FFI-rapport 2009/01610

Search+: An efficient peer-to-peer service discovery mechanism

Magnus Skjegstad and Frank T. Johnsen

Norwegian Defence Research Establishment (FFI)

24 September 2009

FFI-rapport 2009/01610

1086

P: ISBN 978-82-464-1678-6
E: ISBN 978-82-464-1679-3

Keywords

Nettverksbasert Forsvar

Tjenesteorientert arkitektur

Peer to peer

Søkealgoritmer

Service discovery

Approved by

Anders Eggen	Project Manager
Vidar S. Andersen	Director

English summary

This report looks at commonly used Peer-to-Peer (P2P) protocols and aims to identify strengths and weaknesses they may have in a military environment. We further describe some of the P2P based solutions for service discovery that exist today and argue that unstructured P2P protocols are theoretically more suitable for military needs as they are more robust and adaptable.

Unstructured overlays have a tendency to give inaccurate search results and consume more bandwidth than structured alternatives. We implement and evaluate the search algorithm ASAP, which has shown promising results in simulations in terms of bandwidth and accuracy.

Finally, we develop and implement Search+, a novel search algorithm for unstructured overlays that requires little bandwidth while providing close to 100% accuracy.

Sammendrag

Denne rapporten tar for seg de vanligste Peer-to-Peer (P2P)-protokollene, og identifiserer styrker og svakheter disse protokollene kan ha i militære nettverk. Videre undersøker vi ulike typer P2P-baserte service discovery-mekanismer. En viktig konklusjon er at ustrukturerte P2P-protokoller i alle fall teoretisk sett er bedre egnet for bruk i militære nett enn strukturerte protokoller — dette fordi de er mer robuste og adaptive.

Ustrukturerte nett har en tendens til å gi unøyaktige søkeresultater og kreve mer båndbredde enn de strukturerte alternativene. Vi har implementert og undersøkt søkealgoritmen ASAP, som har vist lovende resultater i simuleringer med tanke på båndbredde og søkenøyaktighet.

Vi har identifisert noen ulemper ved ASAP, og har derfor utviklet en ny søkealgoritme for ustrukturerte nett, Search+, som krever lite båndbredde men som likevel oppnår nær 100% treffsikkerhet.

Contents

	Preface	**9**
1	**Introduction**	**11**
1.1	Service Oriented Computing	11
1.2	Peer-to-peer	12
1.3	Problem statement	13
2	**Background**	**13**
2.1	Service Oriented Architecture	13
2.2	The three-layered approach	14
2.3	Summary	16
3	**Peer-to-Peer (P2P)**	**16**
3.1	Introduction	17
3.2	Taxonomy	17
3.2.1	Centralisation	17
3.2.2	Structure	18
3.3	Distributed Hash Tables (DHT)	18
3.4	Consistent hashing	19
3.4.1	Plaxton mesh	20
3.5	Gnutella 0.4	20
3.6	Gnutella 0.6	21
3.7	JXTA	22
3.8	Peer-to-peer Simplified (P2PS)	23
3.9	P2P based Service Discovery frameworks	23
3.9.1	SP2A (JXTA based)	24
3.9.2	WSPeer (P2PS based)	24
3.10	Evaluation: P2P in tactical networks	25
3.10.1	Protocol robustness	25
3.10.2	Robustness against physical disruptions	26

3.10.3 Bandwidth requirements 26
3.10.4 Conclusion 26
3.11 Summary 27

4 Search in unstructured overlays 27
4.1 Blind algorithms 27
4.1.1 Flooding 27
4.1.2 Random walk 28
4.2 Informed algorithms 28
4.2.1 Superpeer based algorithms 28
4.2.2 Flooding with index caching 29
4.2.3 Flooding with query caching 29
4.2.4 Flooding with shortcuts 29
4.2.5 Flooding with routing 30
4.3 Algorithm requirements 30
4.4 Advertisement based Search Algorithm for unstructured P2P (ASAP) 31
4.5 Summary 33

5 Design and implementation 33
5.1 Technology basis 33
5.1.1 Gnutella 34
5.1.2 ASAP 35
5.1.3 Advertisements 35
5.1.4 Join process 36
5.1.5 Maintenance 37
5.1.6 Search mechanism 37
5.2 Search+ design 40
5.2.1 Join process 40
5.2.2 Maintenance 41
5.2.3 Search mechanism 42
5.3 Implementation 43
5.3.1 OpenCOM and Juno 43
5.3.2 Gnutella 43

5.3.3 Gnutella with ASAP 48
5.3.4 Gnutella with Search+ 50
5.4 Summary 52

6 Evaluation 52
6.1 Peer configuration 53
6.2 Queries 54
6.3 Services and topics 54
6.4 Bloom filter 55
6.5 Discovery mechanism 56
6.6 Choosing TTL 56
6.7 Topology 57
6.8 Experiments 58
6.8.1 Experiment 1: Query mechanism 58
6.8.2 Experiment 2: Distribution mechanism 58
6.9 Evaluation plan 59
6.10 Gnutella 59
6.11 ASAP 59
6.12 Search+ 59
6.13 Summary 60
6.14 Test environment 60
6.15 Measurements 60
6.16 Gnutella 61
6.16.1 Response times 61
6.16.2 Bandwidth use 64
6.16.3 Success rate 65
6.17 Gnutella with ASAP 65
6.17.1 Response time 66
6.17.2 Bandwidth 66
6.17.3 Success rate 68
6.17.4 Success rate after 10 hours 68
6.17.5 Distribution mechanism 68
6.18 Gnutella with Search+ 68

6.18.1 Response time 70
6.18.2 Bandwidth 70
6.18.3 Success rate 72
6.18.4 Distribution mechanism 72
6.19 Discussion 72
6.20 Suitability in tactical networks 75
6.21 Summary 76

7 Conclusion 76

8 Future work 77
8.1 Reducing bandwidth 77
8.1.1 Search+ with shortcuts 77
8.1.2 Bloom filters 77
8.1.3 Improve implementation 77
8.2 Dynamicity 77
8.2.1 Liveness 78
8.2.2 Mobility 78
8.2.3 Actual radios 78

Bibliography 83

Appendix A Generating topologies 84

Appendix B Bloom filters 86

Appendix C Tools 88

Preface

This report is mostly based on the Master Thesis "‘Search+: An efficient P2P service discovery mechanism"’, written as a part of Magnus Skjegstad’s master degree in Computer Science at the University of Oslo. The work was supervised by FFI Researcher Frank T. Johnsen, and is a part of the service discovery research activities in project 1086 "‘Secure Pervasive SOA"’.

1 Introduction

For a technology to be suitable for use in a military operation, it must be robust, self adapting and often bandwidth efficient. Modern warfare further requires that forces from different alliances and countries must be able to cooperate and share resources. Ideally, with as little preconfiguration as possible.

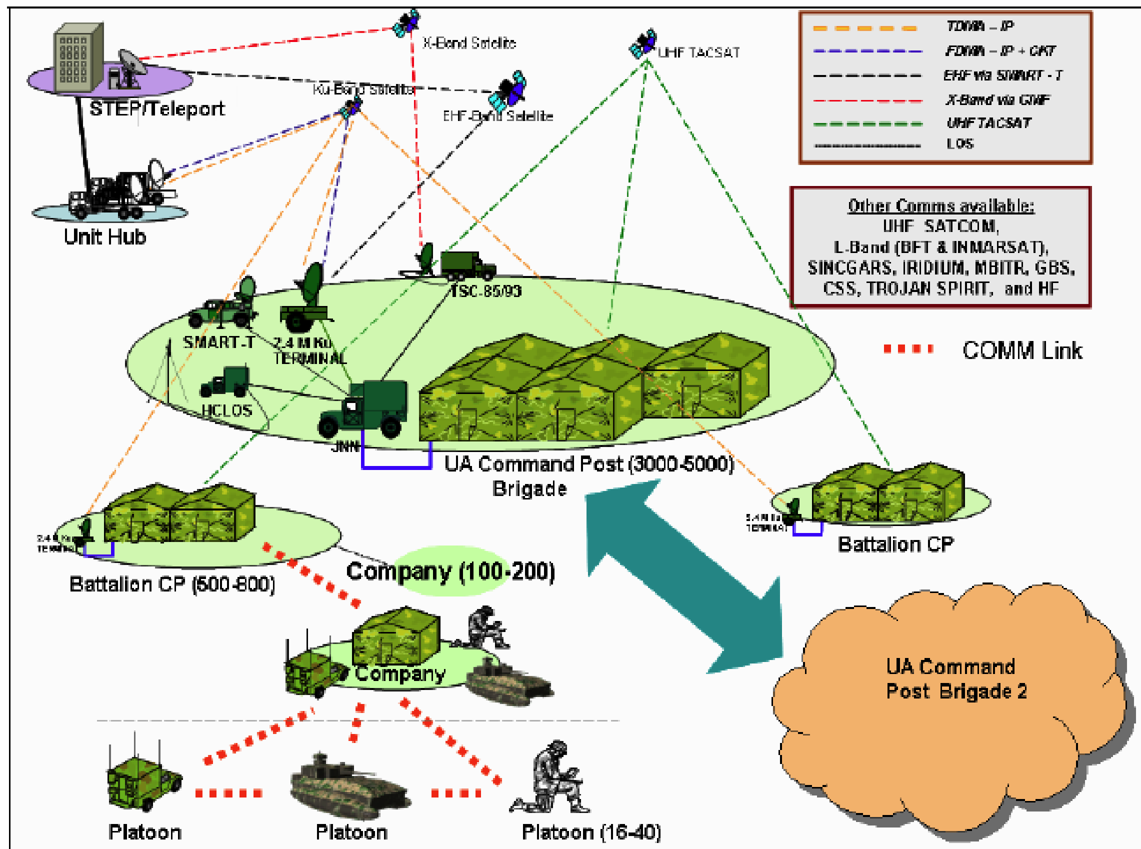

Figure 1.1: Operational network (from [54])

1.1 Service Oriented Computing

In 2005, NATO published a feasibility study [51], that identified *Service Oriented Computing* [38] as a viable solution to many of the problems associated with exchanging information and services in the alliance. As a NATO member, Norway also considers this paradigm to be an important part of the Norwegian Network Based Defence initiative.

In Service Oriented Computing all the capabilities in a network are seen as potential services. In a military context, a service may be a cannon ready to fire, as well as a radar or simply a camera. If a commanding officer needs to have a picture taken of a given object, he would search the network for

soldiers with cameras close to the object's position, and subsequently tell a nearby soldier to take a picture.

In Service Oriented Computing terminology, the officer is called a service *consumer*, while the soldier is the service *producer*. To be able to locate service producers effectively, the consumers would also need a service discovery mechanism to help them. A simple and common approach is to have a central server — a *registry* — maintain an index over the available services. Computer systems that are implemented using the principles from Service Oriented Computing are often called *Service Oriented Architectures* (SOAs).

An operational military network is shown in Figure 1.1. Clearly, the network is quite complex and heterogeneous, which makes it difficult to design a single service discovery mechanism that suits all the participants in the network. We have in our earlier work [40] evaluated service discovery solutions for a SOA in military networks. We conclude that a clear candidate for further research on service discovery is peer-to-peer (P2P) based technologies, due to their potential robustness and self adapting properties. A P2P overlay would allow the service consumers and producers to contact each other directly, without having to rely on a central registry.

1.2 Peer-to-peer

In P2P networks — or *overlays* as they are often called — each participant is considered a *peer*. Each peer may function as both a client and a server, and the underlying protocol defines how work is distributed among the participants.

P2P overlays are often designed with file or content sharing in mind. These mechanisms can be extended to support service discovery by letting the file name be a unique service name and the file content be a service description. How advanced the searches for services can be, depends on the search mechanism in the P2P protocol, but on the Internet it is common to perform keyword based searches to find shared files.

A P2P system used for service discovery must perform three important tasks:

1. Create and maintain the P2P overlay.
2. Enable peers to locate services.
3. Enable peers to invoke services.

First, this thesis looks at how existing overlays that have previously been used for service discovery construct their overlay and evaluate them theoretically with focus on robustness in military networks.

Second, two bandwidth efficient algorithms are evaluated by experimentation in terms of bandwidth requirements and search accuracy. One of these algorithms is Search+, a novel search algorithm developed as part of this thesis.

The actual invocation of the service is beyond scope of this thesis. In [47], Lund et al. show that it is possible to invoke a service with a known location in an operational military network.

1.3 Problem statement

This thesis aims to answer the following questions;

- Which P2P overlays are most suitable for service discovery in military operational networks? The overlays must be able to adapt to mobile environments and be robust against attacks and other disruptions.
- Is there currently any search algorithms that meet the bandwidth requirements of a military environment?
- What are the weaknesses in existing search algorithms and how can they be improved?

2 Background

As mentioned in the previous chapter, a concept that fits well with the idea of network based defence is a SOA. This chapter describes the underlying principles in SOAs. It further explains how we envision a three layered approach with P2P as the middle layer to be suitable for an operational military network.

2.1 Service Oriented Architecture

In a SOA there is a loose coupling between the service consumer and producer, enabling consumers to find and invoke the services[1]they believe are most suitable at the time. This is very convenient in military networks, where the network structure can be unknown until the last moment and changes in topology can occur rapidly and without warning.

Figure 2.1 shows the three aspects of a SOA: The provider of a certain service (*producer*), the client of a certain service (*consumer*), and finally the directory providing a means of discovering available services (*registry*). Basically, a consumer and a provider need not know about the others' locations

[1] In a military context a service can be thought of as an interface to a particular capability which can be used by other services or applications. The service is therefore a unit of work offered by a service provider to achieve desired end results — a capability — for a service consumer [29].

in advance, only where a registry service can be found. The provider will publish its service in the registry, and the consumer will use the registry to search for, i.e. discover available services. Following the registry lookup, the client can invoke a service that a provider has published.

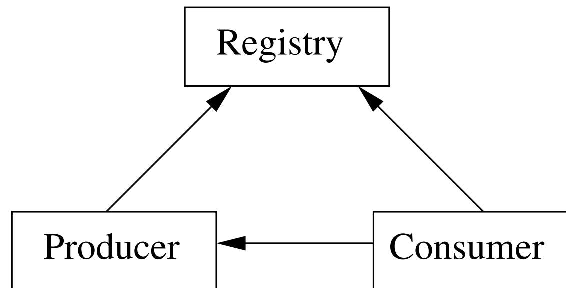

Figure 2.1: The SOA triangle, showing the relationships between consumer, producer and registry. The consumer consults the registry for a matching service, the registry responds with information about the producer, whereupon the consumer contacts the producer to invoke the service.

SOAs are often realised through the use of Web services [26] and standardised protocols. Previous research by Lund et al. [47] has shown that by using data compression, optimised data representation and proxy servers, it is possible to employ a SOA using Web services in tactical networks. Thus, provided you already know where the service you want to invoke is located, you can use Web services in a military network to invoke it. However, the service discovery mechanisms that are currently available for Web services are not ideal for use in tactical environments.

The UDDI [22] registry is one of the commonly used registry solution for Web services today. A competing standard is ebXML [31], which offers similar functionality. Both these solutions are based on a central registry being available, thus introducing a single point of failure. It is possible to increase robustness by configuring the registries in a so called *federation*, in which several registries cooperate by replicating data between them. However, this replication itself would require a lot of bandwidth, which makes this solution undesirable at the tactical level where bandwidth is scarce [32]. Furthermore, if the network is partitioned, the different partitions may not be able to contact a registry and thus be unable to invoke services, even if the services are available within the partition itself. A registry may thus not be the most suitable service discovery mechanism for a military network, at least not as the only available method.

In [32], Gagnes describes a set of requirements for service discovery in a network centric battlefield, along with a discussion of how one can achieve a functional service discovery solution in operational networks. The article discusses how ideas from P2P research, together with registry technology, can be utilised. A P2P based discovery model is shown in Figure 2.2.

2.2 The three-layered approach

In [41], we identify three levels of operation that can be used to classify service discovery requirements in the battlefield, shown in Figure 2.3.

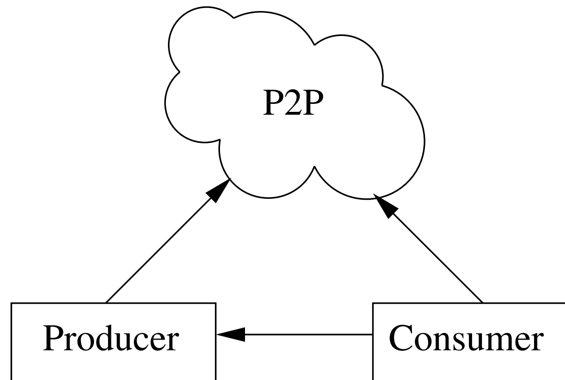

Figure 2.2: A SOA where the registry has been exchanged with a P2P network.

At the lowest level, there are *mobile tactical networks* that mainly consist of highly mobile units taking part in an operation. They may have high bandwidth links between each other, but contact with the the rest of the network must pass long-range reach-back links, with correspondingly low bandwidths. The service descriptions in this layer will be minimal, typically just predetermined identifiers or unique names. Search will mostly consist of broadcasts with requests for a specific service. For more advanced search mechanisms a gateway with access to a higher level must be contacted. We expect that service discovery at this level will be dominated by discovery solutions for ad hoc networks, like the solution proposed in [30].

At the top of the hierarchy is the *strategic network*, where the units are stationary. Communication is done via high capacity radio links and land lines. The high bandwidths make registries well suited for service discovery at this level. By using a registry, the participants will be able to perform complex queries and use detailed service descriptions. Since bandwidth is abundant, single points of failure can be avoided by replicating content on multiple locations. Commercial registry solutions like UDDI and ebXML will be suitable at this level.

Between these two extremes, a large part of the network exists that has units with varying bandwidth and resource requirements. These units typically take part in *deployed tactical networks* and are either temporarily stationary, like command posts, or larger mobile units. They have the ability to carry larger and heavier equipment, thus giving them more resources and bandwidth than the lowest level, but not quite enough to use registry based solutions at the top of the hierarchy. Often, the network topology will not be known in advance, which means that the units must be able to adapt to new, and possibly unknown network configurations while being robust to attacks. We envision a P2P based discovery solution to be appropriate at this level [41].

A P2P overlay is decentralised, requires little configuration in advance and is highly adaptable. Moreover, P2P has successfully been used for service discovery in previous works, including SP2A [16], WSPeer [36] and PWSD [45]. These P2P systems, however, are not designed for low bandwidth tactical networks.

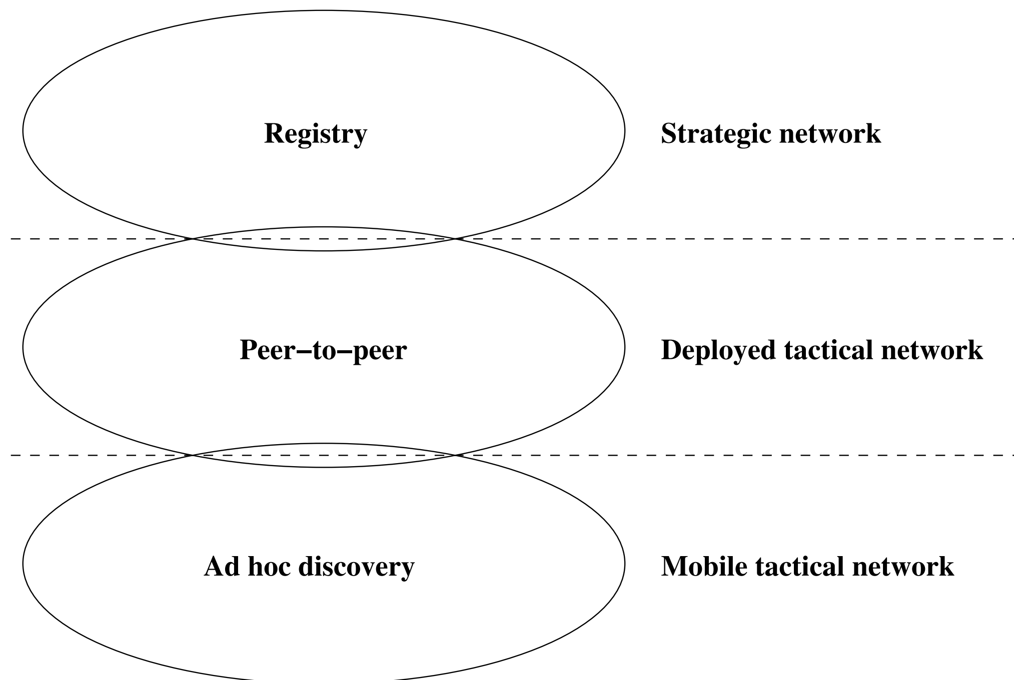

Figure 2.3: Service discovery mechanisms for each operational level as suggested in [41].

2.3 Summary

We argue in [41] that an operational military network has three levels with different needs and requirements for service discovery: a strategic level, a deployed tactical level and a mobile tactical level. We have previously proposed using P2P for service discovery at deployed tactical levels.

The requirements for P2P based service discovery in deployed tactical networks can be summarised as:

- Low bandwidth: The solution must function well with low bandwidth.
- Decentralised: Single points of failure must be avoided.
- Resilient: The solution must be able to adapt to changes in the topology.
- Robust: The solution must continue to function even if large or central parts of the network become unavailable.

3 Peer-to-Peer (P2P)

P2P networks have several properties that are useful in tactical networks — they are decentralised, resilient and self-adapting. Unfortunately, they can often have high bandwidth requirements and be

susceptible to mobility issues when many nodes must be reconfigured simultaneously.

In this chapter, P2P as a technology is presented and the differences between structured and unstructured overlays are explained. Moreover, some of the existing P2P protocols and frameworks that have previously been used for service discovery are described.

The chapter concludes with a theoretical evaluation with focus on suitability in tactical networks.

3.1 Introduction

In the classic server-client model, a service is located on a well-known server and the clients can connect to it whenever the service is needed. In a P2P model, however, all the participants may function both as servers and clients at the same time.

The term *peer* is derived from the idea that instead of having smaller clients connecting to a large server, a true P2P network treats all nodes equally, or as peers. Commonly, a P2P network is created by forming an overlay on top of an existing network. This can be seen as a network within the network, where the P2P protocol defines addressing and routing mechanisms and maps them on top of the underlying network or transport layer, e.g. TCP/IP.

Generally, a connecting node needs to know only one existing peer in the network. After establishing a connection to this node, the connecting party can discover more nodes using the overlay. These newly discovered nodes can be connected to right away for redundancy, or kept available if other nodes become unresponsive.

A P2P protocol commonly also defines how data can be located. This adds to the complexity of implementing a P2P overlay. Compared to connecting directly to a server, the protocols tend to get quite sophisticated, as we will see later in this chapter.

3.2 Taxonomy

P2P overlays are categorised in terms of *centralisation* and *structure*.

3.2.1 Centralisation

In [18], Androutsellis et al. identify three different degrees of centralisation; *purely decentralised*, *partially centralised* and *hybrid decentralised* architectures. In a purely decentralised architecture, each node performs the same tasks and all peers are treated equally. In partially centralised systems, some peers have more responsibilities than others and function as *superpeers*, i.e. by caching queries on behalf of the other nodes — superpeers are described in more detail later in this chapter. In hybrid decentralised overlays a central server will assist the P2P network. The server may for instance maintain a searchable index over content available in the network.

3.2.2 Structure

In *unstructured* overlays, «the placement of content (files) is completely unrelated to the overlay topology» [18]. In other words, the P2P protocol cares very little about where data is actually located in the network, and to find it one must actively search for it. The main challenge in unstructured P2P networks is actually finding all the data that have been published. Examples of unstructured overlays are Gnutella [23, 44] and Freenet [39]. Search algorithms in unstructured overlays are further discussed in Chapter 4.

In *structured* P2P overlays however, all the data is neatly organised to make it easier to locate. Usually this is done by employing an addressing scheme where data can be located by a unique identifier, usually a hash function. This identifier determines where in the network the data should be stored. Some of the first structured approaches were Pastry [57], Chord [60], CAN [56] and Tapestry [68].

Hybrid P2P overlays combine functionality from structured and unstructured protocols. An example of a hybrid overlay is JXTA [61].

A P2P taxonomy based on structure type is shown in Figure 3.1.

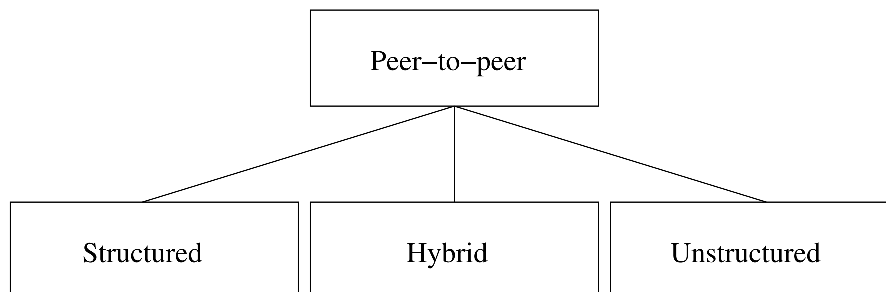

Figure 3.1: A simple taxonomy for P2P. P2P protocols can be categorised as structured or unstructured, or a combination of the two.

3.3 Distributed Hash Tables (DHT)

A DHT is essentially a hash table that is distributed across a P2P overlay in a deterministic fashion — DHTs are therefore structured. File sharing is commonly accomplished by publishing a URL in the DHT, which can be retrieved with the correct keyword or hash.

DHTs can be constructed in different ways, but most commonly they are based on one of two methods; consistent hashing [42] or a Plaxton mesh [52]. These two mechanisms are now briefly explained.

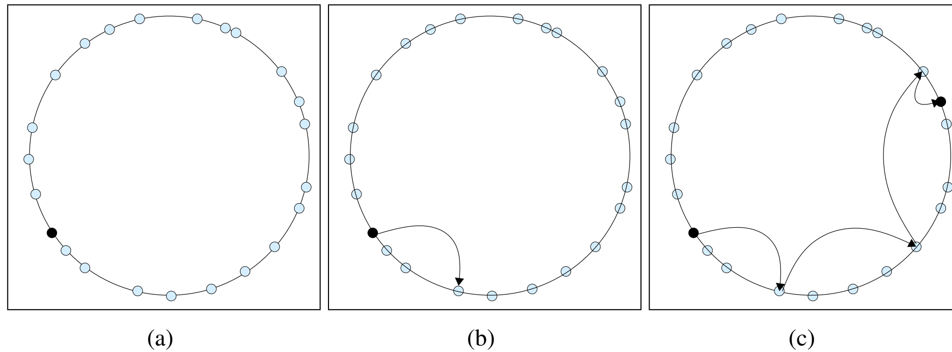

Figure 3.2: *Left:* A node is looking for a hash-value in the overlay. *Middle:* The request is routed towards the node responsible for the address range. *Right:* The request reaches its final destination and the data can be retrieved.

3.4 Consistent hashing

In a consistent hashing scheme, each node and data element is given a unique address in a large address space. Nodes can then be given responsibility for data that has an address which is close to their own. As an example, Chord has a 128 bit address space, allowing 2^{128} unique addresses. 0 and $2^{128} - 1$ are considered next to each other, thus forming a circle, as illustrated in Figure 3.2a.

Data items in the DHT are placed in the same address space as the peers. By using a consistent hashing method, like SHA-1, all nodes can agree on the address given to new data elements.

Retrieving data can be done by following routing tables closer and closer to the address of the requested element, as depicted in Figure 3.2. Each peer knows about a number of peers above and below in the circle. To retrieve a data element with the given address, the interested node can inspect its routing table to find peers that are closer to the address it wants to retrieve. The data element may then be requested from this node and the process is repeated, gradually moving closer to the node that may have it. Alternatively, the querying node may request information about other nodes closer to the data element. In both mechanisms, the process is repeated until the node responsible for the data element is located. The data can then be retrieved. A similar method is used to insert new items into the overlay.

When a new node contacts an existing peer to join the network, the node is first assigned a randomly distributed address within the circular address space, i.e. by generating a hash value based on IP address and TCP port. The existing peer then looks up the node's newly assigned address in its routing table and informs the node about any peers that have an address that is closer to it in the circle. The joining node contacts the other peers and repeats the process until it reaches two nodes placed as close as possible above and below itself in the address space. This is now the new location of the node.

3.4.1 Plaxton mesh

As in the consistent hashing scheme, a Plaxton mesh relies on both nodes and content being identified by a numeric value in the same address space. In a Plaxton mesh, the node's identifier is randomly assigned, while the content is identified by the result of a hash function, e.g. SHA-1. It is important that the identifiers of both content and nodes are evenly distributed in the address space.

In a Plaxton mesh, the distance between two hash values is given by the number of shared bits in the suffix. For example, in 01**11** the suffix is **11**. This means that 00**11** is closer to 01**11** than 01**01** is.

Each node maintains two data structures; a *neighbour table* and a *pointer list*.

The neighbour table is essentially a routing table that contains the addresses of nearby nodes. As with consistent hashing, messages are routed towards their destination by being forwarded to nodes that are considered closer. The neighbour table contains a list of primary and secondary neighbours, used for routing data along different routes, and a list of reverse routes back to other nodes that have the holding node in their neighbour table. More details about the routing mechanism can be found in [52].

The pointer list is a table with pointers to known data elements. For instance, the node may know that an item A is located on node X. If the node receives a request for item A from a node Y, it can tell node X to forward A directly to Y.

When a node inserts a data element D into the mesh, the hash value h is calculated. The node then consults its neighbour table and finds the neighbour that has the address that is numerically closest to h. It then tells the neighbour that it has the content identified by h available. The neighbour caches the node's address along with h in its pointer list, and checks its neighbour table for a neighbour that has an address that is even closer to h. If it finds one, it forwards the information and the process is repeated. When the information reaches the node in the overlay with the smallest numerical distance to the data element, that node will be considered D's root node. The root node will republish information about where the data can be located at regular intervals. The actual content remains on the node that published it first — the overlay only maintains pointers to where it can be located.

3.5 Gnutella 0.4

Gnutella is a search protocol for distributed search in unstructured P2P networks. The protocol defines a basic set of message types that can be used to create and maintain a P2P overlay. The first version of the protocol (0.4) is quite inefficient [53], but it provides a good platform for testing new algorithms due to its simplicity. The protocol is described in more detail in Section 5.1.1.

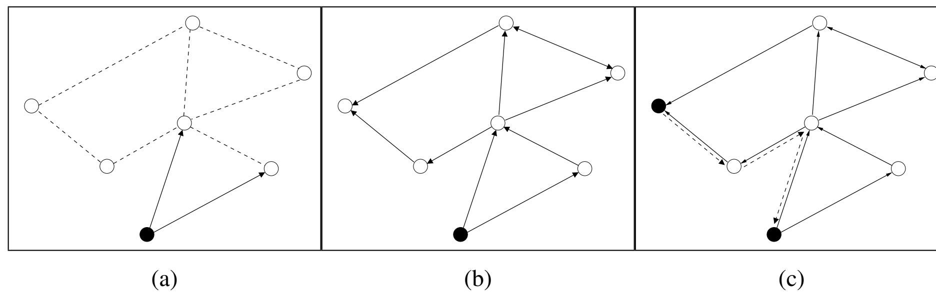

Figure 3.3: A search request in a Gnutella 0.4 overlay. *Left and middle:* A QUERY message is flooded to all neighbours. *Right:* A node with matching content has received the QUERY message and responds with a QUERYHIT.

As can be seen in Figure 3.3, Gnutella nodes search for content by flooding the network with query messages. Whenever a query message reaches a node that has the relevant information, a response is sent back along the same route in the overlay. Generally, this is a very inefficient search mechanism, but it is robust and extremely resilient.

Gnutella 0.4 does not scale well. As stated by Portmann et al. in [53] regarding the dissemination of search messages in Gnutella, the cost increases more than linearly with the number of nodes in the vicinity of the searching node. When the overlay becomes too large, this leads to saturation of central nodes, effectively splitting the network into smaller subnets.

Modern Gnutella clients use version 0.6 of the protocol, which employs superpeers to improve scalability. Gnutella 0.6 is described in the next section.

3.6 Gnutella 0.6

Gnutella 0.6 [44] is an extension to the original 0.4 protocol, and offers functionality designed to make the protocol more efficient and scalable. It includes a caching scheme, which reduces the number of messages sent through the network significantly. Additionally, it introduces the concept of superpeers, called *ultrapeers*.

Ultrapeers are well connected nodes in the overlay, which are chosen to function as *hubs* in the network. Ultrapeers helps with caching search results for their edge nodes and perform overlay maintenance. Gnutella 0.6 uses a variant of *Bloom filters* [20] to exchange search indexes between edge nodes and ultrapeers. Bloom filters are described in detail in Appendix B.

Since this version of the protocol also enables client implementations to add their own extensions, there are many variations in the protocol. Over time the 0.6 draft has evolved and some proprietary client extensions have become de facto standards. Due to the difficulty in determining the exact protocol and the complexity of some of the extensions, a fair evaluation based on the later versions is difficult. This makes 0.6 less suitable for academic experimentation than its predecessor.

The Gnutella 0.6 query process is illustrated in Figure 3.4 below.

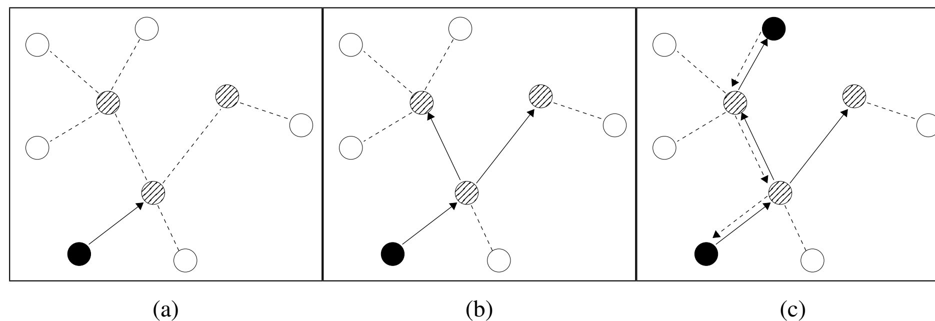

Figure 3.4: A query performed in Gnutella 0.6 with ultrapeers. *Left:* The message is first sent to the ultrapeer. *Middle:* The first ultrapeer consults its routing table and concludes that there are no local matches. The message is forwarded to the other ultrapeers. *Right:* If a match is found a response is sent back to the node that sent the query.

3.7 JXTA

JXTA [61, 62] is a framework developed by Sun for building P2P applications. Communication is done over TCP/IP and HTTP with XML [66] as the underlying message format.

JXTA is actually a family of protocols that are designed to be extendable and scalable. All the included message types can be extended with custom data in XML format. This makes JXTA an obvious choice for implementing service discovery mechanisms for Web services, which are also based on XML.

The creators of JXTA envision a virtual network on top of the Internet that should be able to scale to billions of nodes and services. The details on how they plan to accomplish this is beyond scope of this chapter, but it is described thoroughly in [62].

Like Gnutella 0.6 and later, JXTA lets some of its peers have more responsibilities than others and become superpeers. Their extended functionality mainly include caching messages from other superpeers and maintaining indexes of the content available on their edge peers.

To keep the overall traffic low, the superpeers do not just cache search requests they receive from the edge peers. Instead, the edge peers actively update the superpeers' content index. When the superpeer receives a query from a superpeer or edge node, it first matches the query against its content index and if a match is found, reports it back to the source of the query. The match is not conclusive until an additional message is sent directly from the query source to the edge node to confirm the match. This last step is needed to make sure that the service is available at the time the query is made.

The superpeers are connected to each other through a loosely consistent DHT. The edge nodes form clusters around superpeers and are connected to each other in an unstructured overlay with the superpeers as gateways to the rest of the network. This combination of structured and unstructured mechanism makes JXTA a hybrid overlay.

Several P2P based Web service frameworks have been developed on top of JXTA, including SP2A which is discussed in more detail in Section 3.9.

3.8 Peer-to-peer Simplified (P2PS)

P2PS [65] is inspired by the JXTA architecture, but it is meant to be easier to use. According to the authors, P2PS can in many ways be seen as a subset of JXTA. P2PS is also based on XML and, as in JXTA, all messages can be extended with custom XML data.

A P2PS overlay is organised in clusters of edge peers called discovery subnets, connected through superpeers. Each peer advertises its shared content in its discovery subnet and maintains a cache of other advertisements it receives. To perform a search, a query message is broadcasted to the discovery subnet and nodes with matching advertisements unicast their replies. The discovery subnets are generally based on UDP multicast as discovery mechanism, but P2PS may be extended to support other discovery mechanisms as well. The authors suggest Bluetooth and JXTA as viable alternatives.

The superpeers cache and respond to advertisements and queries on behalf of edge nodes and can forward them to other superpeers in other discovery subnets. The superpeers are connected with TCP/IP, but P2PS is not tied to a specific transport protocol.

[65] describes a grouping functionality that allows edge peers to connect directly to superpeers to send advertisements instead of having use UDP broadcast on a local subnet. The superpeers then create groups of peers that subscribe to different advertisement types. This functionality is reportedly implemented according to their web page [8], but as of March 20th, 2009, the source code is not available for download.

P2PS differs from JXTA in that it does not employ a DHT between the superpeers. It is thereby classified as an unstructured overlay.

3.9 P2P based Service Discovery frameworks

There have been several proposals for services discovery mechanisms based on a P2P overlay [45, 59, 67]. This section takes a closer look at two such mechanisms that are specifically designed for Web services; SP2A [16] and WSPeer [36].

3.9.1 SP2A (JXTA based)

SP2A is a lightweight Service oriented framework for P2P based resource sharing in Grid environments, proposed by Amoretti et al. in [16]. It relies on JXTA as the underlying P2P overlay and has recently been evaluated by the NC3A [48].

An SP2A enabled peer comprises a message handler and router that handles communication. It allows peers to participate in the overlay, uses resource handlers and monitors for resource management. In addition, a set of managers help with security, scheduling and state management. SP2A refers to shared content as resources.

Resources in SP2A are described by name, a textual description, and a uniquely identified owner. In addition, an interface is exposed that specifies how to access it. If the resource is a service, it may also have semantic descriptions of the expectations associated with it. These are divided into three parts; data model, process and behaviour.

Discovery is based on both value matching and semantic search. Discovered resources are ranked against the reference ontology using a semantic matcher.

The source code for SP2A is readily available on the Internet and can be downloaded from [50].

3.9.2 WSPeer (P2PS based)

WSPeer is an API for hosting and invoking Web Services, proposed by Harrison and Taylor in [36]. It is component based, allowing developers to create additional components for other transport mechanisms than just HTTP and SOAP. The framework supports asynchronous communication, which makes it possible to implement P2P based transport components.

The authors argue that Web services and P2P approaches will converge, or rather «cross fertilise» by influencing each other. WSPeer is meant to make it easier to experiment with combining functionality from the two worlds.

Support for P2PS is included. It also has a component for using the UDDI registry.

WSPeer has three main aims:

- To act as a complete interface to both publishing and invoking services, enabling applications to expose themselves as service oriented peers.
- To be applicable to a variety of network architectures including «standard» Web service architectures using technologies such as UDDI and HTTP, and P2P style networks.
- To be flexible and extensible, allowing users to develop application specific service capabilities of their choice.

The source code of WSPeer is available for download from [35].

3.10 Evaluation: P2P in tactical networks

In Chapter 2, the requirements for a P2P solution in a deployed tactical network were summarised. As most P2P overlays are decentralised, this section evaluates structured and unstructured overlays in terms of bandwidth requirements and robustness.

3.10.1 Protocol robustness

Structured overlays are vulnerable to dishonest nodes that exploit their self-structuring mechanisms. The structured mechanisms tend to depend on some degree of trust between the participating nodes, thus often allowing attackers to confuse their neighbours, for instance by giving them invalid or conflicting information. Such attacks may lead to reduced performance and increased bandwidth consumption, and in the worst cases, collapse of the overlay. A detailed overview of attacks on DHTs can be found in [58].

Unstructured overlays are also vulnerable to attacks. One way to attack them is by exploiting weaknesses in the search algorithm. Since these overlays are without structure, a search algorithm based on some form of overlay multicast is often used. The traffic generated by such algorithms tend to increase super-linearly as it travels through the network, and may be used to invoke denial-of-service (DoS) attacks[2]. The possible implications of a query based DoS attack on Gnutella in 2002 is described in [27], along with measures that can be taken to reduce the damage.

Both types of networks are vulnerable to invalid data elements being published in the network. Due to their decentralised nature, it is difficult to enforce an authentication system and anyone may publish or access content in it. Thus, by publishing large amounts of invalid content, dishonest nodes may trick honest nodes into wasting their bandwidth on data that turns out to be useless. This attack can be taken even further, by responding to search queries on behalf of non-existing nodes, polluting cached data and indexes in superpeers and so on. An overview of index poisoning attacks in both DHTs and unstructured networks can be found in [46]. According to the authors, this form of attack has been deployed by the «copyright industry», presumably to fight piracy in P2P networks on the Internet.

All these vulnerabilities depend on the attacker being able to participate in the P2P network to some degree. Although such attacks should be considered and taken into account when designing a P2P system for tactical networks, we must assume that encryption or other forms of access control are used to stop dishonest nodes from entering the overlay in the first place. The most serious weaknesses will then be those that affect the physical structure of the network.

[2] A denial-of-service attack is an attack where a service is slowed down or otherwise rendered useless, thereby denying legitimate clients access to it.

3.10.2 Robustness against physical disruptions

In a tactical network there is always a certain degree of mobility. This can lead to radios going out of range, which in turn may make central nodes unavailable or partition the network. Such effects could also be caused by attacks on central nodes, e.g. by radio jamming.

In structured overlays, joining and leaving is a very bandwidth consuming process, in that nodes must find their place in the overlay and exchange routing tables. They are also vulnerable to partitioning, where gaps can occur in the address space if too many nodes are removed in the same area of the address space at the same time. Unstructured overlays, on the other hand, are more or less unaffected.

Even though unstructured overlays are resistant to partitioning, completely unstructured overlays have a tendency to arrange themselves in topologies following a power law distribution [15]. This means that some nodes have substantially more connections than others. If one of these nodes is destroyed, this can hurt the performance or partition the overlay — although the remaining parts of the network will keep functioning independently of each other. Structured networks however, do not have this problem, because their structure evenly distributes responsibility in the network. It should be noted, that this property of DHTs may actually be a disadvantage in a tactical network, where some nodes may have more bandwidth and as such could handle more traffic than its peers.

3.10.3 Bandwidth requirements

Both structured and unstructured overlays have high requirements for bandwidth, but they use the bandwidth differently.

In structured overlays, most of the bandwidth is consumed for maintenance of the overlay. In a Plaxton mesh, the root nodes must regularly send updates about where content can be located. With a consistent hashing mechanism the peers must keep their routing tables up to date, which can be a challenging task if many nodes are leaving or joining the network at the same time. Fortunately, these efforts in maintenance pays back in terms of less search cost. Content can usually be located with just a few messages, and with algorithms like Beehive [55] in just one hop.

Unstructured overlays require less bandwidth for overlay maintenance, as there is no structure to maintain. They do however require more bandwidth and time for locating content. This means that the bandwidth consumed by an unstructured overlay depends largely on the chosen search algorithm.

3.10.4 Conclusion

Considering the strength and weaknesses of structured and unstructured networks, we see that due to its resistance to physical attacks, an unstructured overlay is more suitable for tactical networks. However, the algorithm used for locating data in the overlay has to be extremely bandwidth efficient.

3.11 Summary

P2P overlays can be categorised in terms of structure. Unstructured networks are resilient to physical attacks, but require bandwidth intensive search algorithms. Structured overlays are more vulnerable, but their structured nature allows content to be more easily located.

P2P has been used successfully for service discovery in earlier works. They are commonly based on a structured or hybrid overlay protocol.

After a theoretical evaluation of overlay types, we conclude that unstructured P2P overlays are better suited for tactical networks because of their resilience to physical attacks. However, the search algorithm employed in the overlay must be extremely bandwidth efficient.

4 Search in unstructured overlays

As we concluded in the previous chapter, the search algorithm in an unstructured P2P overlay largely determines the bandwidth consumption of the overlay. This chapter gives a brief overview of some of the many search algorithms that exist for unstructured overlays. Additionally, the Advertisement based Search Algorithm for unstructured P2P (ASAP) is described. It is evaluated in Chapter 6.

A major challenge in unstructured P2P protocols is actually finding the data that is available. A survey of search algorithms in unstructured P2P overlays up until 2006 is presented in [63] by Tsoumakos et al. The survey distinguishes between *blind* and *informed* algorithms.

4.1 Blind algorithms

Blind algorithms blindly propagate queries to a sufficiently high number of nodes to satisfy the search request. There are many blind algorithms that have been proposed, but flooding and random walk are perhaps the best known, together with variations of these two. Common for all of them is that by searching the network blindly, they require more resources per query than the more informed algorithms.

4.1.1 Flooding

The flooding algorithm can be considered the most basic search algorithm in P2P overlays. When a node performs a search it sends the search criteria to each of its neighbours together with a time-to-live (TTL) value. The neighbours decrease the TTL value and forward the message to their neighbours, continuing this process until the TTL value reaches 0. Each node that has content that matches sends a reply back to the node that performed the search.

The flooding algorithm is easy to understand and implement, but in most situations it is quite inefficient in terms of bandwidth and resource use.

4.1.2 Random walk

The random walk algorithm is similar to the flooding algorithm, but instead of sending the request to each neighbour the searcher picks k neighbours and forwards the query to these selected nodes. Each of the neighbours repeats this process until the TTL reaches 0. Since the query is not sent to all neighbours, it can be sent with a higher TTL and still generate less bandwidth than a regular flood search. This enables the searching node to penetrate deeper into the P2P overlay, and possibly get more accurate search results. In [33] it is shown however, that random walk generally gets better results when the same documents are shared multiple times in the network. This may not be the case in service discovery where a service may only have been published once.

4.2 Informed algorithms

Informed algorithms are algorithms that use previously acquired information about content to optimise the search for it. There are several ways to achieve this, among them is having some peers maintain indexes (i.e. superpeers), or predistributing indexes to all nodes. These indexes can contain exact information about where the content can be located, or provide hints as to where the query can be directed to have a higher probability of success.

As with blind algorithms, several informed algorithms have been proposed. Some of these are described and evaluated by Tsoumakos et al. This section briefly covers some of the algorithms from the survey, together with ASAP, which is described at the end of this chapter.

4.2.1 Superpeer based algorithms

In unstructured overlays it is common to let some nodes have extra responsibilities and become superpeers. These peers have a number of edge or leaf peers connected to them, and can in many ways be seen as their proxy to the rest of the overlay. When one of the peers wishes to perform a query, the query is sent directly to the superpeer, which in turn may forward the query to other superpeers or return a result based on cached information it has about the network.

The superpeers often form their own overlay within the overlay, where information about content on leaf nodes is exchanged. For instance in JXTA such a network is realised by using a loosely structured DHT, as described in Section 3.7. In later versions of Gnutella, the superpeers cache search information in Bloom filters to reduce the number of broadcast query messages.

4.2.2 Flooding with index caching

These algorithms use a blind distribution mechanism to distribute indexes in the network. After the indexes have been distributed, the nodes can perform searches just by consulting the indexes they have received.

Local Indices [63, ref. therein], is an algorithm that broadcasts indexes to neighbours with a given TTL. By doing this, each node in the overlay will be able to respond to queries on behalf of its neighbours within a given radius, depending on the TTL.

This approach gives very good accuracy, but requires a flood-like mechanism to distribute the indexes. In dynamic environments the traffic increases as nodes frequently join the network. Of the algorithms tested in [63], this is one of the most bandwidth intensive, although it gives near 100% success rate in static networks.

4.2.3 Flooding with query caching

In these algorithms, the query mechanism uses a blind distribution method, but keeps the results cached in the overlay. Queries for the same content can then be answered with reduced cost.

The Distributed Resource Location Protocol (DRLP) [63, ref. therein] initially forwards queries in the overlay with a flood-like mechanism. The messages are forwarded to each neighbour with a given probability. When a match is found, a reply is sent back through the overlay following the same path as the request — similarly to a query hit in Gnutella. On each node the reply passes, the location of the found content is recorded. Subsequent queries for the same information can then be responded to much earlier, and in the best cases with only one message.

DRLP is evaluated in [63] and gives near 100% accuracy in static networks. In dynamic networks however, the success rate decreases. When many objects are relocated in the network, the success rate is down to between 40% and 20%. The algorithm is very bandwidth efficient in static networks, since queries are only forwarded the first time the content is requested.

4.2.4 Flooding with shortcuts

These algorithms also depend on a blind distribution mechanism, but after successful replies are received, the searching node connects directly to the node with the matching content. This creates a «shortcut» in the network, based on the assumption that nodes responding to a query are likely to have more relevant content.

Gnutella with Shortcuts [63, ref. therein] creates, as the name implies, shortcuts in a Gnutella overlay by forming direct links to nodes that have previously answered a query. Like DRLP, the algorithm initially uses a flooding mechanism to search for content and peers that provide answers will have their results stored along the path back to the original requester. The algorithm then

assumes that nodes that answered one query have a high probability of being able to answer another one. Subsequent queries are first sent directly to the shortcuts, before the algorithm reverts back to flooding if not enough results are found.

This algorithm achieves a very high success rate, but bandwidth usage increases significantly when there are changes in the network. This is because it falls back to a flooding mechanism when results are not found among the established shortcuts.

4.2.5 Flooding with routing

Flooding with routing uses a blind distribution mechanism to update nodes in the overlay with paths to where information can be found. After the paths have been created, queries can be sent directly to nodes with a high probability of having the requested content.

The authors of [25] propose a query routing mechanism based on predistributed content indexes. The indexes provide information as to which neighbour is most likely to produce matching content if a query is routed through it. Additionally, a stop condition is included in the query, which enables the nodes along the route to determine when enough results have been gathered. The algorithm relies on content and queries being classified by one or more *topics*, so that requests for content of a specific type can be forwarded to nodes that is known to share information of the same type. The work describes a mechanism to classify queries according to topics by generalising terms, e.g. «SQL» becomes «databases», and so on.

According to Tsoumakos et al., this algorithm performs well in terms of bandwidth, but due to the flooding mechanism required to update the routing tables, the algorithm does not work well in dynamic environments. It is also criticised for associating topics with queries, which may lead to inaccuracies in the search results, e.g. if the topic is incorrectly generalised. The algorithm is not evaluated any further in [63].

4.3 Algorithm requirements

From the results presented by Tsoumakos et al. in [63] it can be concluded that informed algorithms consume less bandwidth and have higher search accuracy. However, when the topology changes, many of the algorithms fall back to less bandwidth efficient approaches like flooding.

In a tactical network there are many mobile nodes, which leads to frequent changes in topology. The mobile nodes are expected to have less bandwidth available than deployed or stationary nodes [41]. We assume that in a combat situation both mobility and search activity increases simultaneously. An algorithm that performs worse when it is most needed can not be considered optimal.

As an aid when evaluating new algorithms Tsoumakos et al. present a set of observations based on their work. Their observations can be summarised as follows:

1. Blind forwarding is not adequate for both high performance and low message cost.
2. Index semantics play an important role: Direct location information is effective but sensitive to changes and more demanding (becomes obsolete if a failure/relocation occurs, requires update messages).
3. Adaptation is a key characteristic through which peers that have a prolonged stay in the network enhance their knowledge with time.
4. In many cases the simple protocols are the preferred ones. The simplicity of the mechanisms behind flooding or random walks make them powerful and easy to implement. They can be used either by themselves or in combination with other schemes to improve their performance.

From these four observations we can deduce that the ideal search algorithm for a tactical network should be informed, sensitive to changes, self adapting and simple.

4.4 Advertisement based Search Algorithm for unstructured P2P (ASAP)

ASAP [34] is an informed algorithm that, according to the authors, is developed to reduce bandwidth requirements and improve search accuracy in unstructured P2P networks.

The basis for the ASAP algorithm is four statements made in [34] about file sharing P2P networks on the Internet:

- Large parts of the traffic in file sharing P2P networks consist of search queries.
- The rate at which queries are performed tends to fluctuate with usage patterns, e.g. more queries are performed during the day than during the night.
- Contents shared on many nodes do not change very often, if ever. And they usually do not further share the documents downloaded from other peers.
- Interest clustering is common in P2P systems.

When a node joins the overlay, ASAP broadcasts an advertisement into the network describing keywords for the content it is sharing and which topics are associated with it. The advertisements consist of Bloom filters and a version number, giving a dense representation of the keywords present on each node. Whenever changes are made to the advertisement, e.g. when new content is shared, an update is broadcast in the same way as the initial advertisement, but in a compressed format.

When a node looks up a search term, the node first inspects its locally cached advertisement index. If a match is found, the node knows with a certain probability where the data is located. Since this probability never reaches 100%, a message asking for confirmation needs to be sent directly to the node holding the data. If the node acknowledges the match, the search is considered successful.

If more results are needed or no match is found, the node broadcasts its need for updated advertisements for a set of topics to all its neighbours. The neighbours respond with all advertisements they have in their cache that match the requested topics. This mechanism is intended to gradually move advertisements toward nodes that frequently search for them, based on the assumption that nodes with the same interests often are clustered together in the overlay.

ASAP was not evaluated in [63], as it was not proposed at the time the survey was written. The rest of this section is used to evaluate ASAP in terms of the observations made by Tsoumakos et al. summarised in Section 4.3.

Observation 1 says that informed algorithms perform better than blind in terms of bandwidth. With flooding mechanisms, the accuracy can be high, provided the algorithm uses a large enough TTL, but this comes at the cost of significantly increased bandwidth consumption. ASAP does not blindly search for data, but rather predistributes advertisements to enable peers to directly contact the node that may have the content.

The fact that ASAP uses predistributed advertisements to optimise search may be a desirable feature in a tactical network. Most of the bandwidth cost is when a connection is first made to the overlay and the advertisements are exchanged. Such a solution could be suitable in a military network where units are likely to start out close to each other where bandwidth is high, and later spread out geographically. At this point the advertisements would already be exchanged, and the search traffic can be kept low.

This leads us to observation 2. By predistributing advertisements, ASAP is expected to be quite effective in static networks, but runs the risk of returning outdated search results. This may not be a problem, however, since ASAP broadcasts updates as well — but it should be expected that nodes that are too far away to receive these updates may need some time to acquire the latest information. The algorithm relies on new nodes aggressively distributing their advertisement when they join, but old nodes are less aggressive. This can lead to slow advertisement propagation in growing networks.

Peers using ASAP broadcast request messages when too few results are found. This addresses observation 3. Even if the network is changed, the nodes are able to adapt by requesting new advertisements from their neighbours if the previously received information turns out to be inadequate or obsolete. ASAP does not fall back to a flooding mechanism, thus the bandwidth expense is not as large as in some of the other algorithms — but this is at the cost of accuracy while the advertisements are redistributed. The effectiveness of the request messages should also be considered, as it may take a long time for the advertisements to move towards the right nodes after a reconfiguration.

In their own evaluation, Gu et al. [34] present good results with changes in 20% of the nodes in the network. Since a confirmation message always is sent directly to the content holder before the query is considered successful, ASAP never returns outdated data — but may use more bandwidth for failing confirmation messages.

Regarding the last observation, ASAP may have a disadvantage in that it uses mechanisms that are not as easy to implement as e.g. flooding or random walk. In addition, some extra computation is required in each node to calculate the Bloom filters. It should also be noted that the description of the algorithm in [34] is unclear on some points, e.g. the exact join mechanism they used in their experiments and how update and refresh advertisements should be distributed. These issues may prevent widespread adoption of the algorithm.

4.5 Summary

Common for search algorithms for unstructured overlays is that they either require too much bandwidth (Local Indices, Flooding, Random walk), or that they respond poorly to changes in the overlay (GS, Routing Indices, DRLP) — either by further increasing bandwidth usage (GS), or by getting significantly reduced success rate (DRLP). Superpeers seem to be one of the more flexible solutions. They are used in many of the most popular P2P overlays today (Gnutella, JXTA). However, superpeers may not be the best solution for tactical networks as they make the network vulnerable to targeted attacks to central nodes.

An algorithm that may be suitable for tactical networks is ASAP. ASAP has no single point of failure, adapts to changes, has low bandwidth requirements and, due to the final confirmation message, always returns a fresh view of the services available. There are however some aspects that should be examined further, like the effect of advertisement distribution when new nodes join and the effectiveness of the request message. As far as we know, the algorithm has only been evaluated in simulations by the original authors and in this thesis.

5 Design and implementation

In this chapter, Gnutella and ASAP are described in more detail, as well as how our implementation was made to evaluate their search performance. The new search algorithm Search+ that addresses some of the discovered shortcomings of ASAP is also introduced and implemented.

5.1 Technology basis

ASAP and Search+ are both search algorithms for unstructured P2P networks. They are not tied to a specific protocol and can be used to aid content discovery in any unstructured overlay. Both algorithms are designed to work in pure P2P systems without superpeers. Since Gnutella 0.4 is completely unstructured and well understood, ASAP and Search+ are implemented as extensions to the Gnutella protocol.

Gnutella 0.4 and ASAP are now described in more detail before Search+ is introduced in the next section.

5.1.1 Gnutella

In Gnutella, all communication is done over TCP/IP connections. File transfers, however, are done over HTTP, so each Gnutella peer runs a small HTTP server.

When two nodes connect to each other, the connecting node opens a TCP connection and sends a join request «GNUTELLA CONNECT», whereas the host replies with «GNUTELLA OK» if the join succeeds.

After a node has joined the network it may wish to discover more nodes than just the one it is connected to. In Gnutella this is accomplished by sending a PING message. The PING message usually has a TTL of 7, which means that it will be transferred between neighbouring nodes 7 times. If we assume that each neighbour has 10-20 other neighbours, the total amount of traffic generated by a PING request is quite large and it has been shown that it increases super-linearly [53]. Many of the neighbours are interconnected however, so some of the messages will be discarded as duplicate traffic.

After the PING request is sent, the sender starts to wait for replies. Each node that receives the PING message will respond with a PONG message, which is routed along the same path as the PING message back to the original sender. By collecting the PONG messages a node can build up a cache of nearby nodes, which it later can use as new neighbours or keep as backup in case one of its current neighbours should go offline. The PONG message also contains some additional information about bandwidth and number of files shared, which may enable nodes to make educated choices when determining which nodes to choose as new neighbours.

When a node has joined the network and established a reasonably large number of connections to other nodes, the node may begin to send queries looking for information. The QUERY message works much the same way as the PING message, but only the nodes that actually have the data being requested will respond. The response will be sent as a QUERYHIT message, containing a list of the files that match the original QUERY along with bandwidth information.

The last message type in Gnutella is called PUSH and is sent whenever a node that responded with a QUERYHIT is behind a firewall, making direct download of the files impossible. The downloader may then send a PUSH message with a «servant identifier» which is a 16 bit integer that uniquely identifies the blocked host in the network. This message will then be routed towards the node behind the firewall, and when it reaches its destination the firewalled node will initiate the download. The «servant identifier» needed in the PUSH messages is included in every QUERYHIT message.

Table 5.1 contains an overview of the messages used by Gnutella 0.4.

Message	Descriptor	Carries
PING	0x00	Nothing
PONG	0x01	Number of files and kilobytes shared
PUSH	0x40	Servant identifier, which file to push, ip and port to deliver the file to.
QUERY	0x80	Search criteria
QUERYHIT	0x81	A list of files matching the original query, including name and size. Sender's bandwidth. A unique servant identifier.

Table 5.1: Gnutella 0.4 message types.

5.1.2 ASAP

ASAP is a search algorithm for unstructured P2P networks proposed by Gu et al. in [34]. An overview of its functionality can be found in Section 4.4. The main idea is to spread indexes containing Bloom filters (see Appendix B) across the network to enable nodes to perform most of their searches without having to send anything but confirmation messages through the P2P overlay. Compared to many other search algorithms, this gives a significant decrease in response times and it reduces the bandwidth needed per query. The algorithm does however have some shortcomings that are examined further in Chapter 6.

In ASAP, data is divided into topics. These topics are then assigned to the advertisements to aid the distribution algorithm. It is intended that each node should primarily receive advertisements regarding topics they are interested in. In an Internet based P2P network two topics could be «music» and «documents». If the P2P network was mainly concerned with music, the topics could be more specific, e.g. «pop» and «techno». ASAP makes the assumption that nodes have specific interests and rarely changes them (see the list of ASAP assumptions in Section 4.4). All nodes must agree on the same set of available topics in the network.

5.1.3 Advertisements

ASAP uses three advertisement types, as shown in Table 5.2.

A *full advertisement* comprises a node identity, a Bloom filter describing the node's shared resources, a list of topics covered by the Bloom filter and a version number, as shown in Figure 5.1. Nodes may choose to only cache information that is relevant to them to save resources. A full advertisement is sent by new nodes when they join the network and when any of the nodes requests them. The version number is used to keep track of the latest version of the advertisement.

A *patch advertisement* is sent when changes are made to the shared content of one of the nodes. This is usually whenever they start or stop sharing a service. The patch advertisements are collected

Advertisement	Description
Full	A full advertisement with a complete Bloom filter, topics and version number.
Patch	Changes in the Bloom filter since the last update was sent.
Refresh	An empty update. Tells the recipient that the sender is still available.

Table 5.2: ASAP advertisement types

by the neighbours and applied to the previously received full advertisements.

Refresh advertisements are empty advertisements that tells the other nodes that the sender is still alive, and that nothing has changed since the last patch or full advertisement was sent.

In addition to these advertisements, a node may also send a *request* for advertisements concerning a specified group of topics. When a new node joins the network it will send a request to all its neighbours. This request carries information about all the topics the new node is interested in. Receiving nodes will go through their cache of known advertisements and forward those that match the requested topics.

The advertisements need to be propagated more than to just the immediate neighbours in the P2P overlay to be effective. In [34], Gu et al. test several distribution methods, among them flooding and random walk. The key to achieving a good hit ratio in ASAP is to keep the advertisements thoroughly distributed in the network. Which distribution algorithm that will work best is closely related to the network topology and bandwidth requirements.

Node ID	Bloom filter	Topics	Version

Figure 5.1: A full ASAP advertisement. *Node ID* is a unique identifier for the node that created the advertisement. *Bloom filter* contains a bit pattern used to match keywords against the content held by the creator. *Topics* is a list of topics covered by the content in the Bloom filter. *Version* is a version number that is incremented for each change to the topics or Bloom filter fields.

5.1.4 Join process

The join process is not described in detail in the original paper, but it is mentioned that nodes send an initial request for full advertisements on joining the overlay. We therefore assume that each joining party will send a request for advertisements matching their area of interest. This will result in a collection of full advertisements being transferred between the nodes, containing all cached advertisements matching the requested interests. In addition, we assume that the node will send

an advertisement describing the content it is sharing. This advertisement is sent with the chosen distribution method and a certain TTL into the overlay to properly distribute information about new nodes in the network.

Algorithm 1 ASAP join process. Executed after the underlying overlay has performed a successful join.

$N \leftarrow newlyJoinedNeighbour()$
$I \leftarrow myInterests()$
//Request all advertisement from N that matches our interests I
$requestAds(N, I)$
//Send an advertisement describing shared content
$advertiseSharedContent(N, ttl)$

The join process is shown in Algorithm 1. A *request* for advertisements matching our interests I is sent to the new node N. The request is initially sent with a TTL of 1, but one could extend the algorithm to increase this value if even more results are needed. The reply is handled by the maintenance process, as described in Section 5.1.5.

After requesting content that matches its interests, the node advertises the content it is sharing. This is done by sending an advertisement into the network that describes the local content. The TTL value has to be high enough to get the advertisement properly distributed in the network, while still limiting the bandwidth usage. In [34], a TTL of 6 is used with Random Walk as the distribution method and this yields good results for a 10000 node crawled simulation network.

5.1.5 Maintenance

The maintenance process' main concerns are keeping the advertisement cache up to date and responding to advertisement requests. These two tasks are described by Algorithm 2 and Algorithm 3.

As shown in Algorithm 2, ASAP will look through all the advertisement requests that have been received. For each request r it will look through the local advertisement cache $localAdsCache$. Advertisements having one of the topics matching the requester N's interests I will be added to the set K. If one or more advertisements are found, they are sent to N.

In Algorithm 3 any newly received advertisements are added to the local advertisement cache. The new advertisement $newAd$ is checked against the cached advertisements. If a match is found in the cache, the cache is updated with $newAd$ given that the version number V is higher than the earlier cached advertisement. If this is the first time we receive an advertisement from N, the advertisement in $newAd$ is always stored.

5.1.6 Search mechanism

Algorithm 2 ASAP maintenance process, part 1.

```
//Respond to advertisement requests
for all r ∈ received advertisement requests do
    N ← getRequestSource(r)
    I ← getRequestedInterests(r)
    K ← ∅
    for all ad ∈ localAdsCache do
        if matchesInterest(ad, I) then
            K ← K ∪ ad
        end if
    end for
    //Send the matching advertisements to N
    if K ≠ ∅ then
        sendAdvertisements(N, K)
    end if
end for
```

Algorithm 3 ASAP maintenance process, part 2.

```
//Add newly received advertisements to cache
for all newAd ∈ received full advertisements do
    N ← getAdNodeId(newAd)
    V ← getAdVersion(newAd)
    for all cachedAd ∈ cached advertisements do
        if getAdNodeId(cachedAd) = N then
            if V > getAdVersion(cachedAd) then
                storeAd(newAd)
            end if
        end if
    end for
    if no cached advertisement for node N was found then
        storeAd(newAd)
    end if
end for
```

Algorithm 4 ASAP search algorithm.

```
I ← getOurInterests()
K ← getSearchTerms()
R ← ∅ //R will contain the results
for all ad ∈ localAdsCache do
  B ← getBloomFilter(ad)
  if match(K, B) then
    S ← getAdSource(ad)
    sendConfirmationRequest(S, K)
    if isConfirmed(S, K) then
      R ← R ∪ S
    end if
  end if
end for
if we have enough results in R then
  return R //return the results
end if
//Request more results from all neighbours and append them to A
A ← ∅
for all n ∈ getAllNeighbours() do
  A ← A ∪ requestAdsFromNeighbour(n, I)
end for
//If A contains new ads, look for more matches
if A ≠ ∅ then
  for all ad ∈ A do
    F ← getBloomFilter(ad)
    if match(K, B) then
      S ← getAdSource(ad)
      sendConfirmationRequest(S, K)
      if isConfirmed(S, K) then
        R ← R ∪ S
      end if
    end if
  end for
end if
return R //return the results
```

The ASAP search mechanism is described in Algorithm 4. First, the search terms in K are matched against the Bloom filters B present in the local advertisement cache. When a match is found, a confirmation request is sent to the node S that sent the original ad. If the node confirms the match, the result is added to R.

If the first part of the algorithm returned too few results, the second part of the algorithm will try to request more ads from the neighbours. First, it will loop through all the neighbouring peers and request more ads that matches our interests. Then, if more advertisements are received from the neighbours, loop through all of them and try to match K against the Bloom filter B. A confirmation is sent if a match is found, and the result is added to R. The confirmation process is identical to the confirmation process used in the first part of the algorithm.

When the algorithm completes, the result set R is returned. By repeatedly requesting new advertisements from neighbours that match a given interest, the search algorithm will make sure that advertisements over time move toward nodes that look for them.

5.2 Search+ design

In this section, the new algorithm Search+ is presented. Search+ was created to meet some of the shortcomings of ASAP. ASAP distributes its advertisements primarily during the join phase, which means that only nodes that are already connected to the network will receive the first advertisement. After that, the algorithm relies on search queries to trigger further distribution of advertisements in the network, which in itself requires bandwidth and is not very effective, as is shown in Chapter 6. Search+ will, by establishing subscriptions for advertisements between the nodes in the overlay, provide a more targeted and bandwidth efficient distribution method — enabling good accuracy from the first query.

Search+ follows many of the same principles as ASAP, in that it uses advertisements with Bloom filters and a similar search and confirmation mechanism. The idea in Search+ however, is that each node will *subscribe* to advertisements that match their interests instead of just blindly broadcasting them into the overlay.

Advertisements in Search+ contain a Bloom filter, a node identifier and a version number. This is identical to the full advertisements used in ASAP, as can be seen in Figure 5.1. Other advertisement types can easily be implemented, like for instance the patch advertisement to further compress changes when they are propagated in the network.

5.2.1 Join process

When a new node joins the network, the algorithm described in Algorithm 5 is executed. Its task is to tell the new neighbour what topics we are interested in by requesting a *subscription*. Note that Search+ does not broadcast an advertisement into the network at this point.

Algorithm 5 The Search+ join algorithm

//Send a *subscription* request to the new neighbour
$I \leftarrow getOwnInterests()$
$N \leftarrow$ new node
$sendSubscriptionRequest(N, I, ttl)$

The value chosen for TTL will determine how deep into the network the request is sent. Experiments in Chapter 6 show that the achieved success rate and bandwidth consumption of Search+ is largely determined by the TTL value.

5.2.2 Maintenance

The maintenance process in Search+ has two main tasks; to *process* newly arrived subscription requests and to *publish* advertisements.

In Algorithm 6, each new request is processed and the requested interest is stored in a table. Each node remembers the interests requested from each of its neighbouring nodes.

Algorithm 6 The Search+ algorithm executed while processing newly arrived subscription requests

```
//Process all subscription requests
for all r ∈ newly received requests do
  N ← getSourceNode(r)
  requestedInterest ← getInterests(r)
  ttl ← getTTL(r)
  storeSubscription(N, requestedInterest)
  if ttl > 0 then
    totalInterest ← ∅
    for all interest ∈ active subscriptions do
      totalInterest ← totalInterest ∪ interest
    end for
    forwardSubscriptionRequest(totalInterest, ttl − 1)
  end if
end for
```

Note that new subscription request messages are only sent when two nodes join or when a node's interests change. The request message always contains the topics the node itself is interested in, in addition to the interests requested by its neighbours. This helps establish paths in the overlay that newly inserted advertisements travel along to reach interested parties.

When a node receives a request message with $TTL > 0$ it adds its own interests and its other neighbours' interests to the message before forwarding it.

New advertisements are published as described in Algorithm 7. The algorithm goes through all its neighbours and checks which of its cached advertisements will match one or more of their interests. After retrieving all the relevant advertisements for node N, the ones in $sentAds$ that have already been sent are removed from the set. The result is then sent to N and added to the list of advertisements that N has received.

Algorithm 7 The Search+ algorithm used to publish newly received updates

//Publish new advertisement to interested neighbours
for all $s \in$ active subscriptions **do**
  $N \leftarrow getSubscribingNode(s)$
  $I \leftarrow getNodeInterests(s)$
  $sentAds \leftarrow getSentAdvertisements(N)$
  $updatedAds \leftarrow getAdvertisementsMatching(I) \notin sentAds$
  $sendUpdatedAvertisements(N, updatedAds)$
  $storeLastSentAdvertisements(N, sentAds \cup updatedAds)$
end for

5.2.3 Search mechanism

The search process in Search+ is very straight forward. As described in Algorithm 8, the local advertisement cache is first processed for Bloom filters matching the search terms. For each match that is found a confirmation request is sent to the node possibly holding the content. If this fails, the search ends, as there is no fall back method that could provide more search results, besides sending a subscription message deeper into the overlay. This should only be done as a last result as it will trigger quite a lot of traffic.

Algorithm 8 The Search+ search algorithm.

$I \leftarrow getOurInterests()$
$K \leftarrow getSearchTerms()$
$R \leftarrow \emptyset$ //R will contain the results
for all $ad \in localAdsCache$ **do**
  $B \leftarrow getBloomFilter(ad)$
  if $match(K, B)$ **then**
    $S \leftarrow getAdSource(ad)$
    $sendConfirmationRequest(S, K)$
    if $isConfirmed(S, K)$ **then**
      $R \leftarrow R \cup S$
    end if
  end if
end for
return R

Note that this differs from ASAP that will request new advertisements from its neighbours as a fall back mechanism. This makes sense in the ASAP algorithm, because this is what gradually moves advertisements towards the interested nodes. In Search+ however, this process should already have taken place when the advertisement subscriptions were established during the join process.

5.3 Implementation

In this section, the Gnutella implementation used to evaluate ASAP and Search+ is presented in more detail. The component framework OpenCOM [21, 24] and the newly developed Juno [64] are described, as well as how components were used to implement Gnutella and a pluggable search module that supported ASAP and Search+.

5.3.1 OpenCOM and Juno

OpenCOM is a component framework developed by researchers at Lancaster University. It is available as C++ and Java implementations, but all the components used in this thesis are written in Java.

In Java, OpenCOM makes use of the reflection mechanisms to allow objects to connect to each other at run time. To create a new component, one creates a new class that implements a set of OpenCOM interfaces in addition to having a set of receptacles. The receptacles are essentially public variables of a given type that can be used during run time to create connections between the components. Components can only connect to other components that have a suitable receptacle. The OpenCOM version used in this thesis is described in detail in [24] and all the source code and user manuals are available online.

Juno [64] is a «reconfigurable middleware for heterogeneous content networking» built on top of OpenCOM that allows a more fine-grained separation of the components, in addition to giving extra functionality, like event handling. At the time this implementation was created, the framework did not include support for Gnutella and the code written for this thesis has been submitted to the Juno project.

5.3.2 Gnutella

The Gnutella client implemented in this thesis supports a subset of the Gnutella 0.4 protocol. The actual download features are omitted, including the PUSH messages for downloading files behind firewalls, but otherwise the implementation is a fully functioning client.

The functionality had to be separated into components, as shown in Figure 5.2. As in Juno, the main components supporting the overlay are *construction*, *maintenance*, *forward* and *transport*. In

addition, each message type in Gnutella has been given its own component. All the components, except transport, share the same *state object* that stores the current state of the overlay.

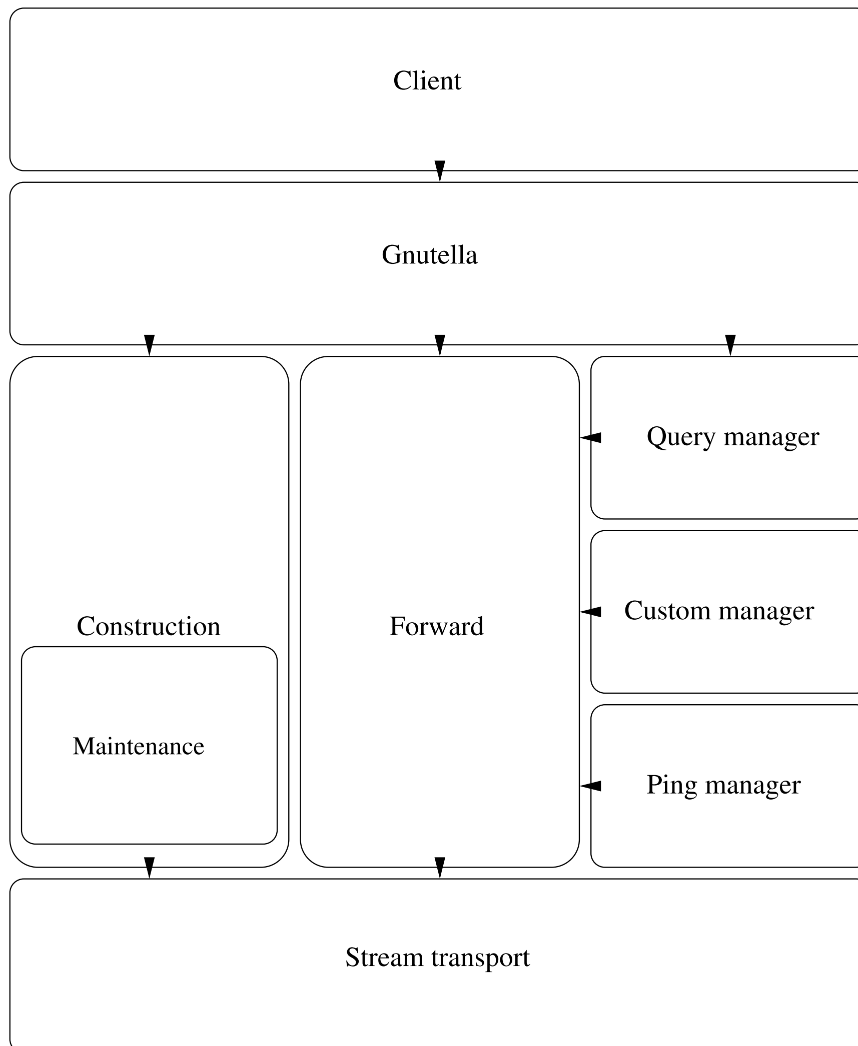

Figure 5.2: The component model used in the Gnutella client.

When the client starts, it first calls the Gnutella driver, which is a short program that initiates all the needed components and connects them to each other. After starting up and connecting the underlying components to the main Gnutella component, the driver returns a normal Java object with the ability to join, leave and perform queries in a Gnutella overlay.

The different parts of the implementation are now described in more detail.

5.3.2.1 Global state object

All components have access to a global Juno repository. In this repository an overlay state object is made available to the Gnutella related components by the driver. This state object contains the following:

- List of nodes that are immediate neighbours.
- List of nodes we know about and whether they are reachable or not.
- List of shared files.
- Routing table containing message type, stream ID's and target node.
- List of message IDs used for discarding duplicates.

Items are removed automatically from the routing table and the list of received message IDs after 10 minutes. The state is maintained by the components and does not contain any functionality, except for some basic book-keeping.

The state object is not shown in Figure 5.2, as it is not part of the component model — although many of the components depend on it.

5.3.2.2 Stream transport component

This is an extended version of the transport component used by Juno. It provides methods for reliably sending and receiving Java objects over TCP/IP. It is essentially an abstraction layer on top of the Java Sockets API, adding support for Juno's message listeners and events.

Message listeners are other components that have registered their interest in a specific message type. The transport component will pass received messages to the listeners if it matches a listener's criteria. Messages may be matched by Juno stream ID, message tag or message type. In Gnutella, the different message types are given their own message tag, thus allowing the different components to only receive messages relevant to the functionality they provide.

The transport component will keep track of all the connections opened to other hosts. Other components need only send the address of the destination together with the object to be transferred, and the rest will be handled automatically. On failures, an event will be passed to listening components.

During the evaluation phase we had to extend the provided Juno transport component to support queueing, so that both outgoing and incoming messages are queued before they are passed to the Java API and the listening components. This was necessary because the platform running the evaluations had a tendency to malfunction if the queues on the network interface became too long. By using thread safe queues, some stability issues under high load in the original implementation were also solved.

5.3.2.3 Construction component

The construction component is responsible for constructing the overlay. In Gnutella, this mainly concerns performing joins and responding to join requests from other hosts. The construction component is connected to the transport component, and uses its send and receive primitives to send

and receive *LEAVE* and *JOIN* messages. Whenever a node successfully joins or leaves, it will pass an event that can be handled by the maintenance component or other components that may be interested.

5.3.2.4 Maintenance component

The maintenance component tries to make sure that the client is connected to a given number of nodes. In the Gnutella protocol specification, this number is set to 10. The reason for having many neighbours is that this provides a certain degree of resilience in the network. By always keeping in touch with a large enough number of nodes, a node will not be significantly affected by suddenly disappearing neighbours or network partitioning. Having many neighbours will also increase search accuracy when doing flood searches, since the request is sent more times into the overlay.

This component is mainly a loop that checks the overlay state for the number of connected neighbours at given intervals. If this number is too low, it will look in the state for earlier observed nodes that are not marked as down or unreachable. It will then ask the construction component to connect to one of them. This is repeated until all known nodes have been tried or the number of neighbours have reached the predefined threshold.

Additionally, the maintenance component listens to the following events: socket failure, node failure, node join and node leaves. Whenever one of these events is received, the maintenance component will make sure that the global overlay state is updated correctly.

5.3.2.5 Forward component

The forward component provides higher level functionality for sending messages within the overlay after it has been constructed. It is connected to the transport component and provides additional routing and broadcast functionality. It also adds support for TTL, which is used by Gnutella to discard messages after a predefined number of hops. Moreover, it maintains the routing table in the global state object.

The forward component provides a *send* method that will decrease the TTL value and discard the message if it reaches 0, before it passes the message to the transport component. It also provides a *broadcast* method that will either send the message to all connected neighbours (flood) or to a number of randomly selected neighbours (random walk). Which broadcast method that should be used is specified during initialisation of the forward component.

Finally, the forward component offers the *forward* method. This method will look at the tag and stream id of the message being sent, and see if it has a route for it in the routing table. If it has, the message will be forwarded to the correct node. By adding backward routes while messages are being propagated through the network, Gnutella enables replies to be routed back to the sender — e.g. the query component would add a backward route for a QUERYHIT with a given stream ID for

each QUERY it forwards to its neighbours. When a QUERYHIT response message is sent by a node that has matching content, it makes sure to use the same stream ID as the original query message, thus allowing the nodes that forwarded the original query to find a backward route in their routing tables.

5.3.2.6 Ping manager

Gnutella uses PING and PONG messages to discover new nodes in the overlay. The discovered nodes are added to a list of known nodes in the global state, where they can be used by the maintenance component as new potential neighbours.

The Ping manager listens to the forward component for messages of type PING or PONG.

When a PING message is received, a backward route is added for a PONG message in the global state before the message is passed to the forward component to be broadcast to other neighbours.

When a PONG message is received, the node information within it is recorded as a known node in the global state, before it is passed to forward. The forward component will consult its routing table for a recorded route for this message, and if one is found (and the TTL is more than 0), the message will be forwarded to the node in the other end of the route — which hopefully is the same node that sent us the original PING message.

5.3.2.7 Query manager

The Query manager listens to the forward component for messages of type QUERY or QUERYHIT. Similarly to the Ping manager, it will broadcast any query message it receives to all neighbours after registering a return route, and when a QUERYHIT is sent back it will forward it to the node it received the query from.

In addition, the Query manager will match received queries against the list of locally shared content in the global state. If a match is found, a QUERYHIT message is generated and sent back to the node that performed the query.

5.3.2.8 Custom manager

The Custom manager is an extension of the Gnutella 0.4 protocol created for this thesis, and it is not used when the implementation is functioning as a proper Gnutella client. The manager provides routing for a new message type, the CUSTOM message. Like the Ping and Query manager, it will record reply paths for all CUSTOM messages it receives, and forward CUSTOMREPLY messages back to the original sender.

The CUSTOM and CUSTOMREPLY messages are used by ASAP and Search+ to distribute advertisements in the Gnutella overlay.

5.3.2.9 Gnutella component

The main Gnutella component ties all the other components together and implements them under a common interface. This interface includes methods for joining, leaving and querying the overlay. It also enables other components to listen in on events being passed by Gnutella components, allowing itself to be extended, as we shall see in the next section.

5.3.3 Gnutella with ASAP

To make Gnutella use ASAP as the main search mechanism, the Query component would have to be exchanged with ASAP. Since this component is an integral part of Gnutella however, the ASAP component was implemented outside Gnutella, but listening to events from the main Gnutella component. This means that the component must be called explicitly to perform an ASAP optimised search — but it also enables the two search mechanisms to coexist in the same overlay and stay compatible with the original Gnutella protocol.

The modified Gnutella component model is shown in Figure 5.3. Two new components are added; the ASAP manager and the datagram transport component.

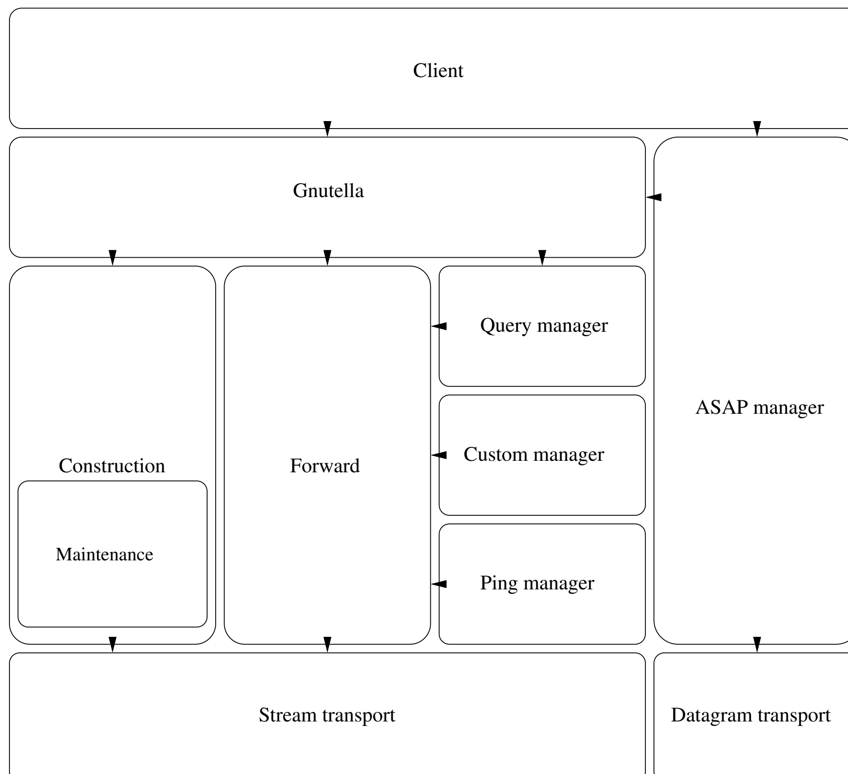

Figure 5.3: The components used when running Gnutella with ASAP. The ASAP component is implemented as an extension to the original protocol. The datagram transport component is needed for confirmation messages.

5.3.3.1 Datagram transport component

The datagram transport component is an abstraction of the Java Sockets API, but instead of implementing TCP like the stream transport component, it implements UDP. UDP is needed by ASAP and later Search+, to confirm the results they find in their Bloom filters.

At the time this implementation was written, the Juno framework did not have a component for UDP transport, so this component is written from scratch, although its structure is inspired by the original Juno TCP Transport component.

Since UDP is unable to guarantee the delivery of the datagrams, each datagram is sent twice with a random delay. To avoid long delays, the component also uses two additional threads to receive and send data over the socket. If the time it takes to receive a message becomes too long, messages could be dropped before we have had the opportunity to inspect them. Messages are therefore immediately placed in a queue, where they are later handled by the appropriate thread.

5.3.3.2 ASAP manager

The ASAP manager implements a query method that can be called from the client component. Additionally, it listens in on events from the Gnutella components so it can be notified when files are shared or new nodes are added or removed from the overlay. The ASAP component also has access to the Gnutella state object and an ASAP state object specific to ASAP.

The ASAP state object contains the following:

- Cache with received advertisements.
- Our own advertisement and Bloom filter.
- List of topics we are interested in.

ASAP listens to CUSTOM and CUSTOMREPLY messages. These messages are used for sending requests for advertisements and replying with the actual advertisements. The routing is handled by the Custom manager component, but ASAP still needs to inspect the messages and extract the content.

When a new node joins, the ASAP component will first locate its own advertisement in its state object and send it to the new node through the Gnutella component with a modified CUSTOM message. Afterwards, it will send a request for cached advertisements that matches our interests. This request is also sent as a CUSTOM message so that the CUSTOMREPLY may be routed back to us with the results. This is the join part of the algorithm, described in Algorithm 1.

When a CUSTOM message is received, ASAP will extract the requested interests and find all advertisements that matches the relevant topics in the state object. These advertisements are then put in

a CUSTOMREPLY message that is routed back to the requesting node. The CUSTOM message may also contain an advertisement that is being broadcast through the overlay. If an advertisement is found within the message, it is stored in the advertisement cache in the state object if it is new or has a higher version number than the previous version.

When a file sharing event is received, ASAP will store the file name in its own advertisement within the state object. If the file has one or more topics, these topics are added to the list of topics we are interested in.

When the query method of the ASAP component is called, the Bloom filters stored in the state object are checked one by one. If a match is found, a confirmation message is sent with the datagram transport component. If the match is confirmed the reply message will contain additional information about the shared content that can be added to the search result. If too few or no results are found, a CUSTOM message with a request for more advertisements concerning our interests is broadcast to all neighbours. The client performing the query will receive a query ID that can be used to check if the query has aggregated more results at a later time.

When a CONFIRMATION message is received from the datagram transport component, it will contain a list of search criteria. These criteria are checked against the files shared by the Gnutella component, and if a match is found we return the file name directly to the requesting node by sending a CONFIRMATIONREPLY message with the datagram transport component. If no match is found, no response is sent.

5.3.4 Gnutella with Search+

Search+ is implemented in much the same way as ASAP, by using a separate component that connects to Gnutella and listens in on events. It also connects to the same datagram transport component for sending and receiving confirmation messages. The Search+ component model is shown in Figure 5.4 — the only new component is the Search+ manager.

5.3.4.1 Search+ manager

The Search+ manager handles advertisement distribution and the search mechanism of Search+.

Search+ has its own state object that contains the following:

- Cache with received advertisements
- A list of topics the node is interested in.
- Our own advertisement and Bloom filter.
- A list of the topics neighbours are interested in.

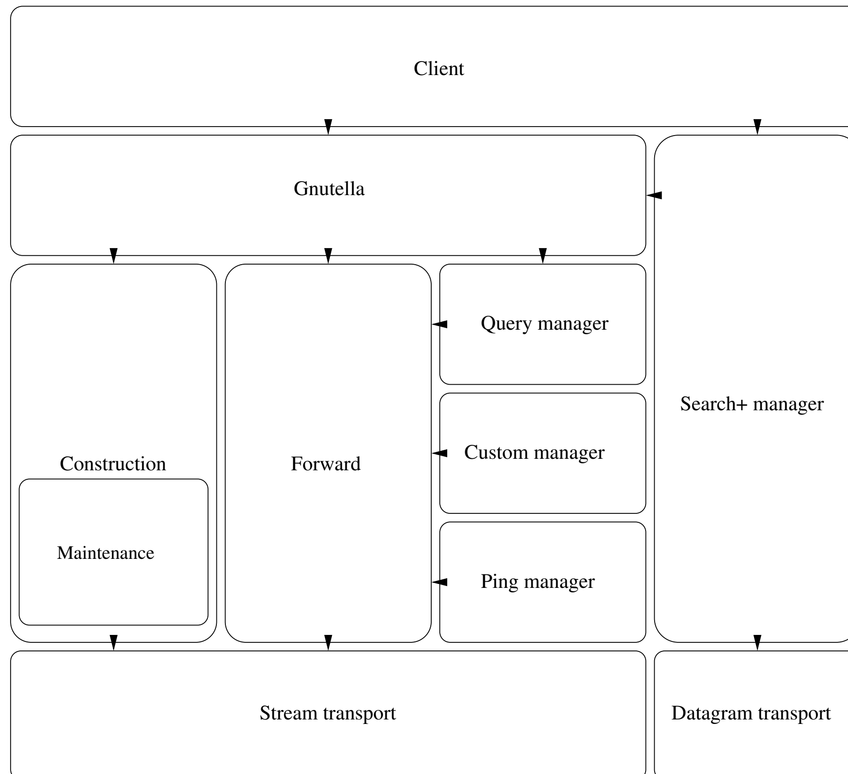

Figure 5.4: The components used when running Gnutella with Search+. The Search+ component is implemented as an extension to the original protocol. The datagram transport component is needed for confirmation messages.

- Cache with advertisements sent to each neighbour.

Search+ listens to CUSTOM and CUSTOMREPLY messages, in addition to the events for nodes joining or leaving and file sharing.

When the manager receives a join event from the Gnutella component, it will gather a list of topics it is interested in from the local state, and combine it with the list of topics neighbours are interested in. This list of topics is then sent as a subscription request to the new node, enveloped in a CUSTOM message with a predefined TTL value.

When a subscription request is received, the interests within it are added to the list of topics neighbours are interested in for the node that sent the subscription request. If the subscription request has a TTL above 0, a new subscription request will be generated, which includes the requested interests, other neighbours' interests and the node's own interests. This request is then forwarded with TTL 0. In this way the subscription request will aggregate the interests of the passing nodes while it travels through the overlay. The Search+ algorithms are described in more detail in section 5.2.

At regular intervals, the Search+ manager will go through the list of neighbours and generate an index of known advertisements that matches each neighbour's interests. This index is then checked against the cache of advertisement that has been sent to the neighbour earlier. If one or more

advertisements are new to the neighbour, a CUSTOM message is prepared with the new information. This message is then sent directly to the neighbour with a TTL of 1. A larger TTL is not needed, as the receiving node will forward the advertisements based on the subscriptions established by its neighbours.

When a CUSTOM message with advertisements is received, each advertisement is checked against known advertisements stored in the local state. If one of the advertisements is new or contains updated information it will be stored in the cache.

Search is performed by going through all cached advertisements for a Bloom filter that matches the search criteria. When a match is found, a confirmation message is sent with the datagram transport component. A match is not conclusive until the node that sent the advertisement acknowledges that it actually has content that matches the query.

If a confirmation request message is received, the search criteria within it is checked against the files shared by the Gnutella component. If a match is found a confirmation message is sent back, containing additional information about the content — e.g. file size and file name. If no matching files are found, no reply is sent.

The Search+ manager also listens for file sharing events, in the same manner as the ASAP manager. If a new file is shared, information about it is automatically stored in our local Bloom filter and our advertisement version number is increased.

5.4 Summary

The lack of suitable search algorithms for tactical networks, combined with the shortcomings of the promising algorithm ASAP, led to the creation of Search+.

Search+ is advertisement based, like ASAP, but the advertisements are distributed using a more targeted algorithm.

A Gnutella 0.4 client is implemented using the component based frameworks OpenCOM and Juno. ASAP and Search+ are implemented as extensions to the original Gnutella protocol.

6 Evaluation

The test environment and topology used in the experiments are now presented, followed by an evaluation of the search algorithms in pure Gnutella, Gnutella with ASAP and Gnutella with Search+ in terms of bandwidth and search accuracy.

The evaluation in this thesis is done by experimentation, using the implementations presented in Chapter 5. To perform the evaluation, an actual P2P overlay is created, with each peer corresponding

to one running instance of the implementation. Each peer performs search queries regularly, until a given time interval passes and all the peers automatically terminate. During the experiment the network traffic is recorded, along with statistics on search accuracy and response times.

The experiments are designed to simulate a group of nodes that meet and connect, and then communicate for a determined time interval.

The algorithms that are tested are;

- Gnutella 0.4 with flooding.
- Gnutella 0.4 with ASAP.
- Gnutella 0.4 with Search+.

They are referred to as Gnutella, ASAP and Search+, respectively.

6.1 Peer configuration

To make it easier to examine the results, the execution is separated in three phases; *initialisation*, *stabilisation* and *query*. Each phase lasts for a known time interval, allowing us to read exact data for each phase from the packet capture.

During the initialisation phase, the nodes are given 30 seconds to start up, configure themselves and read the topology from a file. After waiting the specified time interval, the peers use the topology information to connect to the nodes they have as neighbours. The overlay is then given 180 seconds to stabilise, which we call the stabilisation phase. After this phase is completed, the nodes enter the query phase, and start sending search queries at regular intervals. This phase lasts until the experiment ends.

The following is an overview of the different phases, their duration and the actions performed by each algorithm during each phase:

- Initialisation
  - Duration: 30 seconds.
  - All algorithms: Initialise and read topology file. Share services.
- Stabilisation
  - Duration: 180 seconds.
  - Gnutella: Connect to neighbours.
  - ASAP: Connect to neighbours and send initial advertisement.

    - Search+: Connect to neighbours and establish subscriptions.

- Query
    - Duration: Depending on experiment.
    - Gnutella: Send flood query.
    - ASAP: Search local advertisements, send request for more advertisements if no match. Send confirmation.
    - Search+: Search local advertisements. Send confirmation.

6.2 Queries

The peers perform a search every 20 seconds, with up to 5 seconds random variation. This means that the delay between queries varies between 20 and 24 seconds.

When entering the query phase, all peers send their first query at the same time. This causes traffic bursts during the first parts of the experiments. After a while, the variation in delay between the queries leads to less bursty traffic, as the queries are distributed more evenly over time between the peers. This gradual change in traffic pattern allows us to see how the algorithms respond during high load, how quickly they recover and how well they function when queries are more evenly distributed.

With an average delay of 22 seconds, each node would send 2.72 queries per minute. For a 100 node overlay, that corresponds to 4.53 queries per second.

6.3 Services and topics

Each node is sharing a number of services that are identified by a service name. If we had been using Gnutella for file sharing, rather than service discovery, this would have been the same as the file name.

Since we are testing possible mechanisms for service discovery where a specific service is often sought, each node in the overlay will serve four randomly chosen *unique* services. The success rate of the search algorithms is hence determined by each algorithm's ability to locate one specific service present in the overlay.

The actual number of services in a tactical network is not known, but we assume that in a network with 100 nodes, 400 unique service names would be more than adequate to represent the available services. We want to keep the number of unique services high, as more unique services are expected to have an adverse effect on search accuracy. If we test the algorithms with too few services, the results are not representable of a real life scenario.

After randomly choosing four services, each service is assigned one of 14 topics. The number of topics is the same as the authors of ASAP used in their simulations in [34].

A topic is intended to correspond to a category the service falls in. Examples of such categories could be tracking services, registry services or surveillance services. As we saw in Chapter 5, both ASAP and Search+ rely on the peers announcing which topics they are interested in to steer advertisements toward the nodes that need them.

A node's interests are the topics of the services it is currently sharing. This is based on the assumption that nodes are more likely to request services within the categories they are themselves producing. In a military network, this is not necessarily the case, but as it is an important assumption in ASAP we have chosen to include it in our experiments. However, since our topology does not have interest clustering, the effect of this assumption should not be very different than if interests were assigned randomly. The topics are ignored by Gnutella.

When performing searches, each node in the overlay has access to a list of the services available on the other nodes. The node will pick a name to search for from this list to ensure that the service is present in the network at the time the search is performed.

ASAP and Search+ assume that nodes will mostly search for services relevant to their interests. We have therefore chosen to let the nodes search for services that fall within the same categories as they are interested in with a probability of 0.9. This means that 90% of the queries from each node will be for services that have the same topics as the node's own interest. The remaining 10% are chosen randomly.

6.4 Bloom filter

When using the ASAP and Search+ algorithms, the Bloom filter size affects both the success rate and the bandwidth consumption. A larger filter increases the success rate, while requiring more bandwidth.

As described in Appendix B, Bloom filters can be described by their values m, n and k, where m is the size of the filter, n is the number of elements to be inserted and k is the number of hash functions.

In our experiments, we must choose an n that is large enough to contain the maximum number of keywords that will be inserted in a single node. m and k are then chosen based on what probability we want for false positives.

We have earlier in this chapter chosen to use four unique services per node, but in a real life scenario we expect that a few nodes will have significantly more services than others. We also expect that each node may use more than one keyword to describe a service. If we assume that each node on average will use four keywords to describe a service, and that the node with the most services have 25 services, the value of n should be set to 100.

Using the calculations in Appendix B, we determine that with $n = 100$, $m = 1000$ and k is optimal at 7. The expected probability for false positives is then 0.00819. The size of the Bloom filter used in the experiments is thus 1000 bits, or 128 bytes.

6.5 Discovery mechanism

The Gnutella 0.4 protocol includes a mechanism for discovering new nodes by sending discovery messages at regular intervals. This discovery mechanism will be disabled during our experiments. The process would consume additional bandwidth [53], but since the mechanism is independent of the search algorithm we have chosen to disable it during the evaluation to avoid interference.

Moreover, when using ASAP and Search+, the peers can generate a list of other nodes based on the advertisements they receive, thus making the Gnutella discovery mechanism unnecessary.

6.6 Choosing TTL

The TTL values affect different mechanisms in each algorithm and are not directly comparable. The TTL value chosen in Gnutella reflects the number of hops the QUERY message will travel into the overlay before being discarded. In ASAP, the TTL specifies how far the advertisements should be sent. In Search+, TTL is the number of hops the initial subscription request is forwarded. The TTL must therefore be chosen independently for each algorithm.

6.6.0.2 Gnutella

In [53], Portmann et al. show that in a power law distributed Gnutella network a TTL of 5 reaches more than 95% of the network. The topology used in our experiments also exhibits power law properties, and we have therefore chosen to use TTL 5 when evaluating Gnutella. The experiments are also repeated with a TTL of 4 to examine the difference in bandwidth and accuracy when the TTL is decreased.

6.6.0.3 ASAP

In [34], the authors of ASAP get the best results in simulations using a TTL of 6. To evaluate differences in performance when the TTL decreases, we will also examine TTL of 5.

6.6.0.4 Search+

During the development of Search+ we discovered that a TTL of 3 gave good results in a topology similar to the one used in these experiments. They will therefore be performed with TTL of 3, and additionally TTL of 2 to examine the differences in bandwidth use and accuracy for lower TTLs.

6.7 Topology

We expect a tactical network to follow the principle of preferential attachment, where connecting nodes will have a higher probability of connecting to nodes that already have many connections. In a military network, it is also likely that the nodes will form hierarchical clusters, due to the command structure [17]. Based on these two arguments, we assume that a real life topology form a scale free network with a power law distribution, as described by Barabasi and Alberts in [19].

The topology used in the experiments is a 100 node Barabasi-Albert distributed topology with two edges from each node. To generate the topology, BRITE [49] was used to create a Level 1 AS BA-model with m=2 and n=100, as described in Appendix A. The topologies generated by BRITE with these parameters have an average degree of approximately 4 (3.94).

For our experiments described later in this chapter, we need two sets of topologies. The first set contains only one topology and is used for the experiments that evaluate the query mechanism. We call this *topology set A*. For our experiment that evaluates the advertisement distribution mechanism, we generate 250 additional topologies in the same manner and call it *topology set B*.

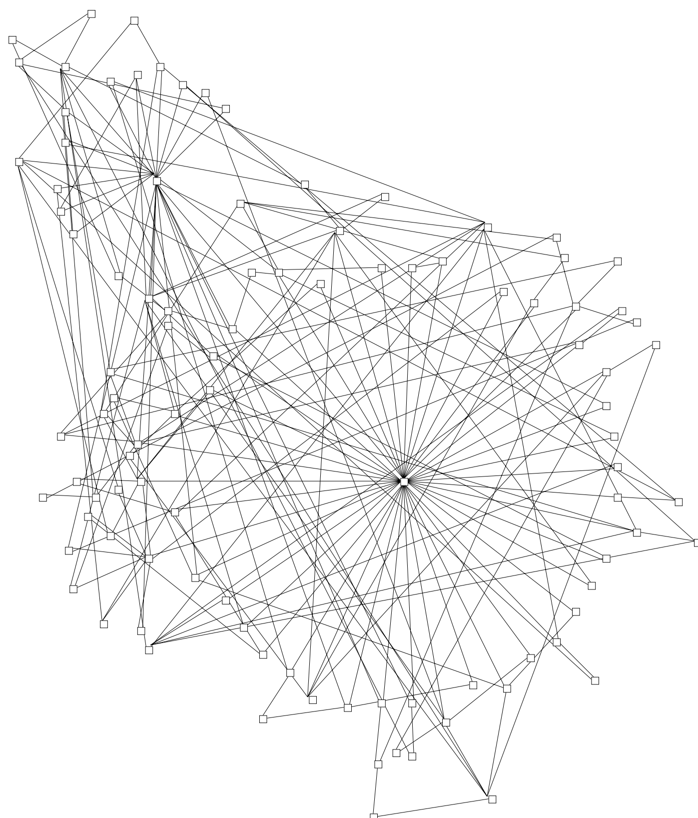

Figure 6.1: A Barabasi-Albert topology with degree 3.94 from topology set A.

Figure 6.1 is generated from the topology in set A. The image is generated by Otter [37]. Most of the edges go to two central nodes, which help form two interconnected clusters, as can also be seen in the edge histogram shown in Figure 6.2.

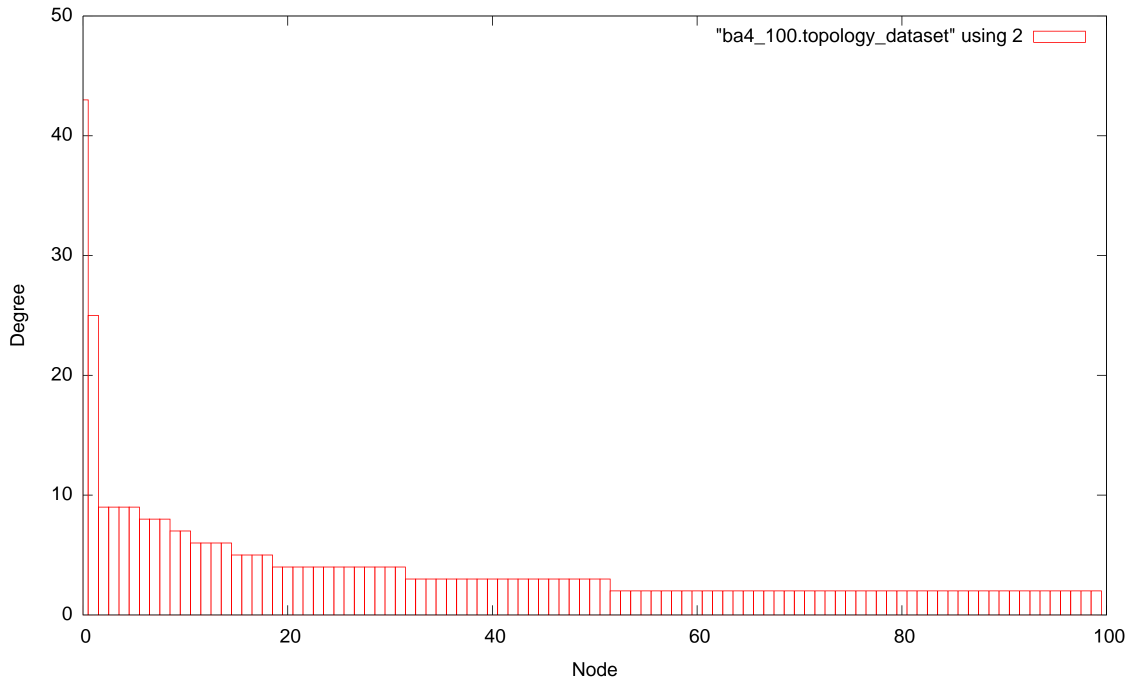

Figure 6.2: Edges per node in topology set A.

6.8 Experiments

The following sections describe the two experiments we use to evaluate the algorithms.

6.8.1 Experiment 1: Query mechanism

The purpose of this experiment is to determine the bandwidth consumption, response times and search accuracy of the search algorithm.

All nodes perform searches for a given time interval. Every execution of this experiment uses the topology from topology set A, which is also shown in Figure 6.1. See Appendix A for more information on how the topology was generated.

6.8.2 Experiment 2: Distribution mechanism

The purpose of this experiment is to evaluate the advertisement distribution mechanism and confirm that this is consistent with the results we find when testing the query mechanism. This experiment does not apply to Gnutella, which does not use advertisements.

All nodes connect and form an overlay, but terminate before they start to send queries. Before they exit they perform an internal test to analyse the potential success rate based on the advertisements they received in their stabilisation phase, i.e. during the three minutes they waited after connecting.

To test the potential success rate, the internal test searches for 1000 random service descriptions in its advertisement cache. The random service descriptions are chosen from a list of service names that are known to be available. The test is only performed internally and no confirmation messages are sent.

6.9 Evaluation plan

The following sections describe how the two experiments from the previous section are used to evaluate the three algorithms. A summary is presented at the end.

6.10 Gnutella

To establish a baseline, the first experiment is performed on the Gnutella implementation using the flooding algorithm described in the original 0.4 specification [23].

Since Gnutella does not perform advertisement distribution, only the query mechanism is tested. The query mechanism experiment is run for 10 repetitions of 30 minutes on topology set A.

6.11 ASAP

The full query mechanism is evaluated first with experiment 1. The experiment is repeated 10 times, each for 30 minutes. We use the topology from topology set A for each TTL.

Since the ASAP algorithm requests more advertisements from its neighbours each time a query fails, the search accuracy is expected to improve over time. The experiment is therefore repeated one time for 10 hours with TTL 6 to see if the accuracy improves.

Finally, we verify our results by running 250 repetitions of the distribution mechanism experiment on the 250 topologies in topology set B for TTL 6.

6.12 Search+

We evaluate the query mechanism in Search+ by running 10 repetitions of experiment 1 for 30 minutes for each TTL on topology set A.

Finally, we verify the results by running 250 repetitions of the distribution mechanism experiment on the 250 topologies in topology set B for TTL 3.

6.13 Summary

Table 6.1 shows the experiments that are performed in the evaluation. The first two experiments use the single topology from topology set A, while the distribution mechanism is tested on the 250 different topologies in topology set B.

Algorithm	TTL	30 min queries	10 hr queries	Distr. mech.
Gnutella	4	10x		
Gnutella	5	10x		
ASAP	5	10x		
ASAP	6	10x	1x	250x
Search+	2	10x		
Search+	3	10x		250x

Table 6.1: Evaluation plan.

6.14 Test environment

The computer running the tests is a Macbook with an Intel Dual Core 2 2.6 GHz processor with 4GB RAM, running OS X 10.5.6.

The implementation is launched as separate threads within the same Java Virtual Machine (JVM). All tests are run on the same jar-file, but with different parameters depending on the actual test being executed.

Each thread will open one listening TCP port on interface 127.0.0.1 (localhost), starting from port 7000. A Python script will orchestrate the actual launch, open a log file for output and make sure that tcpdump [11] is running in the background to capture network traffic.

In addition, key statistics are recorded in log-files by the nodes themselves. These values are: query response time, success rate and queries sent.

After the tests have completed python [10], awk, grep [3] and gnuplot [5] are used to process the results. The tools used in this thesis are further described in Appendix C.

6.15 Measurements

6.15.0.1 Bandwidth

is measured with tcpdstat [28], based on the packet capture recorded with tcpdump. To measure the bandwidth consumed per phase, tcpslice [12] is used to split the capture file in separate files for different time intervals. The bandwidth graphs are generated by tcptrace [13].

6.15.0.2 Response time

is measured by the time it takes from a query is sent to a reply is received. Queries that for some reason do not receive a reply will not be measured. This happens when the query fails.

6.15.0.3 Success rate

is calculated by number of successful queries divided by total number of queries sent.

6.16 Gnutella

The Gnutella flooding algorithm is evaluated by testing the query mechanism 10 times, as described in the evaluation plan in Section 6.9. The experiments are performed with a TTL of 4 and 5.

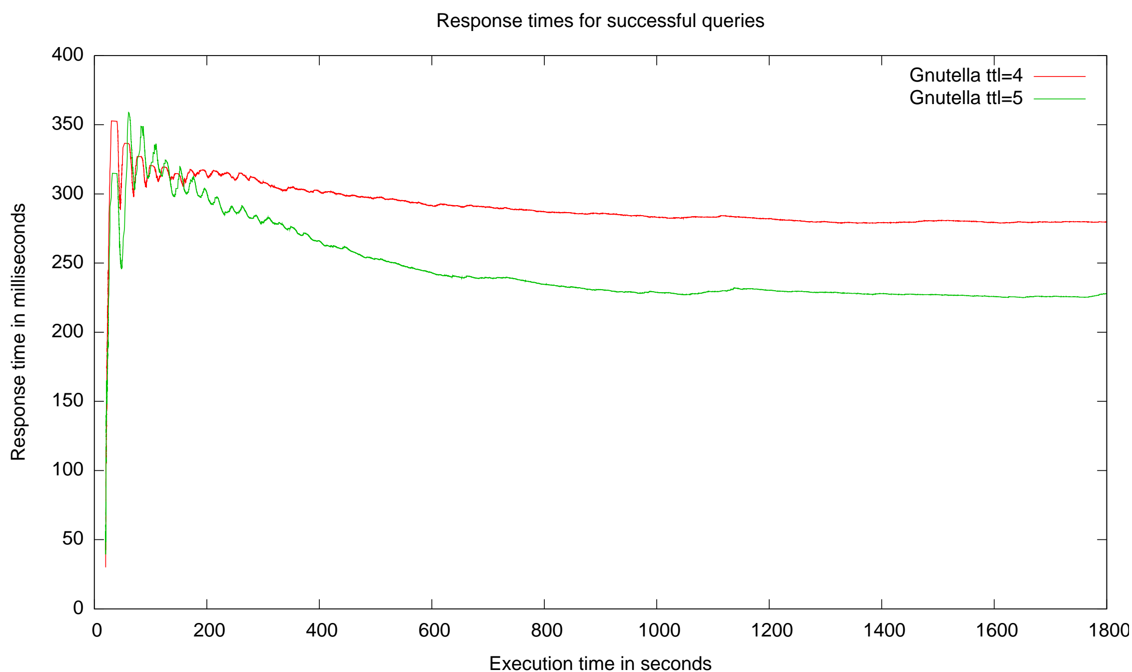

Figure 6.3: Moving average of response times with Gnutella 0.4 and a flooding algorithm.

6.16.1 Response times

In Figure 6.3, the response times for the flooding algorithm are displayed. As we can see, the response times fluctuate in the beginning, as is expected when all nodes send their queries simultaneously. This leads to queues building up in the nodes while they process all the incoming traffic.

After about 400 seconds, TTL 4 seems to stabilise and the algorithm has consistent response times after that. TTL 5 consumes more resources, and stabilises a bit later, after about 600 seconds.

It was expected before this experiment that TTL 5 would have longer response times, due to more traffic in the overlay. As we can see in Figure 6.3, the result was the opposite. We examine this phenomenon further in the following paragraphs.

We assume that the longer response times with TTL 4 are caused by the nodes having to rely on their messages being routed through the highly connected nodes in the topology before they get a response. In Figure 6.2 we saw that two nodes have a significantly higher number of connections than the others. These two central nodes must route a vast number of messages and we have observed that the computer running the experiment does not have the processing power to forward them immediately. This leads to a build up in queues, leading to lower response times.

However, when using TTL 5 the generated traffic should lead to longer queues and delay the traffic even further. It turns out that with a higher TTL, the messages are forwarded through more nodes, resulting in a higher probability that the messages generate a successful hit without having to pass through the delayed central nodes. This can be seen more clearly in Figures 6.4 and 6.5.

In Figure 6.4, we see the data from Figure 6.3 represented as a scatter plot. The results for TTL 4 consistently vary below 8000. When compared to the results in Figure 6.5 for TTL 5, we see that flooding with TTL 5 frequently returns faster responses, but when the response is not received immediately, it tends to take much longer time. This indicates that when the query does not immediately reach the node with the matching service, the reply is delayed along its path before it is forwarded, most likely because of queues in central nodes. It also shows that the delays are longer for TTL 5 than TTL 4, probably due to the increased amount of traffic in the overlay when using a higher TTL.

Rep.	TTL 4	Avg. per node	TTL 5	Avg. per node
1	939477.27	9394.77	1249048.34	12490.48
2	939017.38	9390.17	1248105.55	12481.06
3	939713.01	9397.13	1235840.33	12358.40
4	938363.55	9383.64	1235251.39	12352.51
5	940734.86	9407.35	1229431.82	12294.32
6	941942.77	9419.43	1222237.16	12222.37
7	936721.62	9367.22	1228968.01	12289.68
8	933857.15	9338.57	1222206.18	12222.06
9	936041.28	9360.41	1241343.99	12413.44
10	945276.36	9452.76	1208459.43	12084.59
Avg.	939114.53	9391.15	1232089.22	12320.89

Table 6.2: Bandwidths used by each repetition of Gnutella with flooding, shown in kilobytes.

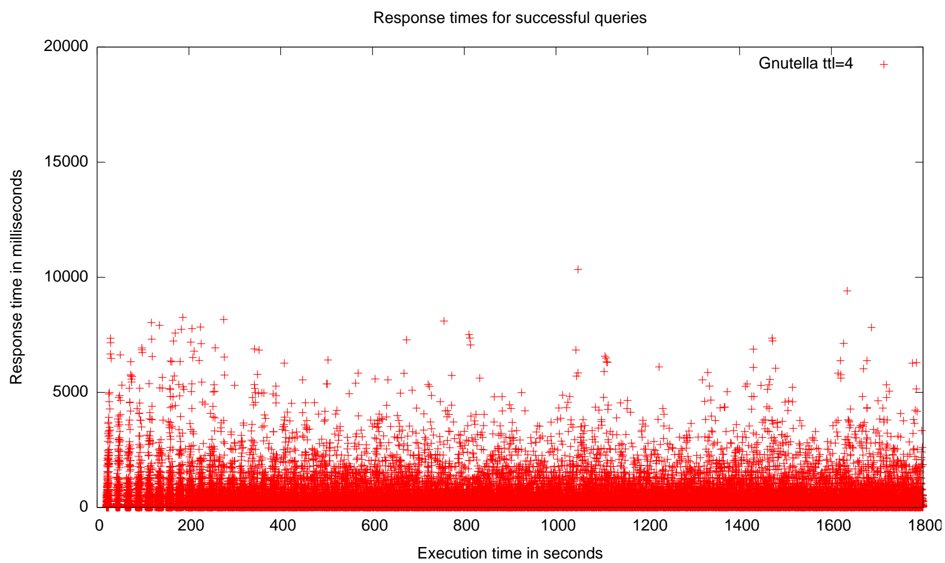

Figure 6.4: Weighted average of response times with Gnutella 0.4, flooding and TTL 4. Response times vary consistently below 8000 milliseconds.

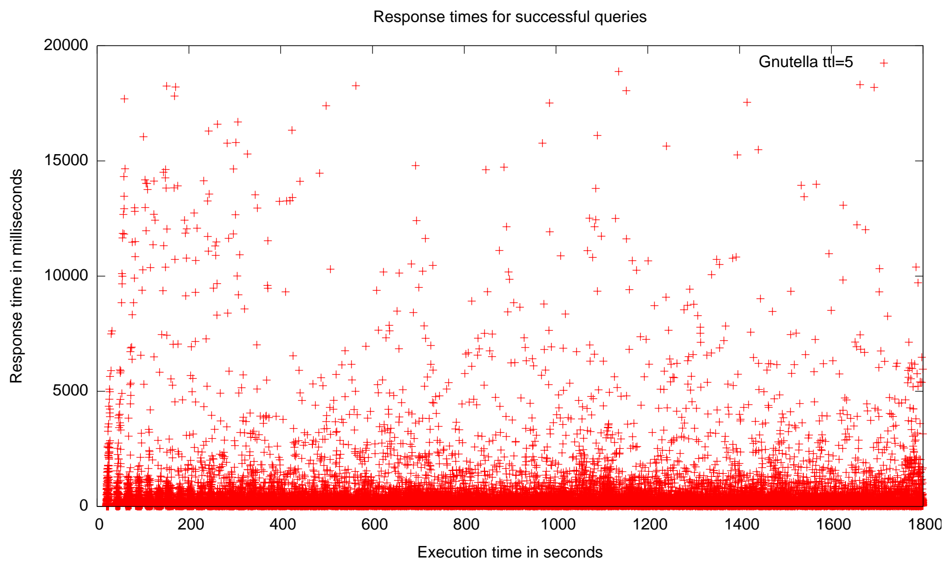

Figure 6.5: Weighted average of response times with Gnutella 0.4, flooding and TTL 5. Most results are returned quickly, but when they are delayed, the delay tends to be higher than with TTL 4.

6.16.2 Bandwidth use

Figure 6.6 shows a bandwidth graph as generated by tcptrace of the 5th repetition of the experiment. The other repetitions show a similar pattern. The graph shows the total bandwidth consumption, along with one graph per node in the experiment. As we can see, the two central nodes consume significantly more bandwidth than the others. The bandwidth consumption is fairly stable during the experiment.

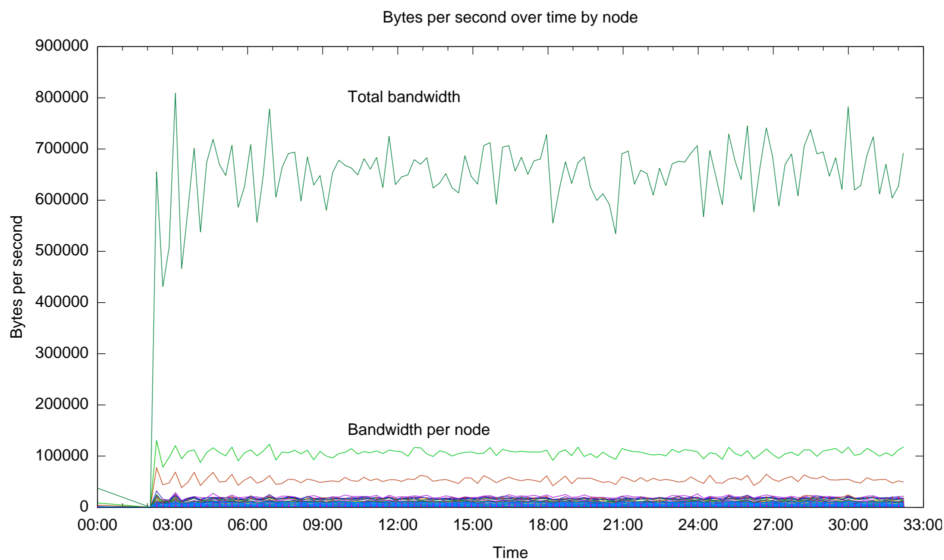

Figure 6.6: The bandwidth graph as shown by tcptrace after the 5th repetition. The largest values are the total, while the other graphs represent bandwidth per port.

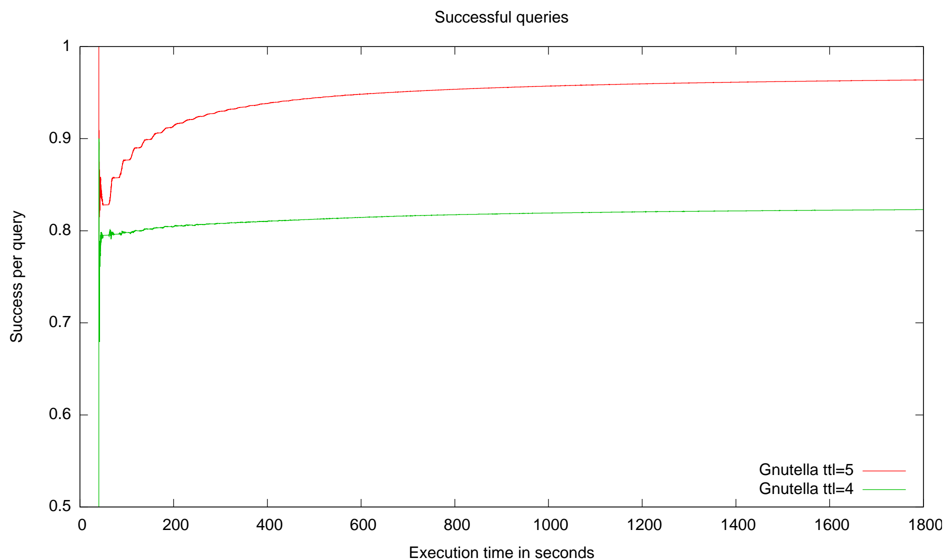

Figure 6.7: Number of successful queries per query sent.

The total bandwidth consumed in each repetition of the experiment is shown in Table 6.2, along with average bandwidth use per node. The average bandwidth for TTL 4 and 5 is 11726.99 and 12320.89

kilobytes per node, respectively. Divided by the execution time, which is 1800 seconds, this gives an average bandwidth consumption per node of 5.22 kilobytes per second for TTL 4 and 6.84 for TTL 5. It should be noted that bandwidth consumption will vary between nodes, as it depends on the number of neighbours they are connected to.

6.16.3 Success rate

The success rate over time is shown in Figure 6.7. As we can see, the success rate increases over time with TTL 5, while it is fairly stable with TTL 4. This phenomenon is caused by queues building up in the central nodes when we use a high TTL, as we also saw when we measured response time earlier in this chapter. Under high load the queries sent from each node will either not arrive in time, or the response will be delayed. After 30 minutes, TTL 5 stabilises at around 0.96, while TTL 4 has a slightly lower success rate at around 0.83.

6.17 Gnutella with ASAP

ASAP is first evaluated with a 10 x 30 minute experiment. After that, the distribution mechanism is tested, before finally a 10 hour test of the query mechanism is performed to see if the algorithm improves its results over time. The first experiment is performed with TTL 5 and 6, while the two latter experiments are only performed with TTL 6.

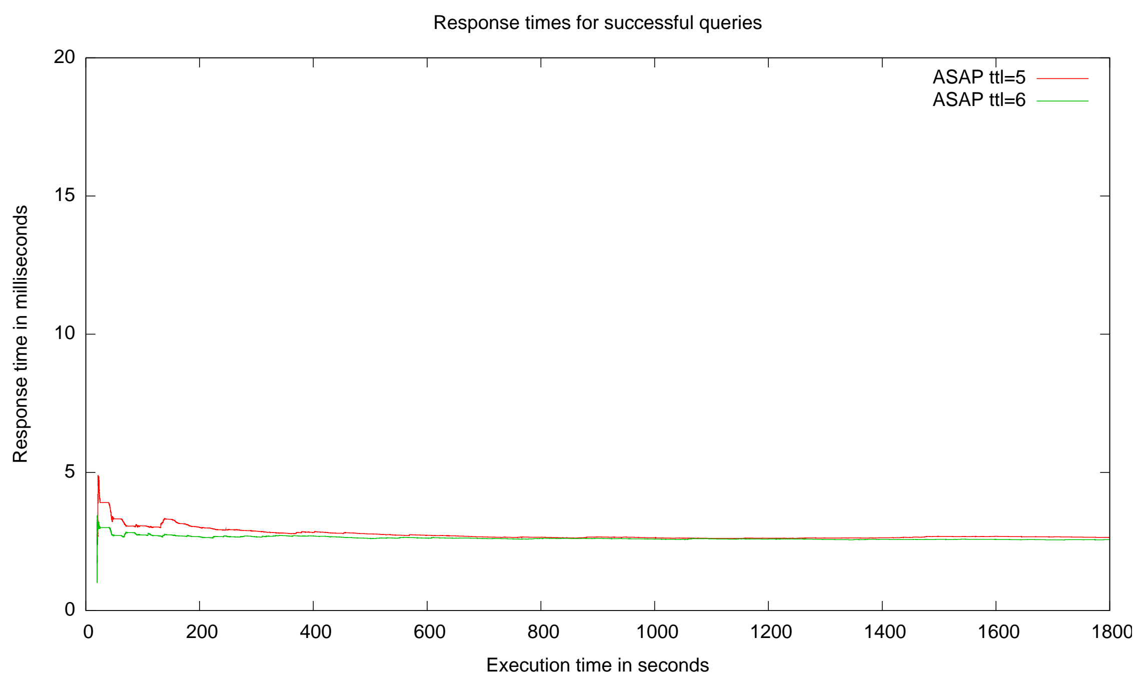

Figure 6.8: Moving average of ASAP response times during the 10 x 30 minute experiment.

6.17.1 Response time

In Figure 6.8 the response times for ASAP are shown. Like Gnutella, ASAP does also have some fluctuations in the beginning, caused by the sudden burst of queries. ASAP's response times still stay well below Gnutella's. TTL 5 seems to have a slightly higher response time during the first few minutes. This is caused by the search algorithm requesting more results from its neighbours. This effect evens out over time, since the results from the new requests will be the same as the previous ones and not lead to any additional results.

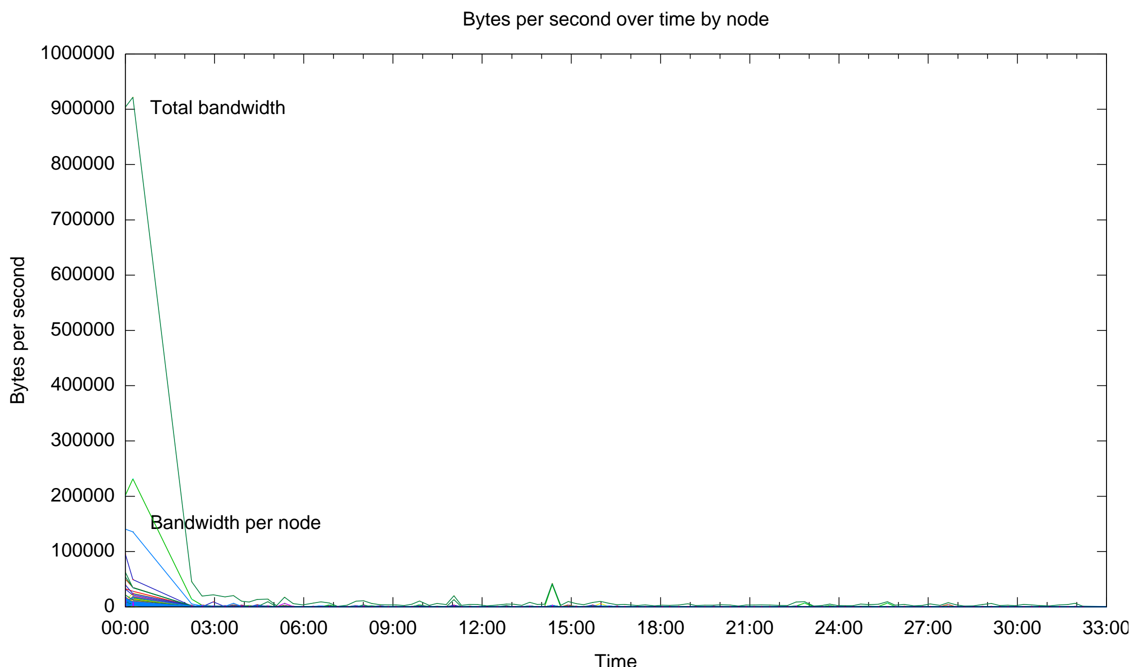

Figure 6.9: The bandwidth graph for the 5th repetition, as generated by tcptrace. The top graph is the total bandwidth consumption, while the rest are bandwidth used per node.

6.17.2 Bandwidth

Figure 6.9 shows a bandwidth graph from the 5th repetition of query mechanism evaluation. The other repetitions show a similar pattern. As we can see, ASAP has a bandwidth peak the first three minutes when the advertisements are distributed. Afterwards, the bandwidth consumption is much lower as it is limited to confirmation messages.

The first three minutes of the packet capture that contains the exchange of advertisements is extracted and the sizes of the packet captures are shown in Table 6.3. The remaining part that contains the actual searches are shown in Table 6.4.

As we can see in Table 6.3, most of the bandwidth is used to distribute the initial advertisements — 324.06 and 373.75 kilobytes per node on average for TTL 5 and 6. That is 1.80 kilobytes per second

Rep.	TTL 5	Avg. per node	TTL 6	Avg. per node
1	34103.39	341.03	38069.29	380.69
2	30601.33	306.01	40237.77	402.38
3	34352.30	343.52	36992.24	369.92
4	34632.19	346.32	37205.64	372.06
5	30483.62	304.84	38134.27	381.34
6	32198.25	321.98	38698.26	386.98
7	33928.73	339.29	37428.21	374.28
8	31747.65	317.48	35855.54	358.56
9	29741.61	297.42	34771.25	347.71
10	32272.60	322.73	36358.01	363.58
Avg.	32406.17	324.06	37375.05	373.75

Table 6.3: Total bandwidth used by ASAP during the first three minutes to distribute initial advertisements, shown in kilobytes.

Rep.	TTL 5	Avg. per node	TTL 6	Avg. per node
1	15942.66	159.43	14085.11	140.85
2	16578.84	165.79	13990.88	139.91
3	15274.80	152.75	13773.01	137.73
4	14910.88	149.11	14336.39	143.36
5	16140.34	161.40	15390.94	153.91
6	16761.02	167.61	16148.79	161.49
7	16597.57	165.98	15388.49	153.88
8	16702.22	167.02	16934.99	169.35
9	16468.74	164.69	15809.59	158.10
10	16795.76	167.96	15383.42	153.83
Avg.	16217.28	162.17	15124.16	151.24

Table 6.4: Total bandwidth used by ASAP while performing searches for 30 minutes, shown in kilobytes.

for TTL 5 and 2.07 kilobytes per second for TTL 6.

In Table 6.4 we see that the average bandwidth consumed during search is 162.17 kilobytes for TTL 5 and 151.24 kilobytes for TTL 6. The experiment ran for 1800 seconds, which corresponds to 92.25 and 86.04 *bytes* per second, respectively. The algorithm consumes less bandwidth during search with higher TTLs, since the nodes have received more advertisements initially and thus sends fewer request message for more advertisements from neighbours afterwards.

6.17.3 Success rate

In Figure 6.10 the success rates of ASAP with TTL 5 and 6 are shown. It can be seen that when using a lower TTL the algorithm uses more time to reach a stable success rate. This is probably caused by the nodes having to request more advertisements from their neighbours in the beginning. After a while the neighbours do not have any new advertisements to offer, and the success rate stabilises. There is a small increase in success rate throughout the whole experiment. We have therefore chosen to repeat it for 10 hours to examine the effect over a longer time period.

6.17.4 Success rate after 10 hours

To evaluate whether the success rate of ASAP continues to increase after our 30 minute evaluation, we performed a single 10 hour experiment with TTL 6 — the result is shown in Figure 6.11. As we can see, the success rate does continue to rise until around 5000 seconds, but after that the algorithm stabilises at around 0.9.

6.17.5 Distribution mechanism

The distribution mechanism is tested by letting 100 nodes stabilise 250 times in 250 different topologies. An internal test is performed on each node to see how much of the shared content it can locate based on the advertisements it has received. ASAP increases its success rate over time, so the actual success rate may be higher than the result of this test. It does however give an indication of whether the topology chosen for evaluating the query mechanism is representative.

The average internal success rate received with ASAP after 250 repetitions is 0.869. The estimated 95% confidence interval is 0.862 to 0.876.

6.18 Gnutella with Search+

Search+ is first evaluated with a 10 x 30 minute experiment. After that, the distribution mechanism is tested.

The experiments are performed with TTL 2 and 3, as explained in the evaluation plan.

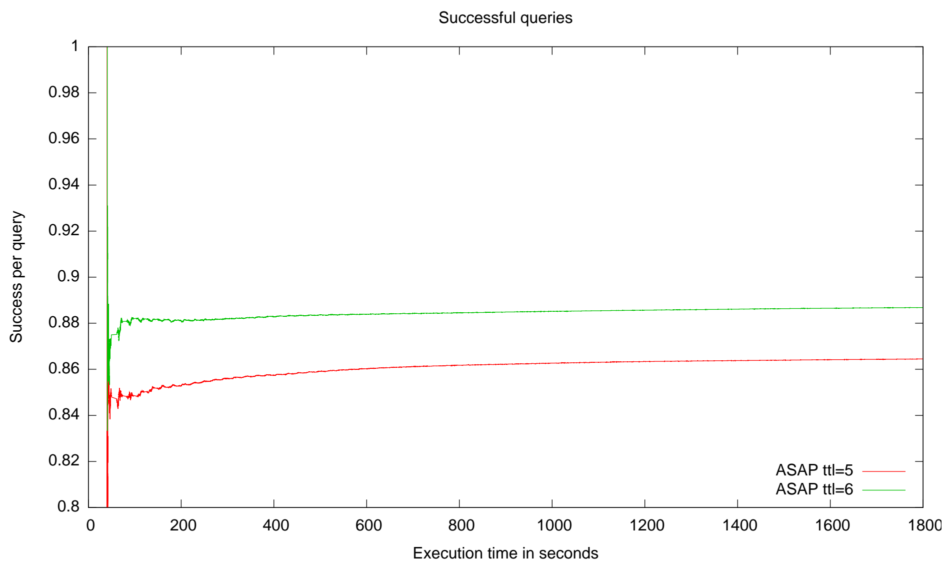

Figure 6.10: Average ratio of successful queries with ASAP during the 30 minutes experiment.

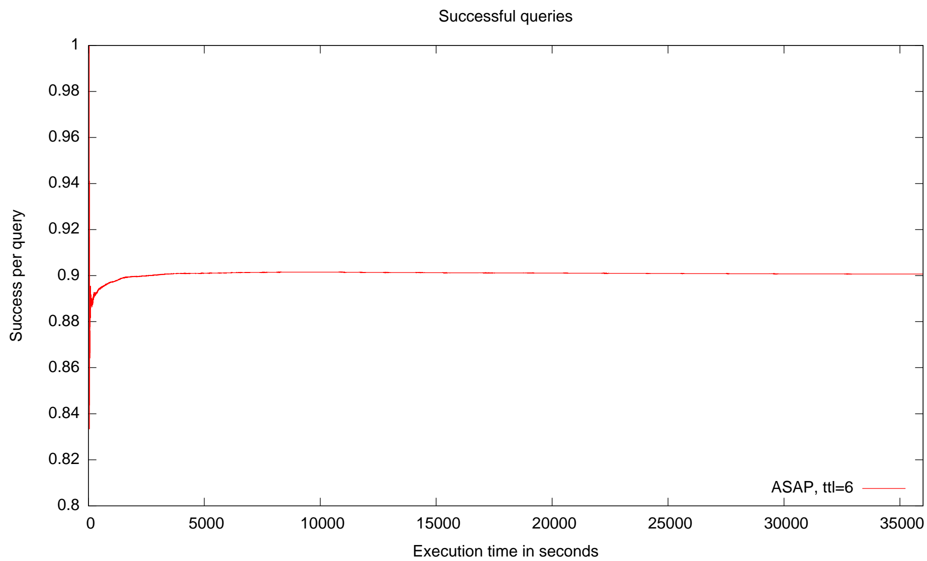

Figure 6.11: Average ratio of successful queries with ASAP over 10 hours.

6.18.1 Response time

The average response times with Search+ is shown in Figure 6.12. The response time is initially fluctuating due to the high load. TTL 3 has a little higher response times, but this is assumed to be caused by the additional overhead caused by an increased TTL. The increase in response time during the experiment does not seem to be related to the algorithm, and may be implementation specific.

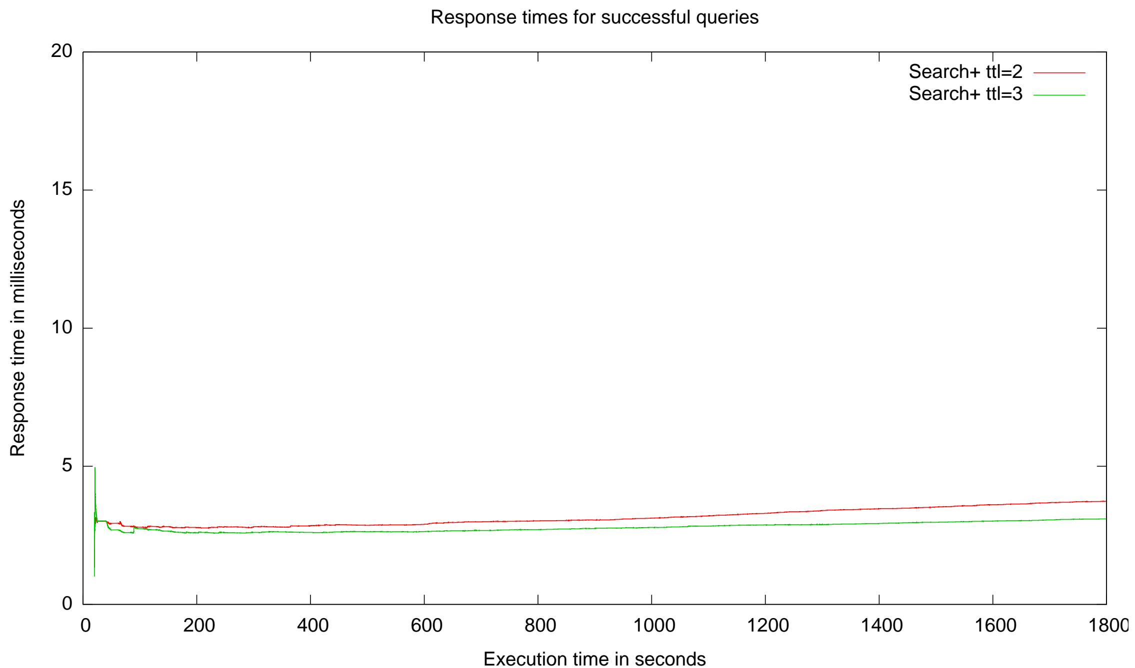

Figure 6.12: Moving average of Search+ response times during the 10 x 30 minute experiment.

6.18.2 Bandwidth

As with ASAP, Search+ consumes most of its bandwidth during the initial advertisement distribution. In Table 6.5, the bandwidth used on distribution of advertisements during the stabilisation phase is shown. The bandwidth spent on queries is shown in Table 6.6.

During the advertisement distribution the average bandwidth used per node is 237.09 kilobytes for TTL 2 and 387.63 for TTL 3. This corresponds to 1.32 and 2.15 kilobytes per second.

The search queries uses on average 63.86 and 67.01 kilobytes per node for TTL 2 and 3, respectively. Divided by 1800 seconds, the average bandwidth consumption is 36.33 and 38.12 *bytes* per second, significantly lower than ASAP's 92.25 and 86.04 bytes per second. The reason for this is that ASAP regularly requests more advertisements from its neighbours when queries fail, which consumes additional bandwidth.

Rep.	TTL 2	Avg. per node	TTL 3	Avg. per node
1	24399.91	244.00	40679.22	406.79
2	23482.15	234.82	37042.41	370.42
3	22751.82	227.52	39651.99	396.52
4	22486.06	224.86	37426.48	374.26
5	23093.86	230.94	37501.54	375.02
6	23781.43	237.81	38737.24	387.37
7	24959.64	249.60	39670.38	396.70
8	24129.27	241.29	38171.91	381.72
9	23594.16	235.94	39881.63	398.82
10	24408.35	244.08	38869.22	388.69
Avg.	23708.67	237.09	38763.20	387.63

Table 6.5: Total bandwidth used by Search+ to distribute initial advertisements, shown in kilobytes.

Rep.	TTL 2	Avg. per node	TTL 3	Avg. per node
1	6290.89	62.91	6725.11	67.25
2	6641.72	66.42	6683.48	66.83
3	6310.09	63.10	6732.23	67.32
4	6256.68	62.57	6749.24	67.49
5	6396.69	63.97	6720.34	67.20
6	6335.27	63.35	6668.98	66.69
7	6624.05	66.24	6713.21	67.13
8	6278.04	62.78	6721.88	67.22
9	6306.50	63.06	6709.10	67.09
10	6417.55	64.18	6588.34	65.88
Avg.	6385.75	63.86	6701.19	67.01

Table 6.6: Total bandwidth used by Search+ while performing searches for 30 minutes, shown in kilobytes.

6.18.3 Success rate

The average query success rate of Search+ is shown in Figure 6.13. After some fluctuations during the high load period in the beginning, the algorithm quickly stabilises at levels slightly below 0.93 for TTL 2 and slightly below 0.98 for TTL 3.

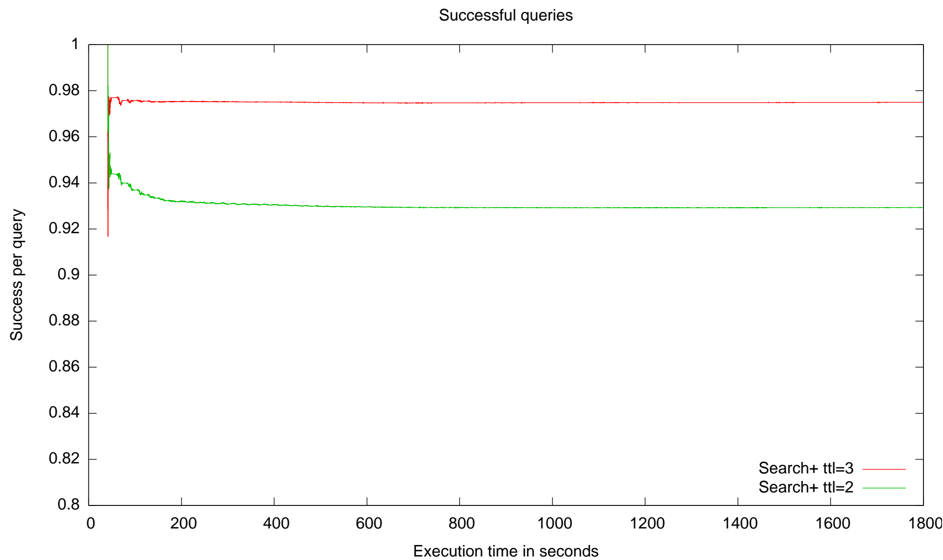

Figure 6.13: Average ratio of successful queries with Search+ during the 30 minute experiment.

6.18.4 Distribution mechanism

After 250 repetitions the average internal success rate of Search+ with TTL 3 is 0.988. With 95% confidence, the confidence interval is estimated to be 0.980 - 0.995.

Surprisingly, our results with Search+ is a bit below the lower limit of the confidence interval, with a success rate of 0.974. This can be caused by a random variation, but it is also possible that some of the confirmation messages were lost during the experiments. When so many nodes send UDP confirmation messages at the same time, there is an increased chance of collisions, even when two copies are sent. This could mean that Search+'s success rate would be higher if the confirmation mechanism is improved.

6.19 Discussion

Our results with ASAP differ from the results found in the simulation conducted by the authors of [34], where ASAP seems to consistently reach values around 0.95. This can in part be explained by two differences in our setup:

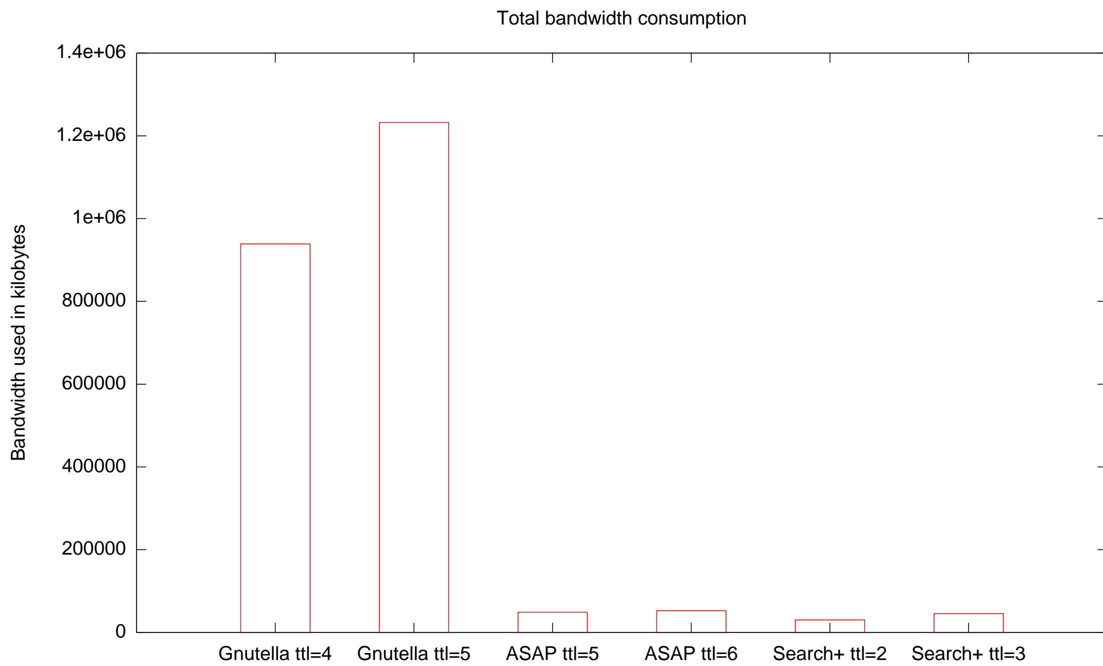

Figure 6.14: The average amount of *total* bandwidth used by each algorithm.

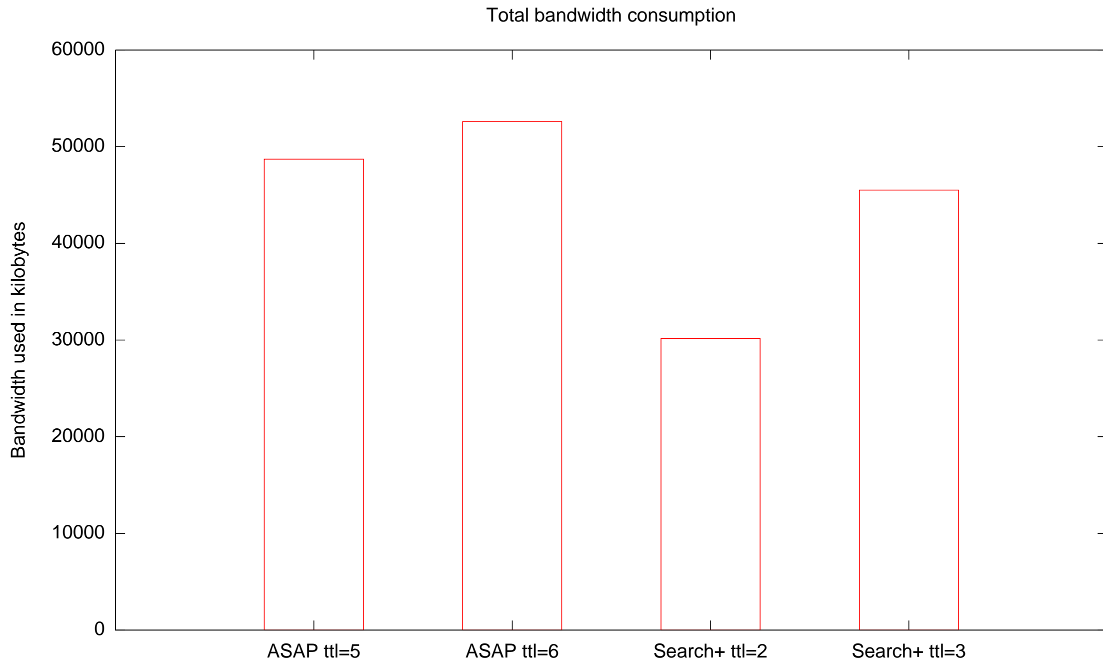

Figure 6.15: The average amount of *total* bandwidth used by each algorithm. Gnutella is removed to give a clearer view of ASAP and Search+.

First, in [34] they distribute the advertisements with 90% of the nodes already connected. This means that the initial advertisement sent on join with TTL 5 (see Algorithm 1) can reach a very high percentage of the nodes, as observed in [53]. This could mean that the success rate of ASAP is largely determined by the initial distribution of advertisements and that the effect of the request message that is supposed to drive advertisements toward interested nodes is negligible. In our experiments we use a TTL of 1 for the request message, as recommended in [34], but increasing this value may yield better results. We did not do this in our experiments, mainly because of bandwidth requirements — as we can see in Table 6.7, ASAP already requires the same bandwidth as Search+. In Figure 6.10 we see that ASAP's success rate rises through the whole experiment, which could indicate that ASAP would perform better in a network that has been running a lot longer than 30 minutes. However, in Figure 6.11, we see that this effect decreases over time. The authors of [34] mention a «warm up period», but they do not say how long this period is. To further increase the success rate, ASAP would have to improve its advertisement distribution — either by distributing the advertisements more effectively when joining the network, or by requesting advertisements from more than the immediate neighbours. Either way, ASAP relies on nodes to keep sending queries, as this is what triggers the distribution process. ASAP may thus be more suitable for larger, stable networks, where there are less stringent bandwidth requirements.

Second, when simulated in [34], ASAP yields the best results in the crawled eDonkey [1] topology. This topology has 1.28 copies of each document (or service in this context) on average. This would increase the perceived search accuracy, as multiple copies of a document increases the probability of finding one item.

In Figure 6.14, the total consumed bandwidth of the search algorithms are compared. As expected, Gnutella with a flooding algorithm requires quite a lot of bandwidth, close to 1.2GB. This gives a good query success ratio of 0.96 as seen in Table 6.7, but the amount of traffic would quickly saturate slow links. As we can see, both ASAP and Search+ require considerably less bandwidth.

In Figure 6.15, Gnutella is removed to give a clearer view of the other two algorithms. We can see here that even if we increase the TTL in ASAP, the consumed bandwidth does not increase significantly. If we compare this to the results in Table 6.7, we can also see that the success rate increases only slightly with a higher TTL. We assume that this is because the lower success rate of ASAP is not caused by a low TTL, but rather that the algorithm relies on a thorough advertisement distribution in the overlay during the join process, as mentioned earlier in this chapter. The problem seems to be that at the time the advertisements are distributed in our experiment, all the nodes are not connected to the network yet. In an ad hoc network this may often be the case, thus ASAP may be more suited for a more stable, Internet based overlay.

Search+ has a more aggressive advertisement distribution scheme, which will send advertisements aggregated in the overlay to new nodes when they start subscribing to topics from their neighbours. This process is targeted, resulting in lower bandwidth consumption than with ASAP initial flood mechanism.

Algorithm	TTL	Success	Response time	Consumed bandwidth
Search+	2	0.929	3.74	30151.36
Search+	3	0.974	3.10	45521.62
ASAP	5	0.864	2.65	48710.58
ASAP	6	0.886	2.57	52598.05
Gnutella	5	0.964	230.05	1232089.22
Gnutella	4	0.823	279.51	939114.53

Table 6.7: Summary of the evaluation results from the 30 minute experiment. Consumed bandwidth is the total bandwidth consumed by each algorithm, including advertisement distribution in ASAP and Search+. Response time is measured in milliseconds, bandwidth in kilobytes.

Algorithm	TTL	Distr. bandw.	Search bandw.	Success
ASAP	6	373.75	151.24	0.886
Search+	3	387.63	67.01	0.974

Table 6.8: Comparison between ASAP and Search+. Distribution and Search columns are shown in kilobytes per node. Success is ratio of successful queries per query sent.

In Table 6.8, ASAP and Search+ are compared using the TTL values that gave the best search accuracy. We can see that although Search+ uses slightly more bandwidth during advertisement distribution, the bandwidth used for queries with Search+ is half of the bandwidth used by ASAP. It should also be noted that the bandwidth spent on advertisement distribution directly affects the search accuracy — we therefore expect ASAP to use more bandwidth if the distribution mechanism is improved to increase the success ratio.

6.20 Suitability in tactical networks

As Search+ with TTL 3 gives the highest accuracy and the lowest bandwidth consumption, it is the most suitable candidate for search in a tactical network. In this section we calculate expected bandwidth consumption in kilobits per second and compare our results to the bandwidths available with military radios.

The average bandwidths consumed by Search+ is 387.63 kilobytes per node during advertisement distribution and 67.01 during search. This corresponds to 17.22 and 0.30 kilobits per second, respectively. This shows that it is theoretically possible to use Search+ for service discovery in a network with low data-rate.

6.21 Summary

In this chapter we have shown that an unstructured Gnutella network based on the 0.4 protocol uses too much bandwidth to be practical in a tactical network. By exchanging the search algorithm with an advertisement based approach, we could with ASAP keep the bandwidth consumption considerably lower, but at the expense of search accuracy. Finally, we showed that with our new algorithm, Search+, the bandwidth consumption can be further reduced, while still maintaining a close to 100% accuracy. We also show that Search+ may be theoretically suitable for use over military radios.

7 Conclusion

We have looked at existing P2P solutions to see if any of them could be used for service discovery in a military tactical network. By theoretical evaluation of existing P2P protocols we conclude that purely unstructured overlays have several properties that are suitable in a military context. They are robust, resilient and self-adapting. However, their bandwidth requirements during search make them unsuitable for use today.

To examine the bandwidth requirements of an unstructured overlay, we implemented Gnutella 0.4 which utilises a flooding based algorithm for search. In addition, we implemented the newly proposed algorithm ASAP that earlier had shown promise in simulations. In experiments, we saw that ASAP did not perform as well as we had hoped.

To address the limitations in ASAP, we developed and implemented a new search algorithm we call Search+, that is specifically designed with tactical network environments in mind.

Through experiments we evaluated Gnutella, ASAP and Search+ and showed that Gnutella is too bandwidth consuming to be used in a tactical environment. Both ASAP and Search+ reduce the traffic use significantly. Unfortunately, ASAP relies on a distribution mechanism that does not perform as well as expected, especially in terms of search accuracy which does not exceed 90% in our experiments. Search+ on the other hand, has the lowest bandwidth requirements and achieves the highest search accuracy, above 97%.

When comparing the bandwidth requirements of Search+ to the bandwidths available in a deployed tactical network, Search+ is expected to perform well.

8 Future work

8.1 Reducing bandwidth

Search+ is quite bandwidth efficient, but there is still room for improvements — some of our ideas are described in the following section.

8.1.1 Search+ with shortcuts

While developing Search+, it was noted that the discovery mechanism in Gnutella used for discovering new potential neighbours could be turned off completely when using Search+. Instead, a list of potential neighbours could be compiled from the senders of the advertisements received through the subscriptions. As an added bonus, the nodes would know which topics the other nodes were sharing, thus enabling them to choose new neighbours with similar interests as themselves. This is very similar to what has been described in other algorithms as *shortcuts*, for instance in [63].

By connecting directly to neighbours known to contain relevant services, the search accuracy can further be increased — and an increase in accuracy may allow us to send the subscription request with a lower TTL, thus ultimately requiring less bandwidth.

8.1.2 Bloom filters

For simplicity, the current implementation uses fixed size Bloom filters in the advertisements as described in Appendix B. Since the time Bloom filters were originally proposed, many variants have been presented, some of which may be more suitable for Search+. This should be examined further.

8.1.3 Improve implementation

The implementation currently bases its network communication on serialised Java objects. By extending the implementation to send data in a more compressed format, the bandwidth consumption could be reduced further. Implementations for military protocols should also be considered.

8.2 Dynamicity

The evaluation done in this thesis is mainly in a static environment. In the future, it would be interesting to examine how well Search+ reacts to changes in both the topology and the shared services.

8.2.1 Liveness

One important aspect in tactical networks is the liveness of information [32]. The search result received by a node should be as up-to-date as possible. The confirmation message sent by Search+ does provide a high degree of liveness, but currently we have not experimented with how well it would perform when many services appear and disappear in the network. We should also examine how well it adjusts to services moving between nodes.

8.2.2 Mobility

The effect of churn, i.e. nodes joining and leaving the overlay, should also be evaluated further in terms of bandwidth consumption. When a new node joins the network a large number of advertisements may have to be exchanged between the node and its neighbours and may have an adverse effect on the overall bandwidth consumption.

In [17], Fongen et al. have developed a hierarchical mobility model for mobile military networks. In future work, this model should be used to examine how Search+ behaves with military mobility patterns.

8.2.3 Actual radios

Ultimately, the algorithm should be tested on real radios in a mobile environment. Radios have several additional requirements that we have not addressed in this thesis. For instance, their bandwidth may vary depending on distance and atmospheric disturbances.

References

[1] eDonkey Network. http://en.wikipedia.org/wiki/EDonkey_network, visited May 4th, 2009.

[2] GNU Emacs. http://www.gnu.org/software/emacs/, visited May 2nd, 2009.

[3] GNU grep. http://www.gnu.org/software/grep/, visited 3rd May, 2009.

[4] GNU Make. http://www.gnu.org/software/make/, visited May 3rd, 2009.

[5] gnuplot homepage. http://www.gnuplot.info/, visited 3rd May, 2009.

[6] Java. http://www.java.com, visited May 2nd, 2009.

[7] Netbeans homepage. http://www.netbeans.org/, visited May 3rd, 2009.

[8] P2PS - Peer-to-Peer Simplified. http://www.trianacode.org/p2ps/, visited March 20th, 2009.

[9] pdftex. http://www.tug.org/applications/pdftex/, visited May 2nd, 2009.

[10] Python Programming Language – official website. http://www.python.org, visited May 3rd, 2009.

[11] tcpdump. http://www.tcpdump.org/, visited May 2nd, 2009.

[12] tcpslice. ftp://ftp.ee.lbl.gov/tcpslice.tar.gz, visited April 19th, 2009.

[13] tcptrace. http://www.tcptrace.org/, visited May 2nd, 2009.

[14] XFig Drawing Program for the X Window System. http://www.xfig.org/, visited May 3rd, 2009.

[15] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernardo A. Huberman. Search in power-law networks. *APS Physical Review E (PRE) 64, 046135, 2001*.

[16] M. Amoretti, F. Zanichelli, and G. Conte. SP2A: a service-oriented framework for P2P-based grids. In *In proceedings of the 3rd International Workshop on Middleware for Grid Computing (MGC05), Grenoble, France*. Distributed Systems Group, University of Parma, Italy, 2005.

[17] Anders Fongen et al. A military mobility model for manet research. In *Parallel and Distributed Computing and Networks (PDCN 2009), February 16 – 18*, Innsbruck, Austria, 2009.

[18] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.

[19] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 289:509, 1999.

[20] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[21] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Middleware*, Heidelberg, Germany, November 2001.

[22] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers, editors. *UDDI Version 3.0.2*. OASIS UDDI Spec Technical Committee, October 19, 2004.

[23] Clip2. The gnutella protocol specification v0.4, document revision 1.2. `http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf`, visited September 9th, 2008.

[24] Geoff Coulson, Gordon Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. OpenCOM v2: A component model for building systems software. In *Proceedings of IASTED Software Engineering and Applications (SEA'04), Cambridge, MA, ESA*, November 2004.

[25] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 23–32, 2002.

[26] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web Services Web, An Introduction to SOAP, WSDL, and UDDI. *Internet Computing, IEEE*, 6(2):86–93, April 2002.

[27] Neil Daswani and Hector Garcia-Molina. Query-flood dos attacks in gnutella. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 181–192, New York, NY, USA, 2002. ACM.

[28] David Dittrich. modified tcpdstat. `http://staff.washington.edu/dittrich/talks/core02/tools/tools.html`, downloaded April 29th 2009.

[29] Anders Eggen. Discussion about core services (personal communication), September 2008.

[30] J. Flathagen and K. Øvsthus. Service discovery using OLSR and bloom filters. In *4th OLSR Interop & Workshop*, Ottawa, Canada, October 2008.

[31] Sally Fuger, Farrukh Najmi, and Nikola Stojanovic, editors. *ebXML Registry Information Model v3.0*. OASIS ebXML Registry Technical Committee, May 2, 2005.

[32] Tommy Gagnes. Assessing dynamic service discovery in the network centric battlefield. In *Military Communications Conference*, pages 601–614, Orlando, Florida, USA, Oct 2007.

[33] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *23rd IEEE International Conference on Computer Communications (INFOCOM), March 7-11*, volume 1, page 130, Hong Kong, 2004.

[34] Peng Gu, Jim Wang, and Hailong Cai. ASAP: An advertisement-based search algorithm for unstructured peer-to-peer systems. In *International Conference on Parallel Processing (ICPP), September 10-14*, page 8, Xian, China, 2007.

[35] Andrew Harrison and Ian Taylor. Wspeer project homepage. `http://www.wspeer.org/`, visited May 1st 2009.

[36] Andrew Harrison and Ian Taylor. WSPeer - An Interface to Web Service Hosting and Invocation. In *HIPS Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models, IPDPS*, Denver, Colorado, USA, 2005.

[37] Bradley Huffaker, Evi Nemeth, and K. Claffy. Otter: A general-purpose network visualization tool. `http://www.caida.org/tools/visualization/otter/paper/`, Visited April 13th, 2009.

[38] M.N. Huhns and M.P. Singh. Service-oriented computing: key concepts and principles. *Internet Computing, IEEE*, 9(1):75–81, Jan-Feb 2005.

[39] Ian Clarke et al. Freenet project homepage. `http://freenetproject.org/`, visited May 4th, 2009.

[40] F. T. Johnsen, J. Flathagen, T. Gagnes, R. Haakseth, T. Hafsøe, J. Halvorsen, N. A. Nordbotten, and M. Skjegstad. Web services and service discovery. In *FFI/RAPPORT 2008/01064*. Norwegian Defence Research Establishment, 2008.

[41] Frank T. Johnsen, Trude Hafsøe, and Magnus Skjegstad. Web services and service discovery in military networks. In *14th ICCRTS: C2 and Agility*, Washington DC, USA, June 2009 (to appear).

[42] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[43] Kenjiro Cho et al. tcpdstat. `http://www.sonycsl.co.jp/person/kjc/papers/freenix2000/`, visited April 29th 2009.

[44] Tor Klingberg and Raphael Manfredi. Gnutella 0.6 (draft). `http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html`, visited April 19th, 2009, 2002.

[45] Y. Li, F. Zou, Z. Wu, and F. Ma. *PWSD: A scalable Web Service Discovery architecture based on peer-to-peer overlay network*, pages 291–300. Springer Berlin / Heidelberg, 2004.

[46] J. Liang, N. Naoumov, and K. W. Ross. The Index Poisoning Attack in P2P File Sharing Systems. In *25th IEEE International Conference on Computer Communications (INFOCOM), April 23-29*, pages 1–12, Barcelona, Catalunya, Spain, April 2006.

[47] K. Lund, A. Eggen, D. Hadzic, T. Hafsøe, and F.T. Johnsen. Using web services to realize service oriented architecture in military communication networks. *Communications Magazine, IEEE*, 45(10):47–53, October 2007.

[48] D. Marco-Pompel. Service oriented peer prototype for mobile users. Technical report, NC3A Technical Note Draft under project SPW001495, nov 2007.

[49] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. Brite: An approach to universal topology generation. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems - MASCOTS'01, Cincinatti, Ohio*, August 2001.

[50] Michelle Amoretti et al. Service-oriented peer-to-peer architecture project homepage. `http://sp2a.sourceforge.net/`, visited May 1st 2009.

[51] P. Bartolomasi et al. NATO Network Enabled Capability Feasibility study, v. 2.0, October 2005.

[52] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, New York, NY, USA, 1997. ACM.

[53] M. Portmann, P. Sookavatana, S. Ardon, and A. Seneviratne. The cost of peer discovery and searching in the gnutella peer-to-peer file sharing protocol. In *Proceedings Ninth IEEE International Conference on Networks, 10-12th October*, pages 263–268, Bangkok, Thailand, 2001.

[54] R. Faucher et al. Guidance on proxy servers for the tactical edge. Technical report MTR 060175, MITRE, September 2006.

[55] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2004. USENIX Association.

[56] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 SIGCOMM conference*, volume 31, pages 161–172. ACM New York, NY, USA, 2001.

[57] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, volume 11, pages 329–350. Heidelberg, 2001.

[58] Emil Sit and Robert Morris. *Security Considerations for Peer-to-Peer Distributed Hash Tables*, volume 2429/2002 of *Lecture Notes in Computer Science*, pages 261–269. Springer Berlin / Heidelberg, 2002.

[59] Kaarthik Sivashanmugam, Kunal Verma, and Amit Sheth. Discovery of Web services in a federated registry environment. In *IEEE International Conference on Web Services (ICWS), July 6-9*, pages 270–278, San Diego, California, USA, 2004.

[60] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM New York, NY, USA, 2001.

[61] Bernard Traversat, Mohamed Abdelaziz, and Eric Pouyoul. Project JXTA: A Loosely-Consistent DHT Rendezvous Walker. `http://research.sun.com/spotlight/misc/jxta-dht.pdf`, visited May 2nd, 2009.

[62] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA 2.0 Super-Peer Virtual Network. `http://research.sun.com/spotlight/misc/jxta.pdf`, visited May 3rd, 2009.

[63] Dimitrios Tsoumakos and Nick Roussopoulos. Analysis and comparison of p2p search methods. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 25, New York, NY, USA, 2006. ACM.

[64] G. Tyson, A. Mauthe, T. Plagemann, and Y. El-khatib. Juno: Reconfigurable Middleware for Heterogeneous Content Networking. In *5th International Workshop on Next Generation Networking Middleware (NGNM), September 22-26*, Samos Island, Greece, 2008.

[65] Ian Wang. P2PS (Peer-to-Peer Simplified). In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 54–59. Louisiana State University, February 2005.

[66] World Wide Web Consortium (W3C). Extensible Markup Language (XML). `http://www.w3.org/XML/`, visited April 29th, 2009.

[67] Tianyin Xu, Baoliu Ye, M. Kubo, A. Shinozaki, and Sanglu Lu. A Gnutella inspired ubiquitous service discovery framework for pervasive computing environment. In *Computer and Information Technology (CIT), 2008, 8th IEEE International Conference on 8-11 July*, pages 712–717, Sydney, Australia, 2008.

[68] B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical report, Technical Report UCB//CSD-01-1141, UC Berkeley, 2001.

Appendix A Generating topologies

The topologies used in the experiments in this thesis were generated by BRITE [49]. BRITE is a topology generator developed at Boston University, that can — among others — generate the Barabasi-Alberts modelled topologies used in our experiments.

BRITE includes a graphical user interface (GUI) to select the model and its parameters. The GUI will in turn generate a configuration file and execute a command line utility to generate the actual topology. The command line utility is included in C++ and Java versions. The full software can be downloaded from `http://www.cs.bu.edu/brite/`.

The following configuration file was created by the GUI and subsequently used to generate the topologies used in this thesis.

```
#This config file was generated by the BRITE GUI.

BriteConfig

BeginModel
        Name =  4           #Router Barabasi-Albert=2, AS Barabasi-Albert =4
        N = 100             #Number of nodes in graph
        HS = 1000           #Size of main plane (number of squares)
        LS = 100            #Size of inner planes (number of squares)
        NodePlacement = 1   #Random = 1, HeavyTailed = 2
        m = 2               #Number of neighboring node each new node connects to.
        BWDist = 1          #Constant = 1, Uniform =2, HeavyTailed = 3, Exponential =4
        BWMin = 10.0
        BWMax = 1024.0
EndModel

BeginOutput
        BRITE = 1           #1/0=enable/disable output in BRITE format
        OTTER = 1           #1/0=enable/disable visualization in otter
        DML = 0             #1/0=enable/disable output to SSFNet's DML format
        NS = 0              #1/0=enable/disable output to NS-2
        Javasim = 0         #1/0=enable/disable output to Javasim
EndOutput
```

As we can see in the configuration file, Otter [37] was also chosen as an output format. This was to be able to view the topologies graphically if any unforeseen phenomenons should occur.

To generate the topology, the following command was issued. Note that this is the Java version of BRITE.

```
java -Xmx256M -classpath Java/:. Main.Brite GUI_GEN.conf \
    ba4_100.brite seed_file
```

The seed file is a generated file that contains the initial values for the random seed. BRITE will use this seed for its random generator, generate the topology and then store a new seed in the seed file.

The following bash-script generated the 250 numbered topology files required for experiment 2 (see Section 6.8.2).

```
for i in {1..250}; do
    java -Xmx256M -classpath Java/:. Main.Brite GUI_GEN.conf \
        ba4_100_top\${i} seed_file;
done;
```

Each generated topology file contains a list of nodes and their properties, along with edges between the nodes. A shortened sample output is shown below.

```
Topology: ( 100 Nodes, 197 Edges )
Model (4 - ASBarabasi):  100 1000 100 1  2  1 10.0 1024.0

Nodes: ( 100 )
0       288     363     19      19      0       AS_NODE
1       258     338     21      21      1       AS_NODE
(...)
99      395     363     2       2       99      AS_NODE


Edges: ( 197 )
0       0       1       39.05124837953327       0.13026094332077315     10.0    0       1       E_AS    U
1       0       2       645.6887795215277       2.153785935206975       10.0    0       2       E_AS    U
2       1       2       671.3337471034806       2.239328339285575       10.0    1       2       E_AS    U
(...)
196     97      0       287.1114766079545       0.9577007991573774      10.0    97      0       E_AS    U
```

In the experiments done in this thesis, only a few of these values are used — the node IDs and the edges. Values that describe bandwidth and location are ignored.

The first column in the Nodes table is the node ID that is assigned to each running P2P-client. After being assigned a node ID, each node will look for occurences of their ID in the second column of the Edges table and connect to nodes with the ID given in the third column. The end result of this process is an overlay that has the same edges as the generated topology.

Appendix B Bloom filters

A Bloom filter is a hash based data structure that provides a membership function with a certain probability of false positives, never false negatives. Bloom filters were first described by Bloom in [20].

More specifically, a Bloom filter comprises an array of bits and a number of indedependent hash functions. When a data element is inserted into the filter, the hash functions are used to calculate a set of hash values representing the data element. For each hash value, the corresponding bit is set in the array.

To check whether a Bloom filter contains a specific data element, the process is the same, except that the generated hash values are compared to the existing array instead of being stored. If all the bits corresponding to the different hash values are true, then the data element can be determined to have been stored in the Bloom filter with a given probability.

A small Bloom filter example is shown in Figure B.1.

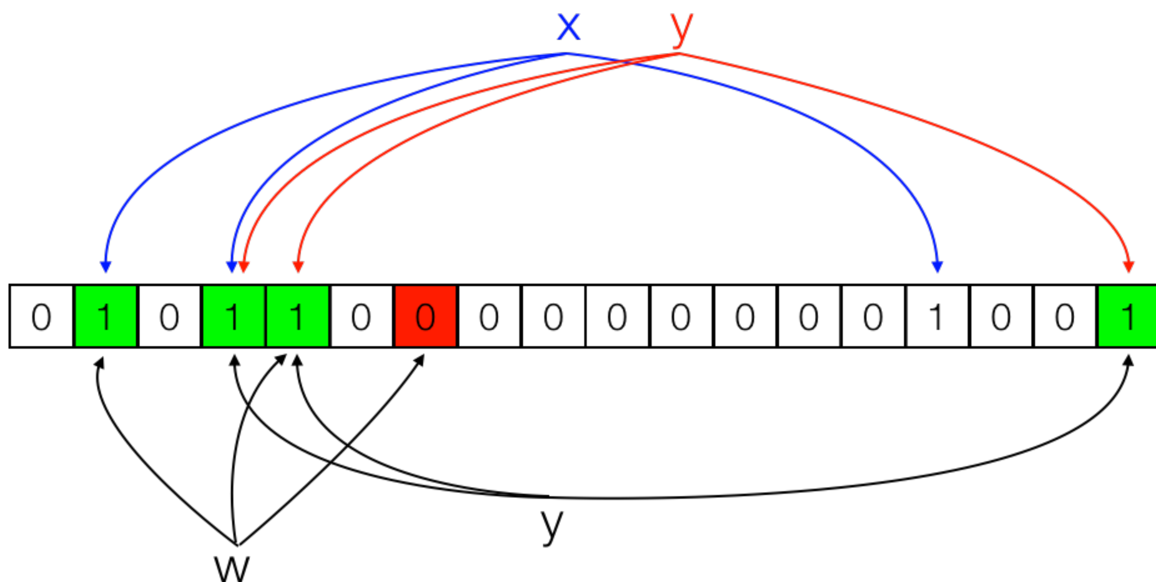

Figure B.1: A Bloom filter with $k = 3$ and $m = 18$. Data elements x and y are inserted into the bit array — one bit is set for each hash function. Elements w and y are checked against the filter — y is present with a certain probability, w is decidedly not since one of the bits is false.

The probability of a false positive in a Bloom filter is the same as the probability of all the bits for a data element already being set.

We can calculate the probability p that one specific bit is not set by a specific hash function in a bit array of size m with:

$$p = 1 - \frac{1}{m}$$

Further, we can calculate the probability p of a specific bit not being set by k hash functions after inserting n elements with:

$$p = \left(1 - \frac{1}{m}\right)^{kn}$$

Therefore, the probability p that k hash functions set k specific bits to true after inserting n data elements can be calculated with:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-k\frac{n}{m}}\right)^{k} \quad \text{(B.1)}$$

Equation B.1 gives us the probability that the bits corresponding to the hash values of a new data element are already set to true — or in other words, the probability of false positives.

Ideally, p should be kept as small as possible, but it can also be desirable to keep k or m small, to save computing resources or storage capacity — or in our case, bandwith. Intuitively, we can see that p will increase when n increases, and decrease when m increases. A high m/n will give a smaller probability of false positives.

The optimal number of hash functions k for a number of given data elements n in a bit array of size m can be determined by:

$$k = \frac{m}{n} \ln 2 \quad \text{(B.2)}$$

In our experiments we wanted to have a Bloom filter with low probability of false positives with 100 stored elements. We chose $n = 100$ and $m = 1000$, with $m/n = 10$. Using Equation B.2, we find $k = 6.93 \approx 7$. The probability of false positives can then be calculated with Equation B.1, giving $p = 0.00819$.

Appendix C Tools

C.0.3.1 NetBeans and Java

The source code is written in NetBeans [7] 6.5.

The implementations are compiled and executed with Java [6] 1.6 update 7 (64 bit) on Apple OS X 10.5.6.

C.0.3.2 Latex, emacs and make

The thesis is written in pdfeTeX [9] version 3.141592-1.21a-2.2 and emacs [2] version 22.3.1. The final document is built using GNU Make [4] 3.81.

C.0.3.3 Xfig

All illustrations are drawn in Xfig [14] 3.2p5, except where otherwise stated. Fig2dev is used to convert the illustrations to PDF format.

C.0.3.4 tcpdump

tcpdump [11] 3.9.7 is used to capture network traffic during the experiments.

C.0.3.5 tcpslice

tcpslice [12] 1.2a3 is used to split the packet capture files into a separate file for each evaluation phase.

C.0.3.6 tcpdstat

We use a version modified by David Dittrich [28] to process the network traffic captured by tcpdump. The original version and documentation can be downloaded from [43].

C.0.3.7 tcptrace

tcptrace [13] is used to generate bandwidth graphs from tcpdump's packet capture files.

C.0.3.8 awk and grep

Apple OS X 10.5.6's awk and GNU grep [3] 2.5.1 are used to extract bandwidth data and calculate averages from tcpdstat.

C.0.3.9 python

python [10] 2.5.2 is used to calculate moving and weighted averages of response times and success rates.

C.0.3.10 gnuplot

gnuplot [5] 4.2p4 is used to generate the all the graphs, except the bandwidth graphs that are generated with tcptrace. The input for gnuplot is processed with awk, grep and python.

C.0.3.11 BRITE and Otter

The topologies used in the experiments are generated with BRITE [49] and rendered graphically by Otter [37]. See Appendix A.