the bottleneck bandwidth from node B to A is 2 Mbps. If node B evaluates whether node A should be a neighbour, it has two values to consider: Its downstream and its upstream bottleneck bandwidth. The downstream QoS metric is announced as a regular property by node A (under the property name "qos.bandwidth.down"). The upstream QoS metric, however, is measured by node B itself. In order to make this process transparent to the "oracle" that computes whether node A should be a neighbour, the result is treated as if it were a property announced by node A (under the property name "qos.bandwidth.up"), even though it was node B that created it locally. Such properties are called "virtual properties", as they do not truly exist on the node they claim to originate from, but for abstraction reasons are treated as if they do.

Propagating QoS related metrics makes little sense: The routing used by the underlying network to connect a node to its neighbour nodes may differ substantially for another node. In this case, the QoS experienced by the involved nodes is likely to be different as well. For example, refer to Figure 4.1: The bottleneck bandwidth from node B to C is 2 Mbps. It makes little sense to propagate this information to node A, as the bottleneck bandwidth from node A to C is higher, namely 5 Mbps. As a result, all QoS related properties are created with a TTL of zero.

Despite this, the QoS experienced to a target node on a certain overlay path can be important information. For example, assume that the above network in Figure 4.1 is fully connected, that node “A” provides a video to the network, and only accepts two neighbours. A new node “D” that joins the overlay can, therefore, choose between two neighbours: node B and C. Given the goal of streaming the video over the network, the bottleneck bandwidth to node A is important. One could argue that QoS related properties should be propagated as far as possible, in order to support this scenario. However, this would cause significant maintenance overhead as QoS related properties are prone to changing. As a result, the properties would need to be re-propagated frequently through large portions of the overlay. Therefore, Flocks use a more sophisticated mechanism based on hierarchical topology aggregation, described in Chapter 6.2, to enable efficient QoS estimation in the overlay.

Uncertainty in Properties

Properties have an uncertainty factor to help modelling situations where the value of a property is predicted and cannot actually be determined immediately. An example would be transmission delay, predicted with common distance estimation methods described in [28, 29, 30]. At the time two nodes first connect and exchange properties, an uncertain property must be set to a reasonable value with a high uncertainty factor. Over time, as both nodes exchange data, the property stabilizes at its actual value and the uncertainty factor decreases.

A node that has too many neighbours takes uncertainty into account when it decides which neighbour to drop. This prevents a new candidate node with high, but uncertain QoS metrics from replacing a node with slightly lower, but stable QoS metrics.

4.1.3 Interests

The concept of properties alone is not sufficient to create overlay networks. Properties describe the assets a node has, but they do not specify what nodes belong together. This is where *Interests* become important.

Essentially, each Flock node uses the properties of other Flock nodes to determine how “interested” it is in them. The idea is that interested nodes “flock” together, whereas uninterested nodes drop their connection to each other, and possibly connect to other nodes that are more "interesting" to them. This behaviour gave rise to the overlay's name, "Flocks".

A Flock node computes its interest in another node locally, using an *oracle* and the property set described earlier. The oracle may be any function that fulfils requirements 4.1 and 4.2, defined to be as follows:

Requirement 4.1 *Given (1) the property set of a target node that should be evaluated and (2) a boolean "connected" that is "true" if the target node is connected, and "$false$" if a connection to the node is only considered, the oracle must return a numeric value greater then or equal to zero that represents how high its interest is in obtaining (in case "connected" is "$false$") or maintaining (in case "connected" is "true") a connection to that node.*

Each Flock node only maintains connections to the n neighbours it has the highest interest in. n can be any number greater than zero[4], and it can differ from node to node. A node should choose n based on how many neighbours it can service. For example, a hand-held device may only be able to maintain a connection to one or two neighbours, whereas a dedicated node may handle dozens of other nodes. The result of the oracle therefore determines directly which nodes are kept as neighbours.

The exact numeric value returned by the oracle is not important, however a higher number indicates a higher interest. As a result, a node that is more interested in maintaining a connection to neighbour node A than to neighbour node B should return a higher numeric value for node A than for node B. If the oracle returns the same numeric value for two different nodes, a tiebreaker must be used to establish an order. A Flock node uses the following two parameters in this case:

- Duration of the connection: The neighbour to which a connection has been maintained longer is assumed to be of higher interest.
- Unique identifier: If two different neighbours have been connected for the exact same time, their unique identifier is used as a tiebreaker.

[4]A value of one makes sense for leaf nodes that are only interested in a single connection. For example, a handheld device may have insufficient resources to relay content between nodes, or to maintain connections to multiple nodes.

Requirement 4.2 *An interest of zero is returned if and only if the connection to the evaluated node must be dropped.*

An interest value of zero is considered a special case, as it indicates a complete lack of interest: A Flock node will not maintain a connection to a node with zero interest, even if it has less than n neighbours. Remember, however, that a non-zero interest value does not guarantee that the connection will be maintained. A node A will still stop the connection to its neighbour B, in which A has non-zero interest, if (i) node A has at more than n neighbours and (ii) its interest in neighbour B is lower than to the other n nodes. In addition, even if A has a high interest in neighbour B, the connection may still be dropped by neighbour B, based on its interest in A.

Keeping interests and properties separated provides a great deal of generality, and is important to support situations in which nodes have different interests and properties. These situations are very popular in large social events: In the previous IronMan example, a publish/subscribe system may be implemented. That way, visitors (which function as the *data sinks*) can "subscribe" to a camera (the *data source*) and receive its video stream. In this simple use case, properties and interests for source and sink are exactly reversed: The sink is interested in the video content and provides Quality of Service (QoS) and network related properties; whereas the source is interested in these QoS and network related properties (because it strives to speed up distribution) and provides the video content.

4.1.4 The Oracle provided by the Flocks Algorithm

Flocks provide a default oracle for applications to use, which returns a numeric value between 0.0 and 1.0 (the final interest value I). However, an application can implement its own oracle as long as it follows the requirements described earlier.

Interest Quadruples

The default oracle provided by Flocks assume that interests are provided to it as a flat list of quadruples. Each of the n quadruples is examined to evaluate how closely the properties of the target node match it. This evaluation results in a *match value* $i_1, i_2, \ldots, i_n$ between 0.0 (no match) and 1.0 (fully matched). The numeric match values of each interest quadruple, and their weights are combined to compute the final interest value I, which the oracle returns. A quadruple contains the following elements:

- Property Name (p): The name of the property that is evaluated through this interest quadruple.
- Value (v): The most desired value of the property that should result in a match of 1.0.
- Threshold (t): Specifies a cut-off threshold. If the match value would fall below this threshold, it is set to zero instead.
- Weight (w): A number greater than 0.0 and less than or equal to 1.0 that represents the importance that this quadruple is matched. A quadruple with a weight of 0.5 affects the final interest half as much as a quadruple with a weight of 1.0.

To compute the match value of a quadruple, the `Match` function is used. `Match` accepts the triple p, v and t, and returns the match value. An application can either use the `Match` function provided by the Flocks overlay, or implement its own. The match function provided by the Flocks overlay is very simple: For numeric values, it returns a higher match value the closer the value of the property is to v. For strings, it tests if both strings are equal (in which case it returns 1.0) or not (in which case it returns 0.0). An advanced application may wish to compute the similarity of the strings, instead of returning 0.0 if both strings do not match.

Finally, the match values are combined to the final interest value I. This can be done in a conjunctive (AND) or a disjunctive (OR) fashion. If only a single interest quadruple exists, it cannot be combined with either of the methods. In this case, the match value becomes the final interest value I and the weight w is ignored.

Conjunctive Method

If interest quadruples depend on each other, they should be combined with the conjunctive method in Equation (4.1). For this purpose, the conjunctive operator, $\wedge$, is redefined as follows:

$$(p_1, v_1, t_1, w_1) \wedge (p_2, v_2, t_2, w_2) \wedge \cdots \wedge (p_n, v_n, t_n, w_n) := \prod_{i=1}^{n} (1 - w_i + w_i \cdot Match(p_i, v_i, t_i)) \tag{4.1}$$

The conjunctive method has several useful properties: First of all, if any interest quadruple with a weight of 1.0 does not match at all, I is set to zero. This is similar to a boolean AND, which also evaluates to zero if any of its components is zero. If this is not desired, the disjunctive method (described in the following section) must be used. Second, each interest quadruple directly affects every other interest quadruple. For example, if the match value of one quadruple with a weight of 1.0 is halved, the final interest value I becomes cut in half as well. In the IronMan example, a visitor may only be interested in swimming activities. In this case, the interests should be set as follows:

$$\begin{aligned} I = (&'Type','stream', 1.0, 1.0) \\ \wedge\ (&'Activity','swimming', 1.0, 1.0) \\ \wedge\ (&'qos.bandwidth.up', MAX, 0.0, 1.0) \end{aligned}$$

With these interests in place, the Flock node on the visitor's device is interested in other nodes that provide a video stream of the swimming activity, with as high bandwidth as possible. As t for the 'Activity' and 'Type' quadruple is set to 1.0, any different value in a neighbour's property causes I to become zero. In this case, the connection to the neighbour is removed. The threshold t of the bandwidth quadruple is set to 0.0, and so its match value returns a value closer to 1.0 the closer to MAX the bottleneck bandwidth of the neighbour node is.

Disjunctive Method

The disjunctive method in Equation (4.2) should be used if the quadruples represent alternatives. It redefines the disjunctive $\vee$ as follows:

Match(A)	Match(B)	Match(A) $\vee$ Match(B)
1.0	1.0	1.0
0.0	1.0	0.5
0.5	0.5	0.5
0.0	0.0	0.0

Table 4.1: Results of combining different match values using the disjunctive method

$$(p_1, v_1, t_1, w_1) \vee (p_2, v_2, t_2, w_2) \vee \cdots \vee (p_n, v_n, t_n, w_n) := \frac{\sum_{i=1}^{n} w_i \cdot Match(p_i, v_i, t_i)}{\sum_{i=1}^{n} w_i} \tag{4.2}$$

With the disjunctive method, the quadruples do not depend on each other: The disjunctive combination of two quadruples does not evaluate to zero if the match value of any quadruple is zero. In essence, the disjunctive method functions as an OR operator, in that it returns a non-zero result if any of the match values are non-zero. However, unlike a boolean OR operator, it returns a higher final interest if both quadruples match. This is a desired property, as it prefers neighbours of which multiple properties match. For example, consider two quadruples (p_a, v_a, t_a, w_a) and (p_b, v_b, t_b, w_b). For readability, the quadruple (p_a, v_a, t_a, w_a) is referred to as Match(A), and the weight w_a is set to 1.0. Match(B) refers to the quadruple (p_b, v_b, t_b, w_b), the weight w_b is set to 1.0 as well. Using Equation (4.2), the final interest values are obtained for different match values are shown in Table 4.1.

Choosing the Appropriate Method

While it is clear that both methods return different interest values, it is important to note that unless all weights are set to 1.0, the *relative order* of the interest values may also be different, and therefore affect which neighbours are kept, and which neighbours are dropped. Specifically, if $(p_a, v_a, t_a, w_a) \wedge (p_b, v_b, t_b, w_b) > (p_a, v_a, t_a, w_a) \wedge (p_c, v_c, t_c, w_c)$, it does not follow that $(p_a, v_a, t_a, w_a) \vee (p_b, v_b, t_b, w_b) > (p_a, v_a, t_a, w_a) \vee (p_c, v_c, t_c, w_c)$. As a result, it is important to carefully consider what method yields the desired results for any particular purpose.

Generally speaking, if the conjunctive method is used, the effect of a single match value increases, the higher its weight, and the lower its match value is. Figure 4.2

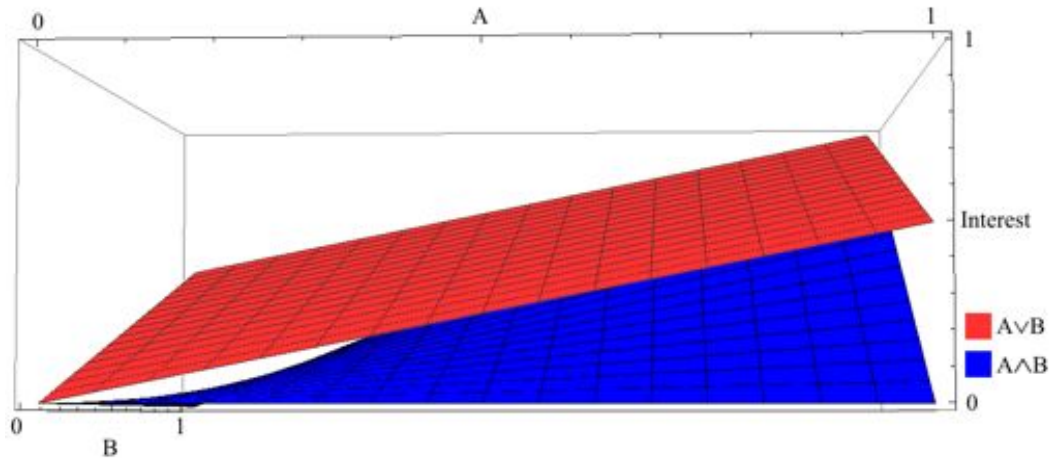


Figure 4.2: Interest values from using different methods to combine matching values

compares the conjunctive and disjunctive methods for varying match values, if both weights are set to 1.0. The interest values that result from combining the match values A and B with the conjunctive and the disjunctive method. The weights of both match values are set to 1.0. The graph shows that when the conjunctive method is used, the effect that a match value has on the final interest value increases potentially the lower the match value is, whereas effects of a match value remain linear if the disjunctive method is used. The difference between both methods is greatest if the weights are both 1.0. As the weights decrease, the interest values of the conjunctive method become closer to the ones of the disjunctive method.

As suggested by the graph, the order of the results between both methods is preserved if all weights are set to 1.0 and none of the match values are zero. This is confirmed by Equation (4.3). In this case, it does not matter what method is chosen.

$$\begin{aligned} \frac{a+b}{2} > \frac{a+c}{2} &\rightleftharpoons (1-a)\cdot(1-b) > (1-a)\cdot(1-c) \\ a+b > a+c &\rightleftharpoons 1-a+b\cdot(a-1) > 1-a+c\cdot(a-1) \\ b > c &\rightleftharpoons b > c, a \neq 1 \end{aligned} \tag{4.3}$$

If at least one match value is 0.0, however, both methods are no longer equivalent. A practical way of deciding which method to use is considering, whether the final interest value should be 0.0 if even only one match value is 0.0. In case this is desired, the conjunctive method is appropriate. In case this behaviour is undesirable, the disjunctive method should be used.

Empty Interest Lists

If the oracle is given an empty list, it will return 1.0 if it can either accept additional neighbours, or *connected* is *true* (that is, it is re-evaluating an existing neighbour). If neither condition holds, it will return a lower, non-zero, value. This results in the neighbour to be dropped, as it is "less interesting" than the n existing neighbours (in which the node has an interest of 1.0).

A Description Language for Interests

Interest quadruples are difficult to read, and do not reflect the process within the Flock node well. For example, only the first three values of the quadruple are passed to the `Match` function, which is treated as a blackbox and returns a normalized match value - a floating point number between 0.0 (no match) and 1.0 (perfect match). The match value and the weight (the fourth value), are then used by the conjunctive and disjunctive methods to compute the interest value.

Therefore, a language for specifying interests has been developed. This language is used both in this thesis, as well as by the default Flocks oracle. Its grammar is shown in Algorithm 4.1.

Combination of Conjunctive and Disjunctive Methods

The grammar shown in Algorithm 4.1 does not permit combining the conjunctive and the disjunctive methods directly. Instead, the group of match values combined with one method must be assigned a weight, before it can be combined with a different method. This is intentional: Consider that each method requires a list of tuples as input, namely (1) a normalized match value between 0.0 and 1.0, (2) and the weight w. However, the output of one method consists of only a normalized interest value. While the normalized interest values from each method are compatible with (1), the weight (2) is lost.

One could assume that it might be possible to compute the weight of a function

Algorithm 4.1 The grammar used by the default Flocks oracle to parse a list of interests, in Extended Backus-Naur form (EBNF).

```
1: InterestList = "[" WeightedGroup GroupExpression "]" | SingleInterest
2: WeightedGroup = "[" Group GroupExpression "," Weight "]" | WeightedInterest
3: GroupExpression = "∧" ConjunctiveGroup | "∨" DisjunctiveGroup
4: ConjunctiveGroup = WeightedGroup | ConjunctiveGroup "∧" WeightedGroup
5: DisjunctiveGroup = WeightedGroup | DisjunctiveGroup "∨" WeightedGroup
6: SingleInterest = "(" MatchFunction MatchParameters ")"
7: WeightedInterest = "(" MatchFunction MatchParameters "," Weight ")"
8: MatchFunction = "Match"
9: MatchParameters = "(" PropertyName "," DesiredValue "," Threshold ")"
10: DesiredValue = StringOrNumber
11: PropertyName = string
12: StringOrNumber = string | Number
13: Threshold = Number
14: Weight = Number
15: Number = float | integer
```

based on its input weights. However, this is not straight forward. Consider the example shown in Equation (4.4):

$$\big[\,[(Match(p_1, v_1, t_1), 0.5) \vee (Match(p_2, v_2, t_2), 0.7)] \wedge (Match(p_3, v_3, t_3), 0.8)\big] \tag{4.4}$$

The tuples in the bracket are combined to a single value using the disjunctive method first. As shown in Equation (4.5), the tuples receive the weights 0.5 and 0.7, and the final interest value computed by this function is a normalized value between 0.0 and 1.0:

$$\begin{aligned} &(Match(p_1, v_1, t_1), 0.5) \vee (Match(p_2, v_2, t_2), 0.7) \\ &= \frac{Match(p_1, v_1, t_1) \cdot 0.5 + Match(p_2, v_2, t_2) \cdot 0.7}{1.2} \end{aligned} \tag{4.5}$$

Even if the individual weights of the previous results are known, it is unclear what the combined weight of this result, which is needed to apply the conjunctive method to this result and $(Match(p_3, v_3, t_3), 0.8)$, should be. Therefore, Flocks require that the weight of each method result is made explicit by adding them to the brackets. This is shown in Equation (4.7), and enforced by the grammar listed in Algorithm 4.1.

$$\begin{aligned}\Big[\,[(Match(p_1, v_1, t_1), 0.5) \vee (Match(p_2, v_2, t_2), 0.7), \mathbf{1.0}]& \\ \wedge\, (Match(p_3, v_3, t_3), 0.8)\Big]&\end{aligned} \tag{4.6}$$

Interest Examples

This grammar permits specifying interests in a flexible and concise manner. Consider again the IronMan example from Section 4.1.2: A visitor, who is interested in video streams from cameras recording the athlete with the starting number 999, specifies this interest as shown in Equation (4.7).

$$\Big[\,\underbrace{\big(Match(''type'', ''stream'', 0.0), 1.0\big)}_{(a)} \vee \underbrace{\big(Match(''athletes'', ''999'', 0.0), 1.0\big)}_{(b)}\,\Big] \tag{4.7}$$

Equation 4.7 (a) requires that the property “type” contains the value “stream”. (b) requires that the property “athletes” contains the value “999”. Both interests are joined with the disjunctive method: This is important for keeping the Flocks overlay connected. A node that uses the conjunctive method instead will disconnect from any node that does not offer a stream, or that does not have any information of athlete 999. Since only a small number of cameras will record a given section of the track at a time, only few nodes - if any - will have information about athlete 999 at any given time. Therefore, it is advantageous for a node to remain connected to many similar-minded neighbours, even if they have no stream, or no information about athlete 999.

Remember that the disjunctive method functions differently to a boolean “OR” operation. Specifically, a node that has both, a stream, and information about athlete 999, will rank higher in interest than a node that only has one of these properties. This is a desired behaviour, as it ensures that nodes which match well become neighbours, while still keeping other, less-desired nodes connected, in case no better neighbours are available.

In order to achieve a high LoS, a node must consider QoS. For illustration purposes, assume that a node desires neighbours with a high bottleneck bandwidth to it. Therefore, we extend the interest specification in Equation (4.7) with an interest in the virtual property “qos.bandwidth.up”, as shown in Equation (4.8).

$$\begin{aligned}&\big[\big(Match(''type'',''stream'',0.0),1.0\big)\\&\quad\vee\big(Match(''athletes'',''999'',0.0),1.0\big)\\&\quad\vee\big(Match(''qos.bandwidth.up'',MAX,0.0),0.5\big)\big]\end{aligned}\tag{4.8}$$

MAX is a constant with a high value (such as INT_MAX). The exact value does not matter, as Match returns a value closer to 1.0 the higher the bandwidth is. The weight of the QoS interest is set to 0.5 to ensure that “qos.bandwidth.up” does not compete with the other properties, such as “athletes” or “type”.

Recall that a property “will_relay” was introduced in Section 4.1.2 to identify nodes that are willing to relay video streams. An interest for this property should be incorporated in a similar fashion, as shown in Equation (4.9). While neighbours that do not relay video streams are not directly useful, they still help in finding nodes that do relay.

$$\begin{aligned}&\big[\big(Match(''type'',''stream'',0.0),1.0\big)\\&\quad\vee\big(Match(''athletes'',''999'',0.0),1.0\big)\\&\quad\vee\big(Match(''qos.bandwidth.up'',MAX,0.0),0.5\big)\\&\quad\vee\big(Match(''will\_relay'',1,0.0),0.5\big)\big]\end{aligned}\tag{4.9}$$

Limitations of the Oracle

While the grammar used by the oracle can express many different situations, it has limitations. Most importantly, it does not support a straight forward way to model conditional interests, as the one shown in Algorithm 4.2. Here, a node has a higher interest in neighbours of class B if it has no neighbour of that class yet.

Algorithm 4.2 An interest that can not be expressed with the grammar used by the default oracle.

```
1: procedure CalculateInterest(candidate_properties)
2:     if $candidate\_properties_{class} = B \wedge neighbours\_with\_class\_B = 0$ then
3:         $interest \leftarrow 1.0$
4:     else
5:         $interest \leftarrow 0.5$
6:     end if
7:     return $interest$
8: end procedure
```

Limitations such as these can be bypassed however, by an application implementing its own oracle based on the requirements laid out in Section 4.1.3.

4.1.5 The Impact of Quality of Service Awareness

QoS-awareness is a pivotal feature of the Flocks overlay. Chapter 2 demonstrates that a large number of robust, flexible and scalable overlays exist, however none of them are QoS-aware. It is reasonable to ask, how much of impact has tacking QoS-awareness on top of an overlay network actually.

In fact, QoS-awareness imposes a number of restrictions on the overlay, particularly if the routing in the underlying network can not be predicted in advance - which is the case, for example, in today's Internet. Specifically, it is not possible to predict how well two nodes match, unless both nodes connect. This poses a major issue for common match-making approaches: A match-maker may find that two neighbours that communicate frequently have a high LoS to each other. As a result, the logical action performed by the match-maker is connecting the two neighbours. To keep the number of links in the overlay constant, match-makers then typically remove a link between themselves and one of the two connected nodes. However, as the newly connected two nodes now communicate directly with each other, the underlying routing protocol may choose a different route, which can result in a different, and possibly worse, LoS.

This example shows that considering QoS is not trivial. The requirement to be QoS-aware has a large effect on how Flocks are implemented. It dictates that Flock nodes must connect to each other in order to determine their true match value, as the routing in the underlying network has to be treated like a black-box. Therefore, a Flock node uses a single transient connection to a neighbour, to exchange properties and determine the LoS. It is important to note that this transient connection may make a Flock node exceed their maximum number of neighbours. For example, a Flock node that has reached its maximum number of five neighbours may still initiate a connection to another candidate neighbour, to evaluate its LoS and its interest to it. However, a Flock node will remove any excess neighbours before initiating a new connection. In this example, the node with the 6^{th} highest interest will be removed.

It is clear that Flock nodes follow a fundamentally different path to the match-making approach, as they essentially match themselves with their neighbours. This gives rise to another issue: In the match-making approach, the match-maker "bridges" two nodes, and makes them direct neighbours. As a result, both nodes gain access to a new match-maker, namely each other. In this way, nodes "move" through the overlay and improvement is achieved.

This is not possible in the Flocks overlay. Without a match-maker, Flocks must pursue a different way of obtaining knowledge of new neighbours. Fortunately, the main idea to achieve this is straight-forward: A node that informs one of its neighbours of all the other neighbours it has, and have them act as their own match-makers. While doing so increases the amount of information that must be exchanged, it enables use cases - such as QoS-awareness - which require that nodes connect to each other. This leads to the introduction of property propagation.

4.1.6 Property Propagation

In order to aid nodes in finding new neighbours, each node sends the properties of every neighbour it has, to all its other neighbours. When propagating a property, each node keeps track of (1) what node a property originally came from, (2) what node it received that property from, and (3) how far that property should still be propagated, expressed as a time-to-live (TTL) value. The TTL value of a property can be used to prevent or limit propagation: the network delay to a node is a property, however as explained earlier, propagating it does not make sense, as any other node would have a different delay to it. In contrast, an important node can be made more visible by setting a higher TTL.

Propagation serves two purposes:

- For one, it allows modelling indirect access to a resource, such as a node that receives a stream directly from a data source and relays it further to its child nodes. As the property gets propagated, newcomers are not only interested in the data source, but also in the nodes that are relaying data from it.
- Additionally, property propagation is the primary means of how a Flock node finds new neighbours. This is because a node, observing a property originating from a node it has not seen yet, will insert that node, and any of the properties it observed into its list of *candidate neighbours*.

4.1.7 Candidate Neighbours

Each Flock node keeps a *candidate list*, which contains a list of nodes that it knows, but that are not currently its neighbours. They are potential neighbours that a node will consider in future, and that also serve as potential backup nodes, should one of the node's current neighbours fail.

Periodically, it picks a candidate node from the list and connects to it, exchanges properties with, and finally evaluates its interest in it. Recall that a node only keeps up to n neighbours. If it currently has less than n neighbours, the candidate node is accepted unconditionally[5]. Otherwise, the node with the lowest interest will be disconnected, which may either be one of its existing neighbours, or the candidate node. Any node that is disconnected is moved to the candidate list.

Several aspects of the candidate list affect the performance of the Flocks overlay. Most importantly, the *size of the candidate list* must be limited. This raises several questions as to the optimal size of the candidate list: A smaller candidate list is preferred under many circumstances, as it reduces the amount of memory used on

[5]Except for the special case in which the oracle returns an interest of zero.

the device. This is especially important on embedded devices that possess only limited resources. However, if the candidate list is too small, a node will only have limited replacement neighbours available that it can contact under churn and failure. In addition, if a node that finds several equally promising neighbours, but cannot store some of them in its candidate list, they might never be considered.

In addition to the size of the candidate list, the *candidate selection strategy* is also important: A Flock node only considers one new neighbour in a (configurable) timespan. Therefore, the improvement of Flock overlay may be substantially slowed if nodes keep choosing the “wrong” candidates. For example, assume a Flocks overlay in which nodes can have up to two content classes. Furthermore, consider a Flock node, that is interested in neighbours with both content classes. It is clear that its performance will be higher if it picks the most suitable candidate from its candidate list right away (namely one with both content classes), than if it continuously picks neighbours that only have a single content class.

The choice of what strategy to use is in essence a scheduling problem, for which several approaches exist. In this thesis, two promising approaches have been identified: A priority based approach, in which candidates that have never been considered before are given priority over candidates that have, and a hybrid approach based on Earliest Deadline First (EDF), in which candidates are assigned an order depending on how “promising” they appear to be. They have been compared against an approach that chooses nodes in a random fashion. Selecting the candidates in order (First In First Out, FIFO) is another popular scheduling strategy, however as the order a node receives its neighbours depends on factors such as network and processing delay, and not on a node’s interest, I have not considered it further. The results in Chapter 5.2.1 show that the hybrid approach outperforms both EDF and the random approach significantly.

4.2 Node States

The life cycle of a Flock node consists of three states: In the *Bootstrapping* state, a node obtains a single neighbour and connects to it. Afterwards, it switches to the *Participation* state. At this point it fully participates in the Flock. Finally, upon leaving the Flock it enters the *Termination* state. This section explains the actions performed by a Flock node during each of these states.

4.2.1 Bootstrapping

To participate in a Flock, a node must first join it. This process is referred to as "Bootstrapping". The process of "joining" a Flock is an implicit one, as no formal registration is required. A Flock is formed as an emerging process, in which numerous nodes connect together.

A new node finds other nodes by connecting to any node (in the following named node $\mathcal{I}$) that has already joined the Flock, and exchanging properties with it. The propagated properties it obtains from its neighbour $\mathcal{I}$ contain the identifiers of nodes they originated from. As a result, the new node can use these identifiers (and the propagated properties themselves) to find other nodes of interest. This process is repeated as resources permit.

The question remains how a new node finds this initial node $\mathcal{I}$ to begin with. Several interesting solutions exist in literature, and they are mentioned here for reference. [112] leverages Domain Name System (DNS) servers, making a named entry (which could be "flock.uni-klu.ac.at") resolve to the address of random nodes already in the network. Another approach is to broadcast the join request, similar to how the Dynamic Host Configuration Protocol (DHCP) [113] protocol works. However, as broadcasts are generally not forwarded across network boundaries, this approach can only be used in Local Area Networks (LANs).Furthermore, unlike the solution employed by [112], it does not scale to a large number of nodes.

Regardless of the solutions proposed in literature, for the evaluations performed in this thesis, I have implemented a centralized *Registration Service* for simplicity and easy of deployment. It is important to note that the Flock Overlay does not require this central component, and any of the solutions above can be used instead. The registration service consists of a lightweight process that runs on a dedicated server known to all nodes, which can be queried to return the unique identifiers of a predefined number of nodes currently in the Flocks overlay. This is similar to BitTorrent, which uses a tracker that all nodes register with and that can be queried for a list of random nodes [45].

Another benefit of using a centralized registration service for my evaluations was its use as a centralized point to collect experimental results. This was achieved using a simple Remote Procedure Call on the registration server, which accepted measurement results and log messages from any node, and saved them for future analysis.

4.2.2 Participation

Once a node successfully connected to its first neighbour and exchanged properties, it enters its maintenance state. During this state, it accepts incoming connections from other nodes. Furthermore, it periodically performs two processes:

- Consider New Neighbours: The node retrieves a candidate node from its passive list, and connects to it to determine its interest.
- Evaluate Existing Neighbours: The node evaluates its interest in its current neighbours, and then disconnects any neighbour that is not among the n neighbours it has the highest interest in. Before disconnecting a neighbour, it informs the neighbour of the reason ("lack of interest"). This causes that neighbour to add the node to its passive list with a low priority, in order to prevent connecting to it again immediately.

Note that both processes do not need to occur at the same time, nor sequentially, nor do they need to use the same period. For example, during my experiments with the Flocks overlay, I evaluated existing neighbours more frequently than considering new neighbours, in order to respond to unrequested incoming connections.

Whenever two nodes connect, the node that initiated the connection sends its properties first. The other node then responds with its own properties.

Maintenance

If a node is informed that one of its neighbours left, it performs the following actions:

- It moves the neighbour from its active list to its passive list.

- It ceases to propagate any of that neighbour's properties.
- It announces that it no longer has access to that neighbour's properties. This is done by simply announcing its properties without the leaving neighbour's properties.

A node may also leave silently. For example, a node may not be able to inform its neighbours if its operating system stops responding. Every node probes its neighbours regularly (with a keep-alive message) to detect this situation. A neighbour not replying to a keep-alive message for a reasonable time frame will be considered to have failed. In this case, the node closes the connection, and performs the steps described above. This also occurs if a disconnection message has been received from the underlying protocol (such as the a TCP RESET message).

If a node cannot connect to the candidate node it selected from its passive list, it inserts that node in its passive list again, with an increasingly longer deadline before a reconnection is attempted. This ensures that inactive nodes[6] are eventually purged from the passive lists of every node in the overlay.

4.2.3 Termination

A node that leaves the Flock permanently should inform its neighbours of its intend, and only then remove the connections to all its neighbours. This is not strictly required. However, by doing so, the neighbours of a node can be certain that the connections were deliberate, and not the result of failure. Therefore, they can perform the same actions as described above, with the exception that they do not move the leaving node to their passive lists, but delete them from their active lists directly.

4.3 Design of a Flock

During the design phase, four dimensions were identified that Flocks focus to offer an improvement in:

[6]Inactive nodes are nodes that are not reachable for a long time, or that left the Flock permanently.

- QoS awareness
- Scalability
- Robustness
- Flexibility

As explained in Chapter 3, some of these dimensions are at odds with each other. This can be observed with many existing overlays: Gnutella, for example, employs flooding to locate content in its overlay. A Gnutella client parses and responds to complex queries, giving its users a high degree of flexibility for locating the content they searched for. However, its performance, both in overlay nodes traversed to locate content, as well as the amount of content recalled, is much lower than using comparatively inflexible "exact-match" queries in structured overlays, such as Chord [15]. By reducing flexibility, some optimizations can be made to improve Gnutella's performance [114].

It is clear that some compromises must be made. The goal of this thesis is to introduce an overlay that offers an improvement to these dimensions, with acceptable compromises, so that the end result is a feasible solution for demanding tasks such as multimedia delivery.

This section will explain how the Flocks overlay achieves this improvement. Implementation details of the Flocks overlay network are discussed where they become relevant.

4.3.1 Performance Considerations

Recall that in a match-making approach, nodes only need to exchange information with their direct neighbours. A Flock node, on the other hand, must learn of the existence of nodes other than its current neighbours through property propagation. As properties are not only sent to a node's direct neighbours, but are then propagated one hop further, the amount of traffic that nodes exchange increases. However, as a result Flock nodes can now connect to their new candidate neighbours directly. This has several important effects:

- A Flock node can measure the LoS to the candidate neighbour before deciding what connection to drop, providing support for *QoS-awareness*.
- Additionally, it can evaluate its interest locally, providing greater *flexibility*. Match-making assumes a common interest - if two nodes match, which is decided based on their "properties", they are connected. However, in many situations, the properties of interested nodes do not necessarily have to match. Imagine, for example, a multimedia streaming scenario, in which multimedia data is sent from the source to a sink. Pretend further that source and sink are exact opposites - specifically, it is reasonable to assume that a source node is not interested in sending data to another source node. This scenario is difficult to implement without additional "knowledge" in the match-maker (such as "match source with sink"), but very easy to implement with Flocks.

An important concern is whether property propagation, essentially a form of "localized flooding", scales with large numbers of nodes. This is a reasonable concern: Gnutella employs a similar technique, in which a query is forwarded for a specific number of hops. It is generally understood that the performance of Gnutella, both in traffic produced, and nodes located, is far from the optimum, and numerous improvements have been proposed. Therefore, it is a valid concern that the Flocks overlay may also suffer from similar issues, even though properties, by default, are propagated much less far than Gnutella queries.

Generally, gossip networks have demonstrated to exhibit good performance and fast stabilization. For example, it has been shown that through only local (direct-neighbour) communication, millions of nodes can achieve a global goal (such as organizing a network in a cube, or a torus) quickly [68].

Flocks function very similar to this class of gossip network. However, since Flocks allow properties to be propagated for more than one hop, they are able to achieve faster stabilization at the cost of additional traffic[7]. Another innovative advantage of the Flocks overlay is that the compromise between faster stabilization and additional traffic is configurable. If the former is more important than the latter, the visibility

[7]For example, consider two "matching" nodes A and B that need to become neighbours, but are three hops apart. In a Flocks overlay that propagates properties for three hops, A is able to see and connect to B immediately, instead of having to connect to a node closer to B first.

of nodes should be increased by propagating property for a greater distance. This can also be adjusted dynamically, if, for example, nodes determine that sufficient bandwidth is available.

In any case, it is important to closely evaluate performance related metrics. These include the traffic overhead produced by the Flocks overlay, as well as how quickly the overlay improves.

4.3.2 Scalability Considerations

The Flocks overlay can be expected to scale well to a large number of nodes due to its lack of centralized component, or global view. Instead, every Flock node only uses a small, local view that contains its direct neighbours, and a fixed size "passive list" of candidate nodes to consider. The storage and computation cost of the local view, the active and the passive list are constant, and independent of the overlay size. In addition, the interests and properties a node stores, as well as the computation of its interest in other neighbours is also independent of the number of nodes in the overlay.

In fact, only two features of the Flocks overlay have a non-constant cost: QoS-estimation and QoS-aware routing. They require a map of the overlay to perform their calculations. However, Chapter 6.2 will present a feasible multi-level topology aggregation solution that scales logarithmically with the number of nodes in the overlay.

Decentralization

A node does not need any central component to find its position in the network, or to find new neighbours. In order to join the Flock, it only needs to connect to any node that is already connected, as described in Section 4.2.1. Finding new neighbours is achieved by the nodes in the Flock themselves. This is done by looking at the propagated properties of any of its neighbouring nodes, and considering the nodes that they originated from as new neighbours. It can easily be shown that a node can find every other node simply through property propagation, provided that (i) the time to live value of a common property is greater than zero, and (ii) the network is not partitioned. The likeliness and implications of the network being partitioned is discussed in the next section.

Incoming Connections

Incoming connections may also cause a node to exceed its maximum number of neighbours. In my simulations, this has not posed an issue, however it is easy to see that popular nodes may receive a lot of incoming connections, which could present an issue. If enough other nodes attempt to connect, they may inadvertently cause similar effects as a Distributed Denial of Service attack (DDoS) on the popular node.

In Section 5.2.1, I will introduce neighbour selection strategies, which use exponential backoff if a node is unavailable or busy, which provides protection against flooding nodes with connection attempts. In this situation, an overloaded node can either (i) not accept the connection at all (ii) close the connection immediately after `accept()` returns, or (iii) inform the connecting node that they have no interest at all, and then close the connection. Any of these actions will cause the connecting node to return the overloaded node to its candidate list with an exponentially longer delay before a new connection is attempted.

4.3.3 Robustness Considerations

Flocks obtain their robustness in part from the underlying mesh structure. Ideally, a node has n active connections and thus, by definition, $n - 1$ alternatives in case any of these nodes fail. Each node also keeps a list of candidate neighbours, which contains nodes that it has not considered yet, as well as nodes that it has not chosen as neighbours due to a lack of interest. If its neighbours fail, it may resort to that list to fill up its connections again. Furthermore, property propagation ensures that all of a failing nodes' neighbours are still known to the nodes surrounding it. This means that increasing the TTL of properties in a Flocks overlay also increases its robustness, at the cost of more traffic through nodes exchanging them.

Partitioning is an issue because once the network is separated, there is no guarantee that nodes still holding references to the other partition will actually connect to them. For example, all nodes affected by the split may actually be more interested in nodes located in their own partition and connect solely to them. Once two partitions are completely split that way, nodes in one partition have no way of finding any other node in the second partition without resorting to the initial bootstrapping mechanism.

A Flock partitioned in two or more Flocks may actually be useful if the interests in each partition are different enough. This raises a new issue: In order to find any nodes of interest a new node joining the network needs to be assigned to the Flock that matches its interest. Continuously resorting to the initial bootstrapping mechanism to find "the correct" Flock is a poor choice: The servers used for bootstrapping would suffer a substantial overhead from maintaining an updated list of nodes in the Flock, as well as undue load from any node in the Flock querying them. They are also not aware of any interests and properties, so that finding the correct Flock would be a matter of chance.

A possible solution is to make each node participating in the Flocks overlay network join an additional, inverse Flock where it keeps neighbours that it has the least interest in. Following this approach increases the robustness of a Flock to a great extend, as sparsely connected nodes in one Flock are strongly connected in the other. This also makes it very unlikely that a Flock will partition. Finally, it allows a newly joined node to find suitable neighbours faster, as it can not only consider the properties and neighbours of the node it initially connects to, but also the neighbours that offer the most contrasting properties.

4.3.4 Flexibility Considerations

Flocks maintain a high level of abstraction and flexibility by separate interests and properties. By adjusting them, a large variety of different effects is achieved.

A default oracle is provided in Section 4.1.4, which calculates a node's interest in a neighbour. This default oracle is sufficient for many common situations, such as Publish-Subscribe overlays, or a peer-to-peer assisted streaming overlay similar to BitTorrent and BASS. However, the application developer can also override the oracle and compute the interest themselves. This achieves the greatest amount of generality by only slightly increasing the implementation effort needed.

The following will explain the interests and properties that should be used to model several common situations. In addition, it will demonstrate the flexibility by showing how each of these models can be extended to provide additional value - such as QoS-awareness.

4.4 Distribution Models

Many use cases can be supported using only the default oracle provided by the Flocks overlay. This section highlights three popular distribution models, and models them using the grammar introduced in Section 4.1.4. It shows that the default oracle alone is already flexible enough to express a set greater than covered by these traditional distribution models.

4.4.1 Client / Server

In the classic client / server scenario, one set of nodes ("servers") provide services to a non-overlapping second set of nodes ("clients"). Assume that no relay is used, so clients may only obtain the service from a server, and not from another client that acts as an intermediate node.

This scenario is modelled by setting a specific property on the "server" nodes, that provide the desired service. A TTL of two hops on that property is sufficient, since "client" nodes will connect to the server directly. Therefore, as soon as any client connects to the server, it will propagate the server's property to its direct neighbours, which in turn connect to the server themselves (and then propagate its property further).

It should be noted that the key difference between client and server is that the server is not interested in the file property, but it provides that property. Clients are interested in the file property, but they do not have that property themselves.

Properties

Assume that a server has the unique identifier 192.168.1.1, and that it provides a file with a unique name "chronicles.mp4". Therefore, its property sextuple is set as follows:

Name	Value	TTL	Confidence	Origin	Propagated From
`file`	`chronicles.mp4`	2	1.00	192.168.1.1	*empty*

Table 4.2: Property announced by the server

Client nodes that are directly connected to the server will propagate this property further, and send it to their direct neighbours after decreasing the TTL field. Each neighbour also populates the "Propagated From" field. As a result, the sextuple recorded by each of its direct neighbours is:

Name	Value	TTL	Confidence	Origin	Propagated From
`file`	`chronicles.mp4`	1	1.00	192.168.1.1	192.168.1.1

Table 4.3: Property further propagated by the clients

The neighbours this property is propagated to now have the identifier of the server, and can connect to it directly. They will not propagate this property any further, because they receive it was a TTL of 1. However, once they successfully connect to the server, they receive the property from the server directly, with a TTL of 2, and propagate that.

Interests

Merely setting properties is insufficient to connect client nodes to server nodes. In order to ensure that clients actively seek and maintain a connection to server nodes, they must be interested in them. This is accomplished by setting interests on clients, as well as server nodes.

The simplest scenario uses a single interest (p, v, t, w) on the client:

$$I = ('\texttt{file}', '\texttt{chronicles.mp4}', 1.0, 0.5)$$

Note that the weight of the interest is set to 0.5. If the weight were set to 1.0, the interest in any node that does not have the property would be zero, causing it to disconnect from any node other than the server. By setting the weight lower than 1.0

(in this particular case, 0.5 has been used), client nodes still connect to each other. This aids bootstrapping: As soon as one client node locates the server, it will inform its neighbours, which, in turn, inform their neighbours upon connecting to the server.

The server has no interest set, and therefore accepts all nodes that it can handle. Nodes that fail to connect to it (because the server is overloaded) will not propagate its property further.

Setting the property and the interest detailed above results in a hub-structure: All client nodes connect to the server node that offers the file they are interested in. No relaying takes place, and the LOS provided to the client nodes is determined by the routing of the underlying network alone.

Relaying Client Nodes

Assume that some of the client nodes are willing to relay the file from the server to other clients. These could be altruistic nodes, or even proxy nodes that are added to the network in order to lower the strain on the server.

To facilitate this, a new property `will_relay` is introduced. A client, C, sets the value of this property to the name of whichever file it is willing to relay to other clients:

Name	Value	TTL	Confidence	Origin	Propagated From
`will_relay`	`chronicles.mp4`	2	1.00	C	*empty*

Table 4.4: The property used by the client to advertise its willingness to relay content.

The interests of client nodes must be changed to consider nodes that are willing to relay. The conjunctive method is used, as it ensures that I will remain non-zero even if both properties do not match.

$$\begin{aligned} I = & \ ('\texttt{file}', '\texttt{chronicles.mp4}', 1.0, 0.5) \\ & \wedge ('\texttt{will\_relay}', '\texttt{chronicles.mp4}', 1.0, 0.25) \end{aligned}$$

Note that the weight of the `will_relay` property is lower than the `file` property. Therefore, a node, given the choice between a server, and a client that relays the file, will choose the server.

The situation that `file` and `will_relay` are both set should not occur, however it is included for completeness sake. Instead, with these interests in place, clients prefer to connect to the server (with an interest of 0.750), if it has resources available. They will also consider clients that relay the file to them (with an interest of 0.500). Clients that do not relay are the least attractive neighbours (with an interest of 0.375). Similar to the previous example, however, maintaining a connection to them results in faster bootstrapping.

This example can be improved further: For example, the server may prefer clients that will relay by setting its interest to:

$$I = ('\texttt{will\_relay}', '\texttt{chronicles.mp4}', 1.0, 0.5)$$

This ensures that resources are spent on clients that relay the file further, thus lowering the load on the server. If the server can handle additional connections, non-relaying clients are still accepted (as I does not drop to zero), however if this is not the case, relaying clients are given preference.

Quality of Service

None of the above configurations considered the LOS clients would be receiving. As a result, the LOS may be very low: Pretend that the server can only handle five neighbours. If all five clients connecting to the server have very little bandwidth, the LOS of all other nodes in the overlay suffers.

Therefore, a different configuration, in which nodes prefer other nodes with high bandwidth, may be considered. Fortunately, this is straight forward to implement: Recall that Flock nodes have the virtual properties *qos.bandwidth.up* and *qos.bandwidth.down*. We assume that a client prefer other nodes can provide the file with a high bandwidth to it (*qos.bandwidth.up*), so that it can download the file as quickly as possible. A server prefers clients that relay, and that can accept and send the file with a high bandwidth (*qos.bandwidth.up* and *qos.bandwidth.down*), so that it can quickly download the file from the server, and upload the file to other clients.

The interests for the client are therefore set to:

$$\begin{aligned} I = \big(&('\texttt{file}','\texttt{chronicles.mp4}', 1.0, 0.5) \\ &\wedge ('\texttt{will\_relay}','\texttt{chronicles.mp4}', 1.0, 0.25), 0.5\big) \\ &\wedge ('\texttt{qos.bandwidth.up}', MAX, 0.0, 0.5) \end{aligned}$$

The server, on the other hand, sets its interests to:

$$\begin{aligned} I = &('\texttt{will\_relay}','\texttt{chronicles.mp4}', 1.0, 0.5) \\ &\wedge \big(('\texttt{qos.bandwidth.up}', MAX, 0.0, 0.5) \\ &\wedge ('\texttt{qos.bandwidth.down}', MAX, 0.0, 0.5), 0.5\big) \end{aligned}$$

With this change in place, the network structure changes completely, and nodes with fast connections end up close to the server, while nodes with slow connections will be distributed at the edges of the network. Note that the resulting network is still a mesh and does not have robustness weaknesses as trees do.

4.4.2 Publish / Subscribe

A publish / subscribe model in Flocks is similar to the client / server model, except that nodes may wish to subscribe to multiple categories. Consider a social event in which some attendees participate in various activities, while others take videos and label them with tags that contain the names of the activity ("Rockband", "Tennis" or "Bowling", which may also be conferred from the GPS position recorded in the video). These labels are stored in a property, "label". A node with the identifier C advertises the properties of all video files it currently provides as shown in Table 4.5.

In the traditional publish / subscribe model, that node would be a publisher. An attendee that attends a tennis match, but is also interested in what happens at the bowling alley, subscribes to that label by setting its interest accordingly:

Name	Value	TTL	Confidence	Origin	Propagated From
`label`	`Rockband`	2	1.00	C	*empty*
`label`	`Tennis`	2	1.00	C	*empty*

Table 4.5: The properties of a node C that publishes content in the publish / subscribe model.

$$I = (\text{'}\texttt{label}\text{'}, \text{'}\texttt{Bowling}\text{'}, 1.0, 0.5)$$

This results in an overlay network in which nodes with similar interests cluster together: Nodes interested in “Tennis” place themselves near nodes that offer said activity. Nodes interested in “Bowling” gather around nodes publishing “Bowling” videos.

Many overlay create a similar network graph for streaming. However, Flocks remain flexible when the requirements or the usage pattern changes. Consider a simple example, in which another group of attendees is attending “Rockband”, but is interested in the outcomes of both the “Tennis” and the “Bowling” activities. They therefore wish to receive both video streams. The nodes of these users can easily express this by changing the interest to match against both values, which includes both events:

$$\begin{aligned} I = & (\text{'}\texttt{label}\text{'}, \text{'}\texttt{Tennis}\text{'}, 1.0, 0.5) \\ & \wedge (\text{'}\texttt{label}\text{'}, \text{'}\texttt{Bowling}\text{'}, 1.0, 0.5) \end{aligned}$$

This causes the structure of the overlay to change: While nodes interested in a single stream still form a group around each respective video source, a third group interested in streams from both other groups is formed.

Assuming that nodes also relay, it can easily be seen that just for the “Tennis” and “Bowling” labels, an overlay network with three clusters is formed, namely (i) nodes interested only in “Tennis”, (ii) nodes interested only in “Bowling” and (iii) nodes interested in both “Tennis” and “Bowling”. It should be noted that this does not actually subscribe the nodes to the publishers, but merely creates a network overlay that places interested nodes close to the event sources. An application developer is free to use any protocol they desire to manage their subscriptions.

Again, specifying only an interest in events does not consider QoS. If the cause of an event is known, that information can be placed in the interest function to create an overlay network befitting this purpose. For example, assume that once an event occurs, a signal is sent out to all interested nodes to indicate the availability of a data stream that needs to be distributed to all subscribers. In this case, it would be beneficial to have high bandwidth nodes close to the event source, so that they can distribute the data stream faster. Similar to the client / server model, QoS can be considered by simply adding an interest in one or more QoS metrics. Assuming clients are interested in receiving the video as quickly as possible, a combined interest of bandwidth and delay may be used:

$$\begin{aligned} I = \Big(('\texttt{label}','\texttt{Tennis}', 1.0, 1.0) \wedge ('\texttt{label}','\texttt{Bowling}', 1.0, 1.0), 0.5\Big) \\ \wedge\Big(('\texttt{qos.bandwidth.up}', MAX, 0.0, 1.0) \wedge \Big(('\texttt{qos.delay.up}', 0.0, 0.0, 0.5), 0.5\Big) \end{aligned}$$

Note that both QoS metrics are weighted differently: It assumes that clients are more interested in obtaining a high bandwidth link, than a low delay.

4.4.3 BitTorrent

BitTorrent clients use a tit-for-tat algorithm to incite sharing: a node will send data to a configurable number of neighbours that upload to them the fastest. One of its neighbours nodes is always sent data (optimistic unchoking), which allows new clients to obtain data they can share. The former aspect can easily be modelled using Flocks if each node C that shares a file F advertises this in a property:

Name	Value	TTL	Confidence	Origin	Propagated From
`file`	`F`	2	1.00	C	*empty*

In addition to that, each node has a property called $UploadRateToMe$ which initially has a high uncertainty factor. The initial value of this property affects the neighbour selection: If every node sets it to the same constant, neighbour selection is random, as with BitTorrent. Nodes may also estimate the upload rate they may provide, which results in nodes preferably connecting to nodes with a high upload bandwidth. Once two nodes connect, each node will update its view of that property with the actual upload rate it receives, and lower the uncertainty factor over time. A node models its interest in $UploadRateToMe$ and $file$ by settings its interest list to:

$$\begin{aligned} I = &('\texttt{file}', '\texttt{F}', 1.0, 0.5) \\ &\wedge ('\texttt{UploadRateToMe}', MAX, 0.0, 0.5) \end{aligned}$$

Optimistic unchoking [45] is indirectly modelled because each node in the Flock looks for better neighbours, using nodes derived from propagated properties. A node that has fully downloaded the file acts as a "seed". Seeds attempt to distribute the file back to the network as fast as possible, and therefore use a different interest:

$$\begin{aligned} I = &('\texttt{file}', '\texttt{F}', 1.0, 0.5) \\ &\wedge ('\texttt{DownloadRateFromMe}', MAX, 0.0, 0.5) \end{aligned}$$

This requires that nodes carry an additional property $DownloadRateFromMe$, set to their maximum theoretical download rate.

4.4.4 Discussion

This section demonstrated the ability of the Flocks overlay to adapt to many vastly different configurations, prompted only by simple changes to their interests and properties. It shows that the concept, as well as the default oracle, are powerful enough to create overlay networks suitable for supporting many common distribution models.

If greater flexibility is needed, the default oracle may be replaced by a custom one, implemented by the application developer. As interests are only evaluated locally, this may occur independently from other Flock nodes. Specifically, as the output from the interest oracle is only used locally, by the same node that calculated it, it is not necessary that nodes in a Flock overlay use the same interests, or even the same oracle.

Suitability for Multimedia Applications

Handling multimedia data imposes high requirements on the overlay network. A Flocks overlay has several important properties that make it suitable for demanding tasks such as multimedia streaming:

First, the overhead of the Flocks protocol is low: It consists of connecting to neighbours, evaluating its interest in each of the neighbours, and deciding whether to maintain a connection to them.

Flock nodes adapt themselves to their current situation: When they first join, and after failures, they search for neighbours aggressively, making large changes to the overlay. Once sufficiently good neighbours are found, they slow down: As a result, fewer changes are made to the overlay, and the protocol overhead is reduced. This allows the upper protocols to use bandwidth for themselves.

In order to evaluate the performance of Flocks, extensive simulations under controlled conditions have been performed. This permits introducing churn and large-scale failures, and observing how well Flocks using different configurations perform, and how quickly they recover. The experiments show that the overhead for the Flocks protocol is low, and the LoS of the overlay improves quickly. Finally, a noteworthy result is after a short time, overlay connections remain sufficiently stable and can quickly be used for longer-lasting activities, such as multimedia streaming.

CHAPTER 5 A First Prototype

A first prototype of the Flocks overlay was developed to investigate a self-organizing, QoS-aware overlay that meets the requirements described in Chapters 3 and 4. This allowed a preliminary performance evaluation, determine the suitability of the approach chosen in this thesis, and identify shortcomings that needed to be addressed in subsequent adaptations.

5.1 Experiment Setup

Practical performance evaluation of the Flocks overlay has been performed through simulation and nodes on the PlanetLab, using a prototype built in C# that implements the concepts described in this thesis. While the use of a network simulator such as NS2 was considered, using C# allowed fast prototyping, and allow me to use a single implementation of the Flocks overlay for my simulations, and for my evaluations under "real" Internet conditions on the PlanetLab.

5.1.1 PlanetLab

The PlanetLab[1] is an overlay of machines that serves as a testbed for network-related research. Participating institutions provide two dedicated machines (so-called "nodes") to the PlanetLab, and may in return create "slices" for their own research projects. A slice permits creating virtual machines on other nodes in the PlanetLab, on which the researched software runs.

Each PlanetLab node is located at one of the participating institutions, and is required to have access to the Internet. Therefore, PlanetLab enables creation of virtual machines on nodes distributed across the globe, which generally experience the

[1]http://www.planet-lab.eu/

same QoS related metrics as other nodes at that institution[2]. For example, a PlanetLab node located in Australia will experience a high delay to an European PlanetLab node. In addition, any service impairment (such as routers between Australia and Europe failing) will affect the connectivity of the aforementioned nodes as well.

Experiments with Flock nodes were run distributed on PlanetLab nodes across the globe, experiencing "real" Internet conditions. This allowed me to demonstrate that Flocks work in practice. I noticed that some nodes were more stable than others: They were continuously available over several months, and selected to run experiments on. However, it is clear that doing so would bias the evaluation towards more stable (and presumably better connected) nodes. As a result, I added less stable and resource-limited PlanetLab nodes to the evaluation as well. However, doing so made it difficult to perform experiments on a large number of PlanetLab nodes over a longer period of time, because a large number would also increase the likelihood of nodes being frequently re-installed or unavailable. Mono [115], an open-source, cross-platform implementation of C# was used to run the Flocks overlay in a cross-platform specific manner.

The Registration Service

A simple "registration service" was implemented to facilitate bootstrapping, which all nodes register with, and that can be queried to return a registered node randomly. The registration service resided on a PlanetLab node that I considered stable and well connected. It doubled as a centralized point to collect experimental results, consisting of the following:

- Network related information from each node: Bytes sent and received, packets sent and received, connections established and lost.
- A list of neighbours from each node, allowing a network map to be drawn.

Results were collected by the registration service every thirty seconds. Each node maintained a connection to the registration service for that purpose.

[2]Institutions may limit the resources available to the PlanetLab nodes. For example, a bandwidth limit may be imposed on any node and any slice by the institutions' "Principal Investigators". In this case, the PlanetLab node experiences worse conditions than other nodes at the institution.

Deployment and Synchronized Startup

Deploying and testing an overlay network on multiple nodes can be a time consuming activity. Every time a new version was compiled, it would need to be copied to every system, verified, executed at roughly the same time, and finally terminated again.

SmartFrog [116] was used to simplify the deployment to the PlanetLab nodes. I made adaptations in order to permit a remote deployment that conforms with the security requirements in the Acceptable Use Policy[3] of the PlanetLab, and in order to support running Mono-based applications. A web-interface was developed to quickly add new PlanetLab nodes to the deployment process, as well as to allow live inspection of the deployed overlay's state.

Commercial Systems

Commercial cloud computing providers, such Amazon EC2, have been considered as an alternative to deploying the Flocks overlay on the PlanetLab. Amazon EC2 has several advantages over PlanetLab:

- Their primary advantages are that the number of nodes they can supply is greater than the number of nodes available on the PlanetLab.
- The nodes are more stable, and they provide a Service Level Agreement[4]. PlanetLab nodes can be reinstalled at any point, and researchers are discouraged from storing the results of their experiments on them[5]. During the experiments, nodes were frequently reinstalled and all data on them was lost. Maintaining an installation on a number of nodes for more than a few days requires substantial efforts.
- Initial experiments showed that configuration of nodes on the PlanetLab differed slightly. For example, some nodes accepted a "sudo" command without a shell, whereas others did not. Nodes provided by Amazon EC2 did not show differences in configuration.

[3]https://www.planet-lab.eu/doc/aup
[4]http://aws.amazon.com/ec2-sla/
[5]http://www.planet-lab.org/doc/guides/user

US East (Virgina)	us-east-1a	us-east-1c	us-east-1d
US West (Oregon)	us-west-2a	us-west-2b	us-west-2c
US West (N.California)	us-west-1a	us-west-1b	
EU West	eu-west-1a	eu-west-1b	eu-west-1c
Asia Pacific (Singap.)	ap-southeast-1a	ap-southeast-1b	
Asia Pacific (Tokyo)	ap-north-east-1a	ap-north-east-1b	
S. America (Sao Paulo)	sa-east-1a	sa-east-1a	

Table 5.1: Sites provided by Amazon EC2 (verified in July 2012).

On the other hand, PlanetLab has the advantage of having more diverse locations. Amazon EC2, in 2012, provides 17 different sites (shown in Table 5.1), whereas PlanetLab provides 537 different sites (verified in July 2012). In addition, maintaining a large number of nodes on Amazon EC2 was not cost-effective, in particular since most experiments were expected to not last longer than three hours, a time span during which the PlanetLab has, through preliminary experiments, been shown to be stable enough. As a result, I did not pursue the use of commercial cloud providers for my evaluations further.

5.1.2 Simulation

Simulations with several thousands of nodes were performed under controlled conditions, in order to evaluate the scalability of the Flocks overlay. A lightweight, custom simulator has been written for that purpose. In addition, the existing code used for the PlanetLab was compiled as a "Class Library" by using the appropriate compiler switch[6].

The simulator loads the class library, and creates the desired number of instances. Each instance runs a separate thread. The simulator network layer provides a network layer to all instances that provides the commonly used socket functions "listen", "connect", "read", "write" and "close", and that is used instead of the socket layer of the underlying operating system. It permits communication between class instances, introduces delay, and measures the traffic exchanged between nodes. The simulation then runs in real-time.

As the simulation layer has a global view, a complete knowledge of the entire

[6]/target:library

overlay, and full access to the traffic sent over its layer, it can accurately measure the overlay quality, structure, as well as the traffic caused by individual nodes, and the overlay as a whole. In addition, it also stored the network graph itself for visual inspections and for evaluation with graph analysis tools [117]. Data samples are collected and saved every second, and analysed after the simulation completes.

As with the experiments run on the PlanetLab, the Flocks overlay uses a registration service for bootstrapping. This service allows new nodes to enter the overlay by providing them with several addresses of other overlay nodes. Flock nodes can also periodically contact the registration service again to find new potential neighbours. However, I disabled this feature in order to evaluate the performance of Flocks as if no centralized component were available, beyond the initial contact with the registration service.

Experiment Phases

Each optimization was evaluated through a series of experiments, which were divided into four phases: Bootstrapping, Failure, Churn and Recovery. Each phase lasted 300 seconds.

The bootstrapping phase begins with a large number of nodes joining the overlay at the same time. The remainder of the phase is used to evaluate how quickly the network converges. At the beginning of the failure phase (which occurs after 300 seconds of the simulation), a large scale failure is simulated by removing 40% of the nodes. This is done by destroying the class instances of the affected nodes. I evaluate the response of the overlay to this failure, and how fast it recovers. During the churn phase (starting after 600 seconds), nodes begin joining and leaving the overlay at the rate of one node per five seconds. In a fourth "Recovery" phase (starting after 900 seconds) the churn stops, and the speed at which the overlay recovers from the churn is evaluated.

Input Parameters

Several input parameters that affect the Flocks overlay can be adjusted through the simulator. They were determined through preliminary experiments and evaluations, and are described here:

- Number of nodes: This simulator parameter determines the number of nodes that initially join the overlay. This number remains constant for the first third of the experiment. In the second third, 40% of the nodes fail. The number of remaining nodes once again remains constant during the churn phase, as we remove as many nodes from the overlay as the number of nodes entering it. Most simulations used 100 nodes, which were performed and evaluated by a single lab computer in real time. Later simulations increased the number of nodes to 200, 300, 400 and 500 nodes, which could no longer be evaluated in real time. Because of that, I changed the simulator to only perform the simulations (which still happened in real time), and to save a network graph every second. After the simulation concluded, the simulator loaded each network graph in order, and performed its evaluations.

- Candidate List obtained through Bootstrapping: The primary goal of the bootstrapping service is allowing a node to join the overlay; however a secondary goal is to include a sufficiently large number of nodes in order to keep the network unpartitioned as long as possible. If each node kept the initial neighbour it is assigned, giving every node other than the the first one initial neighbour would be sufficient to keep the network unpartitioned. However, as Flocks actively search for new neighbours, and drop existing ones, a higher number of initial candidates should be given. The exact number is difficult to predict, as it depends on which interests and properties Flocks use. For the experiments performed in this thesis, I included three nodes in this list, which was sufficient to keep the network unpartitioned.

- Number of neighbours per node: The Flocks overlay assumes that nodes can only maintain a limited number of neighbours. Resources such as bandwidth and CPU are limited, and will become bottlenecks if the number of neighbours is allowed to grow indefinitely. In addition, if nodes in the overlay network are used for streaming or data distribution, their resources will likely need to be shared between all neighbours.

Because of this, it is reasonable to limit the number of neighbours that any one node can have. For simplicity, I used the same limit for all nodes in experiments. In practice, every node may have a different maximum, depending on its purpose and capacity.

Preliminary experiments were performed with different numbers of neighbours: Rings, as shown in Figure 5.1, are formed if only two neighbours are allowed[7], any higher limit of neighbours form a mesh. Allowing additional neighbours increases the resilience of the overlay in large scale failures, but also increases the bandwidth required to maintain the overlay. I found that a limit of five neighbours offers a balance between stability and simplicity for the purposes of this thesis. Increasing the limit to six or higher did not seem to provide a substantial benefit for experiments, but made it harder to evaluate the structure of the overlay visually.

- Candidate List Size and Selection Strategy: As explained in Section 4.1.7, a node can only store the identifiers of a limited number of candidate neighbours - nodes that it considers connecting to in near future. Furthermore, it can only connect to and evaluate a limited number of candidate neighbours as well. Therefore, it is important to consider how many candidate neighbours it can store, and what order they are considered in. I performed preliminary experiments to evaluate the trade-off resulting from different selection strategies, and the performance gained from providing more memory to the candidate list, and thus allowing more identifiers to be stored. The results are presented and discussed later in this chapter.

5.1.3 Performance Measures

A number of metrics that are important and representative for substrate network performance were evaluated: First, I collect the number of messages and bytes sent

[7]Note that two nodes, A and Z, with very different hues, connect at the top left, thus forming a ring. While they have very little interest in each other, they have no better alternatives: All other nodes have two neighbours with a more similar hue than the hues of nodes A and Z.

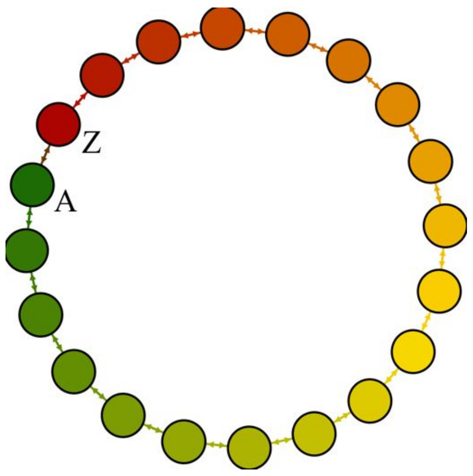


Figure 5.1: A small overlay with 20 nodes interested in neighbours that have a hue/shade similar to its own, forming a ring.

and received for every node to measure the bandwidth required for overlay management. As an indication of how actively the overlay structure changes, I also obtained the number of connections and disconnections since the last sampling. Furthermore, during every sampling I collect and evaluate the network graph itself: this was accomplished with the homogeneity metric used in graph theory [26], and the Jaccard similarity coefficient [118] used in cluster validation. Finally, I computed how close the overlay is to the maximum QoS obtainable by optimal link placement.

Homogeneity

Homogeneity, as shown in (5.1), was introduced by [26]. It uses n as the number of nodes and l as the number of links in the overlay. $v(node_i)$ is the number of nodes of the same type that are linked to $node_i$. Essentially, it measures the purity of each cluster in the overlay as a number between 0 and 1. A cluster consisting of only related nodes has a homogeneity of 1, whereas a cluster with no related nodes has a value of 0. Precision is a similar metric, however homogeneity is more suitable for overlay networks, as it takes the number of links into account, and so penalizes a central node of the wrong class more than precision would.

$$H = \frac{\sum_i^n v(node_i)}{l} \tag{5.1}$$

The Jaccard Similarity Coefficient

While homogeneity is a suitable metric to measure whether each cluster in the overlay only consists of nodes of the correct type, it does not evaluate if all nodes of the correct type are indeed clustered together. For example, assume an overlay with two classes of nodes, A and B. This partitioned into three clusters: two clusters with only nodes of class A, and one cluster with only nodes of class B. The structure of this overlay has a homogeneity of 1, even though it is not optimal: One would expect all nodes of class A form a single cluster, instead of two. This can be detected by comparing two matrices: A cluster similarity matrix (in which the ij^{th} entry is 1 if node i and j belong to the same cluster, and 0 otherwise), and a class similarity matrix (in which entries are 1 if both nodes belong to the same class, and 0 otherwise) [118]. In order to compute the binary similarity between both matrices, the Jaccard coefficient is commonly used in cluster validation [118], which ranges from 0 to 1. It is defined as shown in (5.2), where f_{11} is the number of pairs of nodes belonging to the same class and cluster, f_{10} the number of pairs of nodes belonging to the same class, but are situated in separate clusters, and f_{01} the number of pairs of nodes belonging to different classes, but are situated in the same cluster. If all nodes of the same class belong to the same cluster, and vice versa, the Jaccard coefficient will be 1 (otherwise, it will be lower).

$$J = \frac{f_{11}}{f_{01} + f_{10} + f_{11}} \tag{5.2}$$

QoS Awareness

The QoS awareness of the overlay was measured by evaluating how close to the optimum the available QoS in the overlay is. For bandwidth, this is done by computing the ratio of the bottleneck bandwidth in the Flocks overlay, and the bottleneck bandwidth that would be available by optimal link placement. Only the QoS between nodes of the same content class is considered. If two nodes of the same content class can not reach each other (which means that the overlay is partitioned), the QoS between them is zero. A ratio of 1.0 indicates that no other link placement by the Flocks overlay would increase the bottleneck bandwidth between nodes of the same

Name	Value	TTL	Confidence	Origin	Propagated From
`class`	X	2	1.00	C	*empty*

Table 5.2: The property set for the evaluated scenario.

content class.

5.1.4 Evaluated Scenario

A common scenario was designed in order to evaluate the performance of the Flocks overlay, and used for all experiments, unless noted otherwise. This allows comparing the effect of any changes and optimizations directly. Each experiment was repeated ten times. The results were averaged, and the standard deviation and the standard error were obtained.

For my experiments, I assume a scenario in which the overlay needs to be both content- and QoS aware. This scenario was chosen because it is a common occurrence in SOMA, and because it demonstrates the flexibility of the Flocks overlay. Nodes are primarily interested in content. They still connect to nodes with different content, but they do so only in order to find better neighbours. Secondarily, they are interested in high QoS. Therefore, one cluster is expected to be formed for each content class. Within each cluster, I expect that the overlay optimizes itself to provide the maximum QoS. This scenario was modelled into Interests and Properties as follows.

Content is represented by a number randomly assigned to every node, which expresses the content class this node has, and is interested in. This splits the overlay nodes into distinct classes, namely nodes that are interested in content 1, 2, and so on. A node C uses a single property, *class*, which contained a number (X, from 1 to 5) specifying the class it had, as shown in Table 5.2.

Five different content classes were used, consisting of 25, 23, 20, 17 and 15 nodes respectively. Additionally, I performed experiments with fewer content classes. In doing so, I observed an effect previously mentioned by [26], namely that the stabilization speed of the overlay increases, as nodes are more likely to find a node of the same class in their neighbourhood.

QoS Awareness

I chose bandwidth as the QoS metric that Flock nodes were interested in, and randomly assigned each link a bandwidth class: high, medium or low. This random assignment was saved and used for all experiments with the same number (100, 200, 300, ...) of nodes.

The simplification to three bandwidth classes allows computing the optimality of the Flocks overlay in constant time: An optimal structure groups all high bandwidth links together, followed by medium bandwidth links, which are finally followed by low bandwidth links. It is easy to replace the QoS metric used in Flocks with, for example, delay, or even several QoS metrics, by simply using a different interest. However, computing the optimal structure for such a combination of metrics, in order to evaluate the performance of Flocks, is significantly more difficult.

Interest used in the Scenario

Based on the considerations above, the interest of a node with content class X, was set as shown in Equation (5.4). A small value ϵ is added to the interest, to ensure it remains non-zero. As a result, nodes still maintain a connection to neighbours of a different class if they do not find neighbours of their own class.

$$\begin{aligned}\big[\big(Match(''class'', \mathrm{X}, 1.0), 1.0\big)\wedge \\ \big(Match(''qos.bandwidth.up'', MAX, 0.0), 0.5\big)\big] + \epsilon\end{aligned} \tag{5.3}$$

Note that the nodes in the overlay do not know their own, or any other nodes' bandwidth: Instead, they determine it through measurement, when they connect to another node. The bandwidth between two nodes is determined by the slower node: A high bandwidth node connecting to a low bandwidth node will only observe a low bandwidth on that link. Naturally, if the bottleneck of a connection is the node itself, it will never see neighbours with a higher bandwidth class than its own. Symmetric bandwidth links were assumed. Regarding asymmetric links would raise the complexity of the model and would not essentially change the clustering criteria.

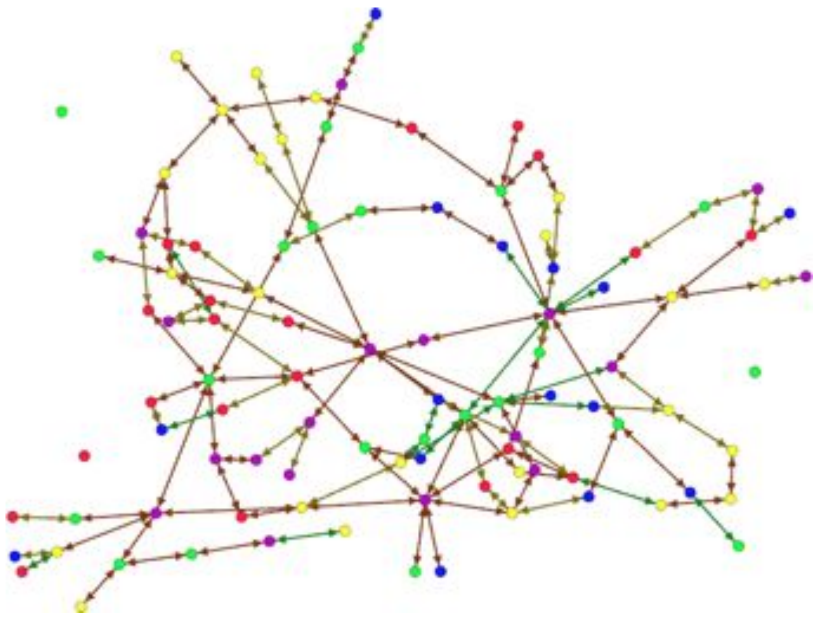


Figure 5.2: An overlay graph after 5 seconds, before any optimizations by the Flocks protocol have been performed.

Illustration of the Optimizations Performed by Flocks

To better illustrate the effect of the Flock overlay, consider Figure 5.2, which shows an overlay graph taken from one of the simulations explained later in this chapter. This overlay contains 100 Flock nodes, divided into five different content-, and three different QoS classes (bottleneck bandwidth), as explained above. Each node is initially assigned a random content-class, which is represented by its colour, and a random QoS-class, which is represented by the colour of its links. The properties and the interests were used as explained in the previous section. The reader should keep in mind that a graph drawing algorithm [119] has been applied to all graphs in this thesis, in order to reach a clearly arranged visualization, and to allow easy identification of which nodes are connected to each other. Because of that, the coordinates of the individual nodes bear no meaning, and may differ from graph to graph.

Figure 5.2 shows the initial overlay, before any optimization. Notice that nodes of the same content class are far apart from each other, separated by nodes of different content classes. This means that content belonging to one class needs to be relayed over several unrelated nodes of a different content class. The graph also exhibits poor

QoS awareness. Consider the node at the bottom right: It has a high bottleneck-bandwidth link to its only neighbour, which is again connected to only one neighbour over a low bottleneck-bandwidth link. Any video stream sent over that link would be need to pass this bottleneck, and so the node at the bottom right can not benefit from its high bottleneck-bandwidth link.

Figure 5.3 shows the overlay graph after one minute, during which the algorithms of the Flock overlay, running on each node, performed optimizations. Notice that nodes of the same content class are now clustered together into five distinct groups. Furthermore, the QoS has improved: Nodes with high bottleneck-bandwidth links are grouped together, followed by nodes with medium, and nodes with low bottleneck-bandwidth links at the edges[8]. As a result, an application using this overlay could now employ a simple limited-flooding algorithm to broadcast content and - without any knowledge of the overlay - reach all interested nodes with high QoS. It is important to note that the clustering is done in a self-organizing manner: Nodes know only their neighbours and the clustering topology is an emerging feature. Furthermore, they have no information about whether they are the bottleneck, or one of their neighbours is: Nodes can only measure QoS related metrics, and have to base their decisions on that.

5.2 Performance Evaluations

The first evaluations were performed under controlled conditions, through simulations. The global view provided by the simulator allowed me to observe the overlay in real time, with accurate network graphs, and allowed me to measure the bandwidth consumed and the traffic used by each individual node, which would be difficult to obtain otherwise.

Preliminary evaluations included selecting the size of the candidate list that nodes use, and the strategy they use to select nodes from their candidate lists. These simulations were performed first, since the rest of the experiments depended on them.

[8]This is particularly evident in the top right cluster of nodes. Some connections are not optimized yet, such as the cluster in the top left. They are getting optimized later in the simulation.

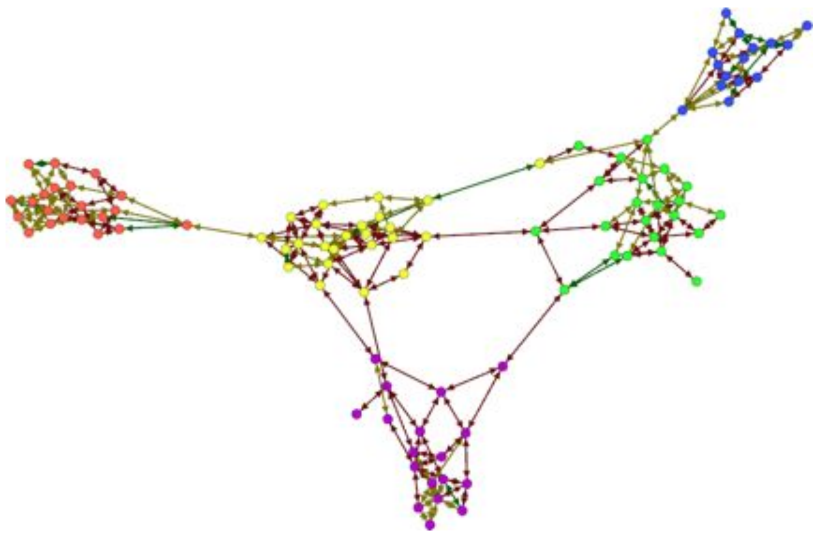


Figure 5.3: Network graph after 60 seconds, during which the Flocks protocol on each node performed optimizations, based on locally available information only.

5.2.1 Candidate Selection

The "candidate list" of a Flock node contains a list of nodes that it is aware of, but is currently not connected to. Each Flock node regularly connects to a node in its candidate list, exchanges properties, and evaluates its interest in it. Only connections to the n nodes it has the highest interest in, are kept. The remaining connections to the remaining nodes are severed, and they are moved to the candidate list again. Because no QoS related information is available until two nodes connect, a node can only accurately determine a QoS-based interest in another node by connecting to it.

A Flock node's candidate list serves (i) as a list of potential neighbours that may improve its situation, by being more interesting than its current neighbours, as well as (ii) a backup, in case one or more of its current neighbours disconnect or fail.

Selection Order

The order in which candidate nodes are considered makes a noticeable difference in the performance of the overlay, as a Flock node only considers one candidate node per optimization step. For example, consider the Ironman-supporting overlay detailed earlier. Visitors may exhibit preferences towards a certain activity, such as "swimming". If only unrelated camera nodes (namely nodes with the properties "bicycling" and "running") are drawn from the candidate list, the overlay performs poorly.

Several strategies have been considered to select nodes from the candidate neighbour list:

- Random Selection: All nodes in the candidate list are treated without bias.
- Earliest Deadline First (EDF): Any time a node is added to the candidate list, it is given a deadline timestamp. The node with the lowest deadline timestamp is selected next. The deadline given to a node depends in (i) whether the candidate node has been connected to before, (ii) whether the local node was interested in the candidate node, (iii) whether the candidate node was interested in the local node, and (iv) whether recent connection attempts failed.

  Nodes that (i) have never been connected to are given a very close deadline, followed by nodes that were (ii) interesting to the local node and (iii) interested in the local node. Candidate nodes that were either not interesting for, or not interested in the local node are given longer deadlines. Whether or not two nodes were interested in each other is determined by checking the value of the oracle function. If it is below a certain configurable threshold, nodes are said to be "little interested", and given longer deadlines. An interest value of zero signifies "no interest", and is given an even longer deadline. Nodes exchange the interest status (interested, little interested, no interest) with each other. Additionally, if the previous connection attempt failed, an increasingly large time span is added to the deadline.

  This approach has the benefit that every node will eventually be considered, and that no starvation occurs. Using EDF results in fast stabilization, and only requires little additional information to be added to the candidate list, namely (i) the node address, (ii) whether the local node and the candidate node were interested in each other, (iii) the deadline. As a result, the memory required by the candidate list remains small, and independent of a node's properties. However, the disadvantage of this approach is that nodes are less sensitive to changing properties. If a candidate node updates its property, the deadline is not affected.

- Hybrid Priority List (HPL): This strategy separates the candidate list into two segments. The "high priority" segment is composed of nodes closer than a configurable threshold t_{prio} to the deadline. The remaining nodes are located in the "low priority" segment. As this approach still uses the earlier mentioned deadline to determine what segment a node is in, "low priority" nodes will eventually become "high priority", and starvation is prevented.

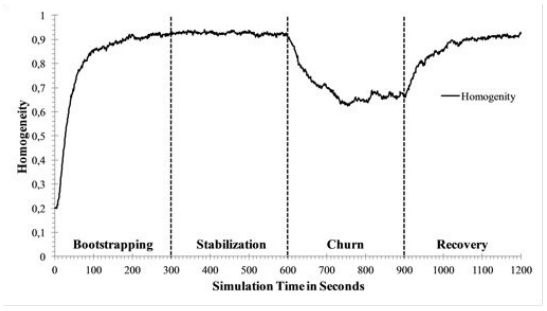
  This strategy first selects nodes from the "high priority" segment. If the high priority segment is empty, the low priority segment is considered. Within the "high priority" segment, nodes are sorted by interest. In order to prevent starvation in the high priority segment, the deadline of any node that is added to the candidate list again is at least as high as the threshold t_{prio}.

  The advantage of this approach is that remains sensitive to property changes. If a node changes its properties, other nodes will propagate this change, as described earlier. This was, other nodes are informed of the change, and update this information in their candidate lists. As a result, they re-evaluate their interest in the changed node, and may update its position in the list.

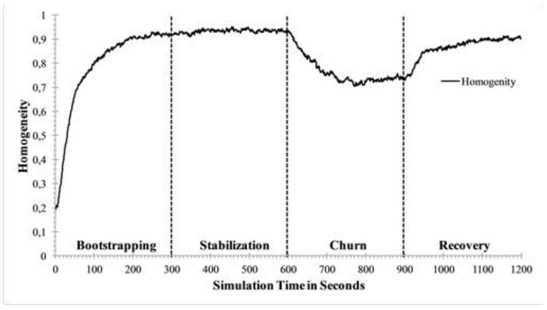
Deterministic selection and replacement strategies make simulations reproducible, and predictable, however they can also pose a problem: Consider that the node removes all information about a candidate that it drops from the candidate list. Therefore, if given the same set of candidates, it will consider and forgo (in case the candidate list is not large enough) the same candidates again. This makes it difficult to escape local optima. Thus, the EDF and the HPL approaches both add a small random number to the deadline of any node when it first enters the candidate list. Due to this random number, the order of nodes in the candidate list will differ slightly, which causes different candidates to be considered and replaced.

Figure 5.4a shows the homogeneity obtained by using the random candidate selection strategy in a simulated overlay network with 200 nodes. Figure 5.4b displays the homogeneity achieved with the same set of nodes, by using the EDF approach. Finally, the homogeneity achieved by using HPL is presented in Figure 5.4c.

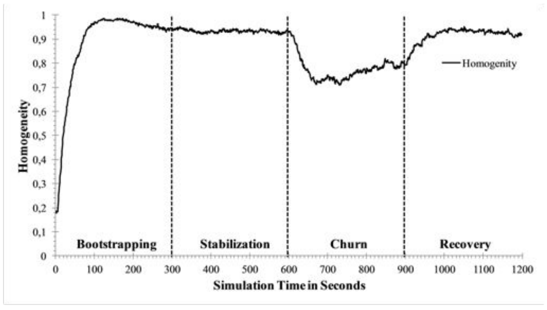
An interesting feature of Figure 5.4c is that the homogeneity, after reaching its maximum, *decreases* again. A visual inspection of the overlay graph revealed that



(a) Random



(b) Earliest Deadline First (EDF)



(c) Hybrid Priority List (HPL)

Figure 5.4: Homogeneity over time in a 200-node network, with different candidate selection strategies.

at its maximum homogeneity, the overlay was divided into five densely connected clusters, with very few inter-cluster connections that resulted because nodes have a low, but non-zero interest in nodes of another class. Therefore, nodes that can still accept neighbours, but cannot find any additional neighbours of their own cluster, may also accept nodes of a different class as neighbours, thus decreasing the homogeneity of the overlay slightly. If the interest in nodes of another class is set to zero, these connections are indeed removed, and the homogeneity reaches 1.0.

Strategy	Bootstrapping	Stabilization	Churn	Recovery
Random	0.805 ± 0.183	0.926 ± 0.006	0.699 ± 0.069	0.865 ± 0.063
EDF	0.786 ± 0.184	0.933 ± 0.007	0.766 ± 0.058	0.867 ± 0.044
HPL	0.879 ± 0.187	0.933 ± 0.007	0.775 ± 0.049	0.916 ± 0.032

Table 5.3: Average homogeneity in a 200-node network, with different candidate selection strategies.

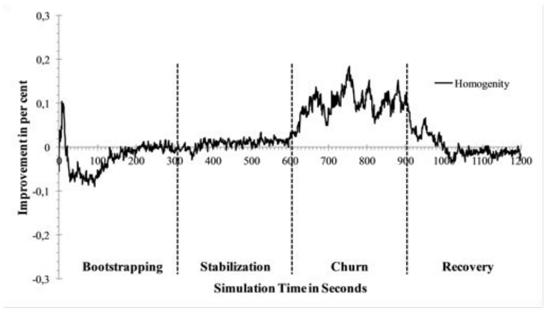
Table 5.3 compares the average homogeneity for each strategy in the four different phases. It can be seen that the Hybrid Priority List outperforms the other approaches.

During the Bootstrapping, Stabilization and Recovery phases the Random and EDF strategies show very similar performance. When the Bootstrapping phases begin, they immediately make large changes to the overlay. After this change, nodes do not join or leave the overlay as it stabilizes again. This is also true for the Stabilization and Recovery phases. Therefore, if the overlay is expected to be stable, it is reasonable to use the simpler random strategy over the slightly more complex EDF strategy.

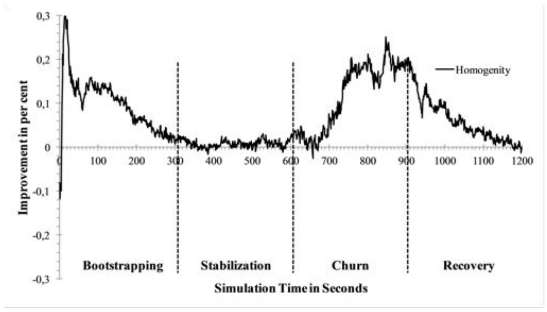
The limitations of the random strategy become relevant in the churn phase. During that time, nodes continuously join and leave the overlay. The EDF and HPL strategies obtain a higher homogeneity, as they use longer deadlines for neighbours that failed, and therefore select a neighbour still in the overlay with a higher likelihood than the random strategy would.

One surprising observation is that selecting neighbours with likely matching interest only has a minor effect on the homogeneity. I initially expected the average homogeneity of HPL to be much higher than EDL, or random selection. However, Table 5.3 clearly shows that HPL and EDF have similar performance in most phases, and that the homogeneity of the random approach is not much lower than HPL if the overlay is stable. The reason for this might not be obvious immediately: A Flock node, once it has sufficiently many neighbours, only keeps neighbours with the highest interest. If its neighbours do the same, nodes quickly flock together into "groups" of nodes that share a similar interest. Since a Flock node becomes aware of other nodes primarily through property propagation, its candidate list fills with candidates that its neighbour nodes find interesting. In many situations the interests of a node and its neighbours match; therefore these candidate nodes will also be interesting to the node itself, and eventually all candidates become relatively good choices. Some candidates may still be more interesting than others, but usually there are few "really bad" choices that a node can make. In this situation, the strategy for node selection becomes irrelevant, which explains the similar performance observed in Table 5.3.

However, even though the performance may be similar in the long term when the overlay is sufficiently stable, the speed at which the overlay quality improves may still differ. To investigate this, I computed the improvement in homogeneity over time for strategies different than the random selection strategy. This was done by calculating



(a) Earliest Deadline First (EDF)



(b) Hybrid Priority List (HPL)

Figure 5.5: Improvement of homogeneity in a 200 node network, if a strategy different than the random strategy is used.

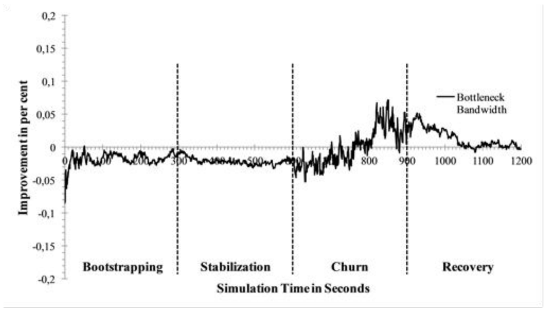
the ratio, as shown in Equation (5.4). The improvement in homogeneity at time t is $h_{improvement,t}$ if strategy X (with a homogeneity of $h_{t,x}$ at time t) is used instead of the random strategy (with a homogeneity of $h_{t,RANDOM}$ at time t).

$$h_{improvement,t} = \frac{h_{t,x}}{h_{t,RANDOM}} - 1 \tag{5.4}$$

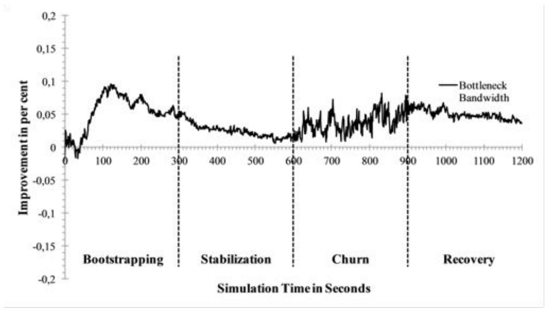
Figure 5.5a shows the improvement for EDF, and confirms the observations above: EDF performs very similar to the random approach if the overlay is sufficiently stable. Under churn, EDF outperforms the random node selection, as unreachable and uninterested nodes are assigned longer deadlines, and the candidates retrieved from the list are more likely to be either nodes that were never considered before, or that were at least somewhat interested before.

Figure 5.5b shows that the higher average homogeneity of HPL during the bootstrapping phase (as seen in Table 5.3) is caused primarily by a much faster improvement of the overlay. HPL reaches a homogeneity of $h = 0.9825$ after 120 seconds, at which point the random approach has a homogeneity of $h = 0.8582$. After 300 seconds, the random approach reaches its maximum homogeneity of $h = 0.94$. It can be seen that if the overlay remains stable, the resulting homogeneity of all approaches is very similar. The benefit from using HPL comes from a quicker improvement under churn.

Figure 5.5a shows that the EDF strategy only provides minor improvements late in the experiment, during the churn phase. Even worse, it initially results in lower homogeneity than the random approach.



(a) Earliest Deadline First (EDF)



(b) Hybrid Priority List (HPL)

Figure 5.6: Increase of bottleneck bandwidth between nodes of the same content-class in a 200 node network, if a strategy different than the random strategy is used.

The same trend is also evident if the bottleneck bandwidth between nodes of the same content class is compared. Figure 5.6a shows the improvement in bottleneck bandwidth (again in per cent) if the EDF strategy is used, and Figure 5.6b the improvement for the hybrid priority list.

The similarity of the bottleneck bandwidth and the homogeneity figures stems from the fact that in an overlay with higher homogeneity, nodes of the same class are closer together, which decreases the chance of having to relay over a low-bottleneck bandwidth link.

Size of the Candidate List

If left unmanaged, the size of the candidate list will grow until it contains all nodes in the overlay. Clearly, this is not an option, and the size of the candidate list must be limited. A smaller candidate list requires less memory, however it decreases the improvement speed of the overlay for a number of reasons. Specifically, a larger candidate list brings the following benefits:

- Choice: Most intuitively, a larger candidate list allows a node to store more new candidates, before it must choose between either replacing an existing candidate, or not accepting new candidates. Pretend that a node can store five candidates. Upon connecting to a node with five neighbours in the overlay, it becomes aware that node's neighbours, and adds them to its own candidate list. It then takes the first node from the candidate list, connects to it, and thus becomes aware of that node's neighbours as well. However, it can only store one additional candidate. Either the remaining neighbours of the second node are not considered, or some of the previously added candidates must be removed without ever been connected to. While this situation will eventually occur for any limit, a larger limit increases the number of candidates a node can retain for later consideration.

- Memory: Nodes that are removed from the candidate list are, essentially, "forgotten". For example, assume a node that has previously connected to a neighbour X, and found that it is not a good match. As it finds more potential neighbours, its candidate list fills up, and eventually, nodes must be removed to make room for new candidates. Once node X is removed, all information is lost. Therefore, the node will consider node X a potential neighbour again when it becomes aware of its presence, even though it will still be a bad match. By reconsidering previous neighbours that are still bad matches, the overlay improvement is slowed. A larger candidate list increases the memory a node can retain, and decreases the likelihood of inadvertently revisiting a candidate.

- Resilience in face of large scale failures: During large scale failures, nodes become disconnected, and the overlay may partition. A node may lose many of its neighbours, or existing neighbours may no longer be the best matches. If the candidate list is small, there is a chance that all of the nodes it contains have failed. A larger candidate list helps to recover from this kind failure, because it provides a larger pool of candidates that a node can choose as new neighbours. If a node fails with the probability of f, the chance that all nodes in a candidate list with n nodes fail simultaneously is f^n.

Therefore, it is desirable to limit the size of the candidate list (in order to save memory), but at the same time to make it large enough to benefit from the above properties.

I performed simulations with varying candidate list sizes, in order to determine the effect of different limits on the overlay improvement speed. Simulations were repeated ten times, with 5, 10, 20 and 50 nodes. A node always adds a new candidate nodes:

If the candidate list already contains the maximum number of nodes, the candidate list with the lowest priority (namely, the one at the end of the list) is removed from the list first. Each simulation used 200 nodes. Based on the previous results, I used the hybrid priority list node selection strategy for the evaluations.

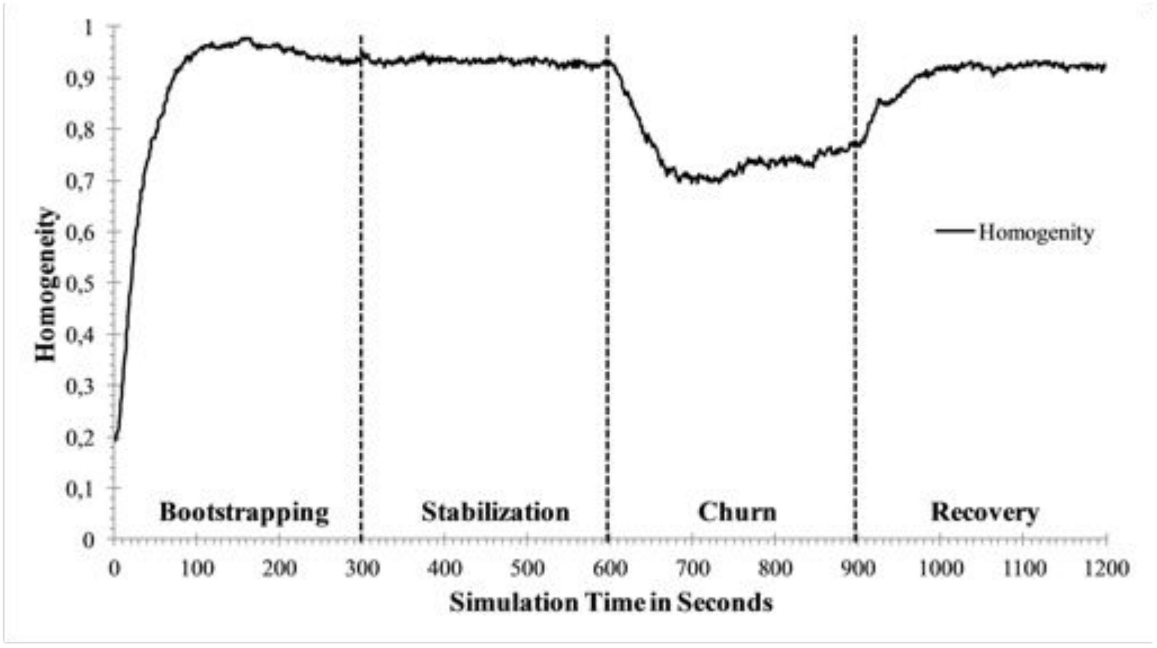
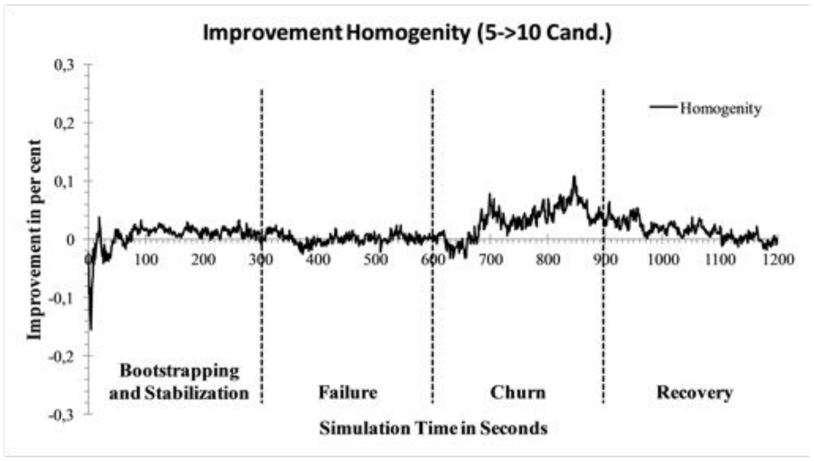


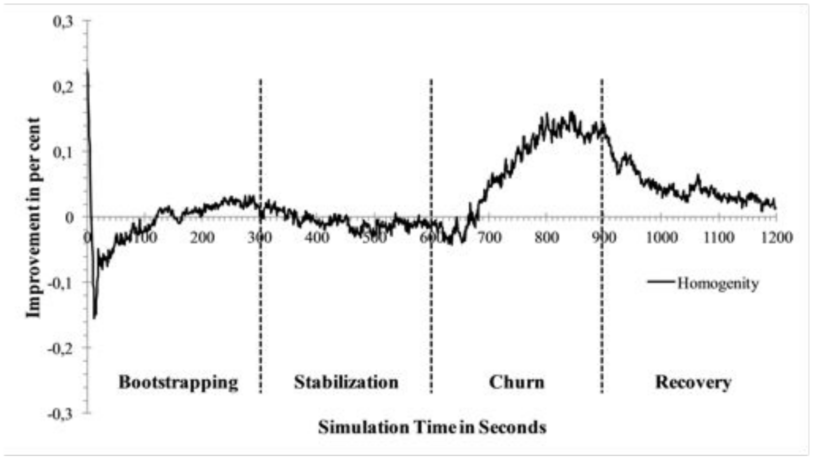
Figure 5.7: Homogeneity in a 200 node network, if each node's candidate list is limited to 5 nodes.

The homogeneity of the overlay for a limit of 5 candidate nodes is shown in Figure 5.7. Figure 5.8 shows the improvement (in per cent) if the limit is increased to 10 (Figure 5.8a), 20 (Figure 5.8b) and 50 (Figure 5.8c) candidate nodes. An important observation from these figures is that a larger candidate list is helpful in dealing with sustained churn, and that the benefit is less pronounced if the overlay is mostly stable.

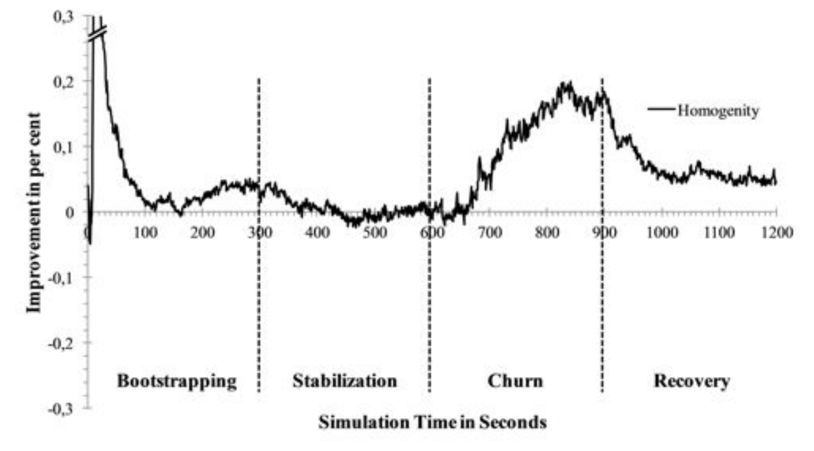
Table 5.4 compares the average improvement in homogeneity for different limits of the candidate list. It can be seen that increasing the limit from 5 to 10 nodes results in an average increase of up to 1.5% during stable phases, and a 3.1% increase during churn. An interesting observation is that further increasing the limit to 20 and 50 only results in a limited improvement during churn. The improvement is greater if the overlay is stable, however it slows down as well: Increasing the limit from 20 to 50 nodes does result in as high of an increase as increasing the limit from 10 to 20 does.



(a) Improvement with 10 Candidate Nodes



(b) Improvement with 20 Candidate Nodes



(c) Improvement with 50 Candidate Nodes

Figure 5.8: Increase in homogeneity if a candidate list with a limit of 10 (b), 20 (c), 50 (d) nodes is used instead of a candidate list limited to 5 nodes.

Based on these results, a candidate size limit of 20 seems to offer a good compromise between memory requirement, stabilization speed and resilience.

Limit(Nodes)	Bootstrapping	Stabilization	Churn	Recovery
10	0.006 ± 0.023	0.001 ± 0.009	0.031 ± 0.028	0.015 ± 0.015
20	-0.004 ± 0.042	-0.006 ± 0.011	0.071 ± 0.064	0.047 ± 0.026
50	0.060 ± 0.089	0.006 ± 0.015	0.100 ± 0.069	0.070 ± 0.030

Table 5.4: Average improvement of homogeneity in a 200-node network, with different limits for the size of the candidate list.

The bottleneck bandwidth achieved with a candidate list size of 5 nodes is shown in Figure 5.9. The improvement by increasing the candidate list is shown in Figure 5.10a (10 candidate nodes), Figure 5.10b (20 candidate nodes) and Figure 5.10c (50 candidate nodes).

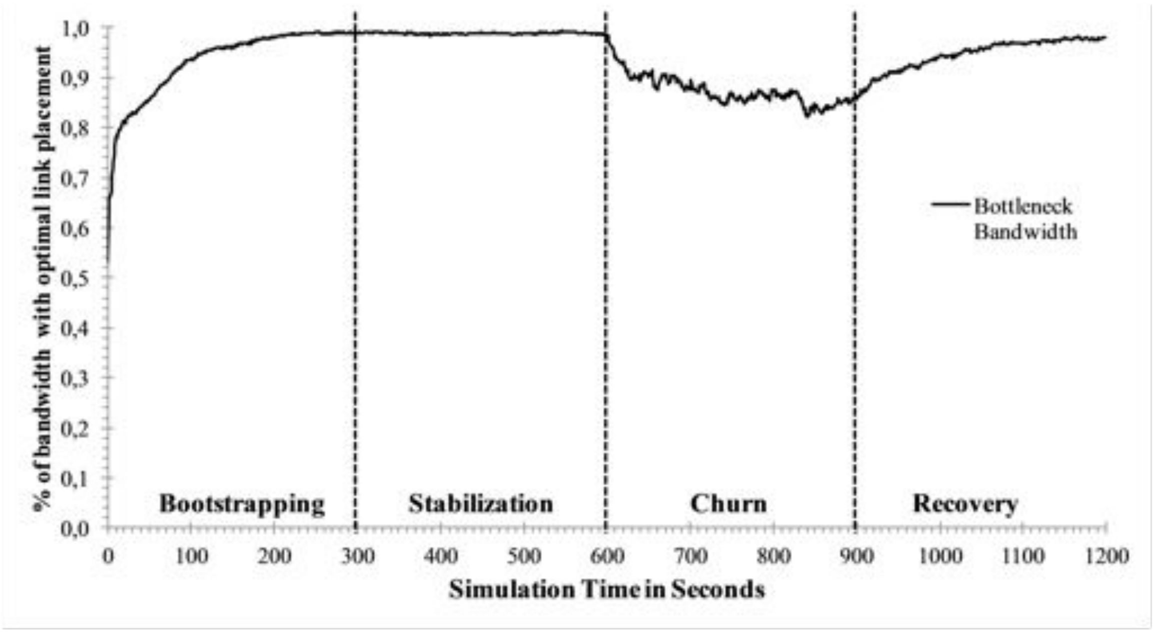
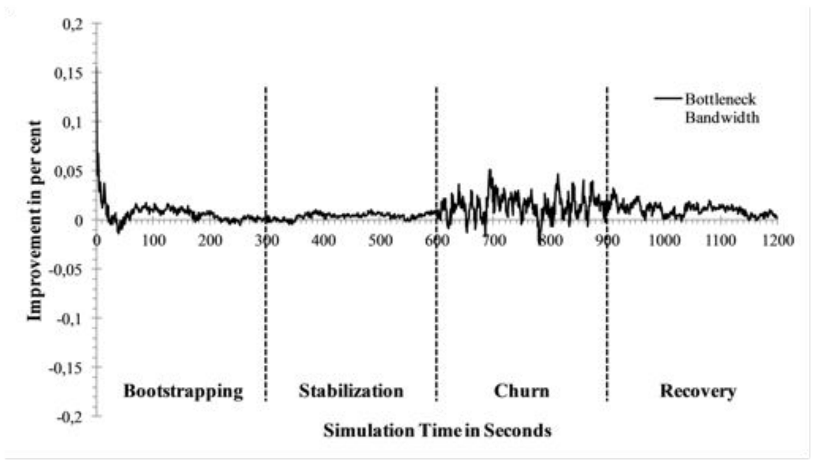


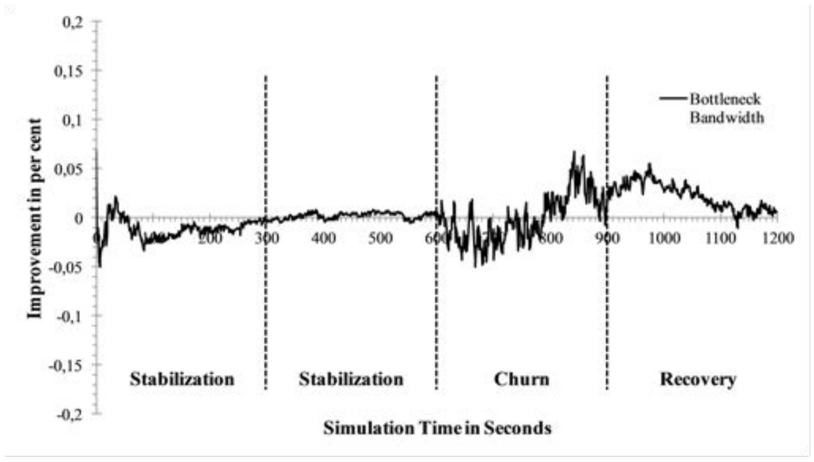
Figure 5.9: Bottleneck bandwidth (in per cent of the global optimum) achieved in a 200 node network, if each node's candidate list is limited to 5 nodes.

5.2.2 Overlay Maintenance Traffic

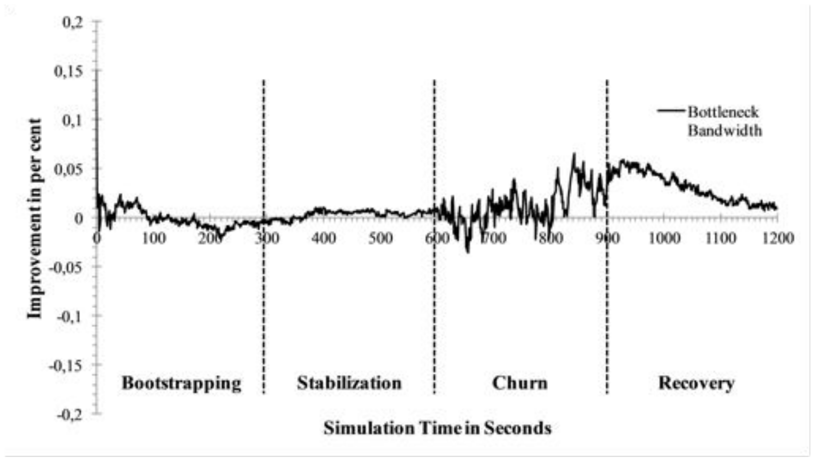
Initial experiments showed that management traffic of the Flocks overlay occurs in bursts, as seen in Figure 5.11. The reason for the bursts is that Flocks overlay nodes



(a) Improvement with 10 Candidate Nodes



(b) Improvement with 20 Candidate Nodes



(c) Improvement with 50 Candidate Nodes

Figure 5.10: Increase in bottleneck bandwidth if a candidate list with a limit of 10 (b), 20 (c), 50 (d) nodes is used instead of a candidate list limited to 5 nodes.

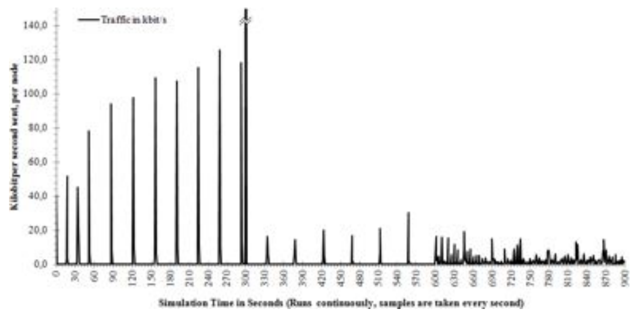


Figure 5.11: Initial maintenance traffic, with a constant timer and no optimizations.

periodically determine when to connect to new neighbours or evaluate existing ones: This approach was initially chosen because it bounds the traffic consumed by the overlay itself for maintenance and overlay improvement. In addition, it allows tweaking the stabilization speed of the overlay based on available bandwidth: Increasing the period consumes less bandwidth, but increases the time required for improving the overlay structure. In an initial simulation, the timer for connecting to new nodes was set to 25 seconds.

Admittedly, the simulations in this section represent the worst case: In Figure 5.11, all nodes joined the overlay at the same time, i.e. the timers are perfectly synchronized. The traffic decreases significantly as 40% of the nodes fail after 300 seconds. Once churn sets in (600 seconds into the simulation), the timer of the newly joined nodes is no longer synchronized with the nodes already in the overlay. Despite this, bursts can still be observed until the end of the simulation. While the required bandwidth is low if averaged over the lifetime of the simulation ($2.5kbit/s$), the peak bandwidth required during the largest burst (not counting the burst in response of the large-scale failure at 300 seconds) averages at $125.6kbit/s$ per node. These high-bandwidth bursts compete with the bandwidth available to the upper layers that build up on the Flocks overlay, and would make the overlay unsuitable for multimedia data distribution.

Therefore, these bursts must be removed. The easiest way for this is randomization: By introducing a variance into the timer, chosen by every node at random, nodes will become active at different times. To prevent the variance from becoming synchronized[9], each node chooses a new random delay when it wakes up to find new

[9]This happens if several nodes pick the same variance, or pick multiples of that variance. For

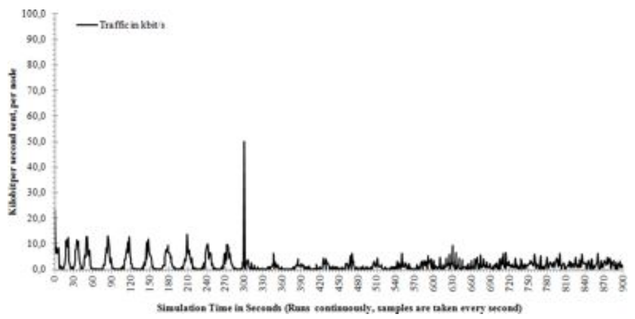


Figure 5.12: Maintenance traffic with a 20-25s variance.

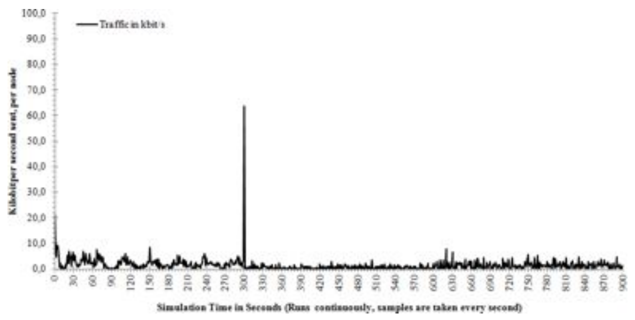


Figure 5.13: Maintenance traffic with a 20-40s variance.

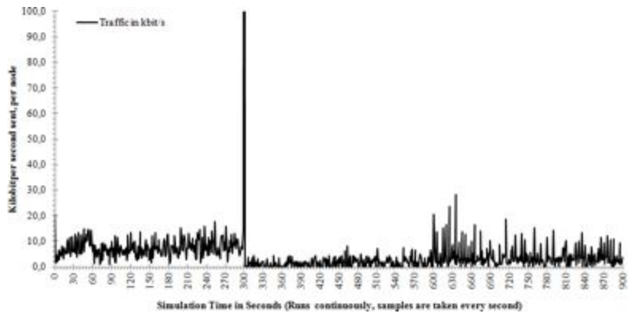


Figure 5.14: Maintenance traffic with a 5-25s variance.

neighbours.

Figure 5.12 shows the traffic profile for a delay ranging from 20 to 25 seconds. It shows that a small variance of five seconds is insufficient for quick synchronization. A delay of 20 to 40 seconds, as seen in Figure 5.13 dampens the bursts, however despite using an attractive average bandwidth ($2.2kbit/s$ over the first 300 seconds, $1.4kbit/s$ over the entire simulation), the stabilization speed of the overlay is slowed down significantly by the long delay. This explains why the bursts can still be observed even after 300 seconds. Using a lower delay of 5 to 25 seconds greatly improves the situation. Figure 5.14 shows that the bursts are removed almost immediately[10]. The resulting traffic profile is much friendlier to the upper layers that build up on the Flocks overlay, as it consumes an even share of bandwidth, with no large bursts. The disadvantage of the lower delay is the increased bandwidth cost. On average, the bandwidth required to maintain the overlay with such a low delay between connections is $7.4kbit/s$ per node, which may be prohibitive for multimedia applications.

Adaptive Timing Profiles

The most obvious way to decrease the maintenance traffic is to increase the timer, and cope with the slower stabilization speed. However, I was able to reduce the traffic *without* slowing the stabilization speed by using adaptive timing profiles.

The main idea of Adaptive Timing Profiles (ATP) is using a short timer during bootstrapping and if any neighbours failed (down to a *lower bound*), in order to quickly discover good neighbours. Once the local neighbourhood of a node becomes stable, the timer starts to increase gradually, slowing the search and lowering the maintenance traffic (up to an *upper bound*). At this point, the overlay is stable, only minor changes are made to its structure, and the management traffic drops.

ATPs also benefit multimedia applications: A node in a sparse overlay region quickly improves its situation by searching for new neighbours more aggressively. Once it is "satisfied" with its current situation, the changes become fewer and slower,

example, should several nodes choose a variance of 10 seconds, and others a variance of 20 seconds, the maintenance traffic would still come in bursts.

[10]It should be noted that large burst that remains in both graphs 300 seconds into the simulation is caused by the aforementioned 40% node failure: All remaining nodes immediately begin looking for replacement nodes, causing the burst seen in both figures.

leaving bandwidth to the upper layers.

Node Satisfaction

How aggressively a node searches for new neighbours depends on how "satisfied" it is. I modelled this satisfaction with an *aggression factor*, which is a floating point value ranging from 0.0 (not aggressive) to 1.0 (very aggressive). Initially, it starts at 1.0, but as the node is getting satisfied with its current situation, a dampening factor d is applied, which gradually reduces the aggressiveness to 0.0. In the experiments presented in this section, I set d to be 0.7. A lower value causes the overlay to settle more quickly, reducing the traffic used for maintenance but increasing the time it takes to reach the optimal overlay configuration.

This aforementioned aggressiveness directly influences the value of the timer used by ATP: If the aggressiveness is 1.0, the lower bound of the timer is used, which causes the node to search for new neighbours quickly, and, as a result, make large changes to its neighbourhood. As the aggressiveness drops, the timer increases, up to the upper bound, which is reached once the aggressiveness is 0.0.

Each node computes its aggressiveness by invoking `Satisfaction`, as shown in Algorithm 5.1, every time it obtains a new neighbour. Essentially, a node is "satisfied" if it has reached it upper bound of neighbours, and has not been able to find better neighbours since the last time it invoked `Satisfaction`.

Algorithm 5.1 Computation of a node's aggressiveness factor

```
1: procedure SATISFACTION($neighbours, new\_neighbours$)
2:     if $neighbours < max\_degree \vee new\_neighbours$ then
3:         $aggressiveness \leftarrow 1.0$
4:     else
5:         $aggressiveness \leftarrow aggressiveness * d$
6:     end if
7: end procedure
```

Figure 5.15 shows the traffic consumed by an overlay with ATP. The lower delay bound is set to six seconds, and used if a node is aggressively searching for neighbours. The upper delay bound is a random value between 30 and 60 seconds, which is used once a node is "satisfied". The traffic initially peaks at $27.1kbit/s$ as all nodes join

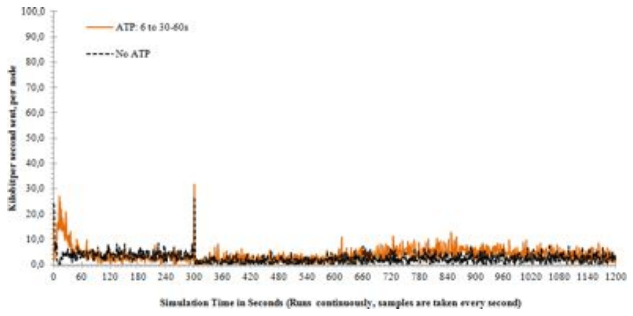


Figure 5.15: Maintenance traffic with and without adaptive timing profiles.

the overlay at the same time. Large changes are made to the overlay structure, which quickly level off as nodes become satisfied. After the first minute, until the large scale failure after 300 seconds, each node uses an average bandwidth of only $2.8kbit/s$. After the failure, the bandwidth peaks, as all remaining nodes begin looking for replacement neighbours, and drops off again afterwards. Finally, as churn sets in, the required bandwidth increases moderately to an average of $3.8kbit/s$, as new nodes join the network and overlay nodes lose neighbours as existing leave.

The ATP shown in Figure 5.15 requires a $2.4kbit/s$ higher bandwidth than the *most bandwidth conserving approach* evaluated earlier, which is a reasonable trade-off, given that both values are sufficiently low, and that the adaptive approach is up to six times more responsive than the non-adaptive one. In response to neighbour changes, nodes search for new neighbours every 6 seconds, as opposed to the 20-40 seconds used by the non-adaptive bandwidth conserving approach. By increasing the lower bound from 6 to $8-12$ seconds, the average required bandwidth of ATP dropped by 26%, at the cost of slower stabilization.

It should be noted that even if the overlay is stable, the maintenance traffic never drops to zero. This is because nodes keep looking for new potential neighbours in order to improve the overlay. As every node only maintains a local view, it is not aware of how close to the optimum the overlay as a whole is. However, as long as no better nodes are found that are available and willing to accept a new neighbour, the structure of the overlay remains stable, allowing the higher level content distribution algorithms to work with without interruption.

5.2.3 Overlay Performance

I collect data samples to compute graph and cluster quality metrics, and comparing the bandwidth available by the current overlay structure throughout the simulation. Content awareness is evaluated using Homogeneity and the Jaccard Similarity Coefficient. In order to evaluate QoS awareness, I first compute the optimal link placement that would result in the highest bottleneck bandwidth to every node, given the current set of nodes. I then compare the bottleneck bandwidth available for every pair of same content-class nodes in this optimal overlay, to the bottleneck bandwidth in the current overlay created by Flocks. These metrics are *global* quality measures of the overlay structure, which we obtain using the *global* view provided by the simulator. Individual nodes are not aware of these quality measures, as they only have a *local* view. Therefore, they may dismiss a neighbour they have low interest in, even if a connection to it would benefit the overlay as a whole. Despite this caveat, the results show that the Flocks overlay stabilizes quickly and improves continuously towards the global optimum.

For comparison, the traffic, the QoS awareness and the homogeneity of an overlay without ATP, using a 15-40 seconds random delay[11], are superposed in Figures 5.15, 5.16 and 5.17.

QoS Awareness

Figure 5.16 measures the QoS awareness of the network in Figure 5.15. Observe that the bottleneck bandwidth reaches 94% of the optimum within 90 seconds. Small dibs can be observed throughout the simulation because each Flock node primarily searches for neighbours with the same class as its own, and only secondarily considers QoS. In other words, given the choice between a node with better QoS and a node of the same class, a Flock node using the interests described in section 5.1.4 will choose the latter. Because of that, the QoS sometimes decreases as a tradeoff between QoS and homogeneity.

As the simulation approaches the 300 second mark, the improvements of the adaptive network begin to slow down. Eventually, the non-adaptive network reaches a

[11]I chose a range of 15-40 seconds because it is the same average delay as an overlay with ATP that uses its upper bound half of the time, and its lower bound half of the time.

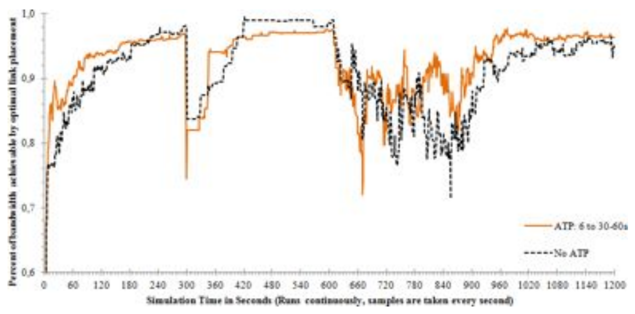


Figure 5.16: Bottleneck bandwidth achieved by the Flocks overlay in percent of the optimal bottleneck bandwidth calculated by a centralized algorithm, with and without adaptive timing profiles.

higher QoS than the adaptive one. This is because all nodes have a sufficiently high number of good neighbours, and become "satisfied", slowing the speed that nodes search for new neighbours. The non-adaptive network, however, continues to use a fast search for new neighbours, at the cost of increased bandwidth (see Figure 5.15). This situation could be improved by modifying `Satisfaction` to include reasonable assumptions about expected QoS, so that a node would only be satisfied if it has at least one neighbour with a similar QoS as its own.

At 300 seconds, the QoS drops as 40% of the nodes fail. As a result of the failure, nodes that have lost all their high QoS neighbours now only have a poor QoS to the remaining overlay, which is represented by a large drop in Figure 5.16. Notice that the Flock nodes quickly remedy this situation and a QoS within 95% of the optimal solution is re-established soon. As churn starts, the QoS begins to vary between 80% and 95% as nodes drop from the overlay, and the Flocks continuously repair and improve the mesh. Notice that the overlay using ATP is consistently more responsive to failure and churn than the non-adaptive overlay, which temporarily drops below 80% of the optimal QoS. This is because "unsatisfied" nodes using ATP search for neighbours at a faster rate, at the cost of more traffic. This traffic slows down once the overlay is stable and nodes become "satisfied" again. In contrast, the overlay without ATP never "settles down", even if the overlay is stable. Because of that, it has a higher QoS than the adaptive approach at the 210 and 240 second marks in Figure 5.15, but also higher bandwidth demands during periods where the overlay is stable.

Homogeneity

The homogeneity of the same network is shown in Figure 5.17. Similarly to the previous cases, it increases quickly at the start of the simulation, as nodes build clusters with other nodes of the same class, and keeps improving continuously. Small dibs in homogeneity can be observed in this graph as well: They are caused by nodes that connect to new candidates during their search for better neighbours. If the new candidate has a different content-class, the homogeneity temporarily drops while both nodes exchange properties.

Generally speaking, homogeneity can be improved by severing a connection to a neighbour of a different class, and replacing it with a connection to a neighbour of the same class. The Interests described in Section 5.1.4 aim to keep the overlay unpartitioned: each node prefers neighbours of the same class, with high bandwidth, but in case none are available, it also retains connections to nodes with a different class. As networks exist in which a perfect homogeneity of 1.0 can not be achieved without reducing the total number of links, a perfect homogeneity of 1.0 is never reached as long as Flock nodes accept all neighbours up to their neighbour limit.

The authors of [26] face a similar situation: The algorithms they evaluate only replace links, but never unconditionally remove them. As a result, the number of links in the network remains the same. To measure how far the algorithms are from their optimum, the authors introduce a new metric, termed "Optimality". It requires computing the "upper bound for homogeneity" first. The homogeneity is then divided by the optimal result, giving optimality as a ratio between the network's current homogeneity and the maximum achievable optimality.

How exactly the "upper bound for homogeneity" should be calculated is not explained, except that a "centralized optimal clustering algorithm" was used. This makes comparing the results of both approaches difficult, in particular as Flocks are not only focused on clustering the network, but are also QoS aware. Despite this, I observed that even without using the less restrictive Optimality metric, Flocks reach a homogeneity of > 0.9 within 95 seconds. In Section 5.2.4 I adjust the interests of Flocks slightly to make them drop links to nodes with a different content class if the overlay is stable, and reach a perfect homogeneity of 1.0 after 135 seconds.

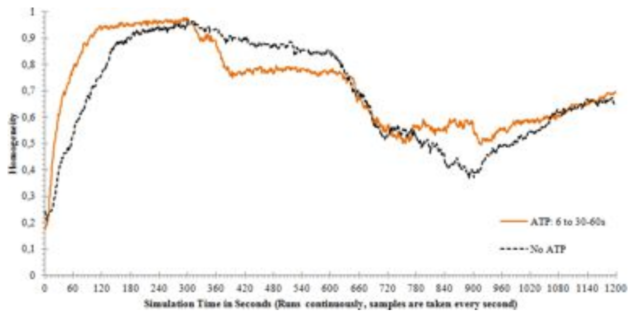


Figure 5.17: Homogeneity of the Flocks overlay with and without adaptive timing profiles.

During churn, the homogeneity drops rapidly as newly joining nodes connect to the nodes they obtained from the bootstrapping service. This is expected, as the bootstrapping service supplies nodes randomly, regardless of what class they belong to. Once a newly arrived node connects to an overlay node, it obtains that node's properties, as well as the properties of its direct neighbours. This information is used to begin searching for nodes of the same class.

Jaccard Similarity Coefficient

I use the Jaccard Similarity Coefficient to determine if all nodes of the same class are found in the same partition, and vice versa. In these first experiments, the interests kept the overlay connected, hence the Jaccard similarity coefficient does not produce meaningful results. It remains stable at 0.197 throughout the during the experiment, which is not surprising: As the overlay remains unpartitioned, there are always fewer partitions (namely one, consisting of the entire overlay) than classes (five).

Effect of ATP on overlay quality

Figures 5.16 and 5.17 show that overlays with ATP improve more quickly than overlays without ATP, at the cost of initially increased traffic. As more nodes become "satisfied", however, the rate at which they attempt to improve the overlay slows down, resulting in *lower* average traffic overall. As a result, they leave more bandwidth for multimedia applications in absence of churn and failure, with brief degraded QoS after failure as the Flock nodes repair the overlay.

In Figure 5.16, the adaptive overlay slows down early: This is because nodes are satisfied with their current neighbours and did not find neighbours with a better QoS recently. As a result, they use the upper bound of 30-60 seconds, which results in a slower improvement speed. This situation can be improved by increasing d in Algorithm 5.1, or by decreasing the upper bound of the rate that Flocks look for better neighbours. Both options would, however, also increase the required bandwidth. Ultimately, Flocks offer a trade-off between how much bandwidth they require for maintenance, and how quickly the overlay stabilizes, that can be adjusted by the aforementioned input parameters.

After failure and during churn, the lack of adaptability of the overlay without ATP becomes evident. The overlay with ATP maintains a higher QoS than the non-adaptive network. In addition, nodes find replacement neighbours more quickly. As the network has fewer nodes, and the number of neighbours each Flock node maintains is constant, the homogeneity must decrease. This happens quickly for the overlay with ATP, and at a much slower rate for the overlay without ATP, due to its non-adaptive nature. The next section analyses the performance of Flocks if the number of neighbours is *bounded*, but no longer constant.

5.2.4 Adaptive Interests with ATP

If nodes are unlikely to relay content of a different (content) class, maintaining connections to such neighbours does not provide any additional benefit beyond allowing newly arriving nodes to find their "own" cluster. In this case, Flocks could safely drop connections to nodes of a different content class by setting their interest to zero.

In a first simulation, I changed the interest so that nodes would only accept neighbours of their own class. However, this strategy makes it difficult to cluster the network: Assume all (random) neighbours that a node obtained from the bootstrapping service were of a different class. In this case, it would be unable to find nodes of its own class. This is because their neighbours would only have active connections to their own classes as well, and could not convey the existence of any other class to that node.

Thus, a different approach has to be followed: Nodes that find that their local neighbourhood is sparse or contains many nodes of a different class will attempt to

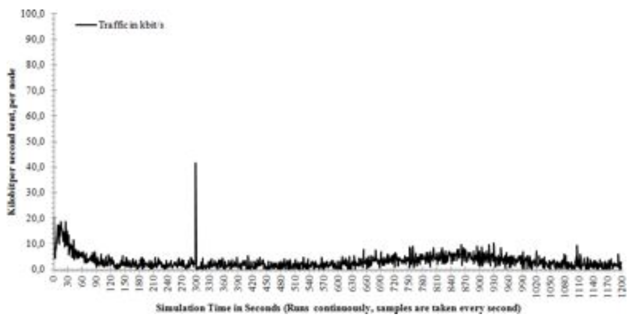


Figure 5.18: Management traffic of the Flocks overlay using adaptive interests and ATP.

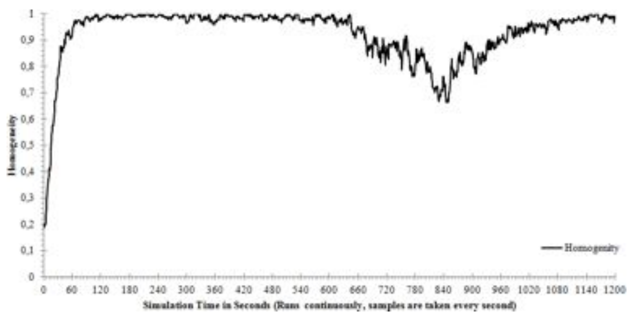


Figure 5.19: Homogeneity of the Flocks overlay using adaptive interests and ATP.

"hold the overlay together" and keep connections to nodes of a different class active. Only once their local neighbourhood is well connected with nodes of the same class, will they begin losing interest in nodes of a different class, causing the overlay to be partitioned into separate clusters.

One of the main strengths of the Flocks overlay is its flexibility. Interests are defined and evaluated locally, and every node can change them at any time without consuming bandwidth. I used this flexibility in order to make interests dynamic: Initially, interests would be the same as described in Section 5.1.4. However, as nodes obtain more neighbours of the same class, they become less interested in maintaining nodes of a different class as neighbours: Once at least half neighbours are of their own class, their interest in other classes drops to zero. This is accomplished by setting ϵ to zero.

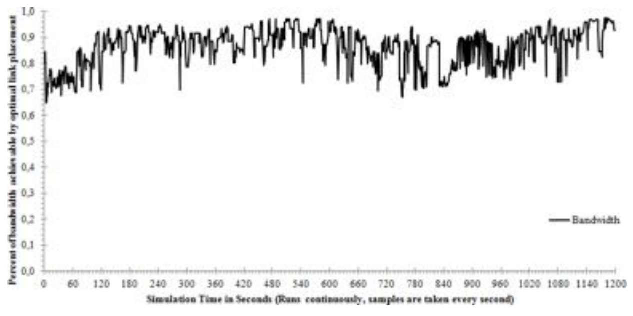


Figure 5.20: QoS awareness of the Flocks overlay using adaptive interests and ATP.

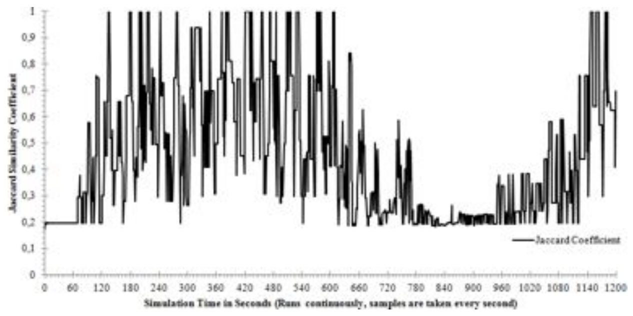


Figure 5.21: Jaccard similarity coefficient of the Flocks overlay using adaptive interests and ATP.

Performance Evaluation

The graphs for the adjusted interests can be seen in Figures 5.18 (management traffic), 5.19 (homogeneity), 5.20 (QoS awareness), 5.21 (Jaccard coefficient). Notice that the homogeneity of the initial random neighbour assignment from the bootstrapping service is very low ($H = 0.188$): the network is initially "reserve-clustered", i.e. nodes of the same class are far apart. Figure 5.19 shows that the first initial link changes made by the Flocks overlay improve the homogeneity quickly, turning the reverse-clustered overlay into a strongly clustered overlay ($H = 0.926$) within 50 seconds. During that time, the management traffic averages $10.5kbit/s$ as large changes are made to the overlay. Figure 5.20 shows that the QoS awareness does not make significant progress during that time: This is expected, as the interests described earlier prioritize homogeneity over QoS. Once the homogeneity improvements start to slow down (as a high homogeneity has been reached), the QoS awareness begins to increase, reaching 95% of the bandwidth that would be available by optimal link placement. Starting 135 seconds into the simulation, the overlay reaches a perfect homogeneity of $H = 1.0$.

At this point, only minor changes are made to the overlay, and the management traffic slows down, averaging $2.5 kbit/s$ from 50 seconds to 300 seconds into the simulation. The traffic increases once again at the time 40% of the nodes fail, but stabilizes again quickly. During the churn phase, the traffic averages $3.9 kbit/s$, as nodes repair the mesh.

Figure 5.21 displays the Jaccard similarity coefficient computed throughout the simulation. An important observation is that during the times that homogeneity reaches its maximum, the Jaccard coefficient also reaches it's maximum value of $J = 1.0$. This shows that when the overlay is partitioned, no two partitions of the same content class exist. Recall that the simulations use five content classes: A Jaccard similarity coefficient of $J = 1.0$ means that exactly five partitions exist, and that for each content class, all nodes are found in a single partition.

Deviations from Optimality

Notice that neither the homogeneity, nor the Jaccard similarity coefficient stay at their maximum of 1.0. The reason is that connections are initiated by Flocks to their previous neighbours, to check whether their properties (and thus the interest in them) have changed since they last contacted them. This also explains the large drops in the Jaccard similarity coefficient: as two nodes from different clusters connect to exchange their properties, their partitions are joined, which is reflected in the Jaccard similarity coefficient. These connects are few, and short-lived, as evident in Figure 5.18 from the low bandwidth consumed by management traffic during that time. As soon as the Flock nodes realize that their previous neighbours still are not interesting to them, they drop the connection again.

Churn and Recovery

During the churn phase, Figure 5.19 shows that the homogeneity starts to drop, as new nodes enter at random positions of the overlay. Recall that the interests we described in Section 5.2.4 assign a non-zero interest to nodes of a different class if the node evaluating them observe that their local neighbourhood has many nodes of a different

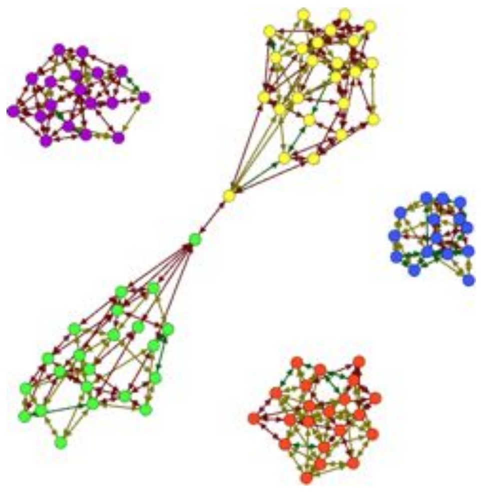


Figure 5.22: Network graph after 299 seconds.

class. As the homogeneity drops, Flocks fall back and start to reconnect. This allows the overlay to improve its homogeneity. After 300 seconds, the churn stops and Flocks quickly recover, and partition the network again once they approach a homogeneity of $H = 1.0$. At the end of the simulation, both the Jaccard similarity coefficient and the homogeneity have reached their maximum values, except for the periodic short-lived connections between Flocks and their previous neighbours explained in the previous section.

Network Graphs

Recall the Figures 5.2 and 5.3 from Section 5.1.4, both of which were taken from this experiment. In Figure 5.2, no optimization has been performed yet.

Figure 5.3 shows that Flocks have strongly clustered the network, leaving only few links between nodes of different classes. At this point, the homogeneity H is 0.9103, whereas the QoS has not been optimized yet (explained in Section 5.2.4). The available bandwidth is 73.03% of the bandwidth achievable by optimal link placement. Figure 5.22 shows the network shortly before the failure that occurs at the 300th second of the simulation. The overlay is clustered and partitioned; one node just initiated a short-lived connection to a node in another cluster, to re-evaluate its interest in it (as explained in Section 5.2.4).

The bandwidth available to the nodes in this final configuration is 91.86% of the bandwidth achievable by optimal link placement. This can be observed in the graph: most links of the same bandwidth are clustered together. Recall that none of the node is aware of its bandwidth: Each node can only measure their bottleneck bandwidth to other nodes by connecting to them, which significantly delays finding the optimum. A possible way to reduce this delay is using reasonable default assumptions and bandwidth-caching.

5.2.5 Changing Interests

In many situations, the interest of a node may change - this might happen in response to an event (something interesting happened, such as the first runner is approaching the goal in the IronMan competition), but might also occur randomly (a visitor, who was watching the cyclists during the IronMan competition, is suddenly curious at what stage of the competition a friend of theirs is, of whom they know the shirt number). In either case, once a node's interest changes, its current neighbours may no longer be good matches, and it is important that the overlay responds quickly.

Individual nodes changing their interests functions identical to the churn phase, that is already evaluated: Since an incoming node can join the overlay at any point, it makes no difference from the overlay's point of view whether the node just joined, or whether it changed its interest.

I evaluated how well the Flocks overlay responds to *large-scale* changes in interest by performing simulations in which a captivating event causes 40% of the nodes to switch their interests instantaneously. For better visualization, refer to Figure 5.23. Figure 5.23a shows the overlay immediately before the interest change. Based on their content, the nodes have grouped into five distinct clusters. At the 300 second mark, 200 nodes change their interest from whichever content class they were before, to content class 2. If the node was already of content class 2, it changed its class to 3. As with failing nodes, the identifiers of the nodes that will change their interest were initially selected randomly, saved, and then remained the same across different simulations for reproducibility. Figure 5.23b displays the overlay immediately after the interest change. Notice that the homogeneity of this overlay is poor, as nodes of class 2 pollute every cluster. Furthermore, two roughly identically sized clusters exist

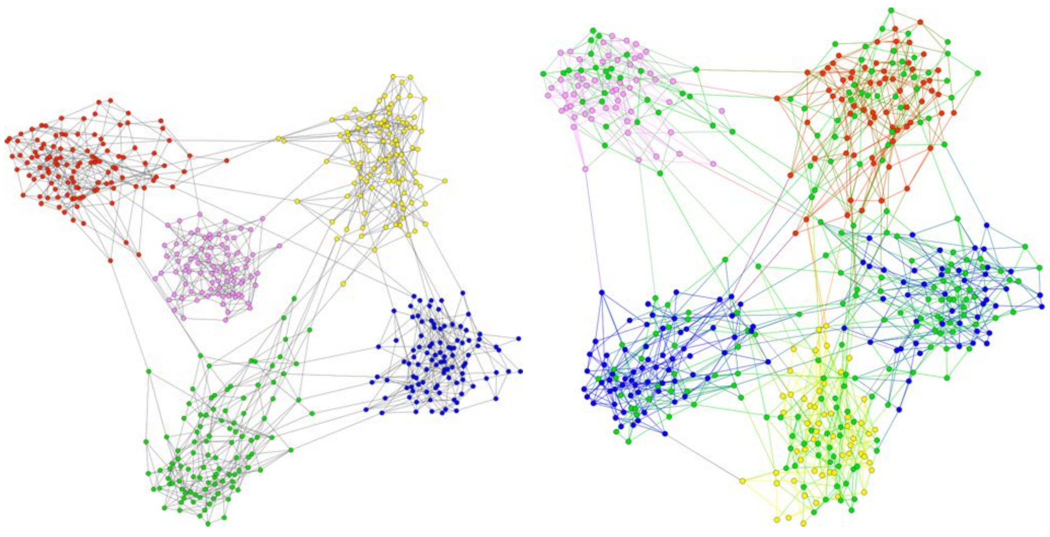

(a) Overlay after 295 seconds. (b) Overlay directly after 300 seconds.

Figure 5.23: The effect of 40% (200 out of 500) nodes changing the interest instantaneously.

at the bottom left and the bottom right that consist of both, nodes of class 2 and nodes of class 3.

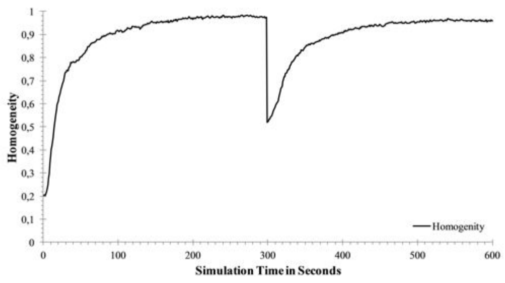
Figure 5.24a presents the homogeneity during the simulation. Even though the homogeneity drops sharply after 300 seconds, due to the large change in interest, it can be seen that the Flock nodes quickly recover from it, making large improvements in homogeneity within the first minute. Indeed, after 75 seconds, Figure 5.25 shows that the overlay has recovered, and all nodes are grouped together in clusters once again. During that time, the traffic overhead produced by the overlay, shown in Figure 5.24b increases modestly, to an average of $7.5kBit/s$.

The remaining improvement in homogeneity that can be seen in Figure 5.24a after the 375 second mark result from nodes strengthening their connections to their own class, and removing connections from nodes with a different class.

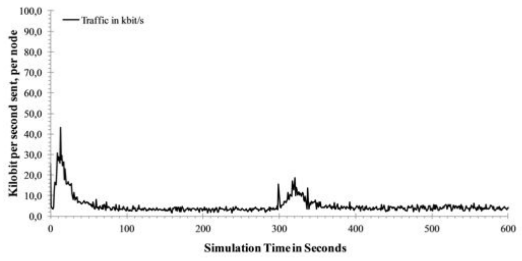
5.2.6 Evaluation on the PlanetLab

After evaluating the effect of the different input parameters and timing profiles through simulations, I conducted additional experiments on the PlanetLab. This was done by removing the network layer of the simulator once again.

Doing so served several purposes: First of all, a common concern was that Flocks



(a) Homogeneity



(b) Overlay after 300 seconds.

Figure 5.24: The performance of a 500 node network, in which 40% of the nodes change their interest after 300 seconds.

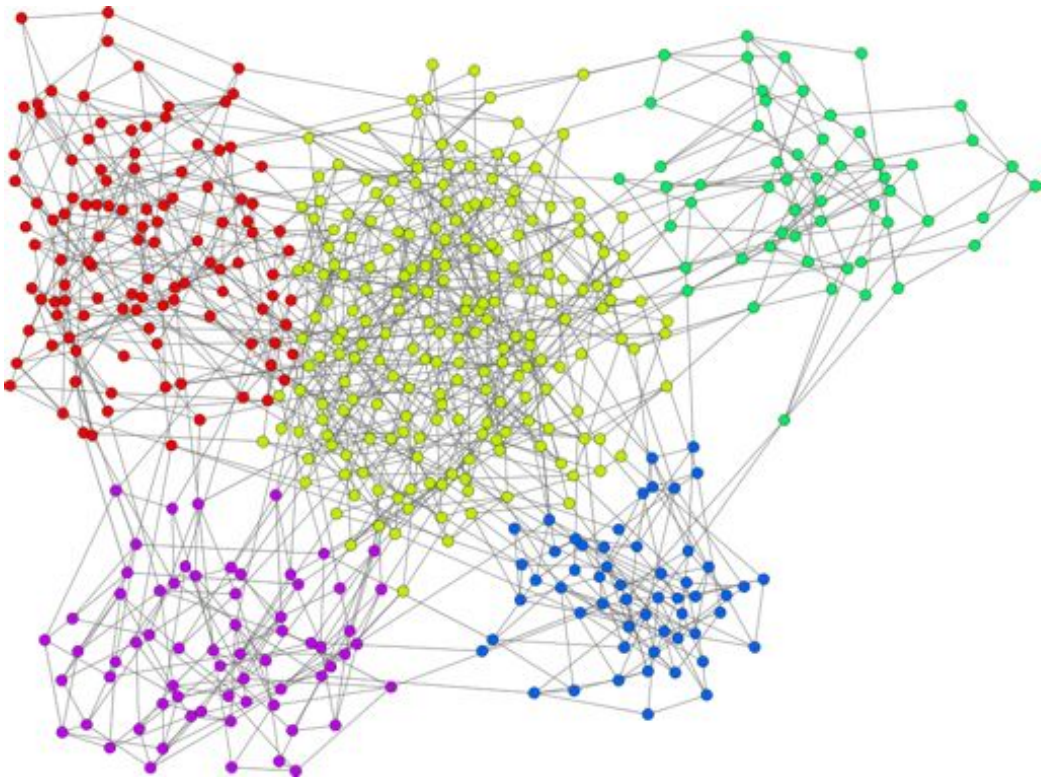


Figure 5.25: Overlay after 375 seconds.

were evaluated through simulations. The PlanetLab consists of hundreds of machines physically connected to the Internet, and allows me to show that Flocks work not only in a simulated environment, but also in practice under "real" Internet conditions. Second, it proves that no information is "leaked" through the simulator: On the PlanetLab, each Flock node is an individual process, and can only communicate with other Flock nodes through socket connections. Finally, it demonstrates that Flock nodes obtain QoS related information themselves, through active and passive measurements, and do not simply rely on pre-measured values or a default.

Experiment Setup

Round-trip-delay was used as a QoS metric, because it allows, together with using sites on different locations on the globe, easy identification whether or not the overlay is QoS-aware.

The Flocks overlay was evaluated on five geographically different sites. They were selected based on their distance. My goal was to create two groups of sites that are distant, and have two to three distinct locations within each group that are relatively close. If the overlay is QoS aware, one would expect that (i) two distinct clusters are formed (one for each distant group), and that (ii) nodes within each cluster are stronger connected, but a distinction between the individual locations can still be observed. In other words, an overlay in which nodes from Asia and America are forming a single, strongly interconnected cluster of nodes, is likely not QoS-aware. Even though nodes that are geographically close should be attracted to each other (forming a cluster of nodes), it should still be possible to tell individual cities or states apart (forming subclusters of nodes).

Each site hosted between 10 and 90 nodes, depending on their capacity: Some sites were overloaded, or had strict bandwidth limits in place. They were included on purpose to evaluate how Flocks perform under poor conditions. In order to avoid contributing excessively to the load on these nodes, I only deployed a limited number of Flock nodes to these sites. Table 5.5 lists the sites that were used during the experiments, the nodes they used, and remarks, such as an unusually high load average[12].

[12]The load average, which indicates how many tasks are runnable during the experiments. If the load average exceeds the number of central processing units (CPUs), tasks are queued and not

Figure 5.26 shows the locations of the sites on Earth.

Graphs in this Section

This section presents two types of graphs: The first type of graph is coloured by content class (nodes of the same colour have the same content class), and is used to evaluate how content-aware the overlay is. The second type of graph is coloured by site (nodes of the same colour reside on the same site), based on the colours shown in Table 5.5 and Figure 5.26. In both graphs, the thickness of the arrows is used to give an idea of how much interest nodes have in each other. Thin lines signify that neither node is much interested in the other node, and will likely replace its neighbour with a different one when they get the chance. Thick lines signify that at least one node has a high interest in the other neighbour. While the thickness may imply that interest is symmetric, it is not the case. In fact, this is caused by a limitation of this particular representation in graphing tool, which displays the higher interest of the two nodes. While the graphing tool allows me to display an asymmetric interest by colour-coding the arrows, the graph becomes overloaded and difficult to evaluate visually. As the exact interest does not carry a lot of meaning, I opted for a more simple and visually pleasing representation.

Institute	Country	AS	Nodes	Remarks
🟥 Nagoya Institute of Technology	Japan	2907	90	[13]
🟧 DePaul University	United States, IL	20130	10	[14]
🟩 National University of Singapore	Singapore	7472	20	[15]
🟦 Hong Kong Polytechnic University	Hong Kong	4616	90	-
🟪 Worcester Polytechnic Institute	United States, MA	10326	90	-

Table 5.5: The sites used for the PlanetLab experiment.

running[120]. For example, on a quad-core system, the load average should remain below 4.00. The load average measurement was taken just before the experiments ran, to ensure it is not distorted by running the Flocks overlay. Therefore, it consists of the average from the five minutes before the experiment.

[13] high performance, 6.44 load average on a 16-core system

[14] heavily overloaded, 13.17 load average on a quad-core system

[15] somewhat overloaded, 24.90 load average on a 16-core system, traffic shaping

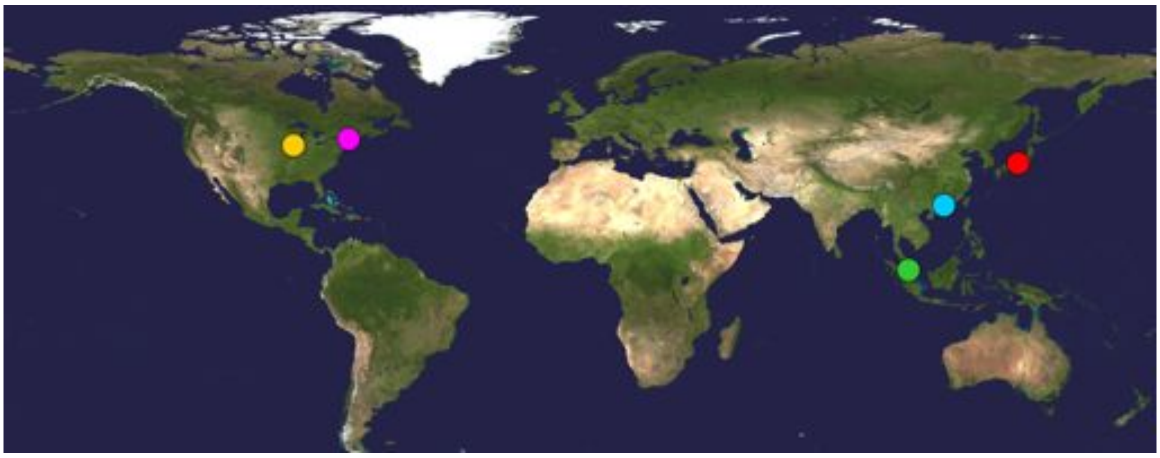


Figure 5.26: An image of Earth reproduced from [6]. The coloured marks were added to illustrate the locations of the sites used for the PlanetLab experiment.

Bootstrapping

An application was written that orchestrates the launch of all Flock nodes on a site. Each Flock node first connects to a centralized registration service for bootstrapping (Section 5.1.2), which provides it with four random identifiers of Flocks nodes that already joined the overlay. This functionality only exists for initial bootstrapping - a node is not able to obtain any more addresses from the registration service, and must rely on the Flocks protocol instead. Furthermore, there is no guarantee that a node is provided with matching neighbours, or that the nodes will even accept it as a neighbour, as the identifiers are chosen by random.

Graph Generation

Each node maintains the connection to the registration service, in order to facilitate creating the overlay graphs shown in this section. The process that creates the overlay graphs is run twenty seconds after the experiment starts, and every thirty seconds after that. It is briefly outlined here: The registration service sends a packet to each node, indicating that it is creating an overlay graph. Each node responds with a list of its own properties, as well as a list of its neighbours and their properties (this includes properties with a remaining TTL of one, such as virtual properties). Once the registration service has received a response of every node, it builds a map of the overlay and saves it for further analysis.

It should be noted that other than the initial four identifiers used for bootstrapping, Flock nodes are not given any outside information. In particular, they have no notion of what a site is, nor do they know whether or not a neighbour is in the same (or a geographically close) location[16].

Experiment Results

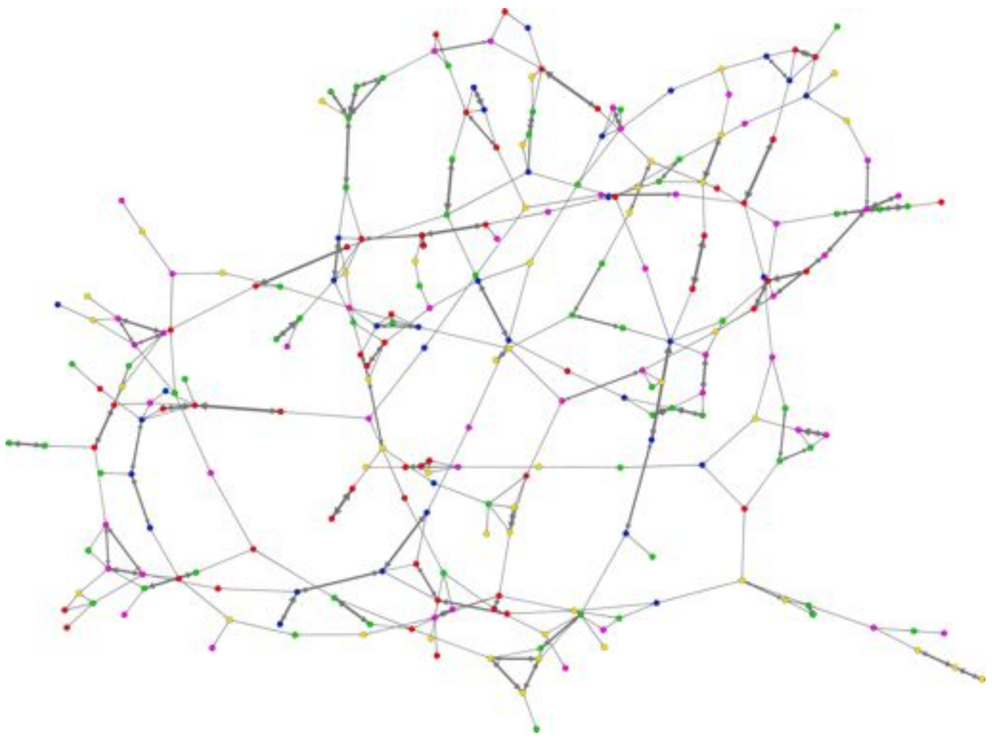
Figure 5.27a shows the overlay graph after 20 seconds, using ATP, colour-coded by content class. This means that Flock nodes only had time to perform very few optimization. Observe that the overlay is loosely connected and largely disorganized. While small groups of neighbours with the same content class exist, they are spread across the entire overlay, and may well be by chance, caused by the random identifiers initially returned by the registration service. A similar situation is evident concerning the QoS-awareness, in Figure 5.27b. Again, some nodes of the same site are grouped together, but no clear clustering, in which nodes from the same site form a cluster, can be observed.

Figure 5.28a shows the same overlay, after 80 seconds. Notice that in only sixty seconds, the homogeneity has greatly improved, and most nodes of the same class are now clustered together. A few nodes - such as the blue node in the middle right, or the yellow node at the bottom right, are clearly in the wrong place of the overlay. The QoS-awareness has improved slightly, and more nodes in the same site have started to group together. However, a clear clustering still cannot be observed.

Figure 5.29a shows the overlay 30 seconds later (110 seconds after the experiment started). At this point, all nodes of the same class are clustered together, and only few connections exist between clusters: These are temporary connections to exchange properties and neighbours, with low interest, as evident by the thin edges that represent them. The QoS-awareness, shown in Figure 5.29b, has improved as well. Many nodes of the same site are now grouped together, such as the clusters of nodes from Massachusetts (seen in the top and bottom left, as well as the middle right). Other nodes are not separated from each other as clearly for two main reasons:

- The nodes from sites located throughout Asia have similar QoS. They separate

[16]A node obviously knows the IP addresses of their neighbours, however it does not use them to infer their locations


(a) Content-awareness. Colours represent content classes.

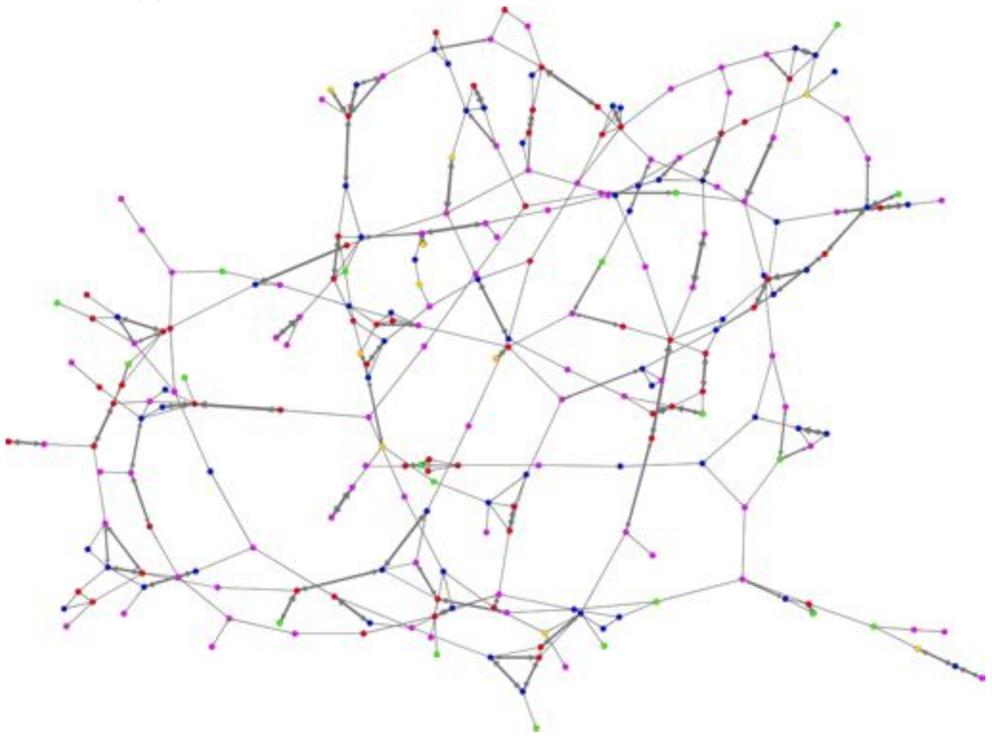
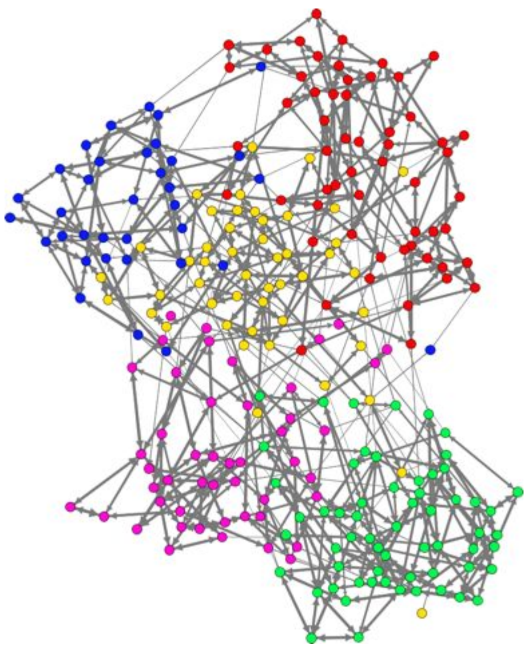

(b) QoS-awareness. Colours represent sites.

Figure 5.27: The first Flocks overlay on the PlanetLab, after 20 seconds.


(a) Content-awareness (colours represent the content class a node has).

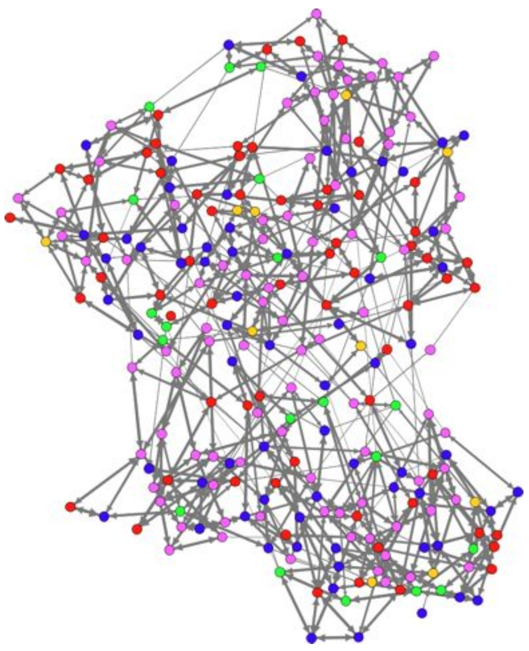
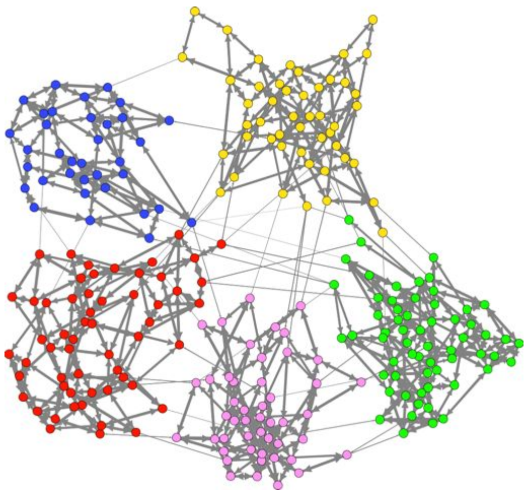

(b) QoS-awareness (colours represent sites nodes reside in).

Figure 5.28: The overlay after 80 seconds.


(a) Content-awareness (colours represent the content class a node has).

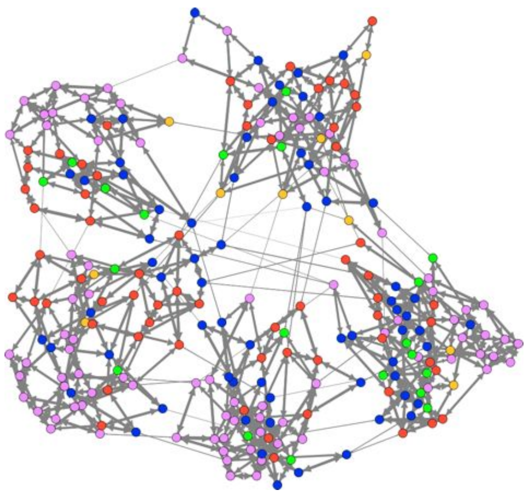

(b) QoS-awareness (colours represent sites nodes reside in).

Figure 5.29: The overlay after 110 seconds.

themselves from the nodes in Massachusetts, to which they have a higher delay. Recall that nodes have no knowledge whether they are located in the same site or not, and they also do not know the number of sites that exist. It would be possible that they were all located in a single site with similar QoS, in which case no clustering should be expected.

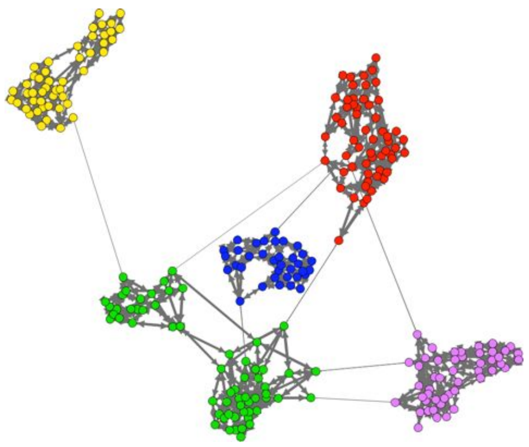
- The nodes residing on the overloaded site in Illinois run slowly, which increases their processing delay, and therefore also the round-trip time they measure. Therefore, while they measure a difference in QoS between themselves, the nodes in Massachusetts and the nodes in Asia, it is much less pronounced, as the majority of the delay comes from their processing delay.

At this point, the homogeneity of the overlay did not decrease any more, and the during the remainder of the execution, the QoS-awareness improved. Figure 5.30b shows the overlay 440 seconds after the experiment started. The links were colour-coded to correspond to the delay measured between both nodes; a red link corresponds to a delay of 575ms, the maximum delay present in this graph. A colour closer to green indicates a lower delay. Notice that in addition to the nodes in Massachusetts, the nodes in Asia also begin to exhibit signs of clustering based on their location. Furthermore, the nodes within each cluster experience a high LOS. Figure 5.30a displays the homogeneity, for completeness.

As observed in graphs constructed later in the experiment, the clustering continues to improve. Figure 5.31a shows the homogeneity, and Figure 5.31b the QoS-awareness 590 seconds after the experiment started. It is clear that the Flock nodes have created an overlay that is content- and QoS-aware. Each content class is represented by its own cluster, and within each cluster, the overlay is organized so that delay is minimal. Recall that the overlay was created without any outside knowledge, except for the four random identifiers that were given to each Flock node during bootstrapping.

Nodes with Shared Interest

In many situations, visitors will be interested not just in one, but multiple content classes. For example, if Tennis and Bowling matches are fought at the same time, a visitor may be interested in both matches. I modelled this use-case on the PlanetLab,


(a) Content-awareness (colours represent the content class a node has).

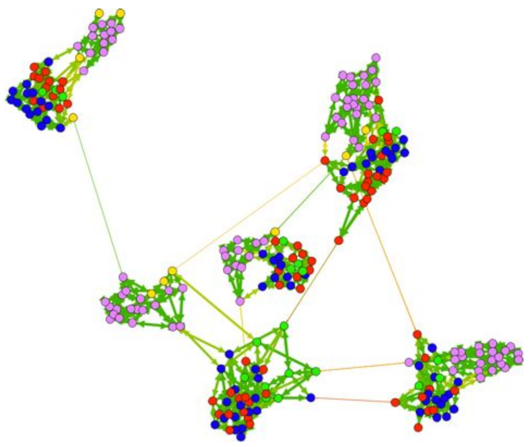
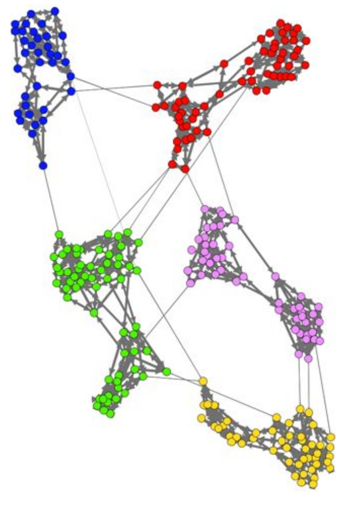

(b) QoS-awareness (colours represent sites nodes reside in).

Figure 5.30: The overlay after 440 seconds.


(a) Content-awareness (colours represent the content class a node has).

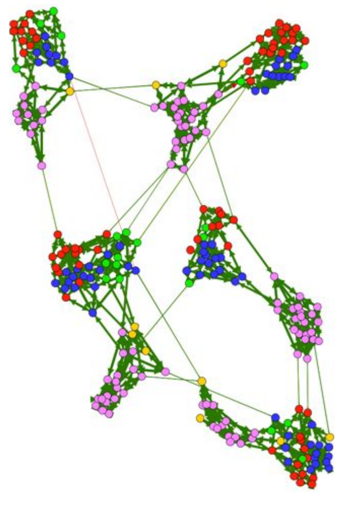

(b) QoS-awareness (colours represent sites nodes reside in).

Figure 5.31: The overlay after 590 seconds.

by assuming that nodes of the content class 3 are interested with the classes 2 and 4 as well. This was done by changing the interest of nodes with the content class 3, as shown in Equation (5.5).

$$\begin{aligned}\Big[\big[\big(Match(''class'',3,1.0),1.0\big) &\vee Match(''class'',2,1.0),0.4\big) \\ &\vee Match(''class'',4,1.0),0.4\big),1.0\big] \\ &\wedge \big[\big(Match(''qos.bandwidth.up'',MAX,0.0),1.0\big),1.0\big]\Big]\end{aligned} \tag{5.5}$$

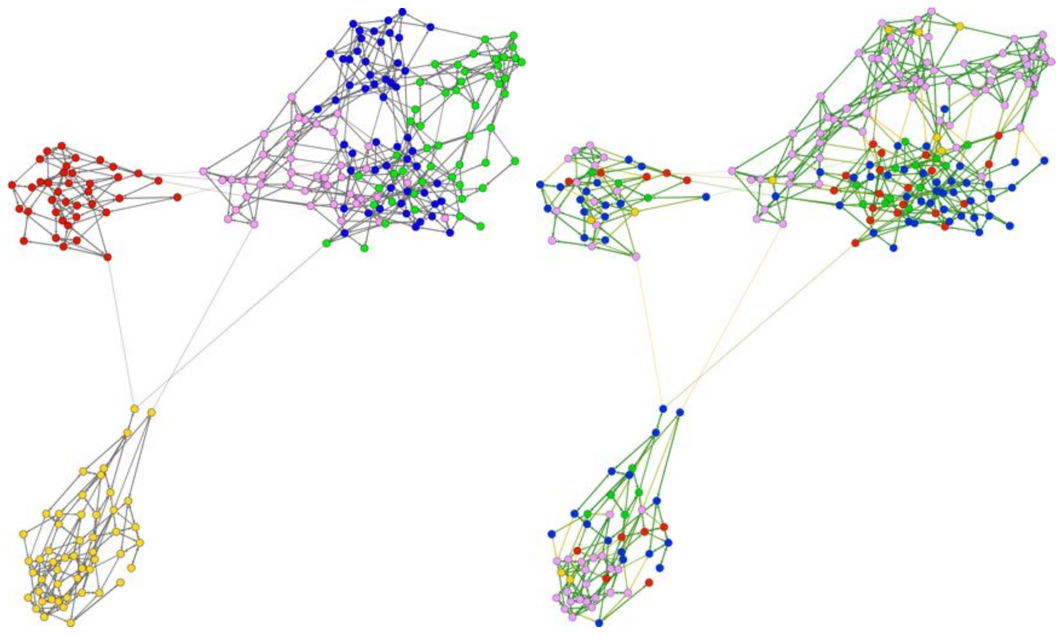
With this interest in place, nodes of class 3 are interested in neighbours that have class 2 or 4 as well. The weights of the interests in class 2 and 4 are both set to 0.4. Therefore, nodes are more interested in neighbours of their own class, 3, which (due to their interest) are likely to have content from nodes 2 and 4 as well. The interest in a neighbour that has class 2 and class 4 is higher than in a neighbour that only has one single class. Finally, as the weights of both other classes are less than 0.5, a node with class 3 will be preferred over a node with the two classes 2 and 4[17]. Table 5.6 summarizes relevant combinations of classes, and the interest each combination results in. The interests of the nodes with the classes 2 and 4 are changed as well: They simply regard nodes of class 3 as a node with the same class as their own. This is because a node of class 3 is able to relay for both content classes, as they receive them from nodes of class 2 and 4.

Class 3	Class 2	Class 4	Interest
Yes	No	No	$0.55\dot{5}$
No	Yes	No	$0.22\dot{2}$
No	No	Yes	$0.22\dot{2}$
No	Yes	Yes	$0.44\dot{4}$

Table 5.6: The interest of a node with class 3 in another candidate node, dependent of what classes that candidate node has.

During the preparations for the experiments, I noticed that several of the sites I used before have become heavily loaded. Therefore, I had to reduce the number of

[17]This can be changed by increasing the weights for classes 2 and 4 from 0.4 to a value greater than 0.5.



(a) Content-awareness (colours represent the content class a node has).

(b) QoS-awareness (colours represent sites nodes reside in).

Figure 5.32: The second Flocks overlay on the PlanetLab, after 170 seconds.

nodes deployed on them. Table 5.7 shows the adjusted number of nodes used in the experiment.

Institute	Country	Nodes
Nagoya Institute of Technology	Japan	30
DePaul University	United States, IL	10
National University of Singapore	Singapore	20
Hong Kong Polytechnic University	Hong Kong	105
Worcester Polytechnic Institute	United States, MA	105

Table 5.7: The new number of nodes deployed on the PlanetLab sites used in the second experiment.

Figure 5.32 shows the overlay after 170 seconds. Refer to Figure 5.32a for content-awareness: Class 2 is shown in green 🟩, class 3 in blue 🟦, and class 4 in pink 🟪. Notice that nodes from class 3 have formed a cluster, and placed themselves between the clusters of nodes with class 2 and nodes with class 4. Furthermore, it can be seen that clustering based on QoS within each class has started to take place: The nodes of class 3 have clearly separated into two clusters that already be visually identified after 170 seconds. The assumption that these two clusters represent sites in Asia and the United States is confirmed in Figure 5.32b. Finally, observe that the LoS received by the majority of nodes is very high, and that the few links with poor QoS are low-interest links between QoS clusters of different content-classes, that are created only to exchange properties and are short-lived.

An observation that resulted from this experiment was that the interests were poorly chosen. In fact, I picked the weights so that nodes of class 3 are still most interested in other nodes of class 3, with the expectation that they will form a distinct cluster in which they can exchange content from both other classes with each other. However, this worked "too well": Nodes of class 3 connected primarily to each other, and only maintained few connections to nodes of class 2 or 4. Figure 5.33, which shows the overlay after 620 seconds, highlights a potential problem: The top cluster of the class 3 nodes only maintains a single connection (highlighted) to a node with class 2. If either one of the nodes connecting the two clusters fails, the remaining nodes in cluster 3 can only obtain the content through a slow inter-pacific connection from the lower cluster.

Therefore, a different approach must be pursued. Conditional interests present the most straight forward solution: A node of class 3 is more interested in nodes of class 2 or 4 than its own, as long as it does not have any neighbour of that class. Once a node has a neighbour of class 2 (or 4), it is less interested in obtaining another neighbour of that class, than it is in obtaining a neighbour of its own class.

Recall from Section 4.1.4 that conditional interests cannot be represented by the default oracle. Therefore, I implemented a custom oracle as shown in pseudo code in Algorithm 5.2 (ϵ is a very small, non-zero value used to prevent partitioning due to a zero-interest).

Ten additional nodes were deployed to the site in Singapore, as its load was very low. Figure 5.34 shows the new overlay after 650 seconds. Observe in in Figure 5.34a that nodes of class 3 remain much stronger connected to nodes with the classes 2 and 4, even after many optimizations. They still build their own cluster, however most nodes have a connection to a node with class 2 or 4, or even both of them. Furthermore, a desirable outcome is that nodes of class 2 and 4 still connect primarily to nodes of their own class. Finally, Figure 5.34b shows that the overlay clustered itself based

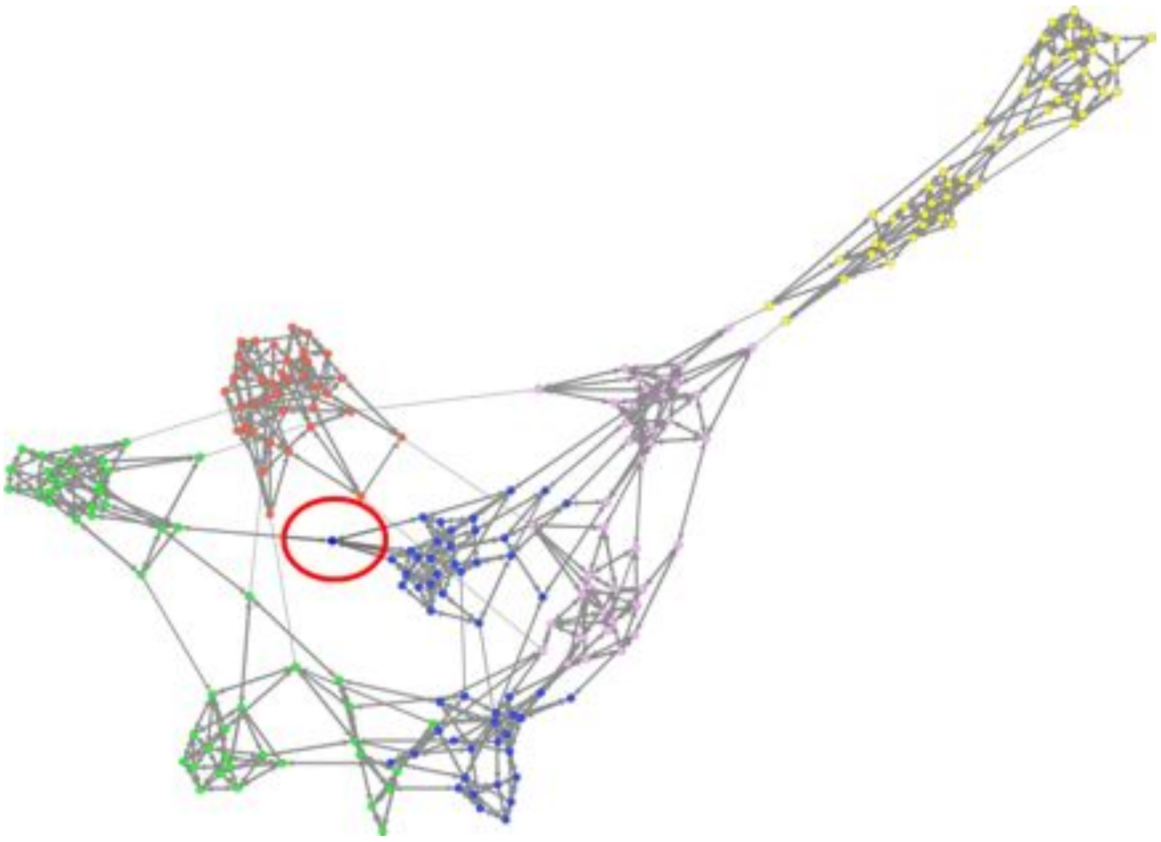


Figure 5.33: The second Flocks overlay on the PlanetLab, after 620 seconds. Receiving content from class 2 hinges on a single connection (highlighted) to the top cluster of class 3.

on the locations of the different sites. The distinction between the Asian sites, and the sites from the United States is already clearly visible, and clustering within each of the clusters has begun as well. In fact, only a single node (the yellow node at the bottom) has not found its correct cluster yet. Finally, note that the LoS within each cluster is very high[18], and that all links with high delay are confined to inter-cluster connections.

5.3 Discussion

The simulations show that the Flocks protocol quickly builds overlays that are both content- and QoS-aware, while only producing a modest amount of management

[18]A red link in Figure 5.34b has again the maximum delay (300ms in this experiment), whereas green links have a delay less than 10ms.

Algorithm 5.2 The interest oracle used by nodes of class 3 in the third experiment.

```
procedure CalculateInterest(candidate_properties, connected)
    if $candidate\_properties_{class} = 2$ then
        if $other\_neighbours\_with\_class\_2 = 0$ then
            $interest \leftarrow 1.00$
        else
            $interest \leftarrow 0.50$
        end if
    else if $candidate\_properties_{class} = 4$ then
        if $other\_neighbours\_with\_class\_4 = 0$ then
            $interest \leftarrow 1.00$
        else
            $interest \leftarrow 0.50$
        end if
    else if $candidate\_properties_{class} = 3$ then
        $interest \leftarrow 0.75$
    else
        $interest \leftarrow \epsilon$
    end if
    return $interest$
end procedure
```

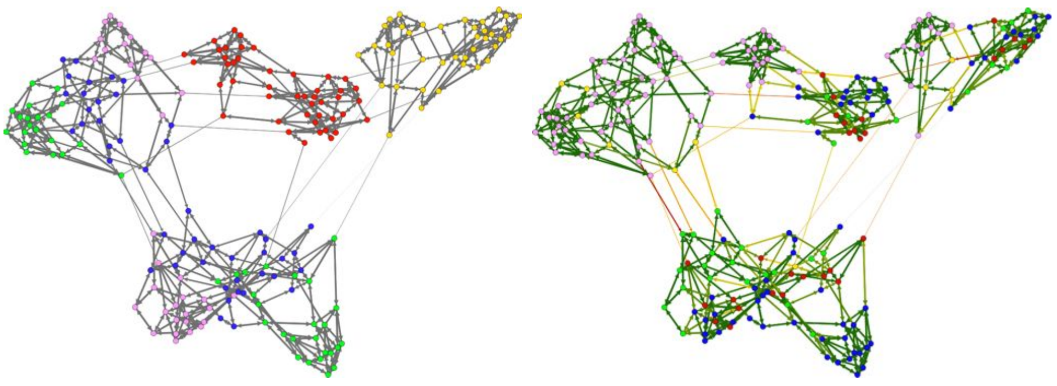

(a) Content-awareness (colours represent the content class a node has).
(b) QoS-awareness (colours represent sites nodes reside in).

Figure 5.34: The third overlay on the PlanetLab, after 650 seconds.

traffic. Even when each node is limited to only investigating one candidate every six seconds, the overlay reaches a homogeneity of greater than 0.9 and a LoS of 94% of the optimum within 90 seconds. Furthermore, the overlay continues to improve, and remains responsive to churn and failure.

Evaluations on the PlanetLab, under “real Internet conditions”, have shown that Flocks also work well in practice. When placed on five different sites, with five different content classes, Flock nodes quickly established clusters based on their content. They correctly identified different locations based on QoS measurements, and build sub-clusters based on that in a completely self-organized way, without any global view or any external information.

During the experiments, I identified a major limitation: Flocks work well in situations where nodes flock together to exchange content. In this case, it is important that the LoS to the neighbouring nodes is as high as possible. However, in many situations, nodes may be interested in QoS to a specific node.

For example, assume once again the IronMan sports event, in which a groups of camera nodes transmit video streams. Nodes connect to the camera nodes to receive the stream, and they also relay the stream to any of their neighbours. Assume that a new node, which is interested in certain camera nodes, joins. However, none of the camera nodes can accept it, and therefore it must choose one of the relaying nodes that are connected to the camera nodes (or another relaying node). In this case, the LoS to the relaying node is not as important as the LoS to the camera node, that it receives from the relaying node. In other words, a relaying node to which I measure the lowest delay may still be a bad choice, if it has a much higher delay to the camera node than any other relaying node.

A potential solution is to propagate QoS related properties: For example, a node (“the relaying node”) can measure the delay to the camera node it is connected to, and then propagate it to its neighbours. Each of its neighbours then adds the delay they measured to the relaying node, and now knows the delay to the camera node they will experience, if they use the relaying node. However, this approach essentially requires that QoS related properties are flooded throughout the network, which does not scale to large numbers of nodes.

CHAPTER 6 Scalable QoS Estimation

QoS estimation not only has many benefits, it is outright required to model several common scenarios. Consider, for example, an overlay in which nodes relay time-critical data from a source node. In such an overlay, nodes are not only interested in the LoS to its neighbour, but also in the LoS that neighbour provides to the data source. A commonly mentioned anecdote is the live transmission of a soccer game. If the delay to the source is too high, one may well hear an excited "Goal!" from next door, several seconds before actually seeing the goal themselves.

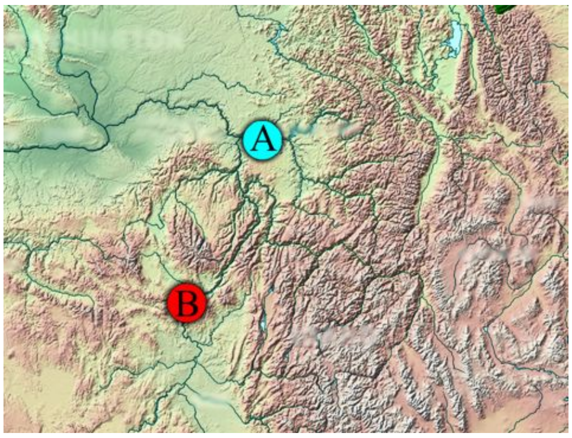
Unfortunately, performing scalable QoS prediction in large overlay networks is a difficult problem [107]. In networks with thousands of nodes, maintaining a global view of the topology is no longer an option. Nodes might have tight memory constraints and are likely unable to store the full topology. In addition, as the likelihood for a LoS change increases with the number of nodes, it is not viable to maintain an up-to-date view of the topology either.

6.1 Introduction to QoS Maps

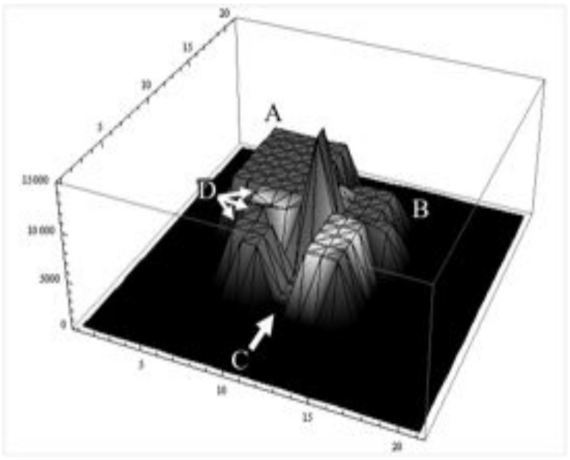
A possible solution that allows scalable estimation of the LoS experienced by any node in the overlay is introduced in this chapter. The general idea is to provide applications in existing large overlays with "QoS maps" - a high level overview of the LoS that it will experience by routing into certain regions of the overlay.

6.1.1 Concept and Definition

The concept, as well as the power of QoS maps is best described through an informal example. Imagine a map from any common atlas. Typically, this map will have labels, to indicate points of interest (such as villages and cities), as well as topographic



(a) A traditional map, displaying topological information, available from [123].



(b) Visualization of the bottleneck bandwidth QoS map generated from an overlay. Several regions offering key insights are highlighted.

Figure 6.1: Similarities of traditional and QoS maps

information that indicate mountains and valleys. A traveller who seeks to venture to a point of interest that lies behind a mountain, can plot their route in several ways: They can take the shortest route, crossing the mountain and taking the effort from climbing that mountain into account. Alternatively, they can plan a route around the mountain, resulting in a longer, but less exerting travel.

A QoS map is somewhat similar: The labels that indicate points of interests in traditional maps are, instead, identifiers of nodes. The topographic information, such as mountains or valleys, describes regions of low and high QoS instead. Finally, the traveller that has to decide between going over a steep mountain, or around it, is a node that can cross a region of low QoS in few hops, or bypass it by choosing a longer route that avoids the low QoS region altogether. Choosing a route locally is straight forward, if such a map is available, as many search algorithms such as the shortest path of Dijkstra [121] or A* [122] can be used to find the best compromise to the target region.

Figure 6.1 offers a visual comparison of the two types of maps. Pretend that in the traditional map, shown in Figure 6.1a, one wishes to travel from point A to B. It is clear that following the river can be expected to be much less exerting than taking the direct route crossing the mountains.

The QoS map, shown in Figure 6.1b, provides a similar insight. Consider that the node generating the QoS map (the "source") is located in the centre, each square represents a node, and the height indicates the best bottleneck bandwidth available by routing to that node, according to the QoS estimation algorithm. Since high bandwidth is desirable, the meaning of a "mountain" is reversed in this visualization: The higher the area, the higher the bottleneck bandwidth, and the easier this "obstacle" can be overcome.

I arranged the other nodes around the source based on distance: Nodes with a distance of one hop are directly adjacent to the source, followed by nodes with a distance of two hops, and so on. The x and the y coordinates have no real geographic meaning, only the regions themselves are important.

Note that the QoS maps themselves are internally represented as complex data structures, and this representation was only chosen for human viewing. Still, a human can draw several conclusions by simply observing this image: First, the region in the bottom is best reached directly, since routing over any other region will introduce a bottleneck. Second, routing over the region in the top (labelled A) to the region in the right (B) carries no bandwidth penalty, as the bottleneck bandwidth of region A is higher. Therefore, this is an excellent alternative route, if the region in the right is no longer reachable directly. Routing over region B to reach the region A will, however, introduce a bottleneck. Finally, the very low bandwidth region of three nodes, region C, below the centre, should be avoided.

It is also possible to make some assumptions about where the bottleneck is located. Consider, for example, region A: The entire region has an uniform height, which indicates that the bottleneck is the link to that region. Furthermore, it shows that no bottlenecks in region A are more restrictive (as this would be represented by a decline in height, as seen in region D).

6.1.2 Aggregation and Scale

Storing a detailed QoS map on every node is akin to storing a global view, and does not scale. Therefore, the information contained in such a map must be reduced. Similar to a traditional map in which (depending on scale) smaller details are omitted, a QoS map does not aim to contain comprehensive information of the overlay, but rather offers a coarse overview. This is reasonable: Smaller surface irregularities do not matter when one considers the fact that we are climbing a mountain.

Reducing such information has two key advantages: First, by removing details, while maintaining the "big picture", the size of the map is reduced. Similar to a world map, that scales to the entire earth by removing details that are no longer relevant in this scale, this allows QoS maps to scale to large overlays. Second, simplifying the topology also reduces the amount of information that needs to be advertised in case of a topology change: Minor, local changes in QoS may be absorbed by the aggregation and do not need to be advertised at all. For example, a small stone that fell into the footpath is a very minor obstacle that does not affect the quality of the footpath much. On the other hand, should a boulder fall and block the footpath entirely, so that it can no longer be crossed, this change in quality still needs to be advertised on a larger scale. The process of topology simplification is achieved through hierarchical aggregation of the overlay. The hierarchical aggregation, together with the aggregated QoS information stored throughout the network is what I refer to as a "QoS map".

6.1.3 Multi-Level Refinement

In some situations, a coarse map is not sufficient. For example, assume the traveller decided to cross the mountain, since going around takes too long. In this case, a more detailed map of the mountain is now helpful: It allows the traveller to identify a footpath much less steep than the rest, leading across the mountain. By walking on the footpath, they avoid much of the exertions from crossing the mountain, and reach the target with only small detours.

QoS maps know a similar concept, except that they "mark" coarse regions in which large differences in QoS exist. This serves as a hint to a node routing through such a region, indicating that looking at a more detailed map may be worthwhile. In the fictional example, a high mountain with a low footpath has a large difference in height. In QoS maps, a low QoS region that is joined with a high QoS region during aggregation will have be represented as a low QoS region, with a high difference in QoS. Based on this information, a node that considers choosing this region may refer to a more detailed representation of this region, and discover the high QoS region. The concept of considering a more detailed representation of aggregated information is referred to as multi-level QoS estimation.

Naturally, anyone who lives in a region knows its surrounding with great detail. This idea is also present in QoS maps: Nodes know the LoS of the network region immediately around them with great detail. This detail decreases with distance. If a node wishes to obtain more detailed information of a certain region, it therefore contacts any node from said region, and requests the necessary information from it.

QoS maps exist for many different QoS-related metrics. Each QoS-related metric, such as delay, bottleneck bandwidth, jitter and loss, has its own QoS map.

6.1.4 Discussion

While seemingly simple, QoS maps serve two important purposes: First, they allow low-cost estimates of the QoS experienced on different paths. Second, they provide applications multiple alternative paths to any target node.

Consider, for example, a video conferencing application with certain bandwidth and jitter constraints. Said application may not find any route that satisfies both QoS constraints. QoS maps provide this application with a fast, coarse-grained estimate of QoS along different paths in the overlay, which may contain routes that satisfy one constraint, but not the other. Let us assume that smoothness is more important for the application than image quality. Therefore, it decides to use a path with low jitter, even if it has low bandwidth and the resolution of the video has to be reduced. At the same time, it discards a route that would have sufficiently high bandwidth for the full resolution video, but has also high jitter. Finally, by using multi-level estimation as described earlier, the application can refine its estimate step by step if desired and increase the accuracy of the estimate.

Every node has a different view of the network, and hence a different QoS map. A single node's QoS map offers it a coarse, but low-cost QoS estimates of any path in the network. By combining the views of the nodes on a route, that aggregated information can be refined. This allows nodes to obtain more accurate information of routes at the cost of a higher computational and network cost caused by the query.

An important feature, which sets QoS maps apart from other overlays that employ a hierarchical organization (such as [47]), is that they preserve the *existing* topology of the underlying network, which is built by the Flocks protocol. In contrast to other hybrid overlays (such as [14], [124] and [58]) that build a hierarchical topology for multicast data dissemination, QoS maps can provide alternative routes and estimates for each route to applications. Topology aggregation, distributed partitioning and QoS epitomes have themselves received significant attention, and are used to maintain scalability. However, to the author's knowledge there is no existing system that combines these existing domains as the QoS maps introduced in this chapter do.

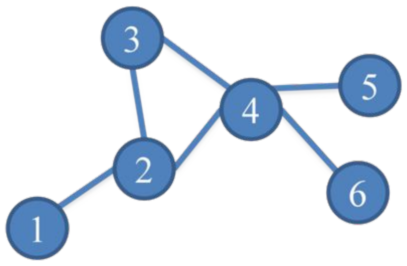
6.1.5 Contributions

The main contributions in this section are as follows: First, it explains how an existing distributed partitioning algorithm should be changed, so that it partitions any overlay into sub-networks ("clusters") that can be used for distributed hierarchical aggregation. Second, it introduces three different methods to deal with hubs - nodes with a lot of neighbours - during partitioning, and evaluates their effects on the aggregated topology. It shows that with the modifications introduced in this section, large overlays can be aggregated into low-depth hierarchies with clusters of a small, bounded size. As a result, nodes only need to store little state about the ON to allow QoS aware routing. Finally, it presents an algorithm to perform multi-level QoS estimation using the aggregated information.

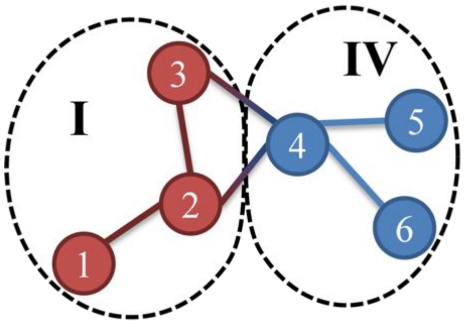
QoS maps make use of three domains: Topology aggregation [3], distributed partitioning, and QoS epitomes [106]. Each of these approaches is applied hierarchically to facilitate multi-level QoS estimation of the usual common QoS metrics. Figure 6.2 illustrates this three-step process on a small six-node overlay.

- Topology Aggregation is used to maintain scalability for large overlays by avoiding a global view. In order to perform topology aggregation, however, one must identify regions of the overlay (shown in Figure 6.2a) that should be aggregated. I determine these regions through distributed partitioning.

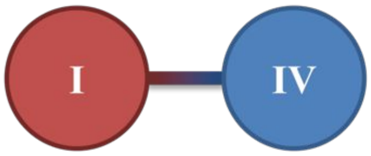
- Distributed partitioning splits the overlay into non-overlapping partitions, as shown in Figure 6.2b. Note that the overlay is not actually partitioned, and



(a) Original, unaggregated overlay



(b) Partitioning



(c) Aggregation

Figure 6.2: Topology aggregation on a six-node overlay.

the connections between each of the partitions remain. This process simply identifies connected clusters of the overlay, that are then aggregated into single virtual nodes by the topology aggregation algorithm. The resulting aggregated overlay is in Figure 6.2c.

- During the topology aggregation stage, QoS epitomes are used to aggregate the QoS of each cluster. A large number of approaches exists for that purpose [106]. For example, geometry based QoS epitomes plot QoS metrics, such as delay and bandwidth, from each source to each destination node on a graph. The resulting graph is then approximated by methods such as line fitting, polylines or curve fitting. While QoS epitomes are used to approximate the QoS of the clusters I aggregate, QoS maps are not limited to a specific method. Various approximation approaches can be used, which have their advantages and disadvantages summarized in [106].

6.2 Hierarchical Topology Aggregation

This section first describes the general topology of the hierarchical aggregation. It then addresses several issues I encountered, which required modifications to the original algorithm, and presents them with evaluations.

6.2.1 Hierarchy Construction

Consider an existing large overlay of n nodes with unique identifiers. For scalability, this network is aggregated into a hierarchy of h levels: The lowest hierarchy level, level 0, is the original overlay and therefore contains all n nodes with no aggregation. This level is then partitioned into several clusters based on the nodes' locality in the original overlay, while bounding the number of nodes per cluster with the constant s. The identifier of a cluster is the largest identifier of the nodes it contains. Each cluster on level l is represented as a single (virtual) "cluster-node" on level $l + 1$. A cluster-node uses the identifier of the cluster it represents as its own identifier. Connections between nodes of different clusters on level l also exist on level $l + 1$, as a link between the virtual nodes of both clusters. In order to reduce the number of links between clusters, link abatement is used [1]: Similar to aggregating multiple nodes into a single cluster, link abatement aggregates parallel links between clusters into a single link. The process continues building hierarchy levels until the uppermost hierarchy level consists of at most s clusters.

Figure 6.3 shows this concept with a hierarchy of three levels and $s = 4$. Parts of the overlay and the aggregated topology have been omitted for clarity, and links to them are represented as dashed lines. Level 0 contains the original overlay, and is partitioned into several clusters. Each cluster is represented by a cluster-node at level 1. This process is repeated, and at level 2, the entire network is represented by only three clusters, making the structure of the network easy to grasp.

When the partitioning and aggregation are described in the remainder of this section, the term "node" shall refer both to a single node in the overlay and to an entire cluster represented as a single node on the next hierarchy level. This simplification of terminology is justified by the fact that both entities behave identically for the purpose of the algorithm.

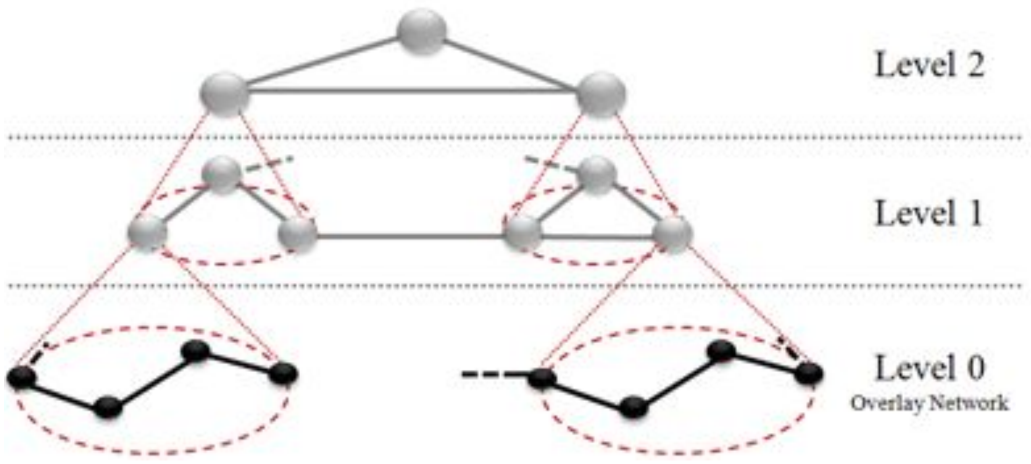


Figure 6.3: A three level hierarchy with $s = 4$.

6.2.2 Simulation Setup

The algorithms explained in this chapter were evaluated through simulations on random graphs with varying density, ranging from 500 to $5,000$ nodes. For this purpose, the same simulator presented in Chapter 5 was used.

Recall that the evaluations in the previous chapter were performed on the Flocks nodes themselves. I ran any simulations in real-time, as it allowed me to use the same implementation that I used in simulation on the PlanetLab as well.

In order to evaluate my changes to the heuristic, as well as the combined algorithm itself, I no longer found it necessary to use the PlanetLab. This is because the aggregation of a network graph is no longer directly concerned with QoS, which was the primary reason that Flocks were written with the PlanetLab in mind.

Therefore, I changed the simulator to no longer run the simulation in real-time. Instead, it uses a simulation time, which it advances in discrete steps. Nodes exchange messages to partition the network and aggregate the topology. At each step, every node may read all messages it has received, and may send messages to its *directly connected neighbours.* This is reasonable, as the average time to parse and respond to a message is negligible (8.38 microseconds on a 2.67 GHz single core lab computer) and the number of messages, as well as the traffic produced are low (an average of 312 messages are sent per node on a $5,000$ node network to fully build the hierarchy, with an average of 4.4 bytes per message).

The simulation steps can directly be related to the expected runtime on "real" networks, if multiplied by the average packet propagation and queuing delay on the network. For example, aggregating a dense network with $3,000$ nodes takes $2,503 \pm 58.3$ simulation steps. If we assume an average propagation and queuing delay of $50ms$, a full aggregation takes an average of 125.15 ± 2.92 sec. In practice, the average aggregation cost will be lower, as most changes in the overlay only require a much cheaper *reaggregation*.

Reaggregation becomes necessary when nodes join and leave the overlay. As only a small region of the overlay is affected, fewer messages and less time are needed. I evaluated the impact through simulations, in which two five-node clusters are joined into a ten-node cluster. Results show that the process takes 24.63 ± 5.51 iterations, during which each of the ten nodes exchanges an total of 40.39 ± 5.62 messages. With the above propagation and queuing delay, the reaggregation completes in an average of 1.23 ± 0.28 seconds.

Completion Criteria

Once all clusters are in a finished state and the number of clusters in the uppermost hierarchy level is not larger than a cluster size limit s, the simulation is considered complete. This situation can be determined without a global view, as the cluster head in the uppermost hierarchy level will find that its cluster has no other inter-cluster edges in this case.

6.2.3 Distributed Partitioning

Before the topology of an overlay can be aggregated, it first has to be split into distinct partitions. As graph partitioning is known to be an NP-complete problem [125], heuristics have to be used. A detailed review of the approaches that have been considered is given in Chapter 2.

In order to determine the distinct partitions needed for topology aggregation, I build up on the partitioning algorithm described in [33], which partitions a network consisting of nodes with unique identifiers by building clusters. Each node (a potential "cluster head") attempts to add surrounding nodes with increasing hop-count ("layers") to its own cluster. The i^{th} layer consists of all nodes with a distance of i hops from the cluster head that do not already belong to a "finished" cluster. If the node succeeds adding a layer, it becomes a cluster-head. Nodes that are added to a cluster assume the identifier of that cluster's cluster-head.

A node that attempts to add a layer of nodes to its own cluster will only succeed if none of the nodes it attempts to add has a higher identifier. Consider a single node with the identifier "10", which begins to build a cluster by adding layer 1 (all nodes with a distance of 1 hop). If none of the nodes it attempts to add has an identifier greater than "10", the process succeeds, all said nodes become part of node "10"s cluster, and assume the identifier "10". Node "10" then attempts to add the next layer.

If a node attempts to add a node with an identifier greater than its own, the process fails, and the node's cluster drops its outmost layer. The dropped nodes assume their original identifiers, and are no longer part of a cluster.

A stopping condition is used to decide when a node stops adding layers: The original heuristic stops growing a cluster if the ratio between cluster members and neighbours exceeds a threshold that is determined by a configurable constant and the total number of nodes in the network. If, after adding a new layer, this stopping condition is fulfilled, that last layer is dropped again, and the cluster is declared to be "finished". A cluster is also finished if no more nodes are found that could be added. Finished clusters no longer participate in the algorithm.

Several changes to the partitioning algorithm were necessary to partition the overlay for QoS maps:

- Its stopping condition was replaced by one that only relies on local knowledge.
- The message format was changed to allow running partitioning algorithm on multiple levels of the hierarchy simultaneously. This removed the need to explicitly finish partitioning a hierarchy level, before starting with the next.
- Finally, the heuristic itself was modified to explicitly deal with hubs.

The following section describes each of the modifications in detail.

Stopping Condition

The original algorithm [33] bounds the number of inter-cluster edges, depending on the network size. This requires knowledge of the total number of nodes in the network graph. Even if the network is assumed to remain static during the measurement, it would be costly to derive this information for large overlays. An alternative approach that does not require this global information is limiting the number of nodes within a partition. Therefore, I considered introducing a new, simple stopping condition (S) that checks the current cluster size against a size limit, s, a positive integer greater than one[1]. While this works well, preliminary experiments showed that considering the number of inter-cluster edges e in addition to the cluster size (in the following labelled **S**ize and **I**nter-**C**luster **E**dges, SICE) further reduces the runtime. This is summarized in Figure 6.4, which compares the average simulation steps required to build the QoS map with the two different stopping conditions on a 5,000 node overlay.

Limiting the number of inter-cluster edges is beneficial for the QoS maps as well, as this limits the number of entry and exit points to a cluster. This reduces the complexity of QoS epitomes and the complexity of the aggregation in the next hierarchy level. Throughout the experiments, I found setting $e := s - 1$ reasonable, as it decreases the likelihood of aggregating nodes into hubs with more than $s - 1$ neighbour clusters.

Because imposing a hard limit on inter-cluster edges makes it impossible to aggregate highly connected networks, I relaxed this condition for cluster heads that are adding their first layer. In this case, the layer is added even if the number of inter-cluster edges, after adding the layer, exceeds e. The drawback of this approach is that for highly connected networks, the runtime gain of SICE (seen in Figure 6.4) is reduced, and approaches the runtime of S.

[1]If $s = 1$, no aggregation is performed, and the algorithm never terminates.

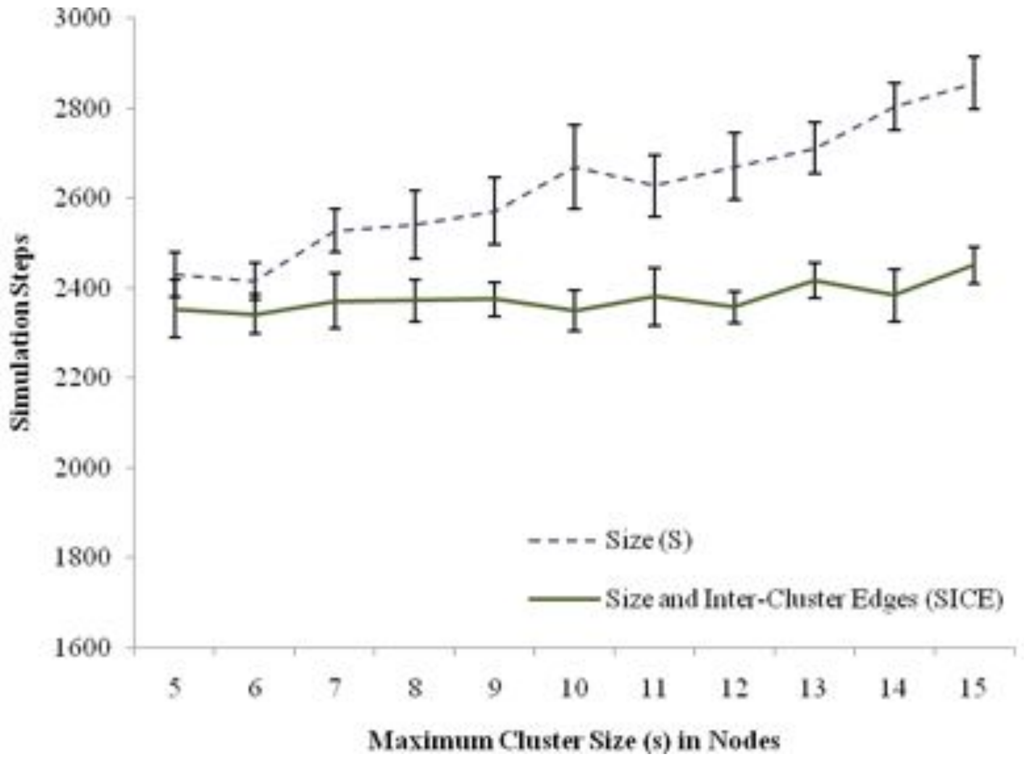


Figure 6.4: Runtime of different stopping conditions on dense 5,000 node overlays.

Cluster Size

The depth of the hierarchy is primarily affected by the size limit s, as it determines how many nodes are aggregated into a cluster. Intuitively, it is desirable to keep s small, because it then becomes feasible to maintain a complete view and employ more complex routing algorithms within a cluster. Furthermore, it reduces the amount of information that needs to be aggregated. Choosing an s that is too small, however, results in a deep hierarchy, which increases the state every node in the overlay needs to store. Preliminary simulations with a variety of different cluster sizes showed that for dense $5,000$ node networks, a cluster size of $s = 10$ offers a good compromise between runtime overhead and hierarchy depth. Therefore, this limit was used for the remaining simulations.

Multiple Hierarchy Levels

As the algorithm runs fully in parallel, without explicit synchronization, no global knowledge exists of when the current hierarchy level is fully partitioned into clusters, and the next hierarchy level can be built[2]. Any finished cluster immediately begins running the partitioning algorithm in the next highest hierarchy level. Therefore, we must consider that different parts of the network may perform aggregation on different hierarchy levels at the same time. Consider any part of a large network that aggregates itself into clusters on level 0 of the hierarchy. At the same time, another part of the same network may already be aggregated into clusters on level 0, and continues to run the partitioning algorithm on level 1. Therefore each node's current hierarchy level is included in every message it sends. Any node that observes a neighbour with a higher hierarchy level considers that neighbour to be part of a finished cluster. This is correct, as a neighbour of a certain hierarchy level is in a final state on all lower hierarchy levels, and would not compete with them in the original algorithm either. Also, a node will not execute the partitioning algorithm until all its neighbours are at least on the same hierarchy level as itself. This does not affect the correctness of the algorithm in [33] either, as it uses implicit synchronization and does not rely on message delivery in a bounded time.

Aggregation of Large Hubs

Hubs with more than $s - 1$ neighbours pose a problem for the original heuristic. Recall that it partitions the network by adding layers of surrounding nodes to the partition of a potential cluster head. If the first layer already exceeds the partition size bound, it is dropped again. This results in a partition with only a single node. Once the entire network consists of such hubs, no further aggregation is possible. Hubs, if existent, will always affect the heuristic, independent of whether they are designated a cluster head (based on their identifiers) or not. For example, assume we need to aggregate a network into clusters of up to s nodes. The network may contain hubs with s or more neighbours. Depending on their identifiers, each hub may either be a cluster head, or not. If the cluster head is the hub itself, no layer can be added without exceeding the limit of s nodes per cluster. If, on the other hand, the cluster head is one of the hub's neighbours, one layer can be added. This layer would contain

[2]An exception to this is the uppermost hierarchy level. As all nodes are for the same cluster, they observe that their cluster has no inter-cluster edges, which indicates that the hierarchy has fully been built.

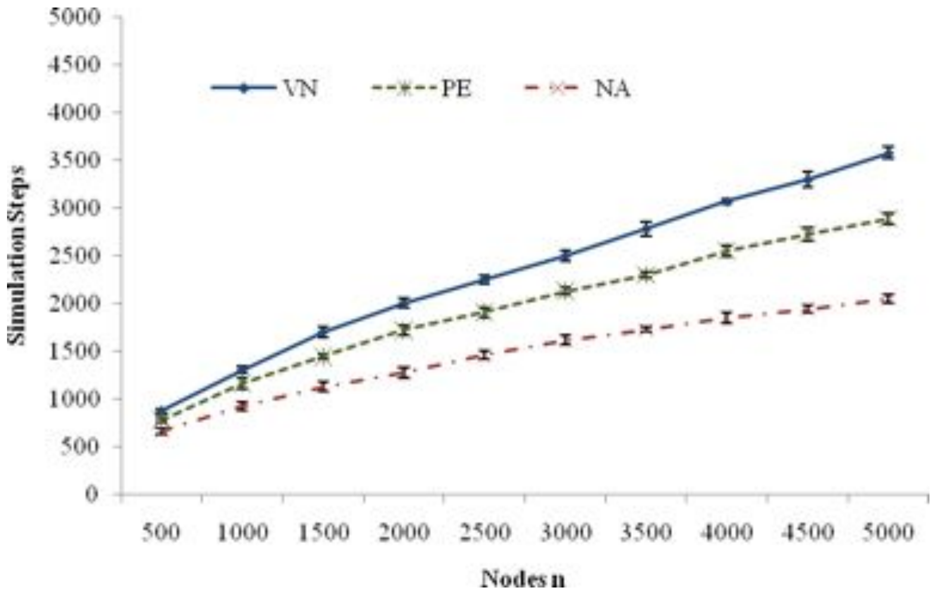


Figure 6.5: Runtime on random dense overlays; different numbers of nodes.

the hub itself, because it is a direct neighbour to the cluster head. As a result, the new cluster would then border the other $s-1$ neighbours of the hub, making it a hub itself in the next hierarchy level.

It should be noted that the random graphs used in the simulations did not usually contain hubs. Hubs emerge primarily during the aggregation of our network. Specifically creating networks with large hubs at the lowest level was not considered necessary, because the algorithm behaves identically for hubs that already exist at the network level and hubs that emerge during aggregation.

As can be seen, hubs frequently occur in higher hierarchy levels during aggregation, and are also very common in overlays. I considered several approaches to deal with them. In a first attempt, labelled *Naïve Aggregation* (NA), cluster heads were permitted to add at least one layer, even if it exceeded the size limit s. Figures 6.5 and 6.6 show this simple approach resulted in low runtime, and low hierarchy depth. However, on simulations with dense $5,000$ node networks, and a target cluster size of 10, the average size of the clusters created on the topmost hierarchy level is 170.17 ± 20.62 nodes. Excessively large cluster sizes like these defeat the purpose of aggregation. As such, while yielding the lowest runtime and hierarchy depth of the evaluated approaches, the structure of its hierarchy is not suitable for our QoS maps.

The root cause why hubs pose a problem lies in the original algorithm, which,

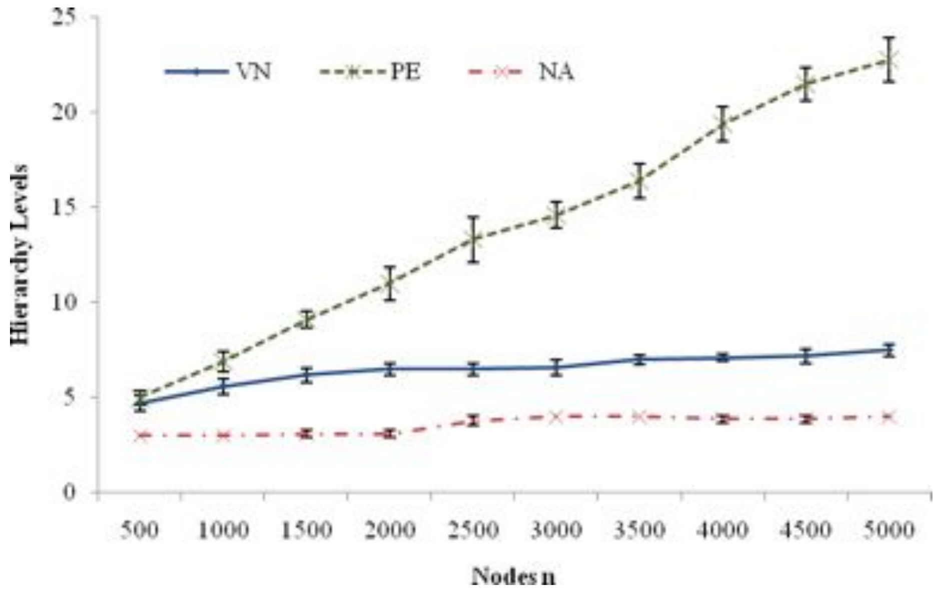


Figure 6.6: Hierarchy depth on random dense overlay; different numbers of nodes.

unrelated to its stopping condition, either adds all nodes of a layer, or none. Therefore, I changed the algorithm so it can accept partial layers as well. If a node is surrounded by more than $s-1$ neighbours, the cluster head randomly accepts $s-1$ nodes into the cluster. The remaining nodes are dropped, as in the original algorithm. Afterwards, the cluster is switched to a finished state. The effect of this process, labelled *Partial Exploration* (PE), can be seen in Figure 6.5. A notable result is that the runtime of the algorithm increases, ranging from 17% for 500, to 41% for 5,000 nodes. Such an increase is expected, as excess nodes are dropped from their cluster and continue to participate in the partitioning algorithm. This happens sequentially (the nodes begin forming a new cluster *after* they were dropped), which explains the increase in runtime. NA simply accepts these nodes, which results in a low runtime, at the cost of creating very large clusters that are unbounded. In the worst case, NA aggregates the entire network into a single cluster with n nodes. In contrast, PE bounds the cluster size. Even with the increase, it retains the sub-linear runtime of the original algorithm. As an upper bound on the cluster size is crucial for scalability, it can be argued that this trade-off is worthwhile.

While PE successfully bounds the cluster size, Figure 6.6 highlights a second problem: The hierarchy depth increases sharply. This is because PE only reduces the number of a hub's neighbours by up to $s-1$ nodes at each hierarchy level. If large hubs with many neighbours exist, the hierarchy depth therefore increases significantly. Consider, for example, a hub with a large number of neighbours. After adding as many neighbours as possible using partial exploration, the cluster containing the hub is finished and no longer participates in the algorithm. Other neighbours that are not part of the cluster are blocked by it, and can only expand in a different direction. It is possible that the number of clusters neighbouring the hub is reduced further, if at least two of its neighbours would join the same cluster. However, to maintain a local view, the partial exploration does not consider the network structure beyond the hub, and therefore does not specifically promote this behaviour. In the worst case, none of the remaining neighbours can be aggregated. A hub with n neighbours and a maximum cluster size of s therefore increases the height of the hierarchy by $\lceil \frac{n}{s-1} \rceil$ levels.

To cope with this issue, I use *virtual nodes* (VN) to "break up" hubs: Once a cluster completes a partial exploration, the cluster head splits into the finished cluster, and a virtual node. The virtual node only exists for purposes of aggregation. It is connected to the cluster, as well as to all remaining neighbours, and continues to participate in the algorithm. With this change, hubs are broken up very effectively within a single hierarchy level, at the cost of additional load on the hub.

Consider again the hub from before. In the worst case, this hub splits into $\lceil \frac{n}{s-1} \rceil$ nodes during the aggregation, resulting in $\lceil \frac{n}{s-1} \rceil$ times as much load as other nodes. Still, if such large real hubs exist in the network, they are likely to be sufficiently powerful to handle the additional load. In addition, a route may include both, the virtual node and the node they split up from. A possible optimization would be conveying the information that both node are the same entity to the routing algorithm. However, as I focused primarily on QoS estimation, I did not consider this optimization further. Alternatively, the algorithm can be configured not to use virtual nodes. This decreases load on large hubs, but increases the hierarchy height, as well as the amount of state every node needs to maintain.

One could reckon that simply aggregating the entire hub with all its neighbours into a single cluster would be more efficient, as the hub has full knowledge of the QoS in its cluster anyway. This was not considered an option, because the number of neighbours of any node is then unbounded. A cluster consisting only of a hub with n neighbours could be traversed in $(n-1)^2$ ways, which greatly increases the advertisement size of the cluster, and the complexity of creating its QoS epitome.

Figure 6.5 shows the runtime increases over PE and NA. Again, this is expected, as a hub splitting into two nodes effectively introduces new (albeit virtual) nodes into the network. On the other hand, Figure 6.6 shows that with VN, the hierarchy depth decreased once more. In fact, with a slight - but still sub-linear - increase in runtime, we have removed the large clusters, and still obtain hierarchy depths only one to two levels deeper than the theoretically optimum of $\lceil \log_s n \rceil$. As a result, by using VN, we are now able to reliably build low-depth hierarchies with clusters that are no larger than the desired cluster size, independently from the underlying network's structure - all important properties for QoS maps.

6.2.4 QoS Epitome Creation

Whenever a cluster is finalized, QoS epitomes of any desired QoS metric are created. *The QoS epitome of a cluster approximates the QoS experienced for traversing that cluster from any entry node to any exit node.* This reduces the amount of information that needs to be advertised to other clusters.

While focusing on the concept of QoS maps themselves, I used the *Full Mesh* (FM) topology transformation method to generate the QoS epitomes. FM stores the QoS from each entry to each exit point of a cluster in a matrix. Therefore, if B denotes the number of entry and exit points of a cluster, FM has a spatial complexity of $O(B^2)$. Other approaches to QoS epitomes may be more efficient: For example, the Partial Mesh (PM) topology transformation method only has a spacial complexity of $O(B)$, but subsequent estimations incur a higher computational cost than FM [3].

It should be noted that the QoS maps are not limited to any specific topology transformation method: Any other transformation method can be used instead, as long as it can be used to obtain an estimation for routing through the cluster (this is the case for all the different transformation methods compared in [3]). As their advantages and limitations are already well researched (see, in particular, the previous cited work), I considered a comparison of their pay-off if used for QoS maps beyond the scope of this thesis.

Any cluster in a QoS map may be the *target cluster* of a route. A target cluster contains the target node of the estimation, and therefore does not have an exit node. Due to aggregation, algorithm does not know the exact location of the target node within the cluster, and thus the worst case has to be assumed. Therefore, the lowest QoS metrics from each entry to any node in the cluster is computed and stored in the epitome as well. As the cluster size is bounded, obtaining this information has little overhead.

The resulting QoS epitomes are included in the topology updates that are sent to the cluster members during aggregation. Therefore, every node also knows the QoS epitomes of each cluster in its local view. To use QoS maps, each node in this hierarchy needs to store $s \cdot (\lceil log_s n \rceil - 1)$ epitomes of its reduced view of the topology with $s \cdot \lceil log_s n \rceil$ nodes.

The topology of the overlay is aggregated without considering QoS. Therefore, it may be possible that a cluster contains paths and nodes with vastly different quality. This is intended: QoS maps aim at giving a view (indeed a map) of the landscape of the given overlay. They do *not* intend to change the landscape itself. By comparing the range of QoS metrics in the QoS epitomes, they can be used to identify poor QoS-awareness in the underlying overlay and thus help to find a better one. In any case, QoS maps allow QoS estimation, regardless of how well or poorly the underlay is organized.

6.3 Hierarchical QoS Estimation

Once the hierarchy is created, it can be used for efficient routing and QoS estimation in the original overlay. This section discusses how the information from the hierarchy is used for QoS estimation, and presents the estimation algorithm.

Recall that each node has a local view of the topology, which contains the topology of its own cluster, and the topology of the clusters alongside the hierarchy that contain it. For example, Figure 6.7 shows the view of node “A” for the topology first shown in Figure 6.3. This information is insufficient for multi-level QoS estimation and routing, however: Even though a node can determine a high level route to the target at level 2 and the next cluster it must route to, it does not know which node at level

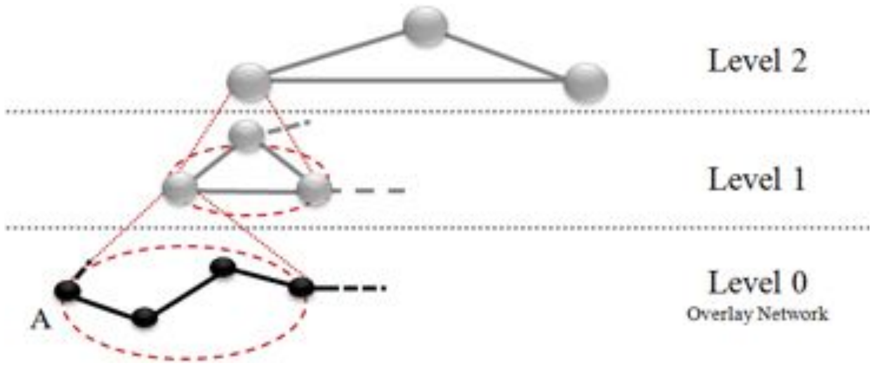


Figure 6.7: Node A's view of the hierarchy.

1 connects to this next cluster. Therefore, it is necessary that a node also knows the identifiers of the clusters that directly border its view at each level. In Figure 6.7, these would be the identifiers of the clusters the dashed lines lead to. This is easy to implement, as that information is already obtained when each cluster is finalized by the partitioning algorithm.

6.3.1 Graphs in this Section

The following section presents an example to understand how estimation with aggregated overlay is performed. Therefore, a brief explanation of the graphs is in order. Consider the overlay in Figure 6.8: The overlay consists of consists of 162 nodes, separated into two clusters, each of which contains 80 randomly connected nodes. Both clusters are connected by two bottlenecks. This particular graph has been chosen because it illustrates a variety of concepts and issues that are encountered during QoS aware routing and estimation.

Note that the graph has been rearranged manually for a clear representation of relevant sections. Due to the high complexity of the graph, the cluster positions in the higher hierarchy levels do not exactly match the positions of the nodes they represent in the lower levels. However, care has been taken that the location of the two bottlenecks, and the general orientation of the graph are consistent throughout the hierarchy levels.

Five data flows have been started in order to generate load, which can be observed through the link colours: Links are coloured from red (load greater than 90%) to blue

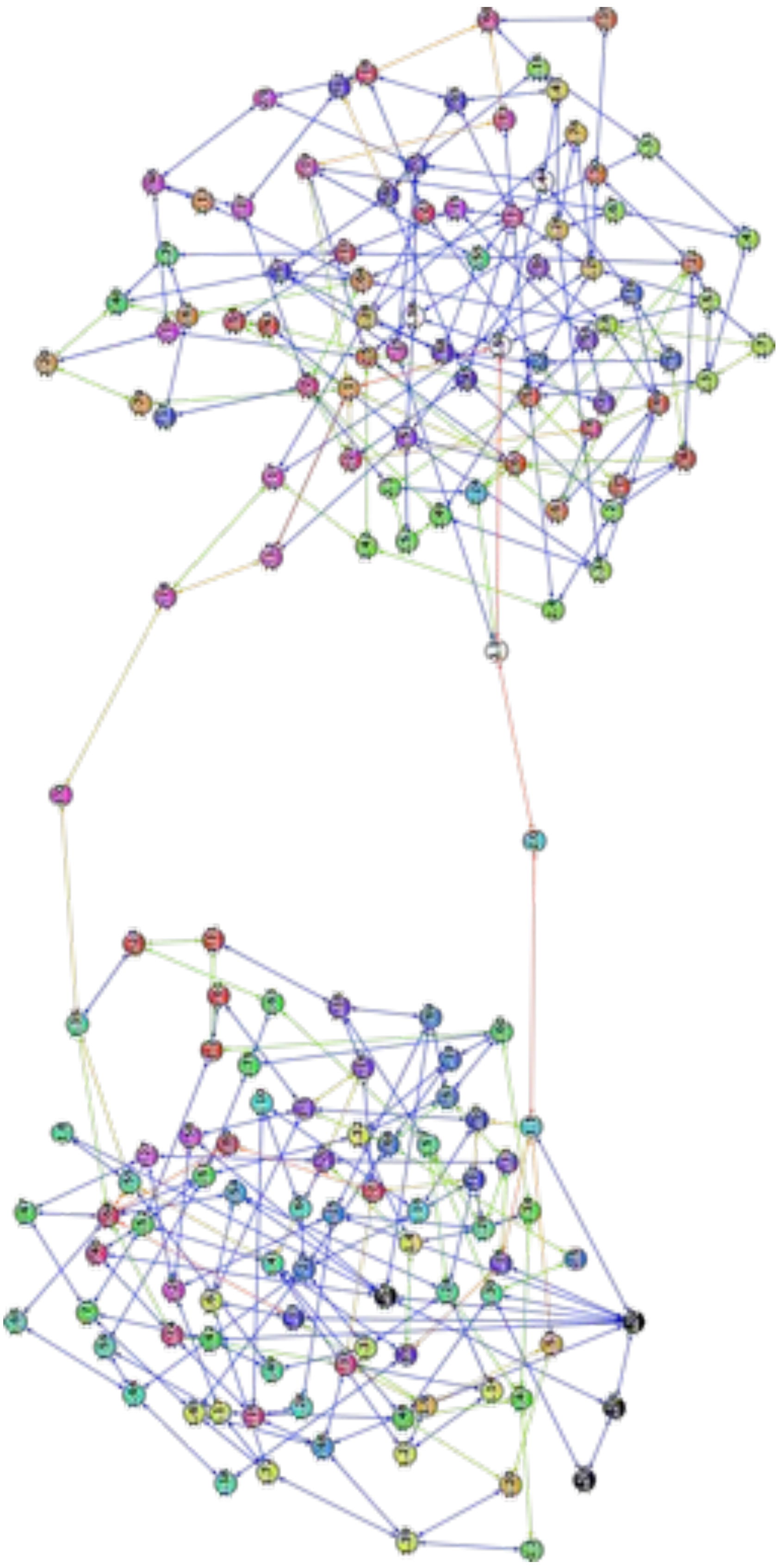
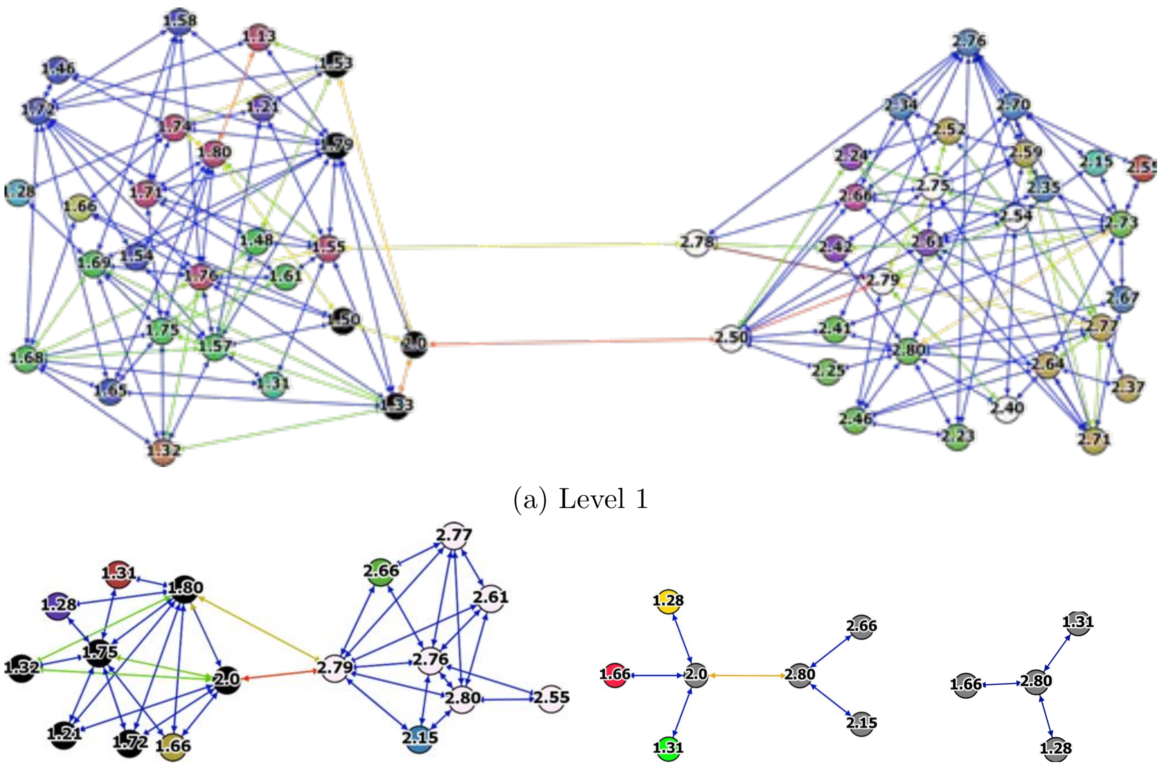


Figure 6.8: An overlay with two bottlenecks. This unaggregated topology is considered the lowest level (level 0) of the hierarchical aggregation.



(a) Level 1

(b) Level 2 (c) Level 3 (d) Level 4

Figure 6.9: The aggregated representations of the overlay in Figure 6.8.

(load less than 1%). They reflect an accurate measurement on the lowest (unaggregated) hierarchy level, and the best-case estimate on the other hierarchy levels. Finally, nodes are coloured based on their cluster, so that nodes with the same colour also belong to the same cluster. The labels on the nodes reflect their true identifier on the lowest (unaggregated) hierarchy level, and the identifier of the cluster they represent on the other hierarchy levels.

I applied the aggregation algorithm introduced in Section 6.2 to the overlay, with a maximum cluster size of six. This generated a hierarchical aggregation with five levels (four aggregated levels, and level 0, shown in 6.8, which represents the original overlay). Figure 6.9 shows the four aggregated levels.

Limitations

Aggregated overlay graphs can be read like traditional overlay graphs. However, due to limitations of the graphing tool used in this thesis, some information is not

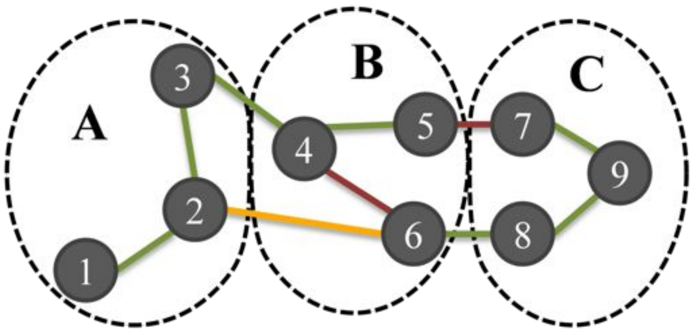


Figure 6.10: A small overlay of nine nodes aggregated into three clusters (A, B, and C).

displayed. This section explains what information is contained in the graph, and lists the omissions. Furthermore, it discusses how the omitted information is used by the QoS estimation algorithm, and how a human reader can simulate the process.

An aggregated node in the graph is displayed as a single circle, with links originating from and terminating in it. This simplification makes the graph easier to read, however some information is lost: An aggregated node often has multiple nodes that connect it with other clusters. For example, cluster A may have two different nodes that link to a node in cluster B. This information is important, not only because the QoS on the two links connecting cluster A and B, but also the QoS experienced within cluster A may differ.

Entry	Exit	Bottleneck-Bandwidth
4	5	Low
4	6	Low
6	4	Low
6	6	Medium

Table 6.1: A naïve QoS epitome for cluster B in Figure 6.10.

Consider the overlay in Figure 6.10: The link between node 3 and 4, which connects cluster A with B, is a poor choice to reach node 9, because any subsequent routing through cluster B will lead over a bottleneck. Detecting this situation is the purpose of QoS epitomes, which are used to calculate the best-case route from any cluster entry to any cluster exit point. In Figure 6.10 all three nodes in cluster B are border nodes. A naïve QoS epitome might therefore look similar to Table 6.1,

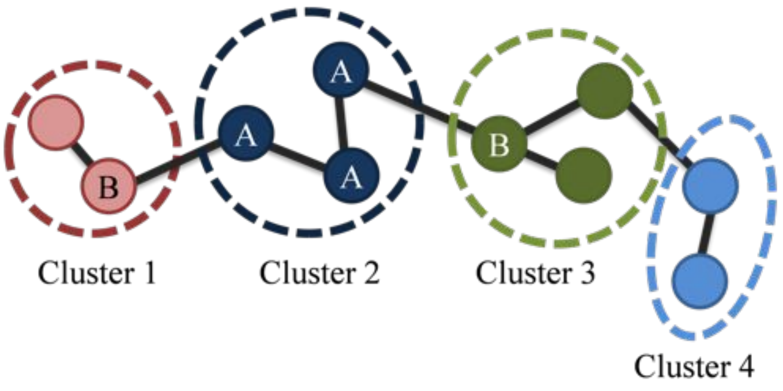


Figure 6.11: A small overlay illustrating the view of nodes in a single hierarchy level.

allowing other nodes to detect the bottlenecks encountered by routing over node 4[3].

As nodes have access to the relevant QoS epitomes, they can use them to avoid bottlenecks within clusters, even without being aware of the details within the cluster. QoS epitomes are not shown in Figure 6.9, due to the amount of complexity they add to the graph. However, as the graphs offer a global view, a human reader can simply look at the nodes in the next-lowest hierarchy level that the cluster in question represents to identify possible bottlenecks.

Note that even though the graphs represent global views, nodes only have a limited, local view. At each level, they are only aware of the following:

- Nodes in their own cluster (these nodes shown in the same colour in Figure 6.8), of which they have full information (namely the identifiers, any available QoS-related information).
- The identifiers of the nodes that connect to their clusters.
- The identifiers of the clusters of said nodes.

Consider the small overlay in Figure 6.11. Any node in cluster 2 has full information of nodes labelled with “A”. In addition, it knows the identifiers of the nodes that connect to its cluster, labelled with “B”, and the identifiers of the clusters these nodes are in (“Cluster 1” and “Cluster 3”). It has no information about the remaining, unlabelled nodes. Note, in particular that in this view, nodes in cluster 2 are not

[3]Better QoS epitomes attempt to reduce the amount of data while keeping their accuracy as high as possible.

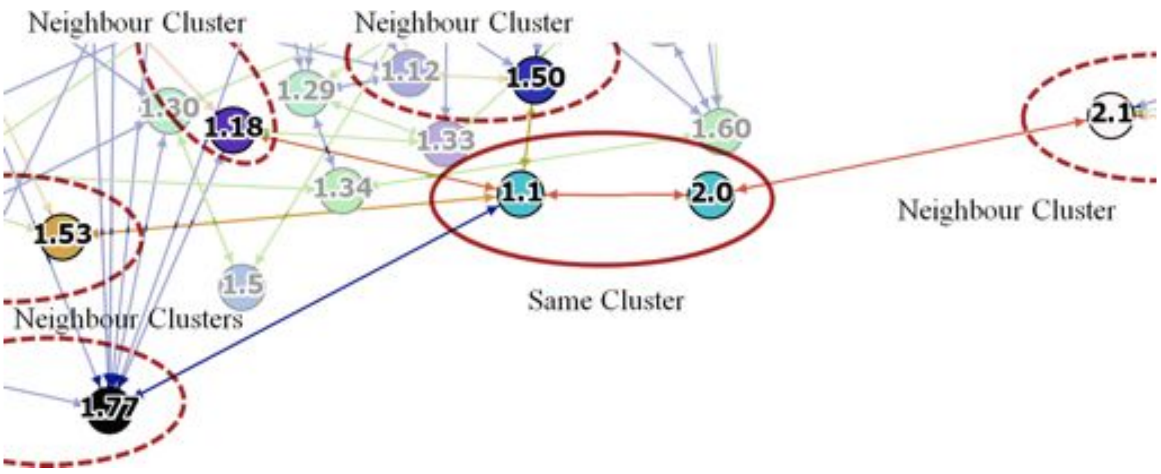


Figure 6.12: An enlarged partial view of the overlay in Figure 6.8, highlighting the nodes included in the view of node 2.0.

aware of cluster 4 at all. They may see cluster 4 as a single, aggregated node in the next highest hierarchy level.

Node Identifier	Relation	Available Information
2.0	Itself	Node/Cluster Identifiers, and QoS
1.1	Same Cluster	Node/Cluster Identifiers, and QoS
1.77	Neighbour Cluster 1.79	Node/Cluster Identifiers
1.53	Neighbour Cluster 1.53	Node/Cluster Identifiers
1.18	Neighbour Cluster 1.33	Node/Cluster Identifiers
1.50	Neighbour Cluster 1.50	Node/Cluster Identifiers
2.1	Neighbour Cluster 2.50	Node/Cluster Identifiers

Table 6.2: Nodes included in the local view of node 2.0 in the lowest hierarchy level.

For example, the view that node 2.0 (in the bottom bottleneck of Figure 6.8) has of level 0 is described in Table 6.2. Figure 6.12 shows an enlarged, partial view of the relevant overlay portion, and highlights the nodes and clusters that node 2.0 is aware of.

6.3.2 QoS-Aware Route Finding: An Example

To motivate this example, assume that node 1.78 (near the bottom left of Figure 6.8), needs to find a high-bandwidth route to node 2.1 (at the right end of the bottom bottleneck). Therefore, the goal of its QoS estimation is to (a) identify that the bottom bottleneck is highly loaded and (b) find the alternative route using the upper bottleneck, avoiding the obvious, shortest path.

Assumptions

It is assumed that node 1.78 knows the location in the hierarchy of the node it wants to communicate with. A node's location is specified by concatenating the identifiers of each cluster it is in, from the lowest to the highest hierarchy level. For example, node 1.78 is located in cluster 1.79, which is itself contained in cluster 2.0 at level 1, cluster 2.0 at level 2, cluster 2.80 at level 3. At level 4, all nodes are contained in a single large cluster, 2.80. Table 6.3 illustrates this. In order to enable an easy location of both nodes and their clusters in Figures 6.8 and 6.9, the clusters containing node 1.78 have been coloured black, and the clusters containing node 2.1 white.

Node Identifier	Cluster the node is located in, at ...				
	Level 0	Level 1	Level 2	Level 3	Level 4
1.78	1.79	2.0	2.0	2.80	2.80
2.1	2.50	2.79	2.80	2.80	2.80

Table 6.3: The location of the source (1.79) and the target (2.1) node in the hierarchy.

This example assumes that no route is yet known. Therefore, we first need to obtain a route that the QoS can be estimated on. The starting point to obtaining a route is the highest hierarchy level - level 4, as shown in Figure 6.9d. On this level, node 1.78 and 2.1 are both located in cluster 2.80. As no other route exists (for example, it may be possible to bypass a bottleneck in cluster 2.80 if a link between cluster 1.66 and 1.31 existed), we consider the next lowest level.

High-Level Route Selection

On level 3, shown in Figure 6.9c, node 1.78 is located in cluster 2.0, whereas node 2.1 is located in cluster 2.80. Both nodes are connected through a link with a medium bottleneck-bandwidth, and there is no other way to route between both nodes. This hierarchy level is significant for three reasons. First, it shows that, in order to get from node 1.78 to 2.1, it is necessary to route from cluster 2.0 to cluster 2.80. Second, as both clusters are connected through a single link with sub-optimal bottleneck bandwidth, it conveys that a bottleneck between both nodes exists. Third, it already offers an upper bound on the best possible bottleneck-bandwidth that can be achieved between both nodes[4]. As a result, node 1.78 now needs to consider the next lowest level, and determine how to route from its cluster 2.0 to a node in cluster 2.80.

Identifying the Bottleneck

On level 2, node 1.78 must find a route from its own cluster, to a node in cluster 2.80. As shown in Figure 6.9b, only node 2.79 (which is in cluster 2.80) borders its own cluster. Recall that at level 2, node 1.78 is located in cluster 2.0. It can therefore route to node 2.79 directly (over a link with poor bottleneck-bandwidth), or it can route to cluster 1.80, which then relays to node 2.79.

This hierarchy level is also significant, because it first shows that two bottlenecks exist, one of which is a worse choice than the other. Based on this information, an application can now identify the highly loaded bottleneck at the bottom, select the route over the top bottleneck (from 1.80 to 2.79) and discard the shorter route over the bottom bottleneck (from 2.0 to 2.79). This satisfies goal (a).

Need for QoS Epitomes

In addition, cluster 1.80 can be reached in several ways. Some possible routes are listed below:

- 2.0→1.72→1.80
- 2.0→1.75→1.80
- 2.0→1.72→1.75→1.80
- 2.0→1.32→1.80
- 2.0→1.32→1.80
- 2.0→1.80

[4]Note that this information may be inaccurate due to recent QoS changes, and the accuracy of the QoS epitome.

In order to determine the best route, an application uses the QoS epitomes created during aggregation. Recall that QoS epitomes permit evaluating the impact on the QoS for routing over each cluster, based on the entry and the exit point of the cluster. For example, the epitome of cluster 1.75 allows estimating the impact if the cluster is entered at 2.0 and exited at 1.80. This is important, as cluster 1.75 may contain a bottleneck that must be traversed if traversed through these two nodes. The information from the QoS epitome is used to select the route to 1.80.

Recall that QoS epitomes are not displayed in the graph. However, one can infer the content of the QoS epitome by considering the next lowest hierarchy level. For example, assume an application needs to decide whether to route directly from 2.0 to 1.80, or to route from 2.0, over 1.75, to 1.80. To make this decision, it needs to know the cost for routing from 2.0, over 1.75, to 1.80.

Chaining QoS Epitomes

In this case, we consider all nodes that belong to cluster 1.75 at the next lowest level 1, starting out at any node belonging to cluster 2.0, building the best case routes that terminating at any node belonging to cluster 1.75 or 1.80. Figure 6.13 illustrates the entry and exit nodes that are considered in the epitome of cluster 1.75 for routing from 2.0 (black nodes), over 1.75 (green nodes), to 1.80 (pink nodes). The links leading from 2.0 to an entry node in 1.75 are highlighted in light blue, the links leading from an exit node in 1.75 to a node in 1.80 are highlighted in red. Links between nodes are coloured based on their QoS, ranging from high QoS (blue) to low QoS (red). Nodes and inter-cluster links that do not belong to clusters 1.75, 1.80 or 2.0 have been faded out.

An application then uses the information from the QoS epitome to "chain" routes together. Consider that each route begins in the node's own cluster, leads over any number of transit clusters, and finally terminates in the target cluster. Using the QoS epitomes, the node knows the identifiers of the nodes over which the route will leave its own cluster, and enter either a transit, or the target cluster. In the above example, the route may leave cluster 2.0 (the black nodes on the right) over 1.33, 1.50, or 1.79. Therefore, the application must take the following information into account:

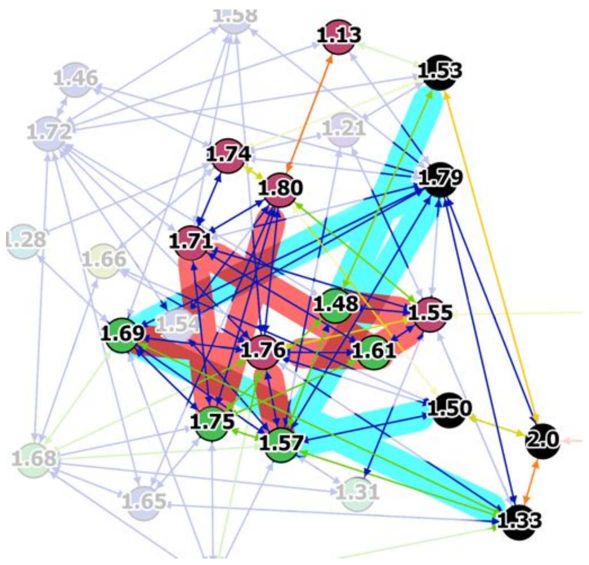


Figure 6.13: A partial graph taken from Figure 6.9a, highlighting the links leading from and to the entrance (blue) and exit (red) nodes of cluster 1.75.

- The cost of routing through cluster 1.75: As the application knows which nodes of its own cluster connect to cluster 1.75 (the light-blue links in Figure 6.13), it consults the QoS epitome of cluster 1.75 to approximate the cost of routing across it from any given entry node, to any exit nodes that then lead to cluster 1.80 (the red links).

- The cost of routing through its own cluster: The application must determine the cost of routing through its own cluster, to reach the exit node(s) considered above. For example, even though node 1.69 is connected to cluster 1.80 with a high QoS link, and can be reached from node 1.33, node 1.33 is not a good choice, because its only link to node 2.0 has poor QoS. The route is essentially chained together, because the entry nodes considered in the previous step determine which exit nodes are valid choices in the current step.

The application can then use a similar process to approximate the cost of routing to cluster 1.80 directly (for example, from node 1.53), or through one of the other clusters that connect cluster 2.0 and 1.80 (which are faded out in the graph). In any case, based on the approximations, it can then select the most suitable route.

Assume that it determines that the most suitable route is the direct route from cluster 2.0 to cluster 1.80[5].

The application then consults the next lowest level (Figure 6.9a), on which it is contained in cluster 1.79. As in the previous step, it has several ways to reach a node from cluster 1.80. To find an appropriate route, it again considers routes within its own cluster, and consults the QoS epitomes each node to calculate the cost for traversing it. Assume that the direct route from 1.79 to 1.55 (which is in cluster 1.80) is chosen. The node again consults the next lowest level, in order to obtain a route from itself to cluster 1.79.

The next lowest level is level 0, the unaggregated overlay seen in Figure 6.8. Recall that node 1.78 only sees a small portion of it, namely the nodes of its own cluster, coloured in black, and the identifiers (of both, cluster and node) of the nodes directly bordering its own cluster. Regardless, it can now calculate a route to node 2.1:

- First, it must route to its neighbour cluster 1.55: As node 1.78 has a full view of its own cluster, and knows the identifiers of the nodes directly bordering its own cluster, any routing algorithm can be used for this purpose. One possible route that traverses only links with a bottleneck bandwidth of 10 MBit/s is 1.78 $\rightarrow$1.79 $\rightarrow$1.56 $\rightarrow$1.8 (which is in cluster 1.55 at level 0).

- The node in cluster 1.55 at level 0, node 1.8, must then route to any node in cluster 1.80 at level 1 of the hierarchy. Since node 1.8 itself is in cluster 1.80 at level 1 of the hierarchy, it does not need to anything for that.

- The node in cluster 1.80 at level 1, node 1.8, must then route to any node in 2.80 at level 2 of the hierarchy. As it has a full view of its own clusters, it can determine a route through cluster 1.80, that exits to cluster 2.80. For example, node 1.8 can route to node 1.55, and then to node 2.78 (which is in cluster 2.80). In this case, it must repeat the route finding process to find a route from

[5]It should be noted that the shortest route in an aggregated representation is not guaranteed to be the shortest route in the underlying representation. For example, routing directly from cluster 2.0 to a node in cluster 1.80 may traverse more nodes in cluster 2.0, than routing to a node in a different cluster would. Nevertheless, it is still possible to determine the shortest route through aggregated information, namely if hop-count is aggregated as well. If an application optimizes against several metrics, such as bottleneck-bandwidth and hop-count, it must consult separate QoS epitomes for each metric. For example, a bottleneck-bandwidth epitome contains the bottleneck bandwidth for traversing a cluster, whereas a hop-count epitome lists the number of hops required to traverse it.

itself to node 1.55 at level 0. At level 0, it finds that it can directly route to node 1.55.

- Node 1.55 repeats the same process to find a route a node in cluster 2.78 at level 0, obtaining the route 1.55 $\rightarrow$1.35 $\rightarrow$1.0 (which is in cluster 2.78 at level 0). This routes over the top bottleneck, bypassing the bottom bottleneck and satisfying goal (b).

Recursion and Termination

A packet following the route described in the previous section is now at the other side of the bottleneck, in cluster-node 2.80 at level 3. This means it has followed the highest level route 2.0 $\rightarrow$2.80 that 1.78 had initially selected at hierarchy level 3.

The task now is to route the packet within cluster-node 2.80 at hierarchy level 3, to target node 2.1. This can be performed by any node that is part of cluster-node 2.80 at level 3, by repeating the route finding process with itself as the route source. In this sense, the process *recurses*, as the route finding process restarts, closer at the target.

Since the new starting point and the target node are contained in the same cluster-node (2.80 at level 3) when the process recurses, the highest common level, of which both nodes share the same view, is always at least one level lower than before: Both nodes see all other level 2 nodes within their cluster (which is represented by cluster-node 2.80 at level 3), and therefore see each other. Assuming a hierarchy has h levels, it is guaranteed that the route finding process terminates after at most $h-1$ of said recursions.

6.3.3 Estimating in Combined Hierarchy Levels

The attentive reader may have noticed that in cluster 2.79 at level 1 all the links leading to the target cluster 2.50 that contains node 2.1 are bottlenecked. Strictly following the procedure introduced in the previous section does not bypass the bottleneck. As source and target node are represented by a single node at the next-highest level, no routes through other clusters are considered, even though cluster-node 2.78

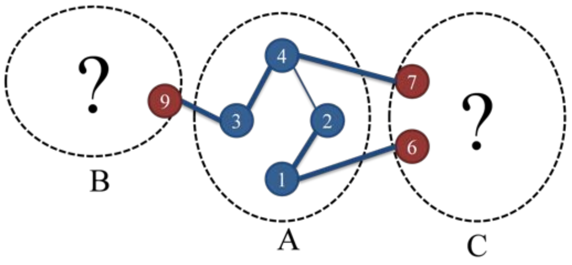
at level 1 could route through cluster-node 2.66, and then to cluster-node 2.50, bypassing the bottleneck. This situation proved to be difficult to handle by the original approach. If such a bottleneck is present, it is necessary to either use it, or return to the higher hierarchy level in order to search for an alternative route that bypasses the bottleneck, and then reconsider the lower level to ensure that no other bottlenecks were introduced. This requires a memory of routes that "should not be taken" and in the worst case[6], this process has to be repeated $s^2 - s$ times.

To solve this problem in a more efficient way, I adapted algorithm as follows:

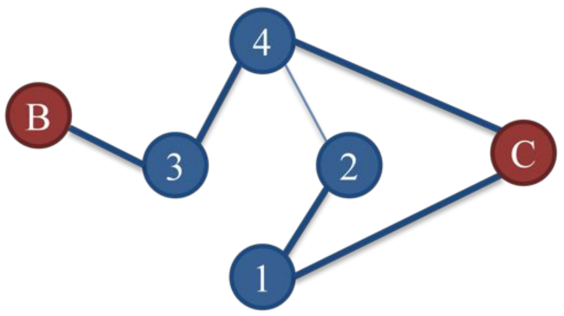
- The algorithm begins at the highest hierarchy level, instead of the lowest common hierarchy level. This is necessary in case the bottleneck exists within a cluster of the lowest common hierarchy level (or one of the higher levels). In this case, the view of a higher hierarchy level is necessary to bypass the bottleneck. By starting at the highest hierarchy level, this is no longer an issue.
- Cluster-nodes of which topology information is available are replaced by the nodes of the next-lowest level that they represent. The other cluster-nodes, of which no detailed topology information is available, remain unchanged.
- The route is determined on this combined view. Aggregated nodes are treated similar to unaggregated nodes for the purpose of determining the route, except that the cost of routing through them is determined using their QoS epitomes.

Figure 6.14a illustrates the process on a small overlay consisting of three clusters, A, B, and C, in which node 1 needs to route to node 4. Both nodes are in cluster A, however they are separated by a bottleneck. To compute the route, the highest hierarchy level is used. For this example, assume that level 1, with the cluster-nodes A, B, and C is the highest hierarchy level. As node 1 has detailed topology information of cluster A, it replaces the cluster-node A on level 1 by the nodes that represent it. The result is shown in Figure 6.14b. Node 1 can now use any routing algorithm on the resulting graph and bypass the bottleneck. Note that node 1 does not need to know the topology of cluster B or C, as it uses their QoS epitomes to determine the

[6]In the worst case, each of the s nodes is connected to a different node in another cluster. Therefore, an algorithm attempting to bypass a bottleneck would need to consider every node as exit and re-entry point, since the cluster could be left multiple times.



(a) The view of the nodes in cluster A, at level 0.



(b) The view of the nodes in cluster A, if both hierarchy levels are combined.

Figure 6.14: A small overlay in which a bottleneck within a cluster must be bypassed.

cost for routing through them. If, for example, a bottleneck existed in cluster C that any routes from entry point 6 to exit point 7 must cross, it would be reflected in its QoS epitome at level 1, and therefore considered by node 1.

Algorithm 6.1 Joining multiple hierarchy levels to bypass bottlenecks within a cluster

```
1: GENERATECOMBINEDVIEW($a$) : –
2: $H \leftarrow$ getHierarchyLevels()
3: $g \leftarrow$ getLocalViewAt($H$, $empty$)
4: for $l \leftarrow H - 1$ to 1 do
5:     $c \leftarrow$ getClusterOfNodeAt($a, l$)
6:     $d \leftarrow$ getLocalViewAt($l - 1, c$)
7:     $g' \leftarrow$ replaceClusterInGraph($g, c, d$)
8:     $g \leftarrow g'$
9: end for
10: return $g$
```

Algorithm 6.1 describes the adapted estimation process in high-level pseudo code. It accepts one parameter, a, which represents the estimating node, and makes use of five functions to combine the locally available parts of the hierarchy into a single view: It introduces four additional functions which are needed because the estimating node no longer operates on the original hierarchy, but generates a new view that combines all available local information. The additional functions introduced are listed below:

- getHierarchyLevels() returns the total number of levels in the hierarchy. A hierarchy with 3 levels consists of the levels $0, 1$ and 2.

- getLocalViewAt(l, c) returns the graph representing the hierarchy at level l, as stored by the estimating node. If c is empty, the entire graph at that level is returned. Otherwise, c is set to a cluster identifier, and only nodes within this cluster are returned. This is necessary in case the estimating nodes has less aggregated information for cluster-nodes other than the source cluster[7].

- getClusterOfNodeAt(a, l) returns the identifier of the cluster-node at level l that node a is contained in. At level 0, it returns node a itself. As each node stores the identifiers of their parent clusters in an array, this amounts to a single operation.

- replaceClusterInGraph(g, c, d) searches graph g for cluster-node c. It then replaces cluster-node c with the graph d that it represents. Links from and to c are attached to the appropriate nodes in graph d. This is possible as every node has full knowledge of its own cluster and its inter-cluster edges, at every hierarchy level.

The algorithm begins by considering the graph highest hierarchy level (line 3). It then identifies the source cluster (which the estimating node is in, obtained in line 5), and determines which nodes represent it at the next lowest level (line 6). Finally, it replaces the source cluster with the nodes that represent it (line 7). This process repeats until the lowest hierarchy level has been reached.

Figure 6.15 displays the combined view obtained from the estimating node 1.2 during one of the experiments. The nodes in both graphs are colour-coded based on which aggregation level (🟦 level 0, 🟩 level 1, 🟨 level 2, 🟧 level 3 or 🟥 level 4) they are derived from. For reference, the unaggregated overlay is shown in Figure 6.16a (node 1.2 is highlighted). Figure 6.16b shows level 3 of the hierarchy generated from the overlay, and Figure 6.16c level 4 (the highest level). Notice that all the nodes from level 4 are contained in the combined view in Figure 6.15. Node 1.2 is within cluster-node 2.80: As a result, it is split up, and replaced with the nodes of level 3 visible to node 1.2 (these are the nodes in the same cluster, shown in Figure 6.16b).

[7]This information can be obtained through cooperation with nodes from other clusters. It is possible to use it to replace other nodes in the graph by their less aggregated representations, however for simplicity this is not considered here.

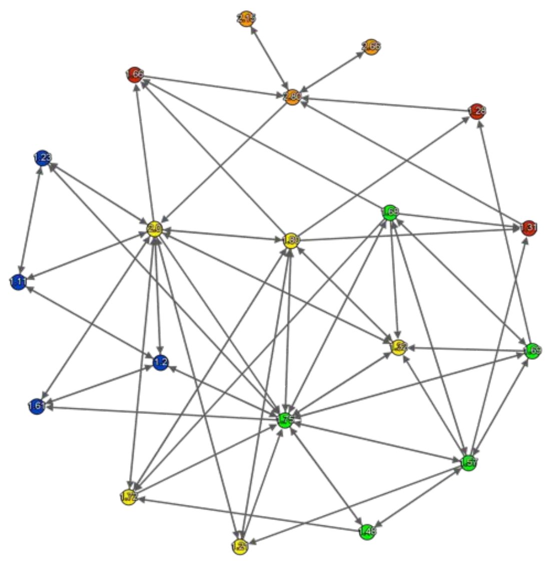


Figure 6.15: The combined view derived by node 1.2. Colours represent nodes derived from different hierarchy levels.

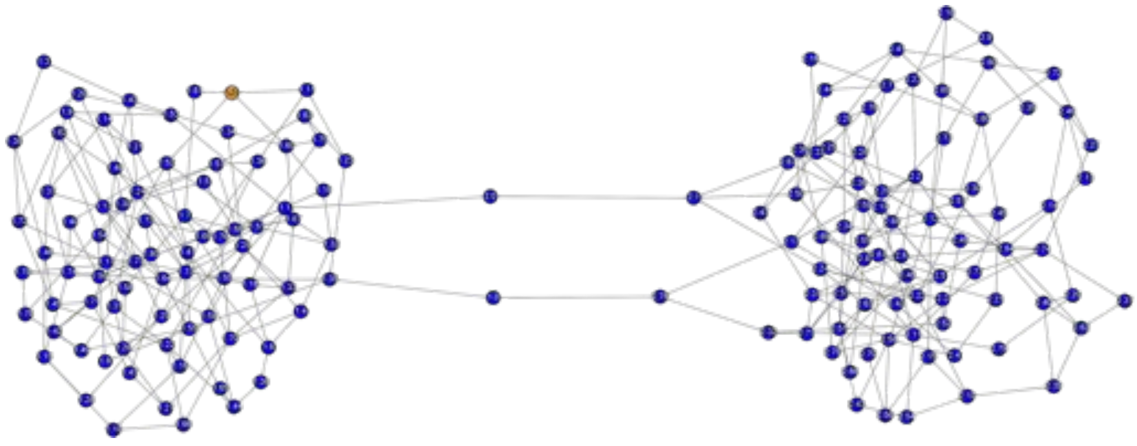
At level 3, node 1.2 is within cluster-node 2.0, and therefore that node is split up next. This process repeats until the lowest level is reached, and results in the graph shown in Figure 6.15.

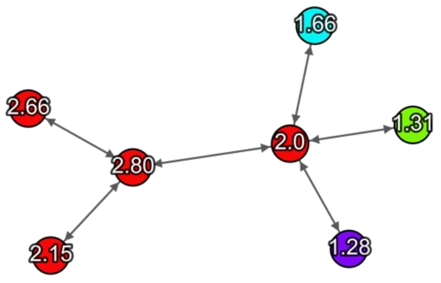
Performance Considerations

With this change, suitable routes can be found even if a bottleneck in the source cluster must be bypassed. However, it also changes the complexity of the estimation and routing algorithms:

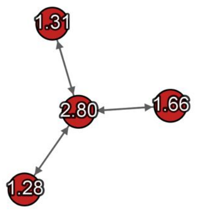
Instead of operating on each level, each with s nodes, separately, the routing algorithm operates on the combination of all levels. Therefore, the maximum graph size that the routing algorithm operates on changes from a constant to a logarithmic function in respect of the total number of nodes in the overlay.

The original algorithm introduced in this section considers s nodes at each of the h hierarchy levels. If no backtracking occurs, it is invoked up to h times, with graphs of up to s nodes. However, backtracking may be necessary to bypass a bottleneck within a cluster. In the worst case, a cluster-node is neighboured by $s-1$ clusters at each of the h levels, and each cluster-node must be considered to bypass the bottleneck. Therefore, even though the number of invocations may be logarithmic in the average


(a) The unaggregated overlay (node 1.2 is highlighted)



(b) Level 3



(c) Level 4

Figure 6.16: The unaggregated overlay (a) and the aggregated (b) levels 3 and (c) 4. Colours represent different clusters.

case, it is exponential in the worst case.

On the other hand, the adapted algorithm that can bypass bottlenecks within a cluster is invoked a single time, with a graph that combines the detailed topology information of the source cluster on each level with the rest of the hierarchy. Therefore, the graph consists of the up to $s-1$ aggregated cluster-nodes (the cluster-nodes not part of the source cluster) at the highest hierarchy level, as well as the source cluster, which is broken up recursively (into $s-1$ cluster-nodes that are not part of the source-cluster at the next lowest level, and the cluster-node that is). Finally, the bottommost level 0 consists of up to s nodes. Based on these considerations, the upper bound of the graph size can be defined as $h(s-1)+1$. Table 6.4 summarizes the differences.

Algorithm	Invocations		Graph Size	
	Average	Worst	Average	Worst
Original	h	s^h	s	s
Adapted	1	1	$h(s-1)+1$	$h(s-1)+1$

Table 6.4: Cost comparison of the hierarchical routing algorithms

The adapted algorithm invokes the routing algorithm on graphs with a higher number of nodes: Consider that the cluster size is bounded with a constant, s, and therefore, the graphs used by original algorithm have a constant size. The graphs used by the adapted algorithm scales linearly with respect to the number of hierarchy levels (and hence logarithmically with respect to the number of overlay nodes).

However, with the adapted estimation algorithm, the routing algorithm is only invoked a single time, whereas it is invoked once for each hierarchy level (on smaller graphs) if the original algorithm is used.

Whether the adaptation is efficient depends on the complexity of the routing algorithm that is used: If the routing algorithm has a high runtime or storage complexity with respect to the number of nodes in the graph[8], the backtracking approach detailed earlier can be more efficient, as it uses graphs with a (small) constant size. However, even though in the average case, the routing algorithm will only be invoked h times, the worst case has an exponential complexity.

[8]An example for a high complexity routing algorithm is the Travelling Salesman Problem (TSP), which is NP-complete.

If a routing algorithm with low complexity is used, the adapted approach should be used instead. Even though the graph size is no longer constant, but logarithmic with respect to the number of hierarchy levels, it only invokes the routing algorithm once.

As efficient routing algorithms exist[9], I consider the graph size increase from constant to logarithmic complexity reasonable, and the adapted algorithm suitable for practical situations.

6.4 Implementation

This section presents the algorithms that perform the steps explained previously. This is done in two parts: First, I present the estimation algorithm that is concerned only with the estimation itself and does not perform any routing whatsoever. It accepts a route from the source to the target, which may consist of nodes and cluster-nodes of different hierarchy levels, and then calculates the estimation. Second, I present the algorithm that also performs QoS-aware routing, as shown in the example in Section 6.3.2. It accepts a source and a target node, and uses the QoS epitomes to obtain a high-LOS route from the source to the target node.

6.4.1 Inter-Cluster QoS Estimation on a Given Route

This case assumes that a route has already been determined, but the application needs an estimate of the LoS that packets will likely experience along the route. This can be used to select the best route out of several alternative routes, or to set reasonable default values on an existing connection. For example, if the bottleneck bandwidth is low, a video stream could start at its lowest quality.

A node that wishes to predict the QoS experienced on a path to a target node first uses its locally stored QoS epitomes to get a coarse, low-cost estimate. This estimate can later be refined.

Each path used in QoS estimation exists of up to three types of clusters: a source cluster, a target cluster and zero or more transit clusters. Note that source and target

[9]Examples include Dijkstra's algorithm[121], or one of the various heuristics that exist even for TSP.

clusters are always different from each other, except for the trivial case in which both nodes are in the exact same cluster at the lowest hierarchy level.

The three types of clusters differ in the amount of local information the source node has, and are treated differently for QoS prediction purposes:

- *Transit clusters* are traversed on route to the target node. As QoS epitomes for that cluster contain the aggregated QoS from each entry to each exit point, they can be used directly to get a high level estimate of the cost of traversing them.

- *The target cluster* is the least accurate cluster type in the estimation process. As with transit clusters, we consider the entry point. However, as they do not have an exit point, it would be necessary to have a view of the target cluster in order to provide a more precise estimation. Without said view, we have to assume the worst case QoS from its entry point to any node in the cluster.

- *The source cluster*, which contains node A, can always be broken down to the next lowest hierarchy level. This increases the quality of the QoS prediction, as it uses less aggregated QoS epitomes. Breaking down a source cluster results in a new (less aggregated) source cluster, and zero or more transit clusters (also less aggregated than the existing transit clusters). The new clusters introduced by breaking down the hierarchy are connected to the cluster that the source cluster was originally connected to.

Note that breaking down the source cluster requires invocation of a routing algorithm, to determine the clusters that will be traversed to leave the source cluster. Since this section is strictly concerned with estimation and no routing, assume that the route given to the estimation algorithm is broken down as much as desired for estimation purposes. If the source cluster is not fully broken down to level 0, the QoS epitome of the source cluster is used in a similar fashion as the QoS epitome of the target cluster. Since the location of the node in the source cluster is not known in this case, the worst case QoS from any node in the cluster to its exit point must be assumed.

Note that this limitation is relaxed in the following section, which always breaks the source cluster down to its lowest level.

Combination of QoS Estimations

In order to obtain the estimate, the QoS for each hop along the route is estimated, and the estimations are combined. If a hop in the cluster represents a cluster-node, its QoS epitome is used for the estimation. For unaggregated nodes, the estimating node has knowledge of the exact QoS measurements[10] and can use them directly.

Depending on the QoS metric, different approaches must be pursued to combine two estimations:

- Additive metrics, such as one-way delay or jitter are added up on a path. Consider a path with L nodes. If $n_{i,i+1}$ is the one-way delay between the i^{th} and the $i+1^{th}$ node, the total one-way delay is defined as shown in Equation (6.1). For example, if a path consists of three nodes that are connected by two links with a one-way delay of $50ms$ (for the link connecting node 1 and 2) and $95ms$ (for the link connecting 2 and 3), the one-way delay of the entire path is $50ms + 95ms = 145ms$.

- Restrictive metrics (also referred to as concave [87] or bottlenecked metrics), such as bottleneck bandwidth, are limited by the lowest value on a path. If $n_{i,i+1}$ is the bottleneck bandwidth between the i^{th} and the $i+1^{th}$ node, the total delay is defined as shown in Equation (6.2). For example, if a path consists of four nodes connected by three links with a bottleneck bandwidth of 4, 3 and 5 Mbps, the bottleneck bandwidth of the entire path is $min(4Mbps, 3Mbps, 5Mbps) = 3Mbps$.

- Multiplicative metrics, such as loss rate. These QoS metrics represent statistical values that are combined on a path using standard probabilistic procedures: If $n_{i,i+1}$ is the chance of loss occurring on the i^{th} and the $i+1^{th}$ node, the cumulative chance that loss occurs anywhere on the path is defined as shown in Equation (6.3). For example, on a path with of three nodes, connected by

[10]Unaggregated nodes are contained in the route in two situations: First, if a source cluster is broken down to the lowest level, it will contain the unaggregated nodes of the estimating node's cluster, which it has a full view of. Second, they may be introduced if a node of a transit or target cluster cooperates and supplies the estimating node with topology information down to its own cluster at the lowest level. In both cases, the estimating node also receives QoS measurements and epitomes along with the topology information.

two links that have a loss rate of 0.5 and 0.25, the loss rate for the entire path is $1.00 - ((1.00 - 0.5) \cdot (1.00 - 0.25)) = 0.625$.

$$n_{1,L} = \sum_{i=1}^{L-1} n_{i,i+1} \tag{6.1}$$

$$n_{1,L} = min(n_{1,2}, ..., n_{L-1,L}) \tag{6.2}$$

$$n_{1,L} = 1 - \prod_{i=1}^{L-1} (1 - n_{i,i+1}) \tag{6.3}$$

Algorithm 6.2 shows this process in pseudo-code. It uses the following high level functions:

- getRouteLength(r) returns the length of route r in hops.
- getClusterOnRoute(r, i) returns the identifier of the i^{th} cluster of route r.
- getAggregationLevelOf(c) returns the aggregation level of cluster c. Level 0 represents a node that is not aggregated.
- getQoSOnLink(i, j) returns the most favourable QoS metric of any links connecting i and j. This is the maximum value for restrictive QoS metrics, and the minimum otherwise. If i is "empty", it instead returns a neutral value that can be combined with other QoS values of the same metric without changing them. For additive QoS metrics such as delay or jitter, the neutral element is zero. For restrictive QoS metrics (such as bottleneck bandwidth) the neutral element is the maximum possible value representable by the data type used by the implementation. For multiplicative QoS metrics, it is either one (probability of no loss) or zero (probability of loss).
- combineQoS(e_1, e_2, e_3) combines the QoS estimates e_1 and e_2, as explained earlier. If e_3 is not empty, it combines the intermediate result from combining e_1 and e_2 with e_3. In either case, the final result is returned.
- getQoSForTraversal(a, b, c) returns an estimate for the QoS needed to traverse cluster a, if entered from cluster b and exited at cluster c. If c is *empty*, the worst QoS from the entry point b to any node in the cluster is returned. If b is *empty*, the worst QoS to exit point c from any node in the cluster is returned.

Algorithm 6.2 QoS estimation on route r.

```
1: EstimateQoSRoute($r$) : −
2: $p \leftarrow$ empty
3: $e \leftarrow$ getQoSOnLink(empty, empty)
4: $l \leftarrow$ getRouteLength($r$)
5: for $i \leftarrow 1$ to $l$ do
6:     $c \leftarrow$ getClusterOnRoute($r, i$)
7:     $qos_\lambda \leftarrow$ getQoSOnLink($p, c$)
8:     if $i = 1$ then                                        ▷ source-cluster
9:         if getAggregationLevelOf($c$) $\neq 0$ then
10:            $n \leftarrow$ getClusterOnRoute($r, i+1$)
11:            $qos_\chi \leftarrow$ getQoSForTraversal($c, empty, n$)
12:        end if
13:    else if $i = l$ then                                   ▷ target-cluster
14:        $qos_\chi \leftarrow$ getQoSForTraversal($c, p, empty$)
15:    else                                                   ▷ transit-cluster
16:        $n \leftarrow$ getClusterOnRoute($r, i+1$)
17:        $qos_\chi \leftarrow$ getQoSForTraversal($c, p, n$)
18:    end if
19:    $e \leftarrow$ combineQoS($e, qos_\lambda, qos_\chi$)
20:    $p \leftarrow c$
21: end for
22: return $e$
```

This process is repeated for each route that is under consideration. At this point, the source node now has a coarse estimate of what LoS it will likely encounter on each of the routes, without consuming any network resources.

6.4.2 Inter-Cluster Estimation Without a Given Route

In many situations, a route to the target may not yet be available. In order to find a suitable route, QoS estimation must be used even in the routing process.

As mentioned earlier, it is assumed that the node already knows the target node it wants to communicate with, and what part of the hierarchy it is located in. This information can be provided by the application, the overlay, or a distributed hashtable. A node's location is specified by concatenating the identifiers of its clusters from the bottom to the top hierarchy level. For example, QoS maps are currently used by a resource reservation algorithm [126]: There, the algorithm provides the node and its location. In other cases, the location may be returned together with the node's identifier by the overlay.

Algorithm 6.3 displays the original algorithm in pseudo-code. As demonstrated in Section 6.3.2, it first determines the lowest common hierarchy level in which both, the source and the target are contained in the same cluster (and thus share a view of that cluster). Depending on how much local information is available, the algorithm either uses QoS epitomes to make an estimation, or recurses and estimates the QoS at the next lower hierarchy level.

Algorithm 6.3 Multi-Level QoS estimation from node a to b.

```
1: EstimateQoS($a, b$) : −
2: $l \leftarrow$ getLowestCommonLevel($a, b$)
3: $r \leftarrow$ getRoute($a, b, l$)
4: $p \leftarrow$ empty
5: $e \leftarrow$ getQoSOnLink(empty, empty)
6: for $i \leftarrow 1$ to getRouteLength($r$) do
7:     $c \leftarrow$ getClusterOnRoute($r, i$)
8:     $qos_\lambda \leftarrow$ getQoSOnLink($p, c$)
9:     if $l = 0$ then                              ▷ $a$ and $b$ are in the same cluster
10:        $qos_\chi \leftarrow empty$
11:    else if $i = 1$ then                         ▷ source-cluster
12:        $b \leftarrow$ findBorderNode($a, c, l - 1$)
13:        $qos_\chi \leftarrow$ estimateQoS($a, b$)
14:    else if $i =$getRouteLength($r$) then        ▷ target-cluster
15:        $qos_\chi \leftarrow$ getQoSForTraversal($c, p, empty$)
16:    else                                         ▷ transit-cluster
17:        $n \leftarrow$ getClusterOnRoute($r, i + 1$)
18:        $qos_\chi \leftarrow$ getQoSForTraversal($c, p, n$)
19:    end if
20:    $e \leftarrow$ combineQoS($e, qos_\lambda, qos_\chi$)
21:    $p \leftarrow c$
22: end for
23: return $e$
```

The algorithm introduces three additional high level functions:

- getLowestCommonLevel(a, b) returns a number indicating the lowest hierarchy level l in which a and b (or any of the clusters they are in) are a member of the same cluster. For example, assume the cluster identifiers of a node are written in a list (each list element represents a hierarchy level, starting with level 0). If node a has the identifiers [1.2, 1.50, 1.75] and node b has the identifiers [2.1, 1.75, 1.75], then the lowest common level is level 2, where both nodes are in a cluster with the identifier 1.75. If a or b are empty, 0 is returned.

- getRoute(a, b, l) returns an array with cluster identifiers that represents a route r from a to b on hierarchy level l. Any common routing algorithm such as in [121] can be used to compute the route, and to provide alternative routes. As the cluster size is bounded by a constant, the complexity of finding this route is constant as well.

- findBorderNode(a, b, l) returns a node n in cluster a that borders cluster b on hierarchy level l. As every node in each cluster has full knowledge of its own cluster and its inter-cluster edges, this amounts to local search through a list with a constant number of nodes.

getRoute and findBorderNode may return alternative routes. Their existence can be conveyed to the application, which may request their computation, in case it is not satisfied with the first result or wants to explore additional routes.

In order to estimate the QoS from a source to a target node, the algorithm first locates the highest hierarchy level that contains both the cluster of the source and the cluster of the target node (see line 2) by comparing the aforementioned identifiers. As each node knows its own location in the hierarchy, it can now find possible routes to the cluster of the target node on that hierarchy level (line 3). Using any common routing algorithm is feasible despite the large number of nodes present in the overlay, as the cluster size and thus the number of nodes in the view of each node is bounded on every hierarchy level (and expected to be small).

Adaptations to Intra-Cluster Routing

The routing algorithm should be made aware of QoS epitomes and aggregation, to ensure that suitable routes are found. This is possible for many common routing algorithms, however it requires that it works with a *modified cost function*, which not only accepts two nodes (namely the current node, and a neighbouring node to calculate to the cost to), but also an "entry point", which the current node is entered from. In addition, it must be considered that the cost of a node may change of a different "entry point" is chosen. This is demanded by aggregation, which estimates the cost for routing through a cluster, from a given entry to a given exit node.

Existing algorithms can be adapted to this requirement in a straightforward fashion: A node n that has $e, e > 1$ outgoing inter-cluster links to a successor node s is split up into one node n' with no inter-cluster links to s, and e distinct nodes, each with only a single inter-cluster link to s. All other outgoing inter-cluster links from n are present in n' (except for the links to s), and any incoming inter-cluster links to n are also present in n' and the e distinct nodes. This process is then repeated on n', until no other successor node that receives more than one link from n'. This increases the complexity of the graph depending on the number of parallel inter-cluster links. As a result, it is important that the number of inter-cluster links is kept low.

The resulting route is again divided into the three clusters explained in the previous section. Transit and target clusters are treated as before. Source clusters, however, are broken down as follows: We set the new target node to any exit point of the source cluster that we consider, the QoS prediction algorithm is applied recursively. This process guarantees that the next hierarchy level we consider in the recursion will be at least one level lower than the previous one. Level 0 of the hierarchy represents the base case. It contains the unaggregated overlay of the node's own cluster that it has a full view of. Therefore, it uses the QoS metrics measured by the other nodes of the cluster and itself directly to obtain an estimate of that level.

6.4.3 The Adapted Algorithm

Note that Algorithm 6.3 is *indeterministic*: It assumes that the correct exit point from the source cluster is found, before breaking it down. This gave rise to the problem explained in Section 6.3.3: After moving to a lower hierarchy level, the algorithm may find itself bottlenecked, because all routes from the estimating node's cluster to the exit point are bottlenecked. In this case, it must move to a higher hierarchy level and find another route that uses a different node to exist the source cluster at.

A straightforward approach that does not require "backtracking" is to break up any source-cluster before running the routing algorithm. This results in a larger graph, however the increase is reasonable: First, the algorithm no longer needs to recurse and therefore the routing algorithm is only invoked once. Second, the graph size is still bounded, and only increases logarithmically with the total number of nodes in the overlay.

Algorithm 6.4 Multi-Level QoS estimation from node a to b.

```
1: EstimateQoSAdapted($a, b$) : −
2: $g \leftarrow$ GenerateCombinedView($a$)
3: $l \leftarrow$ getLowestCommonLevel($a, b$)
4: $t \leftarrow$ getClusterOfNodeAt($b, l$)
5: $r \leftarrow$ getRouteInGraph($g, a, t$)
6: $p \leftarrow$ empty
7: $e \leftarrow$ getQoSOnLink(empty, empty)
8: for $i \leftarrow 1$ to getRouteLength($r$) do
9:     $c \leftarrow$ getClusterOnRoute($r, i$)
10:    $l \leftarrow$ getLowestCommonLevel($p, c$)
11:    $qos_\lambda \leftarrow$ getQoSOnLink($p, c$)
12:    if $l = 0 \vee i = 1$ then                  ▷ source-cluster, or one of its nodes
13:        $qos_\chi \leftarrow empty$
14:    else if $i =$getRouteLength($r$) then               ▷ target-cluster
15:        $qos_\chi \leftarrow$ getQoSForTraversal($c, p, empty$)
16:    else                                               ▷ transit-cluster
17:        $n \leftarrow$ getClusterOnRoute($r, i + 1$)
18:        $qos_\chi \leftarrow$ getQoSForTraversal($c, p, n$)
19:    end if
20:    $e \leftarrow$ combineQoS($e, qos_\lambda, qos_\chi$)
21:    $p \leftarrow c$
22: end for
23: return $e$
```

Algorithm 6.4 shows the adapted estimation process. It uses the Generate-CombinedView function introduced in Algorithm 6.1 to generate a new view that combines all available local information. Note that combined view is valid for every subsequent estimation. Therefore, it does not need to be recomputed as long as the hierarchy itself does not change. In addition, a new function, getRouteInGraph(g, a, b) is introduced. It is identical to getRoute(a, b, l), except that it operates on the combined graph g instead of a specific hierarchy level l.

- getHierarchyLevels() returns the total number of levels in the hierarchy. A hierarchy with 3 levels consists of the levels $0, 1$ and 2.

- getLocalViewAt(l) returns a graph representing the hierarchy at level l as seen by the estimating node.
- getClusterOfNodeAt(a, l) returns the identifier of the cluster-node at level l that node a is contained in. At level 0, it returns node a itself.
- replaceClusterInGraph(g, c, d) searches graph g for cluster-node c. It then replaces cluster-node c with the graph d that it represents. Links from and to c are attached to the appropriate nodes in graph d. This is possible as every node has full knowledge of its own cluster and its inter-cluster edges, at every hierarchy level.

Using this estimate, node a now has a route and a high-level estimate to node b, using only locally available information.

6.4.4 Multi-Level QoS Estimation

There are two ways to increase the accuracy of this estimate. First, recall that the node performing the estimate usually does not have a view of the target cluster[11]. If the target node is cooperative, it can estimate the QoS from each entry point of its cluster to itself as described before, and transmit that information to the source node. At this point, the accuracy of the estimate is only limited by the accuracy of the QoS epitomes themselves.

In addition, with the cooperation from any node of any transit cluster, its next-lowest hierarchy level of said transit cluster can be considered in place of the QoS epitome. As the algorithm uses less aggregated information of the transit cluster, the accuracy of the estimation improves as well. This process is repeated for every transit cluster until the desired accuracy is reached. Note that level 0 contains the original overlay: Therefore, if the topology of any cluster is traversed down to level 0, it will contain the most accurate and current QoS information for that cluster. This kind of stepwise refinement could be used by a number of applications in a very reasonable

[11]The only exception is a trivial case, in which the node and the target node are in the very same cluster at level 0.

way. For example, a Video-on-Demand (VoD) system could, on the most coarse level, find a path, through which delivery can start with a minimal start-up delay. After having started the (batch of) stream(s), it can start a fine-grained QoS estimation in parallel. If a better path is found, the VoD application could switch to this. Clients could be totally unaware of this and be satisfied by fast start-up and good overall quality.

6.4.5 Complexity Analysis

In order to consider a large number of paths to a target node, a low computational complexity of the estimation algorithm is important. Of the high level functions introduced in the previous section, all but two operate within a cluster. As the number of nodes within a cluster is bounded by the constant s, the complexity of the intra-cluster functions is O(1). The following functions do not operate on a cluster:

GenerateCombinedView generates a combined view from the individual subgraphs visible to the estimating node on each hierarchy level. The replacement itself, as shown in Algorithm 6.1 on page 185 operates within a cluster. This replacement must be performed on every hierarchy level. Furthermore, when a cluster is replaced, each of the underlying levels must potentially be considered, in order to determine the border nodes the links terminate in. Therefore, the runtime has a complexity of $O(h^2)$, or, since the number of hierarchy level is logarithmic with respect to the number of nodes in the overlay, $O(\log^2 n)$. The size of the combined graph has a complexity of $O(\log n)$, as one cluster-node at each hierarchy level is replaced by the nodes it represents, which is bounded by the constant s.

GenerateCombinedView, and the routing itself use getLowestCommonLevel, which needs to compare the identifiers of both nodes in order to determine the lowest common level of both nodes. This operation is also linear with respect to the number of hierarchy levels, and thus terminates after $O(\log n)$ iterations.

Consider the pseudo-code in Algorithm 6.4. The loop, from lines 8–22 iterates through every hop on the route. While the number of nodes within the combined view has an upper bound that is logarithmic with respect to the number of nodes in the overlay, we must consider that clusters can be traversed multiple times. This is due to bottlenecks within a cluster that can be bypassed by entering the cluster from a different entry point.

The maximum amount of times each cluster can be traversed depends on the number of individual paths through the router. This number, in turn, depends on the number of nodes inside the cluster, which is bounded by a constant. Therefore, the number of traversals for each cluster-node is bounded by a constant as well.

Therefore, the loop from lines 8–22 is executed a limited amount of times that is bounded by logarithmically with respect to the number of nodes in the overlay. The functions executed within the loop have a complexity of O(1), except the call to GETLOWESTCOMMONLEVEL in line 10. The result of this call is only used to verify whether both nodes are in the same cluster at the lowest level. It can be replaced with a constant-cost function that checks only if the lowest cluster-identifiers of both nodes are identical.

If the common case of a hierarchy depth close to $\log_s n$ is assumed, the runtime complexity of the QoS estimation is $O(\log^2 n)$.

To have sufficient local information for QoS estimation, each node needs to store the identifiers and the epitomes of all nodes in its local view of the hierarchy, as illustrated in Figure 6.7. On the overlay level, level 0, each node stores the identifiers and the QoS metrics of all other $s-1$ nodes of their cluster instead of a QoS epitome. In all remaining hierarchy levels, a node stores up to s identifiers and s QoS epitomes from its local cluster, and up to s identifiers of the neighbour clusters that its own cluster borders (illustrated by dashed lines in Figure 6.7). Assuming the common case of $\log_s n$ hierarchy levels, each node therefore needs to store O($\log n$) identifiers, and O($\log n$) epitomes. Since identifiers and epitomes are likely to be very small, this approach scales very well with the number of nodes in the network.

Finally, note that the application may wish to use a specific route, based on its QoS estimation. In that case, the high level route should be included in the data packet as well. The increase in packet size is identical to the complexity of the high-level route determined earlier. Techniques such as DLDS may be used [127] to reduce the message size and the time needed for running the routing algorithm for subsequent packets.

6.4.6 Hierarchy Management

It is important to maintain the hierarchy after it has been generated. Churn changes the topology of the overlay itself, whereas changes in the original network (such as traffic that reduces the available bottleneck bandwidth, or a congested router outside the overlay's control that increases delay on a path) need to be reflected in the epitomes to ensure that the estimates remain accurate.

Hierarchy management is concerned with performing small, localized changes on the hierarchy in response to these changes. The goal is to avoid large-scale reorganization by confining these changes to as few clusters as possible, which can be reaggregated individually. This allows a *partial re-aggregation*, and avoids an expensive *full re-aggregation* of the entire overlay. Flocks follow a similar approach to [47], in that clusters are split if they would exceed the bounded cluster size s of the original aggregation, and merged if the number of members drops below $\lceil s/3 \rceil$. This section summarizes the approach of [47] and how it is used by the Flocks overlay.

Three cases must be considered: Nodes that join the overlay, node that leave the overlay, and QoS changes in the underlying network.

Leaving Nodes

A node that leaves the hierarchy causes all links connecting it to other nodes to be removed. Afterwards, the node itself is removed from the hierarchy.

If a cluster-head (as explained in Section 6.2.3) leaves their cluster, the cluster identifier must change to account for the fact that the leaving node may return, as cluster identifiers need to remain unique. This causes a mandatory map update, which is described below.

Mandatory and Optional Reaggregation

A mandatory map update needs to be sent out if a localized change in a cluster must to be propagated to other nodes, as it causes additional route changes beyond cluster boundaries. Without loss of generality, assume two clusters, a and b. The highest common level (in which both clusters, or their parent clusters, are contained) is l. In this case, the QoS epitome of both clusters is recomputed. The cluster heads of both clusters send the results are sent to all other nodes of their cluster in the next highest hierarchy level. This process continues until level l. Each node, except the nodes that the change originated from, also sends the update down to their children. The number of nodes that receive the message is s^{l+1}. As these updates may quickly reach a large amount of nodes, optimization is necessary. Literature suggests a logarithmic update frequency.

Optional map updates are processed identically to the mandatory map update described above, except that they may be skipped, as they only affect the accuracy of the QoS epitomes.

Link Changes

Links that are added or removed from the underlying network are also added and removed from the hierarchy. If a new inter-cluster link is created, or an existing inter-cluster link is removed, a mandatory map update is sent out. Intra-cluster links that are created do not necessarily generate a map update: If an intra-cluster link is created or removed, and the QoS epitome does not change, no actions are necessary. Otherwise, an optional map update is sent out.

A node may lose all links to its original cluster. If this occurs, it is treated as a node that leaves the overlay, and then joins it again at a different cluster. This process is explained below.

Sparse Clusters

A node that leaves may cause the cluster to become too spare. We consider a cluster too spare if the number of remaining cluster members n falls below a specified threshold t ($t/2 < t < s$). In response, we search our neighbouring clusters for clusters that are sufficiently small to accept all our nodes (their size must be at most $s - n$). These clusters are considered “merge candidates”. If at least one merge candidate is found, one cluster from the merge candidates is picked, and joined. Joining involves setting the cluster-identifier of our entire cluster to the identifier of the cluster we join. The previous cluster head becomes a regular node, and our previous cluster is no longer represented in the next highest hierarchy level. Links that connected to the previous cluster in the next highest hierarchy level now connect to the cluster we joined. In addition, the following must be considered:

- The QoS epitomes of our cluster and the target cluster need to be merged. This is a simple operation, in which we combine the QoS epitomes from both clusters. Paths between both clusters are no longer represented in the QoS epitome. Instead, paths entering from one cluster and exiting from the other have to be computed.

- The number of inter-cluster edges from our cluster e_a and the new cluster e_b are combined. Inter-cluster edges leading from our cluster to the new cluster ($e_{a \to b}$) no longer affect the QoS epitome.

- Previously, the number of paths included in the QoS epitome were $e_a \cdot (e_a - 1)$ for a, and $e_b \cdot (e_b - 1)$ for b. The combined cluster, c, has $(e_a + e_b) \cdot (e_a + e_b - e_{a \to b - 1})$ paths to include in the QoS epitome increases by $2e_a \cdot e_b - e_{a \to b}(e_a + e_b)$.

- $e_a + e_b - e_{a \to b}$ neighbour clusters are affected, and as each one of these may contain s nodes, up to $s \cdot (e_a + e_b - e_{a \to b} + 1)$ nodes need to be informed of the new cluster identifier. This number increases by the hierarchy level, as each one of the s nodes may be a cluster head for up to s leaf nodes. Therefore, on level $l, 0 <= l <= L$, QoS epitomes need to be sent out to up in a mandatory map update, to s^{l+1} nodes. New identifiers need to be sent out to $s^l \cdot (e_a + e_b - e_{a \to b} + 1)$ nodes, assuming that links between clusters are not aggregated[12].

- The merging process may recurse if merging two clusters causes their parent cluster (in the next hierarchy level) to become sparse.

If no suitable cluster is found, the algorithm is retried once more nodes depart, as the cluster may then be small enough to fit into another cluster. If a node departs a cluster that only consisted of that node, the cluster becomes extinct. Neighbouring nodes delete the cluster from their neighbours, and from the next-highest hierarchy level.

[12]Therefore, two inter-cluster links at level zero are represented as two inter-cluster links at level one.

Joining Nodes

A node that joins the underlying network queries its neighbouring nodes for their cluster identifiers, as well as the cluster size. It then assigns itself to the cluster with the fewest nodes. If more than one cluster has the same amount of nodes, the size of the clusters in the next highest hierarchy level is taken into account, to contain the effects of overfull clusters. If the joining node is connected to more than one cluster, new routes may have been introduced. In this case, a mandatory map update becomes necessary. The map update is optional if both clusters were neighbours before. This situation is detected without race condition by the cluster's root node.

Overfull Clusters

We define a cluster as overfull if its number of member nodes exceeds s. If none of the clusters neighbouring the joining node contain less than s nodes, the cluster picked by the joining node becomes overfull. In order to maintain a bounded cluster size, the cluster is split into two roughly equal-sized partitions. The partition without the root node chooses a new one. As with the original algorithm, the node with the highest identifier is used for that. This root node is added to the next highest hierarchy level, and the links of the previous cluster and itself are updated to reflect the new arrangement of links in the hierarchy. A mandatory map update is necessary to convey this topology change to the network.

Note that a new cluster is introduced to the next highest hierarchy. This cluster behaves the same as a joining node, picks a new cluster, and may cause it to become overfull as well. In this case, the process described above repeats at that hierarchy level.

6.5 Evaluation

The QoS map and the QoS estimations were evaluated through simulations in an overlay with 166 nodes. This section briefly explains the experiment setup, as well as external algorithms that were used during evaluation, presents the outcome, and concludes with a discussion of the results.

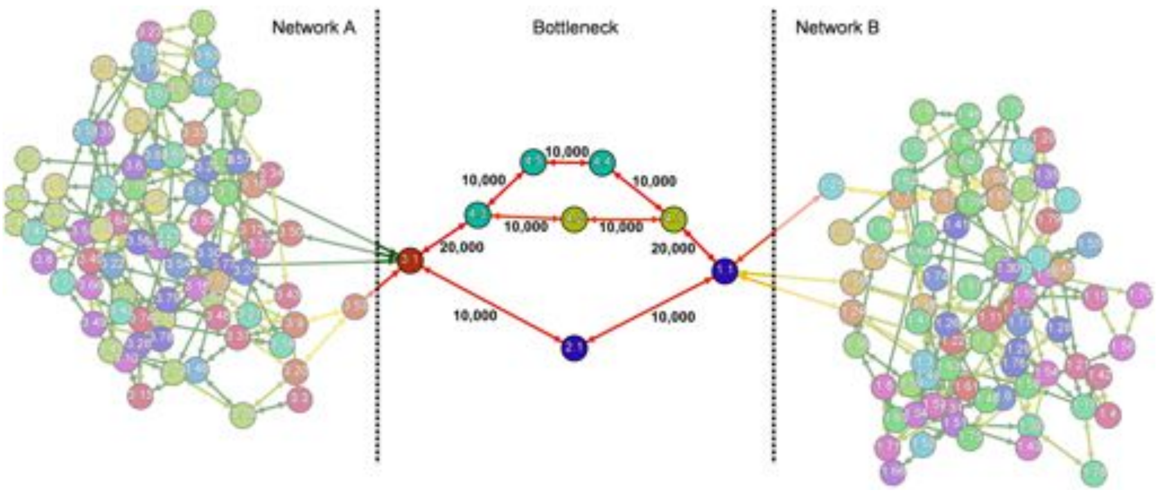


Figure 6.17: The overlay used for evaluating the QoS maps. The bottleneck and the two separate parts that reservation requests originated from, are highlighted. Portions of the overlay not relevant to the bottleneck are included for completeness, and have been faded out.

6.5.1 Experiment Setup

Overlay Configuration

The evaluation was performed on an overlay that consists of two randomly generated segments, with 80 nodes each. Both segments are connected by a bottleneck consisting of six nodes. Figure 6.17 shows the overlay taken from one of the simulations, highlighting the two segments (labelled as "Network A" and "Network B") and the bottleneck (labelled as "Bottleneck"). The total available bandwidth is indicated in black print on relevant links. Several aspects are should be noted: Three routes to traverse the bottleneck exist, each of which has a capacity of $10,000 KBit/s$ (the links to the top segment of the bottleneck have been assigned $20,000 KBit/s$, therefore the total capacity the bottleneck can provide is $30,000 KBit/s$). The two segments on either side of the bottleneck only serve to increase the size of the overlay, and the height of the hierarchical aggregation. As their exact configuration is not important for the experiment, they they are faded in the illustration.

In order to ensure a high hierarchy and a large numbers of clusters within each level, the cluster size limit s has been lowered to 5. This results in extensive use of aggregated information for routing and QoS estimation. During the simulation, random nodes reserve bandwidth for a data flow to a specific node on the other side of the bottleneck. Nodes in “Network A” make reservations to “1.1”, whereas nodes in “Network B” make reservations to “3.1”. As a result, every flow must cross the bottleneck.

As discussed in Section 6.4.4, it is possible to increase the accuracy of the estimation with the assistance of nodes from other clusters. This was not done in the simulations, so that each node must base its estimations and decisions only on locally available information.

REBOOK

I use a resource reservation protocol to determine the benefit (if any) the routes selected through QoS maps provide. This was done because resource reservation allows easy measurement of the following representative metrics:

- Rejected Flows: Resource reservation controls whether a new flow can be admitted (admission control). A flow is only be admitted if enough resources are available on a path to meet the requirements of the flow, otherwise the flow must be rejected. This metric is important because it detects any attempts by the overlay network to route a new flow over an already congested link.

- Committed Bandwidth: This number reflects the amount of bandwidth that has been reserved for an individual flow. The ideal amount of committed resources varies on a number of factors (for example, an important flow may be assigned resources with a higher priority; users of an Internet Service Provider that offers certain classes of services should not reserve more resources than the service they paid for permits), however if we consider a *fair* distribution of resources, each admitted flow should receive as many resources as possible.

- Unused Bandwidth: The unused bandwidth can be determined by summing up the committed bandwidth of all flows.

I selected the REBOOK resource reservation protocol [128] to obtain these metrics, as it is capable of reducing existing reservations under load.

Using REBOOK, the source node that starts a flow (to the target node) specifies the *minimum* amount of resources that it requires (R_{min}), and a *requested* amount of resources (R_{req}, which indicate the maximum amount of resources that the flow needs) in an initial reservation request packet. Each REBOOK-aware router on the path to the target node reads the reservation request packet and reserves the necessary resources: If the network is not congested and enough resources are available, the full amount is reserved. Otherwise, only the available amount is reserved, and the R_{cur} value in the reservation request packet is updated to reflect this reduction. If any router on the path cannot provide the minimum amount of bandwidth, the flow does not start and any resources that are reserved up to this bottlenecked router are released.

After the reservation request reaches the target node, a reservation acknowledgement is sent to the the source, which contains the actual amount of resources committed to the flow. Keep-alive packets are sent from the source to the target at regular intervals. They serve two purposes: First, a router can detect reservations that have not been released properly (for example, due to a failing node), due to the lack of keep-alive packets. Second, they communicate changes in allocated resources. To illustrate an example in which allocated resources change, consider that a small network with four nodes: the source (S), two routers (R1 and R2) and the target (T).

Figure 6.18 illustrates this example. The numbers shown on top of R1 and R2 represent the amount of available resources on that router. S attempts to reserve up to 10 resources[13], and it can accept a reduction to 2 resources. It therefore generates an appropriate reservation request (shown below R1) and forwards it to R1. R1 reserves the full amount, as it has 10 resources available. R2 only has 5 resources available, and therefore reserves 5 resources and records the reduced amount in the reservation packet (R_{cur}). Finally, T receives the reservation request, generates a reservation acknowledgement packet containing the final reserved amount of resources and sends it to S. At this point, S knows that 5 resources have been reserved for it, however R1 is still not aware of this fact. Once S generates keep-alive packets, it includes the current amount of reserved resource (5, in this case) in the packet, and forwards it

[13]Note that REBOOK is not limited to use bandwidth, but can rather be used to reserve any kind of resource.

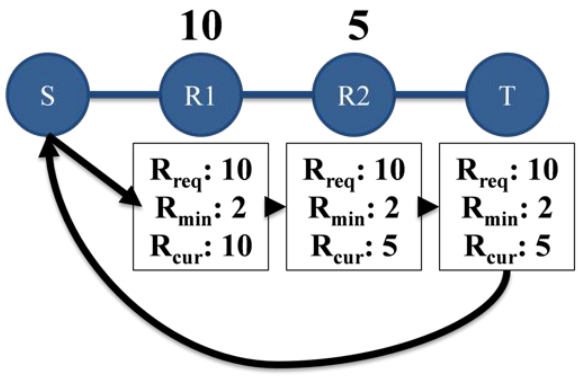


Figure 6.18: A REBOOK reservation request in a four-node bottlenecked network

to R1. As R1 is now aware that only 5 resources have been reserved, it updates its resource allocation table and releases the extraneous resources it allocated.

Once a flow is committed resources, REBOOK may change this commitment under certain circumstances. Should a router have insufficient resources to admit a new flow, it first rejects the reservation request, and then attempts to reduce the amount of resources allocated to existing flows (under the assumption that the rejected reservation will be re-attempted in future). When this happens, the change is not effective immediately[14], as sender and receiver are not aware of the change in allocated resources. Only once the sender transmits a keep-alive packet, the relevant routers will update the resource amount recorded in the keep-alive packet, and release the resources accordingly. As a result, resource release is never instant when a reservation request is rejected, but rather delayed until the affected flows sent a keep-alive packet. This is necessary to make the reservation process deterministic.

In addition, REBOOK also begins reducing the committed resources if a reservation can no longer be completely fulfilled (that is $R_{min} \leq R_{cur} < R_{req}$). This avoids a fully congested link, in which any new reservation request must be rejected, and ensures that the committed resources are redistributed fairly among the active flows[15].

[14]Therefore, the new flow must be rejected even if resources could be released.

[15]For example, consider a link with 10 resources. Two reservations arrive, each of which requests 6 resources. The first reservation is allocated the full 6 resources, whereas the second reservation receives only 4. In this case, the goal of a fair redistribution of resources is achieved if both flows receive 5 resources.

Finally, a sender can request a reservation upgrade. This is done by sending an “Upgrade Reservation Request”, in which case each router attempts to allocate additional resources to the flow (up to R_{req}). This change is effective immediately.

Simulation Setup

Two sets of simulations were performed to assess the effectiveness of QoS maps. Both sets use the hierarchical aggregation in order to perform routing. This is necessary, because the original Flocks overlay itself, as presented in Chapter 5 is only responsible for creating the overlay, and does not perform routing. Furthermore, the optimizations of the Flocks overlay were suspended during the experiment, in order to maintain the overlay configuration (with the bottleneck) and to allow evaluating the effects of QoS maps.

Each simulation lasted 2000 seconds. A new flow was started every 20 seconds, alternating between the two segments (Network A and B in Figure 6.17), and reserving resources to a node in the other segment[16]. Each flow initially requested $6MBit/s$, and could tolerate a reduction down to $100KBit/s$. Rejected reservation requests were not re-attempted[17]. This simplification is justified because new reservations are attempted throughout the simulation; therefore they will attempt to reserve the resources that the previously rejected reservation would have requested. Furthermore, note that every node in the experiment is also REBOOK-aware.

The simulator measured the number of admitted flows, and the average bandwidth committed to each flow after every simulated second.

In both sets of simulations, a node made its routing decisions based on whatever information it had available. In the first set, no QoS maps were available. Therefore, high level routing decisions within the aggregated topology were made based on the path length (short paths were preferred). Since Flock nodes have detailed information of their direct neighbours, decisions between neighbours are made based on bottleneck bandwidth.

In the second set, QoS maps were enabled and nodes chose the path with highest

[16]For example, after 20 seconds, a node from Network A attempts to reserve resources for a flow to a node in Network B. After another 20 seconds, a node from Network B then attempts to reserve resources for a flow to a node in Network A.

[17]Recall that REBOOK assumes a rejected reservation will be re-attempted in future.

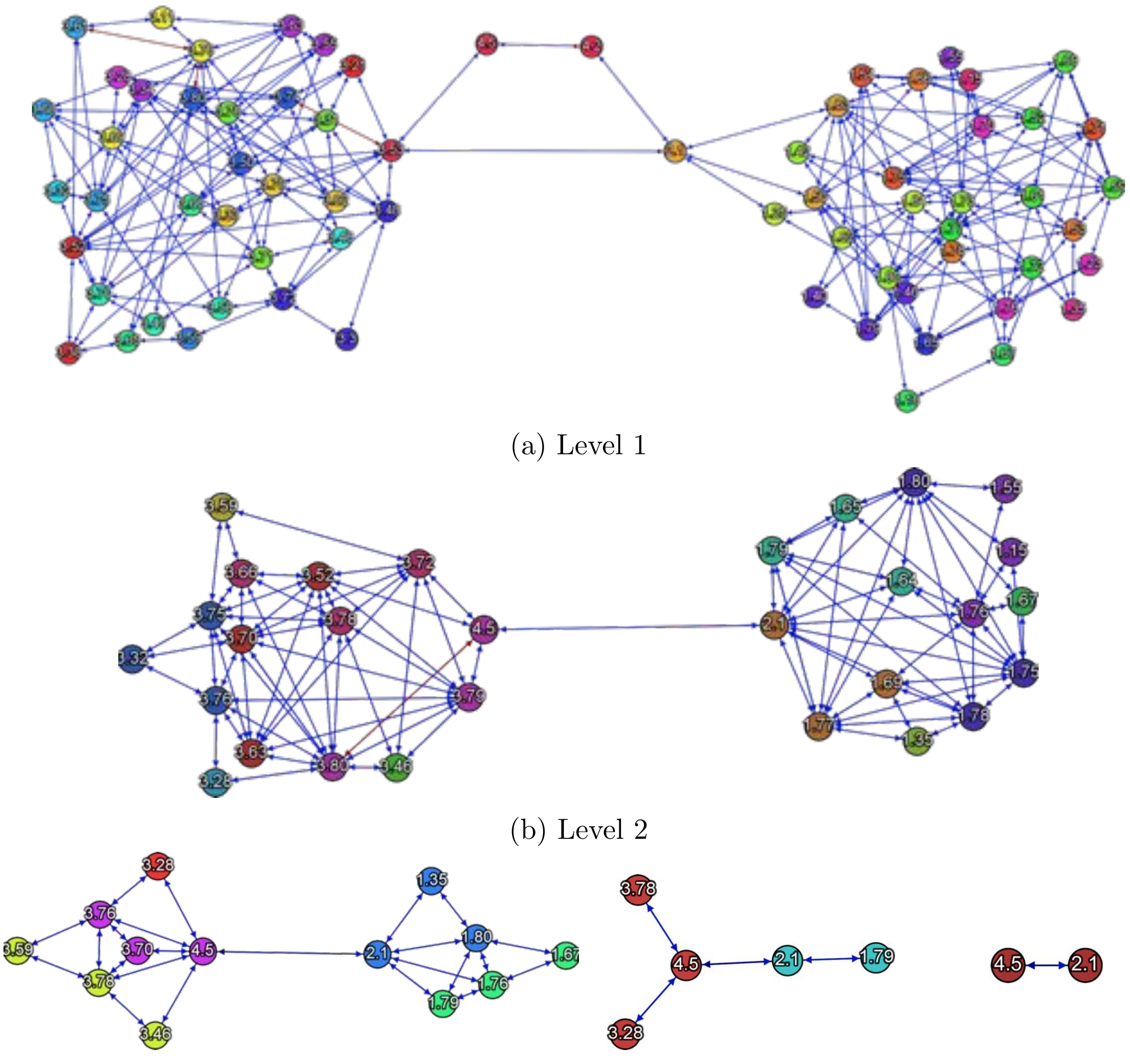
bottleneck bandwidth, based on the aggregated information available to them at the time of the decision. A node queried REBOOK for the amount of committed bandwidth (the sum of R_{cur} from all active flows crossing the node) and subtracted this information from the total link capacity, to determine the bottleneck bandwidth on the link.

As REBOOK attempts to reserve up to the requested amount of bandwidth, a link of $10,000KBit/s$ becomes fully loaded after two flows, and all links in the bottleneck become fully loaded after five flows. At this point, the remaining bandwidth is zero and even an up-to-date aggregation would not be useful to make routing decisions. However, once a new reservation is attempted, REBOOK reduces the resources allocated to the existing flows. Recall that the reduction in REBOOK does not occur immediately, as it has to be communicated to the sender. This has two implications: First, the new reservation will be rejected, even if all flows *could* theoretically be reduced to $100KBit/s$. Second, bandwidth will not be available until the sources of each flow send a REBOOK keep-alive packet, which gives REBOOK the opportunity to communicate the resource reduction to them. As a result, I found a delayed, event-based reaggregation approach, which triggers ten seconds after a reservation request cannot be fulfilled, suitable.

6.5.2 Experimental Results

Figure 6.17 presents the clusters at level 0, determined through the distributed partitioning algorithm. Each cluster is given a different colour. Note that the bottleneck consists of four different clusters. As a result, none of the nodes have a full view of how loaded the bottleneck is, and must rely on QoS maps to make their routing decisions. Figure 6.19 shows each individual hierarchy level, after a single flow has been reserved[18]. In order to perform estimations and route between "Network A" and "Network B", nodes must start at the highest level, in which "Network A" is represented by the left cluster-node, and "Network B" by the right one. They then break up the hierarchy of the network they are in, and gradually move to lower hierarchy levels.

[18]This is the reason for the single congested path in the graph.



(a) Level 1

(b) Level 2

(c) Level 3

(d) Level 4

(e) Level 5

Figure 6.19: The aggregated representations of the overlay in Figure 6.17.

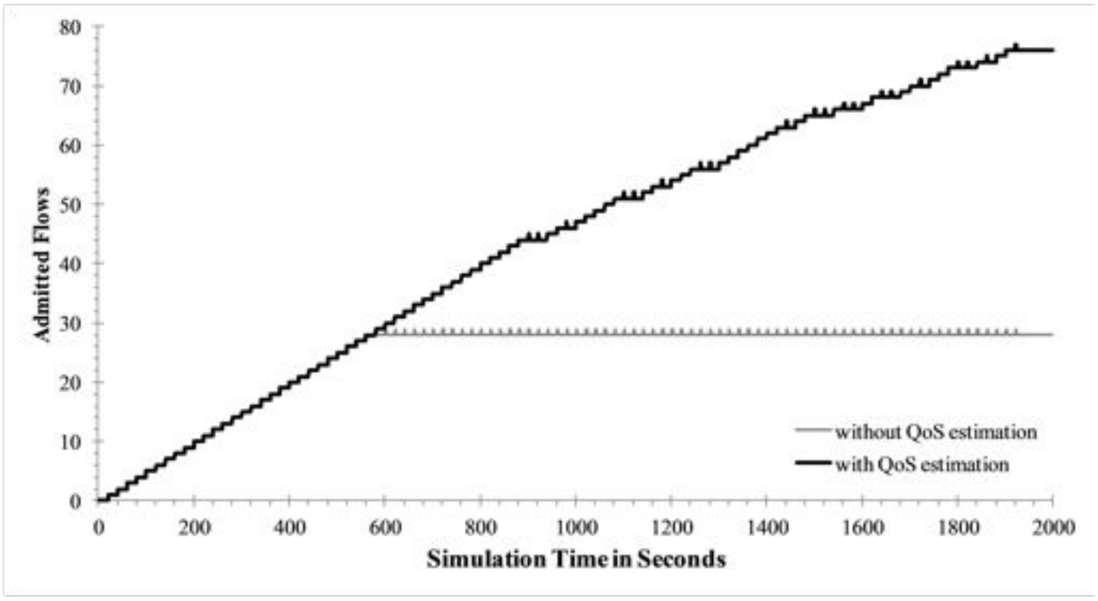
Level 1, shown in Figure 6.19a, is of particular interest because at this stage, the nodes at the bottleneck (represented by the cluster-nodes 4.5 and 2.1 at level 2, shown in Figure 6.19b) make a first decision on how to route across it. The bottom path represents the path 3.1 →2.1 →1.1, whereas the top path leaves a choice between 3.1 →4.3 →4.5 →4.4 →4.1 →1.1 and 3.1 →4.3 →4.2 →4.1 →1.1 at level 0. Therefore, it is critical that an accurate estimate of the load in the bottleneck is available on this level. If the bottom path is chosen, the second routing decision concerned with the bottleneck is made at level 0.

Figure 6.20 presents the results: Figure 6.20a compares the number of flows that are currently admitted at any point during the experiment. These consist of reservations that have been initiated through REBOOK's SENDER_RESERVATION_REQUEST function, and have not received a "reset" notification (which would indicate a request that cannot be fulfilled). As a result, new reservation attempts can be observed in the graph even if the reservation itself did not succeed, and are represented by small spikes. Figure 6.20b illustrates the bandwidth that each of the admitted flows receives in average.

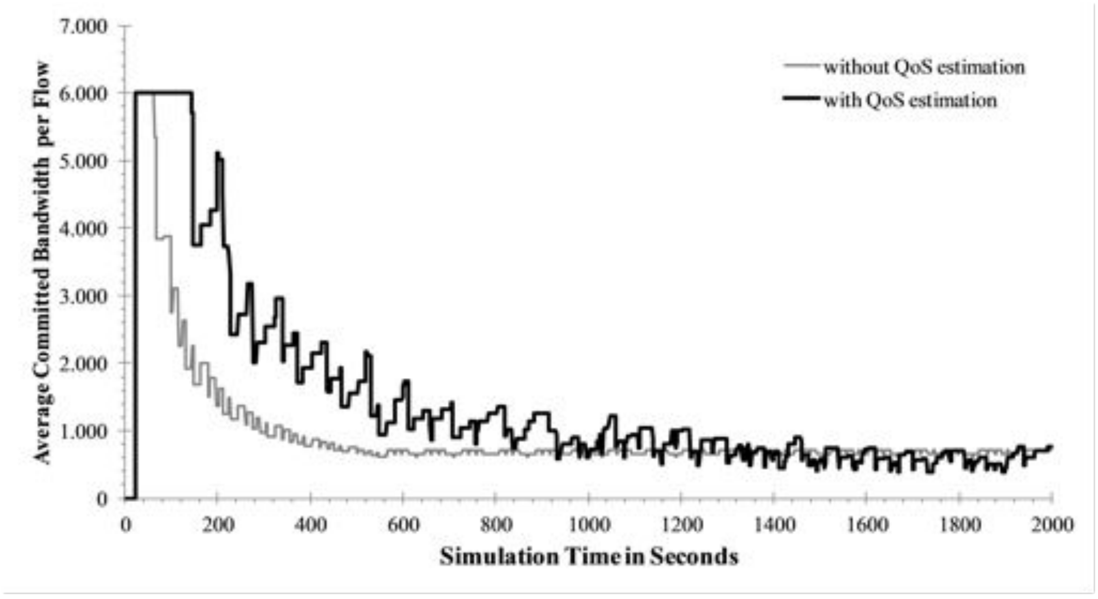
Both graphs allow several interesting insights: An initially surprising result is that few reservations are rejected initially, even if no QoS maps are used. This effect is accomplished by REBOOK, which, as described in Section 6.5.1, begins reducing resources as soon as a new flow can no longer be granted all the resources it requested. As a result, a large number of flows can be accommodated even on congested paths, before any new flows have to be rejected. Figure 6.20b confirms this: Even though no flows are rejected, the average committed bandwidth decreases sharply after only a few flows.

Notice that at the same time, a higher amount of bandwidth is maintained for the simulation set using QoS maps, as nodes identify alternative paths through the bottleneck, and new reservations are made on less congested paths. Without QoS maps, the decision at level 1 did not consider the upper path, due to its apparent greater length.

Even though the bottleneck could, in theory, accommodate every flow at reduced service, flows are still dropped despite the use of QoS estimations. This is caused by REBOOK, which must reject a flow if not enough bandwidth is available on a path.



(a) Admitted Flows



(b) Committed Bandwidth

Figure 6.20: The effect of using QoS maps on the overlay shown in Figure 6.17.

Recall from Section 6.5.1 that REBOOK already begins to reduce resources if a reservation can no longer be fulfilled completely. While this was the case here as well (note the reductions, and the stepwise increases in Figure 6.20b), the released resources had already been redistributed to existing flows by the time the new reservation arrived, resulting in the new reservation being rejected.

It must be noted that for the experiments detailed in this section, stale QoS epitomes would eventually cause the same effect, even if REBOOK did not reject the flows[19]. This can be countered with reasonable assumptions on when a re-aggregation should be performed. In this simulated scenario, REBOOK and the re-aggregation synergize: Once insufficient resources are available, REBOOK must reject a new flow, but at the same time releases resources. This is an optimal time to perform a reaggregation, which allows the bottleneck to be detected, and the flows to be routed past it.

A desirable result that can be seen by observing both graphs is that not only does the number of admitted flows increase by utilizing QoS maps, but for a significant time the average bandwidth committed to each individual flow is higher as well. This is because every individual path through the bottleneck has been identified, and all its available bandwidth is used.

The average committed bandwidth with QoS maps (labelled G_{QoS} from now on) only drops below the average without QoS maps (labelled G_{Orig}) in the second half of the experiment, first starting after 931 seconds. At that point, two reservation requests get rejected (one originating from Network 1, the other one from Network 2), resulting in REBOOK releasing resources in both directions[20] of the bottleneck, and causing a drop in the average bandwidth. However, the same reservations were also rejected in G_{Orig}. This raises the question why a rejected reservation caused the average committed bandwidth of G_{QoS} to drop below the average of G_{Orig}.

A closer investigation revealed that the rejection took place for a flow routed through the top segment of the bottleneck (3.1 $\rightarrow$ 4.3 $\rightarrow$ 4.5 in Figure 6.17). As the link from 3.1 $\rightarrow$4.3 has a higher bottleneck bandwidth ($20,000 KBit/s$) than the

[19]Recall that only rejected flows trigger a re-aggregation in the affected clusters. If a re-aggregation does not happen, the aggregated information that indicates the available bandwidth through a cluster becomes inaccurate as flows are admitted.

[20]from Network 1 to Network 2, and Network 2 to Network 1.

bottom segment of the bottleneck (3.1 $\rightarrow$ 2.1, with $10,000KBit/s$, which is the only path used by G_{Orig}), a larger amount of resources was released. As a result, the average committed bandwidth of G_{Orig} temporarily dropped below the average of G_{Orig}, until the subsequent reservation upgrade requests described in Section 6.5.1 recover any excess released resources.

Another noteworthy feature of the graph is that the average bandwidth of G_{QoS} increases slightly at the end of the experiment. At this time, no more flows arrive, and therefore the reservation upgrades allocate any resources that REBOOK pre-emptively released to the flows. In fact, the average committed bandwidth with QoS map is higher than the average without QoS, however this bears no additional meaning (additional flows that are accepted will continue the trend that can already be observed Figure 6.20b, namely that the average bandwidth allocated to each flow decreases).

In Figure 6.17, each link's colour represents the unused bandwidth on it (a red link is fully loaded) at the end of the simulation. It can be seen that if QoS maps are used, every link within the bottleneck is loaded, and there is no unused capacity. The attentive reader may have noticed that some of the links leading to 3.1 (namely, the top three links coloured green) are not fully loaded. This is because most of the flows are routed through the bottom link leading to 3.1, as it is the shortest path in higher hierarchy levels. While the shortest path in an aggregated topology is not a good measure (as explained previously, it may be longer in less aggregated levels), it was chosen as a tie-breaker for simplicity if more than one path to 3.1 had the same (best-case) bottleneck bandwidth. As a result, paths that are longer in the higher aggregated levels and still bottlenecked at 3.1 were not be considered. This behaviour can be improved by including the (true) hop-count for the highest bottleneck-bandwidth in the aggregation.

6.5.3 Classification of the Hierarchical Aggregation

[1] presents a detailed classification scheme for QoS-aware topology aggregation techniques. The classification scheme is then used to compare a comprehensive list of existing techniques. Some of the criteria do not directly apply to the aggregation

used by in this thesis, as Flocks do not limit themselves to the use of a specific QoS epitome. For example, from the aggregation categories, one implementation of the Flocks overlay may use a symmetric star representation for topology aggregation, whereas another may use a full mesh representation. The complexities therefore depend on the representation used, not the Flocks overlay.

Other criteria, however, are covered in this thesis, and can be classified by the scheme presented in [1]. Table 6.5 lists all criteria, regardless of whether it applies to the Flocks overlay, the classification, and a brief explanation.

Criteria	Classification	Notes
Category	Varies	[21]
Strategy	D/C	Distributed and Centralized[22]
Probabilistic?	Varies	Depends on the QoS epitome generation method used.
Symmetric?	Varies	[23]
QoS Param. Type	Varies	[24]
Selection Criteria	Varies	Depends on the topology transformation method used.
Precision	Varies	[25]
Time Complexity	$O(\log V) + X$	[26]

(continued on next page)

[21]The Flocks overlay does not limit itself to a specific topology aggregation category, and any of the topology transformation methods presented in [1] can be used. In the experiments presented in this thesis, the Full Mesh transformation method has been used, in which the QoS between each pair of border nodes is aggregated. However, the Flocks overlay is not limited to this transformation method.

[22]The overlay with $|V|$ nodes is divided into distinct clusters using the partitioning algorithm presented in Section 6.2.3, which happens in a fully decentralized fashion. Within a cluster, where the number of nodes is bounded by the constant s, the epitome is computed in a centralized fashion.

[23]Depends on the QoS epitome generation method used; if the QoS epitome supports asymmetric links, the Flocks overlay makes use of the information, resulting in better estimates.

[24]Depends on the QoS epitome generation method used; the Flocks overlay also supports multiple (separate) epitomes per cluster and can make use of it.

[25][1] only distinguishes between a precise and an imprecise aggregation, but not how accurate an imprecise aggregation is. For Flocks, the precision and the accuracy of the aggregation depends on the topology transformation method that is used. Note that Flocks provide support to break down aggregated clusters during the estimation process, to increase the accuracy of the estimation. If each cluster is broken down to the lowest level, the estimation is always precise.

[26]X represents the complexity of the QoS epitome generation method (B^2 for the Full Mesh). The

Table 6.5: (continued)

Criteria	Classification	Notes
Decode Complexity	Varies	[27]
Reaggregation	Partial	[28]
Advertisement Complexity	Varies	[29].
Inter-AS Table Complexity	$O(X \cdot \log V)$	[30]
Path Selection	HSR	Hierarchical Source Routing
Update Trigger	Threshold	[31]
Network Dynamics	Yes	[32]
Evaluation Metric	Various	[33]

Table 6.5: Classification of the Hierarchical Aggregation according to the criteria in [1].

6.6 Discussion

This section contributed two features to the Flocks overlay that I consider important for multimedia applications: Hierarchical QoS-aware routing and QoS estimations. This is accomplished through a novel concept, QoS maps, a decentralized hierarchical approach to perform scalable QoS estimation in large scale overlays. It is based on a decentralized version of the Basic Partition algorithm [33], which must be modified in

generation of the hierarchy, including the partitioning has a complexity of $O(\log V)$ in the average, and $O(V^2)$ in the worst case.

[27]Depends on the topology transformation method used (Full Mesh does not need to be decoded).

[28]Reaggregation is only performed in clusters that are affected by the change. Unchanged clusters are not reaggregated.

[29]Depends on the topology transformation method used (Full Mesh has an advertisement complexity of $O(B^2)$)

[30]X represents the spacial complexity of the QoS epitome generation method that is used (Full Mesh has a spacial complexity of $O(B^2)$. Border nodes must store the epitomes for up to s clusters on each hierarchy level, whereas s is the constant representing the upper bound of (cluster-)nodes in a (super-)cluster.

[31]Reaggregation takes place once the changes in QoS exceed a threshold selected by the application.

[32]The routing in the simulation adapts to load, avoiding bottlenecked links and thus increasing the number of accepted requests, as well as the average bottleneck bandwidth available to each request.

[33]SR (Success Ratio), BR (Blocking/Rejection Ratio) and Average Bandwidth per Flow are used in the evaluation.

order to provide suitable partitions for topology aggregation. With the modifications illustrated in this section, Flocks are able to aggregate the topology of large overlay into a hierarchy with small cluster sizes and low depth, while benefiting from the low runtime and small message sizes of the original Basic Partition algorithm.

QoS estimation is performed by a multi-level estimation algorithm, which uses the QoS epitomes stored in the QoS maps to predict QoS on any path between two nodes, starting with only local information. A large number of routes can be evaluated in little time, due to the low runtime complexity of $O(\log^2 n)$. This allows an application to chose the most beneficial route from many alternative routes. Since the QoS estimation algorithm starts at the highest hierarchy level, the initial alternative routes cover large parts of the overlay. Route changes become more and more localized, as lower layers are evaluated in greater detail, and infeasible routes are discarded.

The algorithms and the QoS maps themselves were evaluated through extensive simulations, performed with a resource-reservation protocol capable of dynamic resource reduction [128]. By using QoS maps, Flocks are able detect and bypass bottlenecks independent of the size of the overlay. Furthermore, they are able to give a multimedia application an estimate of the QoS likely encountered on the path, which could then, for example, use reasonable parameters when requesting resources such as a video stream.

CHAPTER 7

Conclusion

This thesis has investigated flexible self-organizing overlay networks for multimedia data dissemination at their lowest level: the optimization of each nodes neighbourhood based on their content- and QoS-related requirements.

The concept of a self-organizing overlay network is not new, and a lot of work exists in that field. Several overlays operate at a similar low level as discussed in this thesis. They determine suitable neighbours through matchmaking, in which a third node connects two previously unconnected nodes based on how closely they match. The closeness of the match is determined by a general oracle function, which must be implemented by the application using the overlay.

QoS-aware overlays have received a large amount of scientific attention as well. They are optimized against certain metrics, such as a minimal delay, or a maximum bandwidth. However, to the best of my knowledge, no work has previously been presented that combines the generality achieved through matchmaking, with QoS-awareness. Even more, QoS-awareness, as discussed in this thesis, is fully integrated into the oracle: an application implementing the oracle therefore needs to no longer be aware of a difference between content- and QoS-related properties, and is able to treat them equally.

Because of the focus on QoS-awareness, traditional matchmaking approaches could not be used. The routes from any two nodes to a target node may differ substantially, and as a result, an overlay must treat the routing in the underlying network like a black-box and must not make any assumptions on the QoS measured by other nodes. Therefore, this thesis introduced a novel approach: a node no longer relies on a third node as matchmaker. Instead, it announces its properties ("what I have") to nodes in a limited radius around it, and uses its interest oracle ("what I want") to find a good match in other nodes that send it their properties. As the node is performing the match itself, it can connect to potential neighbours and thus base its decisions on the actual QoS available to it based on the routing of the underlying network, without being aware of what these routes actually are.

A prototype, the Flocks overlay, has been developed, which implements the concepts presented in this thesis. This prototype provides the same flexibility as existing self-organizing overlays using oracles, but is more general, as it additionally allows the consideration of QoS-related properties, which is not possible with matchmaking based approaches. Its flexibility has been illustrated by modelling several existing distribution approaches, using only the aforementioned interest-property concept. Each of the approaches has been been extended with QoS-awareness, by making simple modifications to the interests it bases on.

The robustness, scalability and flexibility of the Flocks overlay have been evaluated through extensive simulations, and experiments under "real" Internet conditions on the PlanetLab. They have demonstrated that the Flocks overlay quickly recovers from large scale failures, in which 40% of all nodes in the overlay fail at the same time, and efficiently reorganize the overlay on-the-fly if the interests or properties of its nodes change. Finally, the evaluations confirm that the protocol overhead from exchanging properties and discovering neighbours is very modest, and that the overhead can be adjusted by the nodes according to their needs.

An important area of improvement was identified during the evaluations: in practice, a node that considers the suitability of a neighbour is not only interested in the LoS to that neighbour, but the LoS a route over this neighbour provides to certain other nodes in the overlay (for example, the nodes providing relevant multimedia data). For this purpose, "QoS maps" have been developed, implemented and evaluated. QoS maps allow nodes to perform fast and scalable estimations of the QoS to any other node in the overlay, using only local information. With the cooperation of other nodes, and a small amount of additional network traffic, these estimations can later be refined if greater accuracy is desired. In order to scale to large overlays, QoS maps are built using existing hierarchical topology aggregation methods. To partition the overlay for topology aggregation, the Basic Partition algorithm has been used, which permits fast, decentralized partitioning of the overlay graph at a sublinear runtime cost.

During the implementation, several shortcomings have been identified in the Basic Partition algorithm: its stopping conditions required knowledge of global overlay parameters, and the algorithm was unable to deal with hubs - nodes with a a large number of neighbours - if the partition size was bounded. As a result, several modifications have been made to the algorithm, and their impact has been evaluated thoroughly in simulations. The results show that the modified algorithm successfully bounds the cluster size, with only a modest increase in hierarchy height and runtime cost, and successfully partitions even large hubs.

Evaluations of QoS maps have shown that they permit an application to identify bottlenecks ahead of time, and discover suitable alternative routes without a global view.

During the course of this thesis, I came across a number of questions and extensions that remain subject of further research efforts:

- A distributed hashtable, such as Chord, could be used to locate content in the Flocks overlay. Another interesting approach, however, could use an aggregation scheme, similar to the QoS epitomes used by QoS maps, except that it aggregates properties instead. Using this aggregated "content map", nodes that join, or that change their interests the speed at which matching neighbours are found may be increased.

- The influence model [103] discussed in Section 2.4.3 was considered as an alternative to performing distributed partitioning, as it partitions the overlay at "weak" links. Its partitions, therefore, promise to reflect any clustering of nodes in the overlay. However, its partition size is not bounded, and it is unclear how well the algorithm scales with graph size. As a result, the Basic Partition was used by this thesis. An intriguing extension could combine both partitioning approaches, using the Basic Partition to bound the partition size, and then applying the aforementioned influence model to further divide them on weak links. Consider a partition created by the Basic Partition on an overlay optimized based on network delay. The influence model may further divide this partition into sub-partitions of nodes with similar delay. This would provide an improvement to QoS maps, as the error caused by the aggregation of nodes with similar values is smaller.

- The pace at which the Flocks management protocol searches for new neighbours could be adjusted on-the-fly. If a large amount of bandwidth is available, the Flocks overlay can use this bandwidth to optimize the overlay more quickly. If bandwidth is needed by the application, the optimization speed could be slowed down, reducing the traffic it causes.

- The partitioning algorithm, and hence the topology aggregation introduced in this thesis are not resilient against failure. A node that leaves the network during the partitioning causes the partitioning algorithm to stall, as nodes are indirectly synchronized with each other. Currently the partitioning algorithm is restarted for clusters that are stalled, however more efficient approaches (such as a partial restart) may be possible.

- Reaggregation of QoS epitomes is a well studied problem, and considerable work discusses how and how often reaggregation should be performed. However, Flocks have more control than overlays generally considered in these discussions. For example, a Flock node that replaces one of its neighbours with another could estimate the impact of this change. If a change has little effect on the overlay, or if more changes are expected, reaggregation could be delayed.

- A user-interface could allow easy experimentation with the interest-property concept, and the grammar introduced by this thesis, and visualize the overlays generated by user-entered interests and properties.

Bibliography

[1] S. Uludag, K. Nahrstedt, K.-S. Lui, and G. Brewster, “Comparative analysis of topology aggregation techniques and approaches for the scalability of QoS routing,” DePaul University, Tech. Rep. TR05-010, 2005.

[2] H. Wedde, M. Farooq, and Y. Zhang, “Beehive: An efficient fault-tolerant routing algorithm inspired by honey bee behavior,” in *Proceedings of the 4th International Workshop on Ant Colony Optimization and Swarm Intelligence (ANTS 2004)*, ser. Lecture Notes in Computer Science, M. Dorigo, M. Birattari, C. Blum, L. M. Gambardella, F. Mondada, and T. Stützle, Eds., vol. 3172. Springer, 2004, pp. 83–94.

[3] S. Uludag, K.-S. Lui, K. Nahrstedt, and G. Brewster, “Analysis of topology aggregation techniques for QoS routing,” *ACM Computing Surveys*, vol. 39, no. 3, 2007.

[4] Y. Wan, S. Roy, A. Saberi, and B. Lesieutre, “A stochastic automaton-based algorithm for flexible and distributed network partitioning,” in *Proceedings of the Swarm Intelligence Symposium (SIS 2005)*, 2005, pp. 273–280.

[5] D. Peleg, *Distributed computing: a locality-sensitive approach.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000.

[6] R. Stöckli. NASA goddard space flight center image. [Online]. Available: http://visibleearth.nasa.gov/view.php?id=57752

[7] R. Steinmetz and K. Wehrle, *Peer-to-peer systems and applications*, 17th ed. Springer, 2005.

[8] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris, “Resilient overlay networks,” in *Proceedings of the 18th ACM symposium on Operating systems principles (SOSP 01).* New York, NY, USA: ACM, 2001, pp. 131–145.

[9] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, “Peer-to-peer computing,” HP Laboratories Palo Alto, Tech. Rep. HPL-2002-57 (R.1), 2002.

[10] I. T. Union, “Definitions of terms related to quality of service,” *ITU-T Recommendation*, Aug. 1994. [Online]. Available: http://www.itu.int/rec/T-REC-E.800/en

[11] C. Aurrecoechea, A. T. Campbell, and L. Hauw, “A survey of QoS architectures,” *Multimedia Systems*, vol. 6, pp. 138–151, 1998.

[12] S. Akhshabi, A. C. Begen, and C. Dovrolis, “An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP,” in *Proceedings of the 2nd annual ACM conference on Multimedia systems (MMSYS 11).* New York, NY, USA: ACM, 2011, pp. 157–168.

[13] A. Rowstron and P. Druschel, “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems,” in *IFIP/ACM Int. Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001, pp. 329–350.

[14] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, “SCRIBE: The design of a large-scale event notification infrastructure,” in *Proceedings of the 3rd International COST264 Workshop (NGC 01).* London, UK: Springer-Verlag, 2001, pp. 30–43.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” *SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[16] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz, “Tapestry: a resilient global-scale overlay for service deployment,” *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, Jan. 2004.

[17] M. Brogle, D. Milic, and T. Braun, “QoS enabled multicast for structured P2P networks,” in *Proceedings of the 4th IEEE Consumer Communications and Networking Conference (CCNC 2007)*, Jan. 2007, pp. 991 –995.

[18] A. Legout, N. Liogkas, E. Kohler, and L. Zhang, “Clustering and sharing incentives in BitTorrent systems,” in *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS ’07. New York, NY, USA: ACM, 2007, pp. 301–312.

[19] P. Shah and J.-F. Paris, “Peer-to-peer multimedia streaming using BitTorrent,” in *Proceedings of the IEEE International Conference on Performance, Computing, and Communications (IPCCC 2007)*, Apr. 2007, pp. 340 –347.

[20] A. Vlavianos, M. Iliofotou, and M. Faloutsos, “BiToS: Enhancing bittorrent for supporting streaming applications,” in *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM 06)*, Apr. 2006, pp. 1 –6.

[21] C. Dana, D. Li, D. Harrison, and C.-N. Chuah, “BASS: Bittorrent assisted streaming system for video-on-demand,” in *7th IEEE Workshop on Multimedia Signal Processing*, Nov. 2005, pp. 1 –4.

[22] C. Wu, B. Li, and S. Zhao, “Exploring large-scale peer-to-peer live streaming topologies,” *ACM Transactions on Multimedia Computing Communications and Applications*, vol. 4, pp. 19:1–19:23, Sep. 2008.

[23] D. Ciullo, M. Garcia, A. Horvath, E. Leonardi, M. Mellia, D. Rossi, M. Telek, and P. Veglia, “Network awareness of P2P live streaming applications: A measurement study,” *IEEE Transactions on Multimedia*, vol. 12, no. 1, pp. 54 –63, Jan. 2010.

[24] G. D. Caro and M. Dorigo, “AntNet: Distributed stigmergetic control for communications networks,” *Journal of Artificial Intelligence Research*, vol. 9, pp. 317–365, 1998.

[25] S. S. Aman, M. R. Akbarzadeh-Totonchi, and M. Naghib zadeh, "A novel approach to distributed routing by Super-AntNet," in *Proceedings of the 10th IEEE Conference on E-Commerce Technology (CEC 2008)*, 2008, pp. 2151–2157.

[26] E. Di Nitto, D. Dubois, and R. Mirandola, "Self-aggregation algorithms for autonomic systems," in *Bio-Inspired Models of Network, Information and Computing Systems, 2007. Bionetics 2007. 2nd*, Dec. 2007, pp. 120 –128.

[27] F. Saffre, R. Tateson, J. Halloy, M. Shackleton, and J. L. Deneubourg, "Aggregation dynamics in overlay networks and their implications for self-organized distributed applications," *Computer Journal*, vol. 52, pp. 397–412, Jul. 2009.

[28] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: a decentralized network coordinate system," in *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '04. New York, NY, USA: ACM, 2004, pp. 15–26.

[29] M. Costa, M. Castro, R. Rowstron, and P. Key, "PIC: practical internet coordinates for distance estimation," in *Proceesings of the 24th International Conference on Distributed Computing Systems, 2004*, 2004, pp. 178–187.

[30] P. Sharma, Z. Xu, S. Banerjee, and S.-J. Lee, "Estimating network proximity and latency," *SIGCOMM Computer Communication Review*, vol. 36, pp. 39–50, Jul. 2006.

[31] G. Wang, B. Zhang, and T. S. E. Ng, "Towards network triangle inequality violation aware distributed systems," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007, pp. 175–188.

[32] S. Guha, N. Daswani, and R. Jain, "An experimental study of the Skype peer-to-peer VoIP system," in *Proceedings of The 5th International Workshop on Peer-to-Peer Systems (IPTPS 2006)*, 2006.

[33] B. Derbel, M. Mosbah, and A. Zemmari, "Sublinear fully distributed partition with applications," *Theory of Computing Systems*, vol. 47, no. 2, pp. 368–404, 2010.

[34] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[35] S. Ion. (2003, Apr.) Lecture Notes CS 268. [Online]. Available: http://inst.eecs.berkeley.edu/ cs268/sp03/notes/Lecture19.ppt

[36] H. Zimmermann, "The ISO model of architecture for open systems interconnection," in *Innovations in Internetworking.* Norwood, MA, USA: Artech House, Inc., 1988, ch. OSI reference model, pp. 2–9.

[37] D. Clark, B. Lehr, S. Bauer, P. Faratin, R. Sami, and J. Wroclawski, "Overlay networks and the future of the internet," *Communications and Strategies*, pp. 109–130, 2006.

[38] N.-F. Huang, Y.-J. Tzang, H.-Y. Chang, and C.-W. Ho, "Enhancing p2p overlay network architecture for live multimedia streaming," *Inf. Sci.*, vol. 180, pp. 3210–3231, Sep. 2010.

[39] N. Magharei, R. Rejaie, and Y. Guo, "Mesh or multiple-tree: A comparative study of live P2P streaming approaches," in *26th IEEE International Conference on Computer Communications (INFOCOM 2007)*, May 2007, pp. 1424 –1432.

[40] C. Diot, B. Levine, B. Lyles, H. Kassem, and D. Balensiefen, "Deployment issues for the ip multicast service and architecture," *Network, IEEE*, vol. 14, no. 1, pp. 78 –88, Jan. 2000.

[41] Y. Cui, B. Li, and K. Nahrstedt, "oStream: asynchronous streaming multicast in application-layer overlay networks," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 91 – 106, Jan. 2004.

[42] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *Proceedings of the First International Conference on Peer-to-Peer Computing*, Aug. 2001, pp. 101 –102.

[43] T.-Y. Tseng, R. Lee, S.-W. Lin, and T. Han, "Mixed client server and peer to peer system for internet content providers," in *IEEE International Conference on Systems, Man and Cybernetics (SMC 06)*, vol. 4, Oct. 2006, pp. 3336 –3341.

[44] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: characterization and implications for CDNs and web sites," in *Proceedings of the 11th international conference on World Wide Web*, ser. WWW '02. New York, NY, USA: ACM, 2002, pp. 293–304.

[45] C. Bram. (2008, Feb.) The BitTorrent protocol specification. [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html

[46] S. Saroiu, K. P. Gummadi, and S. D. Gribble, "Measuring and analyzing the characteristics of napster and gnutella hosts," *Multimedia Systems*, vol. 9, no. 2, pp. 170–184, Aug. 2003.

[47] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," *SIGCOMM Computer Communication Review*, vol. 32, pp. 205–217, Aug. 2002.

[48] D. Tran, K. Hua, and T. Do, "A peer-to-peer architecture for media streaming," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 121 – 133, Jan. 2004.

[49] B. T. Loo, R. Huebsch, I. Stoica, and J. M. Hellerstein, "The case for a hybrid P2P search infrastructure," in *Proceedings of the Third international conference on Peer-to-Peer Systems*, ser. IPTPS'04. Berlin, Heidelberg: Springer-Verlag, 2004, pp. 141–150.

[50] M. Castro, M. Costa, and A. Rowstron, "Debunking some myths about structured and unstructured overlays," in *Proceedings of the 2nd conference on*

Symposium on Networked Systems Design & Implementation - Volume 2, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 85–98.

[51] J. Buford, H. Yu, and E. Lua, *P2P Networking and Applications.* Elsevier North-Holland, Inc., 2008.

[52] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in p2p systems," *Commun. ACM*, vol. 46, pp. 43–48, Feb. 2003.

[53] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "SkipNet: A scalable overlay network with practical locality properties," in *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS 03).* Berkeley, CA, USA: USENIX Association, 2003, pp. 9–47.

[54] P. J. Keleher, B. Bhattacharjee, and B. D. Silaghi, "Are virtualized overlay networks too much of a good thing?" in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. London, UK: Springer-Verlag, 2002, pp. 225–231.

[55] H.-C. Hsiao, H. Liao, and P.-S. Yeh, "A near-optimal algorithm attacking the topology mismatch problem in unstructured peer-to-peer networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 7, pp. 983–997, Jul. 2010.

[56] Y. Liu, "A two-hop solution to solving topology mismatch," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 11, pp. 1591 –1600, Nov. 2008.

[57] Y. hua Chu, S. Rao, S. Seshan, and H. Zhang, "A case for end system multicast," *IEEE Journal on Selected Areas in Communicationsie*, vol. 20, no. 8, pp. 1456 – 1471, Oct. 2002.

[58] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: high-bandwidth multicast in cooperative environments," *SIGOPS 2003*, vol. 37, pp. 298–313, Oct. 2003.

[59] V. Padmanabhan, H. Wang, and P. Chou, “Resilient peer-to-peer streaming,” in *Proceedings of the 11th IEEE International Conference on Network Protocols*, Nov. 2003, pp. 16 – 27.

[60] H. Kobayashi, H. Takizawa, T. Inaba, and Y. Takizawa, “A self-organizing overlay network to exploit the locality of interests for effective resource discovery in P2P systems,” in *Proceedings of the The 2005 Symposium on Applications and the Internet*, ser. SAINT ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 246–255.

[61] Y. Liu, L. Xiao, X. Liu, L. Ni, and X. Zhang, “Location awareness in unstructured peer-to-peer systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 2, pp. 163 – 174, Feb. 2005.

[62] A. Manzalini and F. Zambonelli, “Towards autonomic and situation-aware communication services: the CASCADAS vision,” in *IEEE Workshop on Distributed Intelligent Systems: Collective Intelligence and Its Applications (DIS 2006)*, Jun. 2006, pp. 383 –388.

[63] D. Devescovi, E. Di Nitto, D. Dubois, and R. Mirandola, “Self-organization algorithms for autonomic systems in the SelfLet approach,” in *Proceedings of the 1st international conference on Autonomic computing and communication systems*, ser. Autonomics ’07. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST), 2007, pp. 26:1–26:10.

[64] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, “Bimodal multicast,” *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 41–88, May 1999.

[65] B. Pittel, “On spreading a rumor,” *SIAM J. Appl. Math.*, vol. 47, no. 1, pp. 213–223, Mar. 1987.

[66] J. Leitao, J. Pereira, and L. Rodrigues, “HyParView: A membership protocol for reliable gossip-based broadcast,” in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 07)*, Jun. 2007, pp. 419 –429.

[67] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "The peer sampling service: experimental evaluation of unstructured gossip-based implementations," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, ser. Middleware '04. New York, NY, USA: Springer-Verlag New York, Inc., 2004, pp. 79–98.

[68] M. Jelasity, A. Montresor, and O. Babaoglu, "T-MAN: Gossip-based fast overlay topology construction," *Comput. Netw.*, vol. 53, pp. 2321–2339, Aug. 2009.

[69] J. Leitao, J. Marques, J. Pereira, and L. Rodrigues, "X-BOT: A protocol for resilient optimization of unstructured overlays," in *28th IEEE International Symposium on Reliable Distributed Systems (SRDS 09)*, Sep. 2009, pp. 236 –245.

[70] (1997, Sep.) Integrated services (intserv). [Online]. Available: http://datatracker.ietf.org/wg/intserv/

[71] (1998, Dec.) Differentiated services (diffserv). [Online]. Available: http://datatracker.ietf.org/wg/diffserv/

[72] Z. Li and P. Mohapatra, "QRON: QoS-aware routing in overlay networks," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 29–40, Jan. 2004.

[73] J. W. Stewart, III, *BGP4: Inter-Domain Routing in the Internet.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[74] (2005, Jan.) Route views project page. [Online]. Available: http://www.routeviews.org/

[75] X. Dimitropoulos, D. Krioukov, and G. Riley, "Revisiting internet AS-level topology discovery," in *Passive and Active Network Measurement*, ser. Lecture Notes in Computer Science, C. Dovrolis, Ed. Springer Berlin / Heidelberg, 2005, vol. 3431, pp. 177–188.

[76] L. Subramanian, I. Stoica, H. Balakrishnan, and R. H. Katz, "OverQoS: offering Internet QoS using overlays," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 11–16, Jan. 2003.

[77] D. Rubenstein, J. Kurose, and D. Towsley, "Detecting shared congestion of flows via end-to-end measurement," *IEEE/ACM Trans. Netw.*, vol. 10, no. 3, pp. 381–395, Jun. 2002.

[78] T. Kim, Y. Shin, E. Powers, M. S. Kim, and S. Lam, "Application of wavelet denoising to the detection of shared congestion in overlay multimedia networks," in *6th IEEE Workshop on Multimedia Signal Processing*, Sep. 2004, pp. 474–477.

[79] D. Katabi, I. Bazzi, and X. Yang, "A passive approach for detecting shared bottlenecks," in *Proceedings of the Tenth International Conference on Computer Communications and Networks*, 2001, pp. 174–181.

[80] Y. Zhu and B. Li, "Overlay networks with linear capacity constraints," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 2, pp. 159–173, Feb. 2008.

[81] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1990.

[82] Y. Zhu, B. Li, and K. Q. Pu, "Dynamic multicast in overlay networks with linear capacity constraints," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 7, pp. 925–939, Jul. 2009.

[83] R. Schoonderwoerd, O. Holland, and J. Bruten, "Ant-like agents for load balancing in telecommunications networks," in *Proceedings of the 1st International Conference on Autonomous Agents*, ser. AGENTS '97. New York, NY, USA: ACM, 1997, pp. 209–216.

[84] D. Subramanian, P. Druschel, and J. Chen, “Ants and reinforcement learning: A case study in routing in dynamic networks,” in *Proceedings of the 15th International Joint Conference on Artifical Intelligence (IJCAI 97).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 832–838.

[85] S. Liang, A. Zincir-Heywood, and M. Heywood, “The effect of routing under local information using a social insect metaphor,” in *Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002)*, vol. 2, 2002, pp. 1438 –1443.

[86] H. F. Wedde and M. Farooq, “A performance evaluation framework for nature inspired routing algorithms,” in *Proceedings of the 3rd European conference on Applications of Evolutionary Computing*, ser. EC’05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 136–146.

[87] Z. Wang and J. Crowcroft, “Quality-of-service routing for supporting multimedia applications,” *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1228 –1234, Sep. 1996.

[88] J.-H. Choi and C. Yoo, “One-way delay estimation and its application,” *Computer Communications*, vol. 28, no. 7, pp. 819–828, 2005.

[89] (2010, Jun.) RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification. [Online]. Available: http://www.ietf.org/rfc/rfc5905.txt

[90] S. Ubik and V. Smotlacha, “Precise measurement of one-way delay and analysis of synchronization issues,” in *Proc. TERENA Netw. Conf.*, May 2003.

[91] (2011, Apr.) Standard for a precision clock synchronization protocol for networked measurement and control systems. [Online]. Available: http://www.nist.gov/el/isd/ieee/ieee1588.cfm

[92] L. De Vito, S. Rapuano, and L. Tomaciello, “One-way delay measurement: State of the art,” *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 12, pp. 2742 –2750, Dec. 2008.

[93] (2003, Jun.) RFC 3550: RTP: A Transport Protocol for Real-Time Applications. [Online]. Available: http://www.ietf.org/rfc/rfc3550.txt

[94] J. Strauss, D. Katabi, and F. Kaashoek, "A measurement study of available bandwidth estimation tools," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, ser. IMC '03. New York, NY, USA: ACM, 2003, pp. 39–44.

[95] M. Jain and C. Dovrolis, "Pathload: A measurement tool for end-to-end available bandwidth," in *Proceedings of the Passive and Active Measurements (PAM) Workshop*, 2002, pp. 14–25.

[96] Y. Zhang and N. Duffield, "On the constancy of internet path properties," in *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, ser. IMW '01. New York, NY, USA: ACM, 2001, pp. 197–211.

[97] V. Paxson, "End-to-end routing behavior in the internet," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 5, pp. 41–56, Oct. 2006.

[98] J. Yu, "Scalable routing design principles," *Internet RFC 2791*, 2000.

[99] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1998.

[100] F. Pellegrini and J.-H. Her, "Efficient and scalable parallel graph partitioning," in *Proceedings of the 5th International Workshop on Parallel Matrix Algorithms and Applications (PMAA 08)*, Neuchâtel Suisse, 2008.

[101] B. Awerbuch, "Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, ser. STOC '87. New York, NY, USA: ACM, 1987, pp. 230–240.

[102] H. D. Simon and S.-H. Teng, "How good is recursive bisection?" *SIAM J. Sci. Comput*, vol. 18, pp. 1436–1445, 1995.

[103] C. Asavathiratham, S. Roy, B. Lesieutre, and G. Verghese, "The influence model," *Control Systems, IEEE*, vol. 21, no. 6, pp. 52 –64, Dec. 2001.

[104] S. Basagni, "Distributed clustering for ad hoc networks," in *Proceedings of the 4th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 99)*, 1999, pp. 310–315.

[105] J. Beal, "Leaderless distributed hierarchy formation," *MIT AI Lab Memo*, 2002.

[106] W. Y. Tam, K.-S. Lui, S. Uludag, and K. Nahrstedt, "Quality-of-service routing with path information aggregation," *Computer Networks*, vol. 51, no. 12, pp. 3574–3594, 2007.

[107] R. Hou, K.-S. Lui, K.-C. Leung, and F. Baker, "Routing with QoS information aggregation in hierarchical networks," in *17th Int. Workshop on Quality of Service*, Jul. 2009, pp. 1–9.

[108] T. Korkmaz and M. Krunz, "Source-oriented topology aggregation with multiple QoS parameters in hierarchical networks," *ACM Trans. Model. Comput. Simul.*, vol. 10, no. 4, pp. 295–325, Oct. 2000.

[109] Y. Tang and S. Chen, "QoS information approximation for aggregated networks," in *IEEE International Conference on Communications*, vol. 4, Jun. 2004, pp. 2107 – 2111 Vol.4.

[110] D. Thaler and C. Ravishankar, "Distributed top-down hierarchy construction," in *Proceedings of the 17th International Conference on Computer Communications (INFOCOM 98)*, vol. 2, Mar. 1998, pp. 693 –701 vol.2.

[111] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr., "Overcast: reliable multicasting with on overlay network," in *Proceedings of the 4th Symposium on Operating System Design & Implementation*, ser. OSDI'00. CA, USA: USENIX Association, 2000, pp. 14–30.

[112] P. Francis. (1999) Yoid: Extending the multicast internet architecture. Unrefereed Report. [Online]. Available: http://mpi-sws.org/ francis/yoidArch.pdf

[113] R. Droms, "Dynamic host configuration protocol," *Internet RFC 2131*, 1997.

[114] Y. Zhu and Y. Hu, "Enhancing search performance on gnutella-like P2P systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 12, pp. 1482–1495, Dec. 2006.

[115] (2012) Mono. [Online]. Available: http://www.mono-project.com/

[116] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, "The smartfrog configuration management framework," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 16–25, Jan. 2009.

[117] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," *International AAAI Conference on Weblogs and Social Media*, 2009.

[118] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining.* Addison-Wesley, 2006.

[119] Y. F. Hu, "Efficient and high quality force-directed graph drawing," *The Mathematica Journal*, vol. 10, pp. 37–71, 2005.

[120] R. Walker, "Examining load average," *Linux Journal Contents*, vol. 152, Dec. 2006.

[121] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[122] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968.

[123] U.S. Department of the Interior. (2012) National park service. [Online]. Available: http://www.nps.gov/carto/

[124] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, "Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination," in *Proceedings of the 11th International Workshop on Network and*

Operating System Support for Digital Audio and Video (NOSSDAV 01). New York, USA: ACM, 2001, pp. 11–20.

[125] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified NP-complete problems," in *Proceedings of the 6th Annual ACM Symposium (STOC 74).* New York, NY, USA: ACM, 1974, pp. 47–63.

[126] P. L. Montessoro, S. Wieser, and L. Böszörmenyi, "An efficient and scalable engine for large scale multimedia overlay networks," in *2012 IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR 2012)*, May 2012, pp. 1–6.

[127] P. L. Montessoro, "Distributed linked data structures for efficient access to information within routers," in *International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT 10)*, Oct. 2010, pp. 335 –342.

[128] P. L. Montessoro and D. De Caneva, "REBOOK: A deterministic, robust and scalable resource booking algorithm," *Journal of Network and Systems Management*, vol. 18, pp. 418–446, 2010.