p-SARS

Peer-to-Peer Search for A Recommender System

Master of Engineering Thesis

Rune Devik

12.15.2003

FACULTY OF SCIENCE
Department of computer science
University of Tromsø, N-9037 Tromsø

p-SARS

Peer-to-Peer Search for A Recommender System

Master of Engineering Thesis

Rune Devik

12.15.2003

FACULTY OF SCIENCE
Department of computer science
University of Tromsø, N-9037 Tromsø

Abstract

WAIF started out in 2002 and the overall goal is to make the computers automatically search for relevant information based on the user's preferences, and to push this information directly to the user wherever he/she is and to whatever device he/she has available. In other words, make the machines serve us, with as little human interaction as possible. This thesis focuses on a specific part of this problem, which is to design and implement a search mechanism and the surrounding distributed system that can find publishers that publish on a given topic. We present a search mechanism that is scalable, fault resistant, self administrative and that utilizes the resources already present in the network. This is done by utilizing the powers of the unstructured overlay peer-to-peer architecture.

However creating an efficient search mechanism for a pure peer-to-peer net is known to be a problem due to the decentralized nature of these overlay networks. Our solution is to incorporate several known techniques. We propose the use of Random Walks supplemented by both a hint cache and a probabilistic gossiping mechanism. The results gathered show that the search mechanism has good coverage but is highly dependent on that the time to live (TTL) set on the query reflects the size of the overlay network and that the nodes individual hint caches are populated.

To verify our design we have both implemented the system and a simulator. We show with throughput testes and simulations that the system designed can scale to millions of users.

Keywords:
Peer-to-peer, unstructured, search, gossip, Information Retrieval (IR)

Preface

History

The ARPANET was conceived in the late 1960's. The main goal was to share computing resources around the US. To do this they created a common network architecture and thus seamlessly integrated heterogeneous networks, making each peer an equal participant. There where no firewalls yet and everybody could connect to any other peer on the net. Everybody consumed yet also produced data. Although the early killer application of the net, FTP [J. Postel, J. Reynolds. 1985] and Telnet [J. Postel, J. Reynolds. 1983], was client-server, the overall usage pattern was peer-to-peer [A. Oram. 2001]. In the early 1990's this would start changing.

With the introduction of the web came a change in the flow of information itself. Web browsers with an easy-to-use interface made the Internet publicly available. People just wanted to surf the web, and request/download was all they needed. As the web grew the ordinary client was no longer sharing, rather only consuming data from central servers. Client – server became the prevailing architecture. This paradigm shift was not entirely positive. As time went by the client computers became more powerful and people started realizing that utilizing these resources would be beneficial. So now it actually seems that we to some extent are going back to what once was.

The break through of peer-to-peer, in modern time, began with Napster [Napster]. This hybrid peer-to-peer application soon became a gigantic success. Literally over night everybody were willing to make their resources available to others. The resource here was bandwidth, and disc space used to trade music (MP3). But Napster was just the beginning. Soon people realized the weaknesses with the Napster architecture and started research to find better completely distributed alternatives and new application areas. Although much work has been done in the research community concerning peer-to-peer, and especially how to do searches in peer-to-peer nets, there are still many unresolved issues before we fully understand this paradigm and its potential.

Thesis Background

This paper is concentrated around peer-to-peer and a relatively new genre, information retrieval (IR). IR is gradually becoming a more and more interesting field for research because of the ever-increasing amount of data out on the Internet. The problem is basically twofold, to get good precision and good recall on searches. Precision is the relevance of the data found. That is, if all data returned from the search is relevant we have a precision ratio of 100%. Also if every available piece of relevant information is returned, we have a recall of 100%.

In 2002, a research project called WAIF [WAIF] was created at the University of Tromsø. Its main object is to change the current trend of information retrieval on the web. Instead of making the user search for information we should instruct the computer to do this work for us with minimal user interaction. When information is ready it should be pushed to the user regardless of where he/she is and what device he/she is using. Our thesis focuses on a part of this problem. That is, how we can find publishers that publish on some interesting topic and what architecture we should base this search mechanism upon.

Stakeholders and Audience

This thesis is written as a part of my Siv.ing. degree in computer science. The work itself is a part of the WAIF project, and the goal is to design and implement a prototype that satisfies the requirements set. The target audience for this paper is people with expertise in the field of computer science.

It is taken for granted that the reader has basic skills in the field of computer science, but he or she does not have to be an expert on the fields presented in this paper. It is however recommended that the reader has some knowledge about peer-to-peer and information retrieval. Knowledge about searching in peer-to-peer nets is especially relevant and the most relevant information will be presented in the theoretical framework chapter.

Acknowledgments

First I would like to thank the WAIF group as a whole for letting me participate in their project and especially Nils Peter Sudmann, who also was my teaching supervisor. He contributed with design ideas and also spent a lot of time helping me structure and review this thesis.

Other people I especially would like to thank are:

- Tom Inge Strøm for reviewing my thesis.
- Karl Magnus Nilsen for critical comments during the process of designing the system and also for reviewing my thesis.
- Jan Heier Johansen, for the cooperating project WAIF Recommender System (WRS) and our cooperation figuring out the interface between our systems (WRS and p-SARS).
- André Henriksen for helping me with mathematical statistics and always having a few cookies on standby.
- Raymond Haakseth, for helpful hints on the Python Language.
- All others that I've had the pleasure of getting to know while I studied here in Tromsø.

Last but not least I would also like to take the time to thank my family for always supporting me and making it possible for me, in the first place, to study here in Tromsø.

Tromsø, December 15th, 2003

Rune Devik

Content

List of figures

List of tables

Chapter 1

Introduction

In this chapter we describe the background for our problem, the problem itself and the scope of the thesis. We will also talk about the method chosen for this work and in the end we'll summarize our major findings.

1.1 Background

Our project is a subproject of WAIF [WAIF]. WAIF is an acronym for "Wide Area Information Filtering". The WAIF project as a whole investigates structuring techniques for future-generation large-scale distributed applications. This includes fundamental research issues like, how to best partition an application into a set of cooperating modules, how to optimize interaction among them, how and where to deploy them, how to interact with the users, how to provide integrity, security and auditing, and how to ensure fault-tolerance. In particular, WAIF is focusing on event-driven architectures supporting a more general publish/subscribe paradigm.

The environment we conjecture in the time frame of the project (2003-2006) is a ubiquitous and pervasive computing infrastructure where a single user (occasionally) might be supported by thousands, or even millions of computers.

WAIF is a joint project between the University of Tromsø, the Cornell University and the University of California, San Diego.

1.2 Problem definition

1.2.1 Problem description

At the lowest level WAIF [WAIF] is a push-based publish subscribe system. The WAIF virtual network of publishers and their clients form a directed graph initially created explicitly. Links in a connected graph may be optimized by analyzing traffic and by creating new links. However there is no way to discover and interlink two disconnected graphs. Because the clients explicitly have to set up connections to the publishers based on recommendations from other humans or what they might stumble upon on the Internet, the graph generated will, with a very high probability, be disconnected. The objective of this work is to implement a mechanism to discover and interlink these disconnected graphs. Since the set of members is dynamic and it's desirable to have a scalable self administrative solution, we will focus our work around a decentralized architecture. The method chosen to perform this work is rapid prototyping.

Problems to focus on:

- Which distributed model is best suited for this application.
- Design, implement, test and analyze the application.
- API to the search engine.

Our system has one cooperative project named WAIF Recommender System (WRS). While the work on the WRS system is concentrated around the protocol for passing the publications to those interested, our work is concentrated around finding (on demand) clients with similar interests. The underlying assumption here is that people with similar interests would like to exchange publications.

1.2.2 Focus

The focus of this thesis lies on design and implementation of a distributed search engine for the WAIF Recommender System (WRS), currently under development. More specifically the work includes:

- Identifying which architecture suits us best.
- Designing a search mechanism and the surrounding distributed system.
- Designing the interaction pattern between p-SARS and WRS.
- Implementing a prototype.
- Empirical studies such as performance testing.

1.3 Implementation environment

1.3.1 Language:

We've chosen the Python [Python] language for implementation. This language is an interpreted language and is supported by both Windows and UNIX. The reason for this choice is that Python is a relatively high level language, and this makes it a good choice for rapid prototyping. The code is also portable as long as the implementation does not include modules that target specific operating systems.

1.3.2 OS:

We chose to implement p-SARS on the Windows XP platform. The main reason for this is familiarity. But because Python is an interpreted language, and our code portable, the p-SARS system will run on all platforms supporting Python run-time.

1.3.3 Hardware:

For the implementation we used an HP pavilion ze4400. This is a laptop with the following important characteristics:

- CPU
    - AMD Athlon XP-M 2400+ (1,8 GHz)
    - 266 MHz front side bus
    - level 2-cache 512 kB
- Memory
    - 2 x 256 MB DDR PC2100 266MHz
- Network
    - 10/100 LAN Ethernet, integrated

1.4 Method

Work in the discipline of computing follow three major paradigms, or cultural styles if you will [P. J. Denning. 1989]:

- *Theory:* This paradigm is rooted in mathematics, and results in the development of a coherent, valid theory.
- *Abstraction:* This paradigm is rooted in the experimental scientific method and results in an investigation of a phenomenon.
- *Design:* This paradigm is rooted in engineering and results in the construction of a system that solves the problem stated.

As our work mainly will be focused around design and the implementation of a search engine prototype, the design paradigm suites us the best as an overall strategy. But in order to design our search engine we also need a research method. Since we are building an experimental prototype we need an experimental method. Because of that, our choice fell on Rapid Prototyping [A. Macro. 1990]. The key reasons for this choice are:

- It’s easy to find the needed requirements during prototyping.
- We get started right away.
- It’s easier to involve other people and get feedback when we can show them something that works and maybe even let them use it.

To a certain degree we have also followed the template in [G. Hartvigsen. 1998] on how to write the thesis itself.

Some UML[1] has been included into the design phase. More specifically we have chosen to model the information flow with the help of activity diagrams. The reason for this is threefold. First, visualizing with UML makes the design easy to read and maintain, second UML is a standard so it’s likely that whoever has an interest in this thesis already is familiar with this modelling language, and third, while modelling the system with activity diagrams a better understanding of the system’s hot spots is obtained.

[1] UML is an acronym for The Unified Modeling Language

1.5 Issues not investigated

Here we'll describe some fields in peer-to-peer computing that, although important, we have chosen not to emphasise on. The reason is that they are not particularly relevant in our research.

- *Free-riders* are a major problem in current peer-to-peer nets. A free-rider is a person who benefits from using the peer-to-peer net but does not contribute with resources to the network himself. Solutions for this problem are in the line of giving incentives to the user that makes him/her more eager to share resources [P. Golle et al. 2001].

- *Security:* The use of peer-to-peer often requires third parties to be allowed access to our computers, and thus our resources such as CPU and disc storage. Security then becomes an important issue. All systems should be able to provide both confidentiality and integrity, and this also applies to peer-to-peer systems. Current solutions inside different organizations are to either totally ban the use of peer-to-peer applications or introduce strict policies to their use [D. Piscitello. 2002].

- *Legal issues:* All press attention on peer-to-peer in recent time has been concerned about legal issues using file sharing applications. Napster [Napster] was not online long before RIAA [RIAA] started the process of shutting it down. The newer applications like Gnutella [Gnutella] are not as easy to stop because everything is completely distributed. Therefore the record companies now have a new strategy. They sue people that share large amounts of illegal files on these systems instead. Fred von Lohmann, who is an attorney, has written an article on what we as developers should be aware of when creating a peer-to-peer system so that we do not end up losing a law suit. In [F. v. Lohmann. 2003] he explains that we essentially have two options. Either we create an architecture that provides total control over our users, or we go for the total anarchy approach where we know nothing about the users.

1.6 Major results

This thesis presents the design and implementation of a scalable, fault tolerant, self administrative and completely distributed search engine. This is accomplished by utilizing the powers of the unstructured peer-to-peer architecture.

However creating an efficient search mechanism for a pure peer-to-peer net is known to be a problem due to the decentralized nature of these overlay networks. Our solution is to incorporate several known techniques. We propose the use of Random Walks supplemented by both a hint cache and a gossiping mechanism.

We have also built a simulator to simulate the efficiency of our proposed search mechanism and two different gossip mechanisms. By comparing the simulations and throughput tests performed on the p-SARS prototype we argue that the system is capable of supporting in excess of one million clients.

We have also discovered that our proposed Random Walk Gossip (RW-G) mechanism performs remarkably well in our simulations. In an overly network of 10 000 p-SARS nodes we have a success ratio of 100% on our searches when we use this gossip mechanism to populate our hint-caches. And furthermore, in our simulations the overall load increases slower than the additional processing capability when we include more p-SARS nodes in the overlay network. That is, our solution seems to scale better the more WRS clients are included into the p-SARS overlay network.

1.7 Outline

Chapter 2 – This chapter presents the theoretical background for our project. It will be concentrated around the search problem in peer-to-peer networks, but also the distributed hash table (DHT) technique will be presented. Replication and bootstrapping is also covered.

Chapter 3 – This chapter presents related work. We will cover information retrieval projects, file sharing projects and the utilization of the resources present in the leaf nodes of the Internet.

Chapter 4 – The requirement chapter first describes our cooperating project the WAIF Recommender System (WRS). Then the functional and non-functional requirements set for our distributed search mechanism p-SARS are listed.

Chapter 5 – This chapter first describes the overall architecture and then the interaction between p-SARS and the WAIF Recommender System (WRS). Then the design of our four system mechanisms; the membership mechanism, the topic update mechanism, the gossip mechanism and the search mechanism are presented.

Chapter 6 – Here we describe the implementation of the modules in the p-SARS system. We use UML to visualize the work flow of each module.

Chapter 7 – This chapter presents the design and implementation of our simulator. We also present the different tuneable parameters and the simulator's input file.

Chapter 8 – This chapter presents the testing and the simulations of our p-SARS system. The results are also compared and discussed.

Chapter 9 – In the discussion chapter we mainly discuss the four system mechanisms identified during design, and enhancements to these.

Chapter 10 – This chapter concludes the thesis.

Chapter 11 – This chapter presents the Bibliography of our thesis.

Appendix A – This appendix contains the full source code listing for our p-SARS system.

Appendix B – This appendix contains the full source code listing of our simulator.

Appendix C – Here we describe every event flowing into, through and out of our p-SARS system.

Appendix D – This appendix contains a CD-ROM containing all source code and test results.

Chapter 2

Theoretical Framework

In this section we'll describe the theory behind peer-to-peer networks. We start out giving a brief overview of the peer-to-peer paradigm as a whole, then delve deeper into the problem of searching these overlay networks. Then we will look at distributed hash tables (DHT), and why searching these overlay networks are hard. In the end we talk about replication and membership in peer-to-peer networks.

2.1 The Peer-to-peer paradigm

Peer-to-peer is a decentralized architecture where each peer has the same (or similar) capabilities and where the peers cooperate to solve a given problem or to offer a service like e.g. file sharing. The architecture itself is often divided into three sub-groups [Q. Lv et al. 2002]:

> *Centralized peer-to-peer*: All nodes have the same responsibilities, but a centralized server performs some service needed by the peer-to-peer net.
>
> *Decentralized and unstructured peer-to-peer (pure peer-to-peer)*: Every node has the same capabilities and responsibilities. There is no central service; everything is evenly distributed onto the nodes.
>
> *Decentralized and structured peer-to-peer*: In this approach there is some structure on the net itself. The strictness of the structuring varies between different approaches, but the overall goal is to make searching/lookup mechanisms better and more effective (scalable, fast and correct). Examples are distributed hash table systems (DHT's), routing based on hints and the notion of super-nodes or super peers.

2.2 Searching in a peer-to-peer net

Searching inside peer-to-peer networks has been, and still is, a subject of much research. To start out our description of the different approaches related to our work, we divide the different solutions into the three different sub-groups identified above.

2.2.1 Centralized (The hybrid approach)

In the centralized approach we have a server that performs the search on behalf of the systems clients. This is also called the hybrid approach and is thoroughly discussed in [B. Yang, H. G. Molina. 2001]. The advantages of this are that the system offer completely accurate searches and a fast response. The disadvantages are that it requires extra hardware (i.e. the server(s)), one or more administrators and that the server itself is a single point of failure.

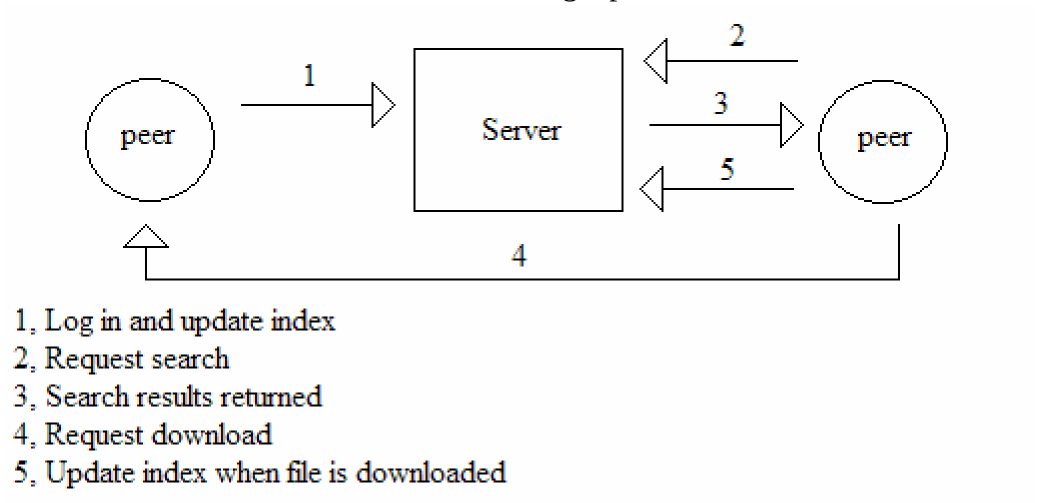

peer
1
Server
2
3
5
peer
4
1. Log in and update index
2. Request search
3. Search results returned
4. Request download
5. Update index when file is downloaded

Figure 1 - The centralized search approach

This approach works as follows:

- A client has to log in (1) and provide, to the server, a list of which objects it shares.
- When a client wants to perform a search, it sends the search request to the server (2).
- The result is returned to the client (3), telling it which peers, if any, has the object in question.
- The client connects to one of the peers provided by the server requesting a download (4).
- When the client receives the object it has to update the server (5) telling it that it now also shares this object.
- The search index must also be updated when a client disconnects. The objects that this client shared are no longer available.

2.2.2 Decentralized

In the decentralized approach we have no administrators, no extra hardware and no single point of failure. But the problem is that some of the desirable capabilities of the centralized approach like e.g. completely accurate and very fast searches cannot be obtained. Because the search engine is completely decentralized we use what's called 'blind searches'. We don't know where to find the data so we are in a sense blind. The solution used in early peer-to-peer applications was the flooding algorithm described in figure 2.

```
if result message:
    if querying node:
        display results
    else:
        # Lookup from which node we received the query from
        node = lookup(query_id)
        # Send results back to this node
        send(result, node)
else:
    if query known:
        # Discard known query
        pass
    else:
        # Register query, and the node from which we
        # received the query
        register_node(query_id, node)
        # Perform search
        result = local_search(query)
        if result != empty
            send(result, node)

        # Decrement TTL
        TTL = TTL – 1
        # Relay search to neighbour as long as TTL
        # is still positive and the neighbour is not the same
        # neighbour as the one that sent the search to us
        if TTL > 0:
            for all neigbours != node:
                send(query, neighbour)
```

Figure 2 – Pseudo code for the flooding algorithm

This algorithm is essentially a limited broadcast, where each node sends the query to all of its neighbours as long as the time to live (TTL) value is positive. When a node receives a query for the first time, it registers the query along with the node that sent it. It's important to note that the node that sent the query needn't be the node where the query originated. The node then performs a local search and sends back the results, if any, to the node registered on this query.

The second stage of the algorithm is the relaying of the query. The TTL is decremented and if it's still positive the query is sent to every neighbour except the one which we received the query from. Any results found are routed back to the querying node using the same path as the search to avoid a message implosion at the querying node. This may happen because a potential high number of nodes may search in parallel and return the results simultaneously. A query is typically processed only once at a specific node. Any duplicates are discarded.

The major disadvantage with the flooding algorithm is that it doesn't scale very well. As more clients connect, the search traffic in the net overwhelms the clients [J. Ritter. 2001]. Possible solutions on this problem are discussed in [Q. Lv et al. 2002] where the most promising one is Random Walk.

Random Walk works as follows: Instead of flooding the query to all neighbours, one is selected at random and the query is forwarded to him. They still use a TTL to ensure termination of the algorithm, but a Random Walk may also terminate when the desired number of hits is reached[2]. To decrease the delay before a hit is found they may also increase the number of walkers. The reason that the answer is routed back to the querying node in the flooding algorithm is as mentioned the fear of message implosion at the querying node. In the case of random walks this will not happen because there are only one or very few peers searching on behalf of the querying node at any given time. Therefore the walker may return the answers directly when they are found, and it can also check back with the querying node to figure out when the query is satisfied.

2.2.3 Decentralized but structured

The decentralized yet structured approach tries to i.a. reduce network traffic by imposing a structure on the peer-to-peer virtual overlay network itself. DHT systems are in this group, but we shall discuss these in a separate section because they essentially do not solve the search problem, but instead what's known as the lookup problem [H. Balakrishnan et al. 2003].

Essentially there are three approaches to create a more effective search than blind search:

- The first approach incorporates what is known as super-nodes. Super-nodes are introduced because the peer-to-peer net is often very heterogeneous. This again implies that some nodes are better suited to handle some services, such as searching, on the behalf of the others. How to best elect these nodes are discussed in [A. Singla, C. Rohrs. 2002], and include the need for sufficient bandwidth, suitable OS, and sufficient uptime. If super-nodes are to handle all searching in the net, they must cache what data their clients share and also decide on a search algorithm to use between the super-nodes themselves. In a way this approach transforms the decentralized peer-to-peer net to a hybrid peer-to-peer net where the super-nodes cooperatively act as a dedicated server for searching. But since the super-nodes are elected dynamically we eliminate the single point of failure.

[2] The query is then said to be satisfied.

- The second approach is known as semantic searches. If a pattern in the data can be found, relations between data can be drawn and this gives rise to more powerful queries. We’re not restricted to search for the name of the resource, but we may now search on the content itself. It’s clearly easier when the searchable data consists of text documents only [C. Tang et al. 2003], than if for example music, pictures [C. Falaoutsos et al. 1994] or video files are included.

- The third approach is to build some sort of routing table. Routing tables can be build by e.g. observing the traffic in the net. It might also be a good idea to group together peers that share some of the same objects, because peers with similar interests are more likely to satisfy each others queries than random peers [E. Cohen et al. 2003]. One hypothesis on how to achieve this grouping of peers is presented in [C. Gkantsidis et al. 2004]. The idea is to use the nodes that previously have answered your queries to populate your neighbour set. This will, according to the hypothesis, lead to a formation of communities of users with similar interests.

2.3 Lookup in peer-to-peer net (DHT)

Distributed hash tables (DHT) do not solve the search problem, but instead what's known as the lookup problem. The lookup problem is stated as follows: Given an object X stored at some dynamic set of nodes in the system, find it. This is discussed in [H. Balakrishnan et al. 2003], where the conclusion is that it's still an open question if it's possible to layer indexing and keyword searches effectively on top of DHT systems. Although these systems do not currently address our search problem, they are highly related.

There are two important properties of a DHT system that we indeed would have liked to incorporate into a distributed search algorithm. First, they guarantee to do a lookup in a fixed amount of steps, often O(log(n)) steps where n equals the number of nodes in the overlay peer-to-peer network. Second, if the object exists, it is guaranteed to be found. The drawback of these systems is that if we want to find an object, we must know its name. That is, we have to know exactly what we are looking for. Another problem is the joining and leaving of nodes. Each time a node either connects or leaves, the network must apply some resources to restructure the overlay network.

All these systems e.g. Pastry [A. Rowstron, P. Druschel. 2001], CAN [S. Ratnasamy et al. 2001], Chord [I. Stoica et al. 2001] and Tapestry [H. Hildrum et al. 2002] are very similar, and therefore we will describe them in general. Each node in the graph is given a node id from an id-space when it connects. When a node wants to publish an object, it hashes the name of the object into this id-space. The node that has the numerical closest id to this hash has to either hold the object or a reference to where the object can be found. A lookup is then straight-forward. First we hash the name of the object, and then we look up the numerical closest node in the id-space. This node will hold the object or a reference to the object, if it exists.

Because of scalability issues each node cannot keep a complete and updated list of all nodes in the overlay network. In their routing tables they therefore keep some of the numerical closest nodes, and some of the numerical distant nodes. When a lookup propagates through the overlay network towards its goal, it's always sent to a node that has a node-id numerically closer to the object-hash. Because of the structure of the routing tables it is ensured that the lookup requires only log(n) steps.

2.4 Replication

One way to make searches more effective is to duplicate the objects, or references to objects, on to different nodes in the net. When an object resides on several nodes the chance of finding a particular object increases. The problem is, however, to find the optimal replication strategy. There are several papers discussing this problem e.g. [E. Cohen, S. Shenker. 2002] [Q. Lv et al. 2002] and they conclude that square root replication is nearly optimal.

In [E. Cohen, S. Shenker. 2002] square root allocation is defined as an allocation where for any two objects the ratio of allocations is the square root of the ratio of query rates. Another observation made in this article is that square root allocation lies between Uniform and Proportional allocation, although surprisingly much closer to the Uniform allocation. In a Uniform allocation all objects are replicated equally as appose to Proportional allocation where more popular objects are replicated more frequently than less popular objects.

Although, theoretically, square root replication is sophisticated it’s shown by [Q. Lv et al. 2002] that it’s actually not hard to achieve in practice. One of the algorithms proposed is path replication. This algorithm replicates the objects on the search path form the querying node to the node holding the object.

2.5 Bootstrapping and Maintaining Membership

When a node wants to join the overlay peer-to-peer network the question arises of how to find an initial node to connect to. The solution widely adopted is one or more centralised bootstrap-servers. Nodes register themselves to a well-known server, and in return they get a list, possibly randomised, of other nodes that already are a part of the peer-to-peer network. The only other solution today is to ask all other nodes on the Internet to find out if they run as a peer in the overlay network in question. This is obviously not a solution to consider. One might think that broadcasting a request on an LAN might be a solution, but there is no guarantee that there are others on that LAN currently connected to this overlay network, in which case we again will end up searching the whole Internet. There is however another solution, but it requires that the node has previously been a member of this overlay network and that it cached the addresses to the nodes it was in contact with. When such a node tries to bootstrap, it can initially try to contact the nodes it already knows, but if this fails it must again fall back to the bootstrap-server approach.

Another problem is that the choosing of nodes to add as neighbours influences the topology of the overlay network. In [A. J. Ganesh et al. 2003] they show how to use random walks to establish an overlay graph that is well connected. When an arriving node has found a node already connected to the overlay network, a join operation is initiated at the discovered node. The join operation is such that the returned nodes are picked nearly uniformly random. This ensures that there is a high probability that the resulting overlay network stays well connected. In their article they also discuss how to keep this desired graph property when nodes join and leave.

In the distributed hash table (DHT) approach, the problem of finding the initial node is also solved with a bootstrap-server. The difference between DHT and unstructured peer-to-peer is the join operation after the initial node is found. In DHT this requires a restructuring of the overlay network itself, and possibly also moving some data between nodes whereas nothing needs to be done in the unstructured approach. The restructuring in DHT is necessary because a new node added to the net gets a node id from the global id-space. This means that the new node possibly has to take over the responsibility of some of the objects distributed on the overlay network, because it is now the numerically closest node. It also has to make its presence known to other nodes so that they may update their routing tables. At last the joining node also has to build up its own routing table.

2.6 Summary

In this chapter we have presented the theoretical background for our project. We have shown that the peer-to-peer paradigm consists of three different architectural approaches; centralized, decentralised and unstructured, and decentralised and structured. We have also shown how the search mechanism has to be adapted to the underlying architecture.

We then presented a nearly optimal replication strategy, square root replication, to make the searches in a decentralized peer-to-peer network more efficient. In the end we discussed the bootstrap problem and that the joining and leaving of nodes affect the overlay network topology.

Chapter 3

Related work

In this chapter we will present work related to our project.

3.1 Information Retrieval projects

In Stumbleupon [Stumbleupon] the idea is similar to the idea in the WAIF Recommender System (WRS). Push relevant information directly to the user from peers or friends with similar interests. The system itself is implemented as a web-browser plug-in. To start up the system the user must configure its interests or the profile, as we will call it, so that only relevant information is received. The user may also include friends in this profile. The assumption here is that close friends share the same interests.

Information is pushed to the users based on ratings. Members rate the web-sites they come over, and the highest rated sites in accordance with the client's profile are presented to the client when asked for. Another feature is that the system learns what the client thinks is relevant information based on the client's ratings. So the more a client participates in the rating process, the more relevant the information pushed to this client will be.

In NewsMonster [NewsMonster], as with Stumbleupon, the user may rate websites and share ratings with other users automatically. But this program also supports news gathering from RDF Site Summary (RSS) streams and automatic caching of articles for offline surfing before the user even requests them. The user may also search through all current subscriptions and already cached articles. As with Stumbleupon, the user must create a profile and the profile is automatically updated based on what the user rates as relevant information. This application is integrated with Mozilla and Netscape.

Konspire [Konspire] is a new type of file sharing application that pushes files to the user before he/she asks for them. The pushing is based on which channels the user currently subscribes to. The user may also start a new channel, broadcasting files to other subscribers.

The Oxygen [Oxygen] project at MIT is a highly related project. Their goal is basically the same as the goal of the WAIF project as a whole: Put the people in centre. Make the machines invisible but in the same instant make them omnipresent. Make them serve us, not the other way around. Push relevant information directly to the user, wherever he/she is and to whatever device he/she might have available. Create software that adapts to changes in the environment or in user requirements. E.g. if a user is at work, he/she might only be interested in work related information. The basic scheme for the project is therefore to make the machine adapt to the users based on where they are, what they do and what they are interested in.

3.2 File sharing applications

3.2.1 Napster

Napster [Napster] was released in the fall of 1999, and was the first file-sharing peer-to-peer application. Although it wasn't a pure peer-to-peer application because the search engine was situated on a dedicated server, the actual download was done directly between peers. This is known as a hybrid peer-to-peer network.

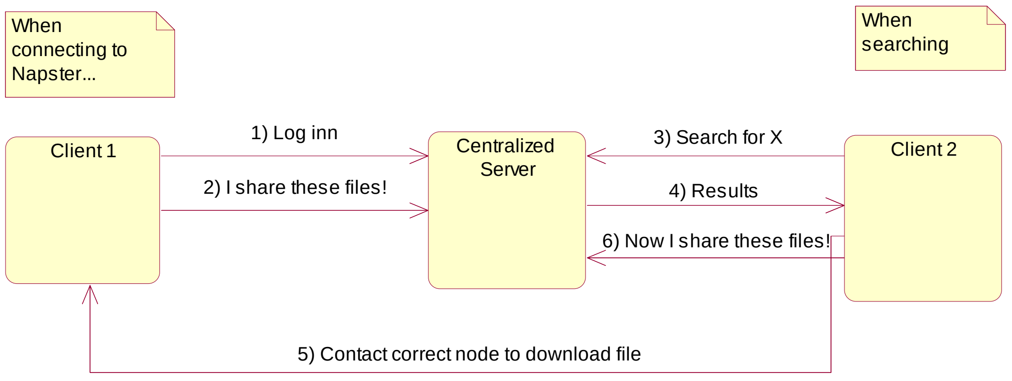


Figure 3 - Searching as it worked in Napster (Simplified)

When a client connects to Napster, which uses a central server, the client logs in (1) and provides information about which files it shares (2). The files are recorded at the central server to satisfy future requests (3). To give the clients an up-to-date view of the system, the search index is updated when a client downloads a file (6) or when a client goes offline. The advantages of this architecture are that the system offers completely accurate searches and fast responses. The disadvantages are that it require extra hardware (i.e. the server(s)), one or more administrators and that the server itself is a single point of failure.

Placing the search engine on a central server gave the system excellent searching capabilities, but it also meant that the Napster crew knew that copyrighted material was shared and that they had the means to stop it. This was also discovered during the trial between Napster and RIAA [RIAA].

The populated user database actually meant more investors and more money. This way the court showed that Napster indeed benefited financially from the file sharing, that they had to know about what was going on, and that they had the ability to control their users. The users could simply be denied access. It was demanded that copyrighted material was filtered out, and that users indulging in trading copyrighted material should be denied access to the system. Napster only got their filters 99% correct, but the judge demanded 100%. This was never achieved and the search service was taken offline, resulting in a total shut down of the file sharing service itself.

3.2.2 Gnutella

While Napster was up and running the interest for the peer-to-peer paradigm was increasing. This resulted in e.g. the Gnutella [Gnutella] protocol where everything including the search mechanism was decentralized. When Napster was shut down people started looking for a substitute and over night the Gnutella net got thousands of new users.

On top of the Gnutella net you find many different client applications. Some of the well known, and still up and running, file sharing applications are Morpheus [Morpheus], BareShare [BareShare], WinMX [WinMX], Grokster [Grokster] and KaZaA [KaZaA]. They are all currently under some kind of dispute with the record companies, but as of yet it seems that the decentralized nature of their systems might save them. The reason is that no servers owned by the companies actually indulge in the file sharing process themselves. Everything is decentralized and run by the users.

Gnutella started out using the flooding algorithm described in the theoretical framework chapter:

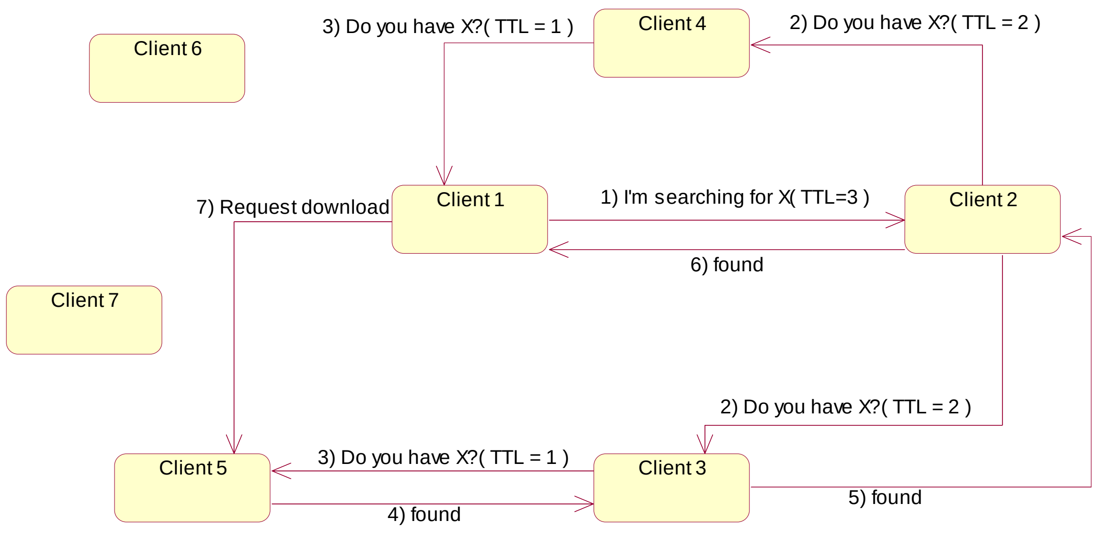

Client 6
3) Do you have X?(TTL = 1)
Client 4
2) Do you have X?(TTL = 2)
7) Request download
Client 1
1) I'm searching for X(TTL=3)
Client 2
6) found
Client 7
2) Do you have X?(TTL = 2)
Client 5
3) Do you have X?(TTL = 1)
Client 3
4) found
5) found

Figure 4 - The flooding algorithm

A search is initiated by a querying node (client 1) sending a search request to all of its neighbours (step 1). The request propagates through the net (step 2 and 3), where each node receiving the query sends it on to all other neighbours, except the neighbour from whom it received the query, and only as long as the time to live (TTL) is still positive. If a node finds the file we are searching for, we have a hit (client 5). The result is propagated back the same way the search came until it reaches the querying node (step 4, 5 and 6). This node then downloads the file directly from the node holding the file (step 7).

Since the flooding algorithm doesn’t scale very well, it was soon discovered that structuring the overlay peer-to-peer network could potentially be a good idea. The most used structuring technique today is the notion of super-nodes or super-peers which we discussed in the theoretical framework chapter. Applications that use this technique are e.g. KaZaA [KaZaA] and Morpheus [Morpheus]. The reason that the distributed hash table (DHT) technique has not yet been widely adopted in these applications is the problem of searching these nets. Another problem is the extra work needed maintaining these nets as nodes join and leave.

3.3 CPU sharing projects

In its widest sense, this is also peer-to-peer. Philosophically, peer-to-peer is to take advantage of the resources in the net’s leaf nodes. In the architectural sense though, these projects fall in under the controller – worker paradigm. But because the sharing of resources is such a vital notion in our project, we’ve chosen to include a description of some resource sharing success stories.

3.3.1 SETI@home

This project aims to “…search out new life and new civilizations.” The goal is to utilize the waste amount of unused computing power out on the Internet, and put it to use analyzing radio signals from space.

This can be accomplished because the problem can be divided, and the results can be computed in parallel. By allowing people to download and install a screensaver, the SETI@home group accomplishes their goal of utilizing idle CPU time. When the client computer becomes idle, the screensaver is activated and starts to process a data chunk from Berkley. After a while the chunk is processed, and the results are sent back. In response, another work unit is received by the client. Their progress and extended background information can be viewed at their homepage [SETI@home].

3.3.2 Intel philanthropic peer-to-peer program

This project was created by Intel to demonstrate the power of distributed computing, and has currently several sub-projects around the world, all in the field of medical research. All projects have the similarity that the problems can be divided and processed in parallel. The most widely known of these sub-projects is probably the project running at the University of Oxford. In their project Screensaver – Lifesaver, they have concentrated their effort in the field of cancer research. The goal is to identify molecules that interact with proteins that in advance have been determined to be a potential target for cancer therapy. Through a process called virtual screening, it will be determined which molecular candidates have a high likelihood of being developed into a drug. As the name suggests, they also follow the SETI@home approach. By creating the application as a screensaver, they manage to tap into the idle resources on millions of computers world wide. Their progress and extended background information can be viewed at their homepage [Screensaver – Lifesaver]. Information on the Intel philanthropic peer-to-peer program can be found at [Cure], including links to other sub-projects.

3.4 Summary

This chapter presented work that is related to our project. We started out describing some information retrieval projects. Information retrieval projects focuses on retrieving information, preferably from different sources, and push this information directly to the users based on their interests. Some projects like WAIF and Oxygen also try to create adaptive software. That is software that is able to change its behaviour based on what context the user currently is in and what device the user currently has available.

We then presented two widely known file sharing applications and i.a. discussed how the search problem was solved in these applications.

In the end we described some CPU sharing projects that with their success shows that there actually is a lot of resources available in the leaf nodes of the Internet. Not only that, but it also shows that people is able and willing to share these resources as long as it is deemed safe to do so.

Chapter 4

Requirements

In this chapter we will present the requirements set for the p-SARS system. We start out by describing the cooperating system named WAIF Recommender System (WRS). Then we will present the functional and non-functional requirements set for the p-SARS system in detail.

The presentation of the requirements has incorporated some ideas from The Volare Specification Template described in [S. Robertson, J. Robertson. 1999].

4.1 Cooperating system

Before we describe the WAIF Recommender System we will take a look at the whole vertical distributed system stack.

4.1.1 The system stack

Figure 5 shows an overview of the vertical distributed system stack. The system stack is divided into four modules. At the top all WAIF applications will run. This is typically GUI applications taking advantage of services the middleware layer provides.

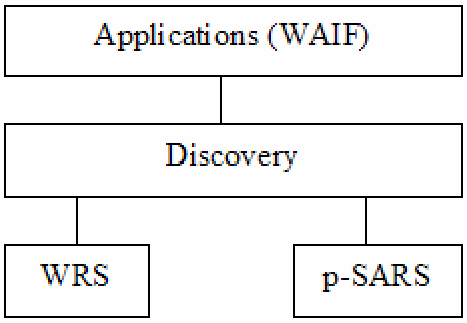


Figure 5 - Overview of the vertical distributed system stack

The main task of the discovery module is to minimize the needed user interaction. It will use both the WAIF Recommender System (WRS) and p-SARS to find and suggest new sources of info based on e.g. user behaviour and interests.

We envision different application to be situated on top of the discovery layer. These are e.g. applications for publishing links and / or files in general. The specific application itself may be standalone or e.g. a web-browser plug in.

4.1.2 The WAIF Recommender System (WRS)

The goal of the WAIF Recommender System (WRS) is to provide users with personalized publications based on the ratings of other users. This is accomplished by running an application on each client that is capable of both receiving and publishing publications.

As a client subscribes to a publisher it also submits a threshold value of accepted personal rating of that publisher. If the personal rating of the publisher falls below this threshold the subscription is terminated. Subscribers rate all publications received and the publishers personal rating is some summarization of all these ratings. This means that the group as a whole rates each publisher, and based on this rating and the threshold value, set independently by each client, each subscriber decides if it still wants to receive publications from this source. This will prevent publishers from spamming a group, and only highly relevant and personalized information will flow through the system to the clients. It's assumed that the WRS system has to run a while before it has accumulated enough data to perform at its best. The system has to learn who is a good producer, and who is interested in what, based on user feedback.

The work in the WRS project is in reality concentrated around finding the appropriate communication protocol for message passing between clients. And also how to ensure that only relevant information is passed on to the subscribers. But to facilitate testing a GUI application is also implemented. This application allows clients to publish and/or subscribe to publications[3].

The goal of our p-SARS system is to extend the WAIF Recommender System (WRS) with an external search level. As shown in figure 6 the WRS system sets up a directed graph between publishers and subscribers. This graph is initially created manually. That is, the client must explicitly set up connections to already known publishers. There is no way for a client to search for a publisher on a given topic and therefore our p-SARS system will extend a subset of the WRS nodes with such a search capability. This subset is selected by the WRS system, and the WRS nodes selected to run our search mechanism are called super-nodes.

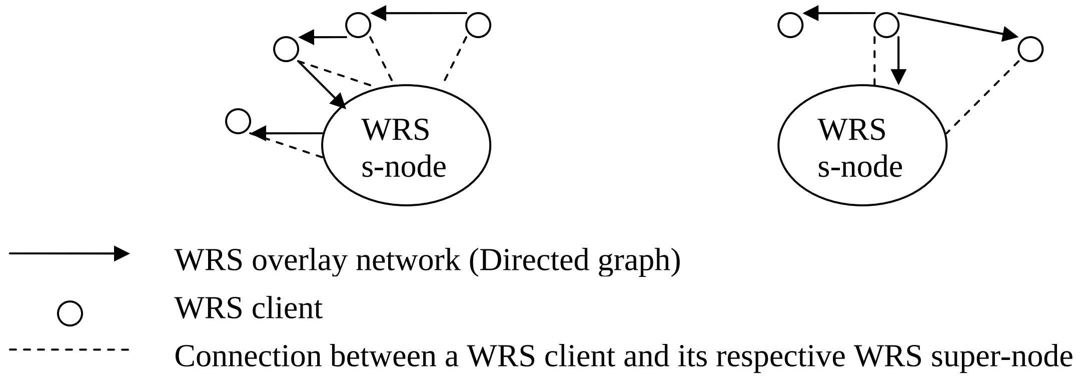

WRS
s-node
WRS
s-node
WRS overlay network (Directed graph)
WRS client
Connection between a WRS client and its respective WRS super-node

Figure 6 - Two disconnected WRS overlay networks

[3] Only plain text publications are currently supported in this application.

It’s important to notice here that the WRS system is thought to scale to millions of clients. That is, this is not an application just created for the members of a single project group. This application could potentially be used by every client connected to the Internet creating a world wide network between publishers and subscribers. Another important issue is that the topics are generated by humans and reflects their interests. We may therefore assume that these topics will change slowly, because the interests of each individual human changes slowly.

Next we will present a summary of the system requirements. The rationale behind them will follow in later discussions.

4.2 Functional Requirements

These requirements state the functions or actions that are to be part of the finished p-SARS system. But only the functions or actions that contribute directly to the goal of the system. That means that e.g. look and feel requirements are not included here. All requirements are stated with a fit criterion to facilitate testing the system functionality.

Requirement 1
Description: ***The system must be able to bootstrap.***
Fit Criteria: *When a node is started up it manages to connect to the p-SARS overlay network, as long as there are other nodes running.*

Requirement 3
Description: ***Provided with a topic the system should support searching for publishers on this topic.***
Fit Criteria: *When the system receives a query request, it should be processed and an answer should be given to the querying WAIF Recommender System (WRS) client.*

Requirement 4
Description: ***An API to the search engine should be provided for external systems like the WRS system.***
Fit Criteria: *The WRS system and the p-SARS system are able to communicate.*

Requirement 7
Description: ***The system must keep a topic set, describing which clients are publishing on which topics.***
Fit Criteria: *The topic set is populated with the right information when the node is up and running so that searches may be satisfied.*

Requirement 8
Description: ***The local topic set should be updated periodically.***
Fit Criteria: Changes in the topic set situated on the WRS super node shall after update be reflected in the locally cached topic set on the corresponding p-SARS node.

Requirement 10
Description: ***The system must try to keep a populated neighbour set.***
Fit Criteria: *When a node is up and running it should have a populated neighbour set >= 1 so that a Random Walk may be performed.*

4.3 Non-Functional Requirements

Non-functional requirements describe the properties that the finished product must have. E.g. operational and performance requirements that applies to the system. In short we may say that everything that is not a part of the system's fundamental functionality is included here. We use the same presentation as with the Functional Requirements above but divide the requirements into two groups starting with the performance requirements.

4.3.1 Performance requirements

Here we will present the requirements we have incorporated into our design to improve the overall performance of the system.

Requirement 5
Description: ***When a local client starts to publish on a new topic, disseminate this information. (Disseminate local view.)***
Fit Criteria: *When a new topic is discovered after the locally cached topic set has been updated a message informing surrounding nodes should be sent.*

Requirement 6
Description: ***The system must keep a hint set (routing table), possibly limited in size.***
Fit Criteria: *When the system is warm, the routing table should be populated with some information on what topics clients of other super-nodes publish on.*

Requirement 9
Description: ***When a node connects to the peer-to-peer system it should make use of already populated hint sets.***
Fit Criteria: *When a node connects it should receive the routing tables of its neighbours.*

4.3.2 Operational requirements

These requirements describe which environment the system is meant to operate successfully in. This include e.g. platform and if there is some special architectural needs.

Requirement 2
Description: ***The system should be decentralized.***
Fit Criteria: *No single failure should bring the system as a whole to a halt.*

Requirement 11
Description: ***The system should be designed as modules.***
Fit Criteria: *The system is easily maintained, and changes are easily incorporated.*

Requirement 12
Description: ***The system should work in both Windows and UNIX***
Fit Criteria: *The code is portable to both operating systems.*

4.4 Summary

To set our system in perspective we started out this chapter describing the vertical distributed system stack and then more specifically our cooperating system the WAIF Recommender System (WRS). The goal of the WRS system is to provide users with personalized publications based on the ratings of other clients. This is accomplished by running an application on each client capable of both receiving and publishing publications. Our p-SARS system will enhance this WRS system with an external search level so that the clients of the WRS system may search for publishers publishing on a specific topic.

In the end we summarized the requirements set for our p-SARS system. We divided the presentation in two; functional requirements and non-functional requirements.

Chapter 5

Architectural design

In this chapter we will present the design of the p-SARS search engine and why we have chosen the solutions we have. While discussing our design choices we will also refer to the corresponding requirements already stated in chapter 4.

We start out explaining our overall architecture then the interaction between p-SARS and the WAIF Recommender System (WRS). In the end we will describe the design of the main system mechanisms[4].

5.1 The overall architecture

Before we started to design our system we needed to decide on the overall network architecture. The choice was basically twofold. Either we design the search mechanism as centralized client-server or we create some sort of a decentralized solution. Since the decentralized approaches has some desirable properties like for instance scalability, failure resistance, self administrating and no need for extra hardware our choice fell on a decentralized architecture known as peer-to-peer (requirement 2).

The centralized client-server solution has several advantages. By caching all information in one place, the searches become very fast and correct. The search coverage is excellent because all searching is performed at a centralized server. This is not true in an unstructured peer-to-peer network. So the trade-off present here is to exchange search precision, speed and better coverage for failure resistance, self administration and better utilization of available resources. To be able to satisfy our WRS clients the p-SARS system needs fairly good search coverage so that a client can find some of the publishers out there, but it’s not critical to the system if we cannot find them all. Therefore we are willing to make this trade-off.

The reason we didn’t choose the distributed hash table (DHT) approach is twofold. Firstly it’s still an open question if it’s possible to layer pattern search on top of DHT and secondly it’s not fully understood how the joining and leaving of nodes in a DHT system affects the scalability [H. Balakrishnan et al. 2003].

If we view our system as a completely isolated system, our architecture is unstructured peer-to-peer. Therefore every node has the same responsibilities and capabilities. The bigger picture is different though. The nodes in the p-SARS system will extend a sub-set of the WRS clients with search capabilities. These enhanced nodes are in effect super nodes. The reason not every WRS node is incorporated into the p-SARS overlay network is scalability. Only the nodes capable of being a part of

[4] The membership mechanism, the topic update mechanism, the gossip mechanism and the search mechanism.

the distributed search engine should participate and only as many nodes as needed. If e.g. a super-node is capable of supporting one thousand nodes only one of a thousand WRS nodes should participate in the p-SARS overlay network. This will lead to better scalability and ultimately better search coverage because we have reduced the size of the overlay network with a factor of one thousand. As an example if we have a network of ten million nodes only ten thousand WRS nodes should participate in the p-SARS overlay network.

It may have been possible to enhance our system's scalability and efficiency by introducing super-nodes with special responsibilities and capabilities. We could e.g. make a sub-set of the p-SARS nodes maintain a more precise routing table. This approach is what we in the theoretical framework chapter referred to as structured peer-to-peer and this architecture also represents the current trend in file sharing applications. It is already realized in e.g. KaZaA [KaZaA] and Morpheus [Morpheus]. We have however chosen not to incorporate this into our design because the WRS system already has chosen the nodes best capable of running as super-nodes. The potential gain here is therefore probably so little that we decided it was not worth the added complexity.

5.2 The interaction between p-SARS and WRS

The WAIF Recommender System (WRS) is itself responsible for the local topic graph connecting subscribers and publishers. Our system only comes into play when a client connected to a WRS super-node tries to extend its local graph with new publishers not already in its local graph. It's important to notice here that a WRS super-node also acts as an ordinary client.

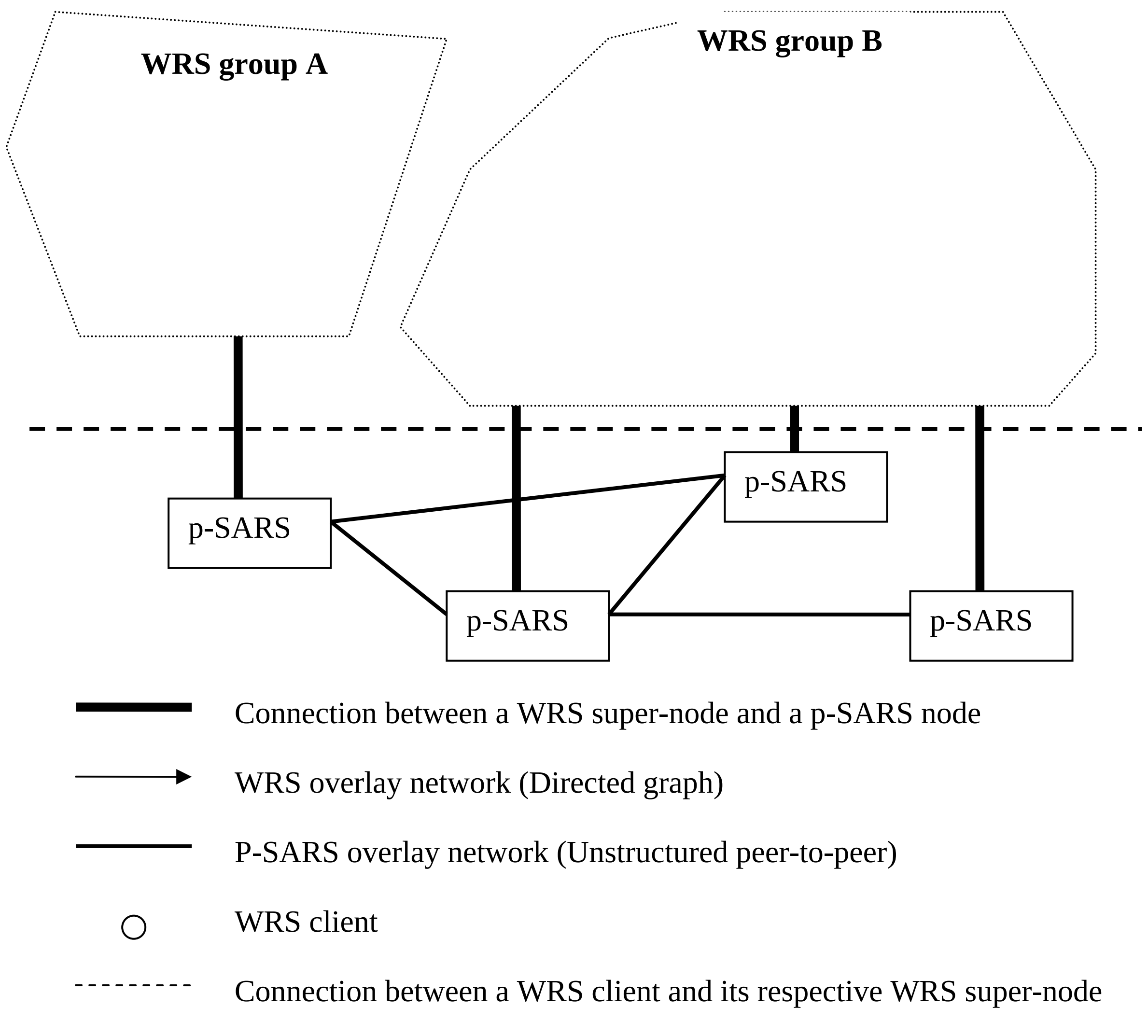

WRS group A
WRS group B
p-SARS
p-SARS
p-SARS
p-SARS
Connection between a WRS super-node and a p-SARS node
WRS overlay network (Directed graph)
P-SARS overlay network (Unstructured peer-to-peer)
WRS client
Connection between a WRS client and its respective WRS super-node

Figure 7 - Interaction between p-SARS and WRS

As shown in figure 7 both the p-SARS system and the WRS system create overlay networks. These two overlay networks are completely distinct and separated. While the WRS overlay network is a directed graph between publishers and subscribers (WRS clients), the p-SARS overlay network is an unstructured peer-to-peer network between the p-SARS nodes. The directed graph in the WRS overlay network describes which clients are publishing, and to whom they publish. The links in the p-SARS overlay network describes which nodes a node considers to be in its neighbour set.

Each WRS super-node is extended with a p-SARS node that provides a search mechanism to this WRS super-node and all of its WRS clients (requirement 4). As an example we can view the two WRS groups formed in figure 7. Although the WRS clients in group A are not connected to any WRS clients in group B in the WRS overlay network, they are indirectly connected through the p-SARS overlay network. Therefore if a client in group A issues a query to find a publisher the client will find this publisher even if he/she is in group B. The WRS overlay network will then be updated and the WRS groups will be joined to one group by this newly found connection.

The WRS system decides at run-time which of its nodes should run as super-nodes. It’s also important to notice that a WRS client is connected to only one WRS super-node at a time in contrast to Gnutella 2 where each client may be connected to several super-nodes. This approach is taken in Gnutella 2 as a safety precaution to minimize the impact a badly functioning super-node will have on a client.

5.3 The main system mechanisms

This section describes the design of the systems main mechanisms. We will start with the membership mechanism.

5.3.1 The membership mechanism

An important aspect of our design is the *neighbour set* which is situated and updated locally on each node in the p-SARS overlay net. For the system to be able to perform any search and gossip activity it needs to connect to other nodes (requirement 1 and 10). A set of nodes are therefore kept in what we call the neighbour set, and continuously pinged to ensure that they are indeed alive and ready to process queries on our behalf.

A neighbour is pinged if nothing is heard from it within a defined time period. If a neighbour dies or we after bootstrap discover that we don’t have enough neighbours, defined by a constant, the system must try different things to find more neighbours. One solution to find additional neighbours when already connected to the overlay network is to extract them from the events propagating through the node. The search events even hold a list of visited nodes, nodes that could be seen as potential neighbours. This is not yet incorporated into our design, with the exception of extracting neighbours from the ping and pong events received.

The designed mechanism works as follows:

- Nodes are pinged, and those that do not answer with a pong event within a specified time interval are removed. If a pong event is received, but it arrives too late, the node is re-inserted into the set only if there’s still room. All nodes presumed dead are also removed. A node is presumed dead when the sending of events to this node fails.

- When a node X pings a second node Y, node Y will include node X in its neighbour set if there is still room. This means that a node X may have node Y in its local neighbour set, but node Y need not consider node X as its own neighbour if it already has the maximum defined number of neighbours in its local neighbour set.

- Periodically the system module will contact the bootstrap server trying to fetch new neighbours, but only if the neighbour set is not yet full.

The bootstrap service is designed as a centralized server. This is due to the known problem of name resolving. We need to know which node to contact, before we can even try to establish a link. The alternative is to search for a node throughout the whole Internet by trying to connect to each one until a member of the overlay network is found. This is clearly not the solution. We could try to search the LAN, but there is no guarantee that someone else on our LAN actually runs WRS then we’ll probably again end up searching the whole Internet. So like many other applications we rely on a centralized service to help us with the initial bootstrapping of our system.

5.3.2 The topic update mechanism

The *topic set* describes which topics each client of the WAIF Recommender System (WRS) publishes on. To facilitate searching each p-SARS node keep a cached copy of this set. Each WRS super-node therefore holds an updated version of the complete topic set of its local clients and offers this set to the corresponding p-SARS node.

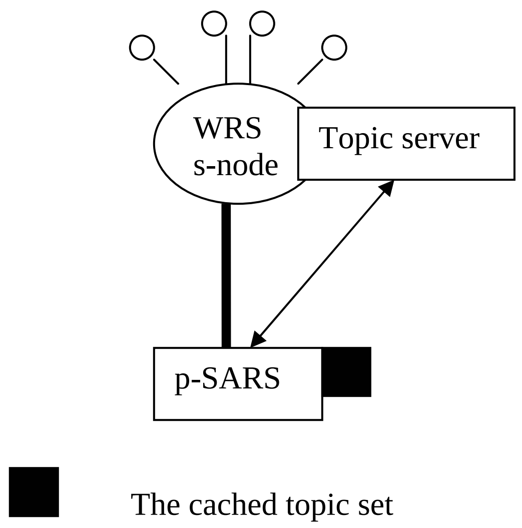


Figure 8 - The origin of the Topic Set

To be able to return any search result to a client of our search mechanism p-SARS, the topic set is essential (Requirement 7). We could fetch the topics from the topic server each time a query came in to ensure that the search results always reflected the real world. We have chosen to cache the data though for two reasons. First it’s not expected that the topics, which essentially describes the interests of a specific human, will change rapidly. Secondly it’s much faster to use a locally cached topic set than to pull the topic server each time we need to process a query. We do have to update the topic set periodical though to prevent that our topic set, in time, will contain only stale data (Requirement 8).

As a byproduct of the update operation we also discover what has changed since last update. If a new topic not already registered is discovered, or if the last node that published on a topic has left, we initiate a gossip (Requirement 5). Gossips are used to build and maintain the hint sets, essentially a nodes local routing table.

The gossip events are important to both speedup the searches and to make them more effective. Next we’ll describe the gossip mechanism in detail.

5.3.3 The gossip mechanism

The goal of the gossip mechanism is to build up what we call a *hint set*, locally on each node (Requirement 6). This set is in fact a routing table that is used to route queries to super-nodes that *may* have clients publishing on a given topic. The emphasis on *may* is very important. When searching is performed we don't really know if the hint is still valid, but the theory is that it's better to occasionally discover a stale hint than always perform a blind search. This means that the routing table don't necessarily contain correct information.

If a matching entry cannot be found in the routing table, the search mechanism degenerates to Random Walk. That is, a random neighbour is fetched and the query is relayed to it. Although the system will function without a hint-set, it becomes more effective if we have a populated one.

Topic	**p-SARS nodes**
Fishing	(128.0.0.1),(128.0.0.2), (128.0.0.3)
Basketball	(128.0.0.3)
......	
Peer-to-Peer	(129.0.0.3),(128.0.0.1)

Figure 9 - Example hint-set / routing table

As we can see in figure 9 the hint-set contains the topics and a list of the p-SARS nodes that may know of a WAIF Recommender System (WRS) client publishing on this topic. The topic set has two important properties. Firstly every unique topic has only one entry with a corresponding list of hints. Secondly each hint points to another p-SARS node where several or none hits matching the query may be discovered. That is, if a p-SARS node should know of more than one WRS client publishing on the same topic it will still be present with only one entry in the hint set. So as an example the p-SARS node with IP 128.0.0.1 could know of more than one client publishing on the topic fishing. This information is recorded in the local topic sets on the p-SARS node with IP 128.0.0.1 and will not be discovered before the query is relayed to this p-SARS node. If we are unlucky though the hint is stale and no hits matching the query are found on node 128.0.0.1.

To prohibit that joining nodes will start out with an empty hint cache they requests the hint sets of their neighbours. This way we manage to utilize the knowledge our neighbours already have accumulated to populate the hint sets of joining nodes.

The designed gossip mechanism works as follows:

- *Gossip start:* When profiles are updated on a node in the p-SARS system two lists are returned. One list contains new topics discovered locally on this node, and the other list contains previously known local topics that have vanished as a consequence of the update. A gossip event is then created. The event contains the two lists received from the update and in addition the hint set of this node. Each gossip event also has a time to live (TTL) value. The entire neighbour set is then traversed and for each neighbour the gossip event is sent to this neighbour with a defined probability.

- *Gossip propagation:* When a node receives a gossip event, it fetches the data, updates the hint-set and decrements the TTL on the gossip event. If the TTL still is positive the entire neighbour set is traversed and for each neighbour the gossip event is sent to this neighbour with a defined probability.

- *Gossip termination:* The gossip event terminates when the TTL reaches zero, or the gossip event becomes discarded. If the TTL is positive the event is propagated from a specific node to at least one of its neighbours with a probability of:
  $P(X>=1) = 1 - P(X=0) = 1 - \binom{n}{0} p^0 (1-p)^n = 1 - (1-p)^n$
  Where p is the defined probability of gossiping to the individual neighbour and n is the number of neighbours.

5.3.4 The Search mechanism

The main task of our system is to enhance the WAIF Recommender System (WRS) with a search capability, so that clients running WRS is able to search for new publishers on a given topic. We therefore need to design a search mechanism.

One solution presented in the theoretical framework chapter was the flooding algorithm. Although it is widely known that this algorithm does not scale [J. Ritter. 2001], it actually has one desired property. It is highly responsive [Q. Lv et al. 2002]. This property is a result of the flooding itself since many nodes receives and process the query in parallel. Another solution to the search problem, also presented in the theoretical framework chapter, is Random Walks. It has been shown that Random Walks are scalable and has nearly as good coverage as the flooding algorithm but that it’s far less responsive. This is however a trade-off we are more than willing to make for better scalability. The clients of the WRS system aren’t in need of a highly responsive search mechanism anyway. They are looking for a potential long time relationship with publishers and a few extra seconds waiting for the search results aren’t that important as long as the results from the search mechanism are of equal quality. We have therefore chosen the Random Walk approach supplemented with a hint cache. The hint cache contains information about other super-nodes that may know of publishers on the topic in question. If no hints are found we perform Random Walk.

In an unstructured peer-to-peer overlay network there is no way to predict accurately when a search result is returned or if one will be returned at all. To offer a more reliably service to the WRS system we have added what we call *pending searches* to our design. The p-SARS node that is local to the querying WRS client creates the pending search when it first receives the query. The pending search mechanism has one responsibility. Make sure the querying WRS client receive the answer for its query when it's satisfied or when the query times out[5]. The timeout value is defined by a constant and all search results received after the pending search has timed out is discarded.

The designed search mechanism basically follows three vital steps on each node it visits:

- The first p-SARS node receiving the query from a WRS client will be reckoned as the querying node in the p-SARS system. This node will create a pending search for this query.

- The node then examines its own topic set to see if it can satisfy the query locally. Each query has a "happy" value that describes how many hits the query needs before it's satisfied. If some results are found they are sent directly back to the querying node where the results are en-queued on the pending search. The query itself is only relayed to another node if both the time to live (TTL) and happy value is still positive. The TTL is decremented by one on each node it visits, while the happy value is decremented with the number of hits discovered.

- If the query is not yet satisfied and the TTL is still positive we try two approaches:

    - First we see if we have some hints about other p-SARS nodes that may know of publishers on the topic in question. If a hint is found the query is relayed to the p-SARS node registered on that hint. Because every search event carries with it a list of the nodes visited we are able to remove all hints already visited by the query. If more than one hint remains we choose randomly among them.

    - If the first approach fails, that is if we don't find any hints, we perform a random walk. A neighbour not already visited has precedence over a neighbour already visited just like when we searched for a hint. If we do not find any neighbours not already visited, one is picked at random from all of them. The reason for this is that even though the query has visited all the neighbours, it may be that some of the neighbours of this node themselves has neighbours not visited and so on. The query is thus always relayed to another node as long as the TTL and happy value still is positive. There is one exception though; if a p-SARS node becomes disconnected and no longer has neighbours it cannot relay the query.

[5] An answer will be returned even if it contains no search results.

5.4 Summary

p-SARS is designed as an enhancement to the WAIF Recommender System (WRS). The main goal is to provide the WRS system with a search service so that WRS clients can search for publishers publishing on a specific topic. To do this we have chosen unstructured peer-to-peer as the overall architecture. The reasons for this are scalability, failure resistance, self administrating and utilizing the resources available in the leaf nodes of the Internet[6].

Our system consists of four main mechanisms:

1. *The membership mechanism.* This mechanism makes sure that a p-SARS node has a populated neighbour set and that the neighbours present in this set are up and running. Nodes presumed dead are removed. The neighbours are used to process queries on our behalf, but they are also needed in the gossip mechanism.

2. *The topic update mechanism.* The topic set describes which topics the local clients of a WAIF Recommender System (WRS) super-node publish on. To facilitate searching a p-SARS node keep a cached copy of this set. This mechanism is therefore needed to periodic update this cached data to prevent it from becoming stale. As a byproduct of the periodical update we learn if a new topic is discovered or if a topic is removed. This byproduct is used by the gossip mechanism.

3. *The gossip mechanism.* The hint set is a p-SARS node's routing table and is used by the search mechanism to route queries to other p-SARS nodes that may know of publishers on the topic in question. The task of our gossip mechanism is to build up this hint set locally on each p-SARS node. This is done by disseminating a nodes local view as hints to other p-SARS nodes. The local view of a node is the node's locally cached topic set and the node's hint set.

4. *The search mechanism.* We have chosen the Random Walk approach supplemented with a hint cache. The approach taken when a query is received can be divided into these steps:

    a. Perform local search
    b. If not satisfied and time to live still positive propagate query:
        i. to a node based on a hint if one not already visited is found
        ii. to a node chosen randomly from the neighbours. Nodes not already visited have precedence over already visited nodes.
    c. Return results to the querying p-SARS node if any is found in step a.

[6] The Internet is the underlying physical network of the p-SARS system.

Chapter 6

Implementation

In this chapter we will describe our implementation of the p-SARS system, but for a complete understanding of the system we will also refer to the full source code listing available in appendix A of this thesis. We start out with a short description of the system as a whole.

6.1 System overview

The system itself is divided into separate modules and the information between modules flow through queues. The queues are multi-producer, multi-consumer FIFO[7] queues and when a process or thread tries to fetch an element from one of these queues, it'll be suspended until an element arrives if not specified otherwise. The reason for dividing the system into modules is twofold. First it divides the work into manageable pieces, and second it makes changes easier. We don't have to re-implement the whole system, just the module we want to change. All modules depict in figure 10 are implemented as threads and all event-passing inside and into the p-SARS system is asynchronous.

Each module fetches an event from a queue and then processes this event, except for the listen and update topic modules. These modules receive marshalled events over a TCP/IP connection. When a module is finished processing the event it may enqueue a response onto another event queue before it again awaits a new event. The pusher module does not however enqueue the event on a queue but marshals this event and sends it to its destination through TCP/IP. All events flowing trough the system is described in detail in appendix C of this thesis.

The system is divided into nine modules as shown in figure 10. In the following sections we'll describe the processing modules in detail, but to understand the system as a whole we'll start out with a high-level walk through of the entire system.

The listen module is the interface between the different p-SARS nodes and the WAIF Recommender System (WRS). When it receives an event it unmarshall it and inserts it into the input queue, and then continues listening. The event dispatcher waits on the input-queue until an event is ready. It then identifies the event and en-queues the event on the right event queue. We have divided the system so that we have four different events, and therefore four different modules processing their corresponding events. When the event processing module is finished processing its event, it decides which response to send, if any, and en-queues this response on the output-queue.

[7] FIFO is an acronym for First In First Out.

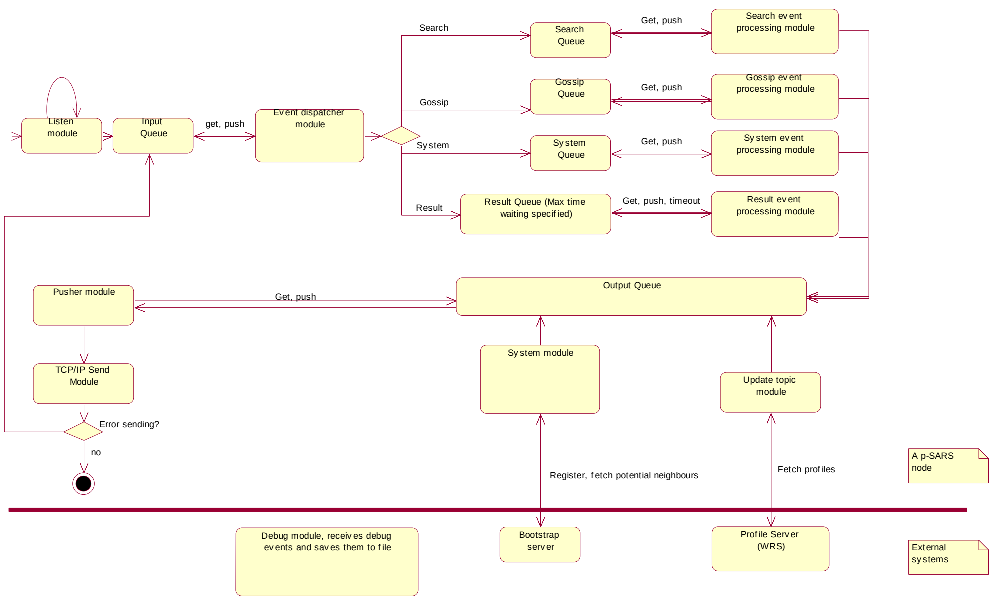


Figure 10 - System overview (The design of a p-SARS node)

The pusher module fetches outgoing events and starts the right module for sending the event to its destination. There are two reasons for starting a separate module for sending. The first is that the sending may take a while, and the second is that it may be desirable to enhance the system later and e.g. support both XML-RPC and TCP/IP. Currently all communication goes through the TCP/IP protocol and this is implemented in the TCP/IP send module. The TCP/IP send module only tries to send the event to its destination a defined number of times. If the sending fails the event is enqueued on the node's input queue with its error field set to true. It's then up to the event's corresponding event processing module which action to take, if any.

The system module communicates with other p-SARS nodes, through events, to maintain the peer-to-peer overlay network. This will be described in the System module section, but briefly this implies managing bootstrap and neighbour set membership. Topic set updates, which are the updating of the topics WRS clients publishes on, are handled by the update topic module. This module pulls information from a WRS topic server and updates the locally cached topic set.

We have also implemented a debug module. This module receives debug events and creates a log. Because there are many nodes running concurrently this will help debugging, and tests can also be run on the gathered data itself to see if the system function correctly.

6.2 The system module

If we take a look at the activity diagram in figure 11 we see that first the system module performs bootstrap by contacting the centralized bootstrap server. The node registers itself at the server and in return gets a list of nodes already connected to the p-SARS overlay network. Neighbours are then extracted from this list. The system module then sends out requests asking for the hint sets of its chosen neighbours, so that this node may populate its own hint set. Since this module is the first one started it also has to initiate and start up other modules. This includes starting up the topic update module and the event dispatching module. When the event dispatching module is started we also indirectly start the listen module, the pusher module and the four event processing modules depict in figure 10.

The event dispatching module investigates which type of event is received from the listen module and sends the event towards the right event processing module by enqueuing the event on the right event queue. We will describe the topic update module in its own section below.

The system module is designed and implemented so that after the initial work just mentioned, it sleeps for a defined number of seconds before it repeats some work which we have chosen to call *system update*. Three main tasks are repeated on each system update:

- Traverse the neighbours to see if any have a ping event pending. If a neighbour has a pending ping event this means that it has not returned a pong event since last system update. Since no pong event is received, the node is presumed dead and removed from the neighbour set. All hints associated with this neighbour are also removed.

- Traverse neighbours again, but this time test to see if it’s time to ping any of the neighbours. Each neighbour has a corresponding timestamp that tells the system how long it’s since last ping. When it’s time to ping a neighbour, a ping event is created and en-queued on the output queue.

- The last task of the system module is to help maintain the size of the neighbour set. If there’s not enough neighbours connected, defined by a constant, the system contacts the bootstrap server in search for new neighbours to add. The node has already registered itself at the bootstrap server so this is not repeated.

This module cooperates closely with the system event module. The system event module, which we discuss next, process and create responses to system events.

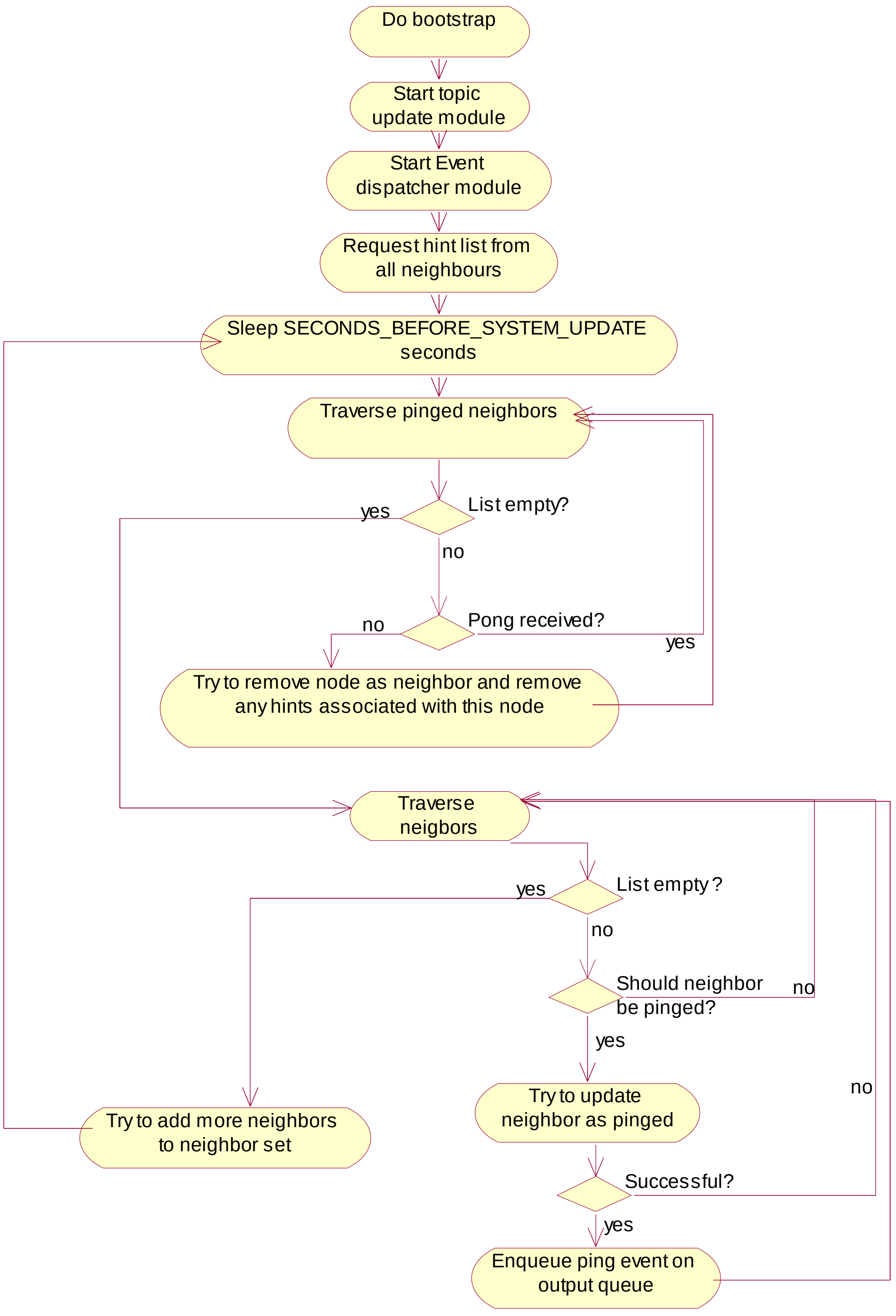

Do bootstrap
Start topic update module
Start Event dispatcher module
Request hint list from all neighbours
Sleep SECONDS_BEFORE_SYSTEM_UPDATE seconds
Traverse pinged neighbors
List empty?
yes
no
Pong received?
no
yes
Try to remove node as neighbor and remove any hints associated with this node
Traverse neigbors
List empty ?
yes
no
Should neighbor be pinged?
no
yes
Try to update neighbor as pinged
Try to add more neighbors to neighbor set
no
Successful?
yes
Enqueue ping event on output queue

Figure 11 - The system module

6.3 The system event module

The main task of this module is to take appropriate action when a node receives ping and pong events. We will start out our description with the activity diagram presented in figure 12.

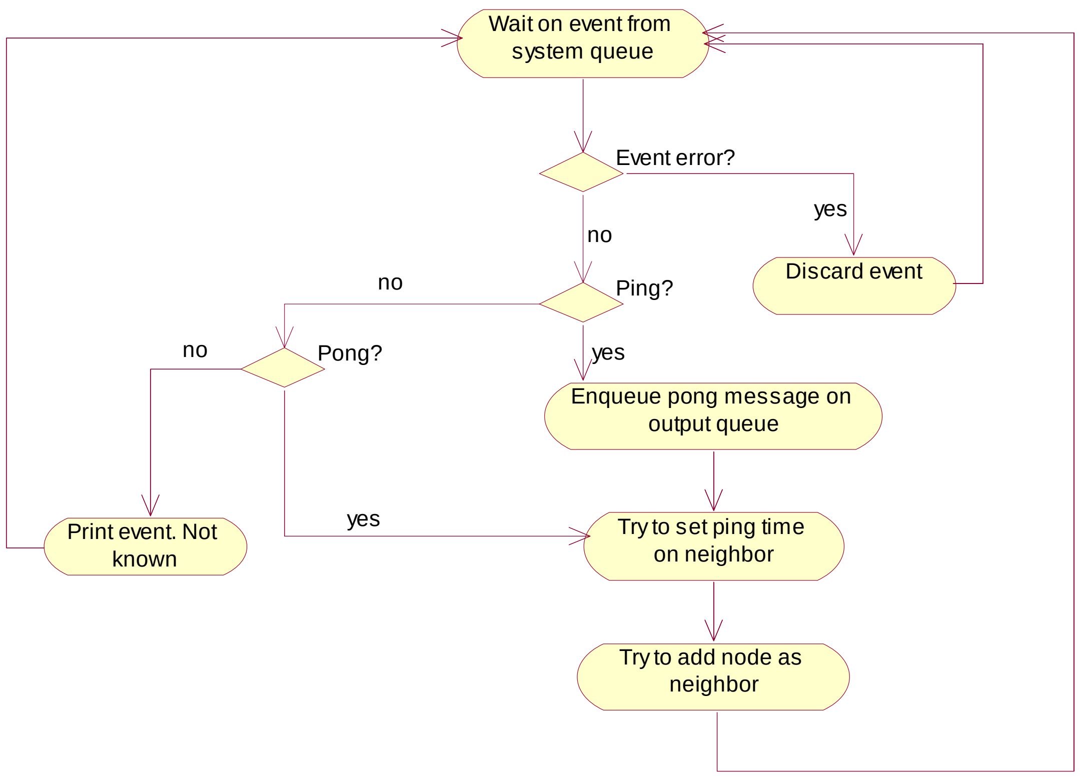

Wait on event from system queue
Event error?
yes
Discard event
no
Ping?
no
Pong?
no
yes
Enqueue pong message on output queue
yes
Print event. Not known
Try to set ping time on neighbor
Try to add node as neighbor

Figure 12 – Processing the system event

When this module receives an event it goes through some tests to identify which type of event it is before the appropriate action is taken:

- *Event error*: If it is an error event it means that the system has tried to send a ping or a pong event but has not succeeded. When an event can't be sent to the addressed node, this node is presumed down and usually deleted from the neighbour set. But because the node will be deleted anyway on the next system update, for now, no action is taken here. The reason we have implemented this feature is that all events that the pusher module fails to send out always will be returned to the right processing node so that an appropriate action can be taken. It may be that we in the future will use this response in a constructive way and not just discard it as we do now.

- *Ping event*: If the event is a ping event a pong event is created and enqueued on the output queue. Then we treat the ping event as a pong event and reset the ping-time of this node but only if the pinging node is already in our neighbour set. If the neighbour set is not full, the last action taken on the ping event is to add the pinging node to the neighbour set if it's not already present. This way we extract neighbours from ping events.

- *Pong event*: If a pong event is received, the module tries to reset the recorded ping-time. This may not succeed though, because the event may be too late and the neighbour already removed. If the event is too late we reinsert the pinging node in the neighbour set if it's not already full. The thought here is that it's better to have a potentially saturated neighbour than potentially none at all.

6.4 The topic update module

This module performs periodical updates of the locally cached topic set to ensure that potentially stale data is updated. The activity diagram for this module is presented in figure 13.

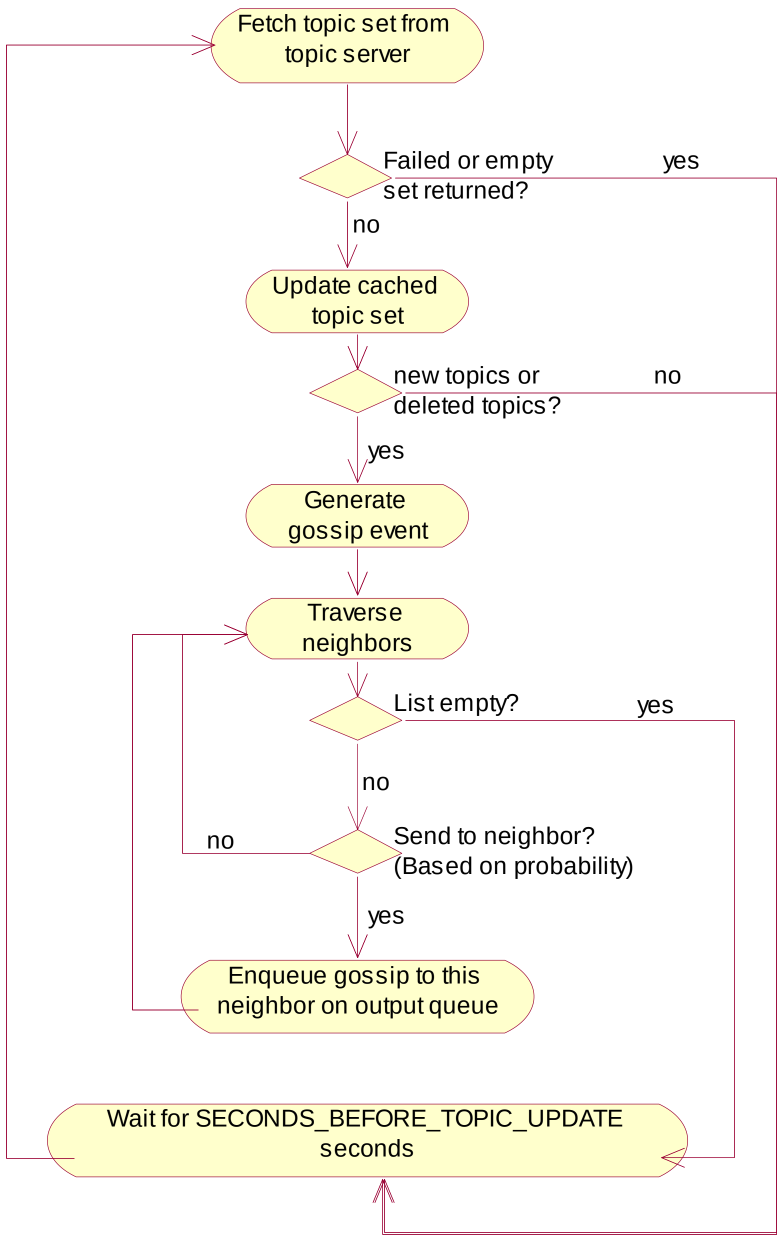

Fetch topic set from topic server
Failed or empty set returned?
yes
no
Update cached topic set
new topics or deleted topics?
no
yes
Generate gossip event
Traverse neighbors
List empty?
yes
no
Send to neighbor? (Based on probability)
no
yes
Enqueue gossip to this neighbor on output queue
Wait for SECONDS_BEFORE_TOPIC_UPDATE seconds

Figure 13 – Topic update

The module has a do-while structure. This means that it's executed once before it's put to sleep for a defined number of seconds and then re-executed. The following takes place on each *topic update*:

- The module contacts a known WRS topic server to fetch the updated topic set. There is one topic server for each WRS super-node. Since the p-SARS node will extend the WRS super-node with search capabilities, it will most likely be situated on the same physical node as the WRS system's topic server. When the topic set is fetched the module updates the locally cached version. The update process returns two lists. The first list contains any new topics discovered and the second list contains any removed topics since last update.

- If the lists returned contain any information, that is either some new topics are added or some old topics are removed, a gossip event is created. Then the entire neighbour set is traversed and for each neighbour the gossip event is addressed and enqueued with a defined probability. The gossip event contains all information received from the update. Both topics removed and new topics are described in the same event. It also contains this nodes entire hint set.

6.5 The gossip event module

Gossips are initiated in the topic update module as described but also in the search module as will be described in the corresponding section below. This module handles the gossip propagation and termination. The activity diagram of this module is presented in figure 14.

When a gossip event is received it is processed sequentially in these steps:

- If the event is an error, this means that an outgoing gossip has failed to be sent. The neighbour is presumed dead, and its data is removed from both the neighbour set and the hint set. The algorithm then precedes to the last step.

- If the event is a gossip *get* event, we have a neighbour that requests our hint set. The response created is a gossip *push* event that contains both the hint set and the topic set of this node. This gossip event is only sent to the requesting neighbour and will not propagate further. Then the module again waits for a new gossip event.

- If the event received is a gossip *push* event, we have received the hint set and the topic set from one of our neighbours. The action taken is to update the node’s local hint set with the received information. Then the module again awaits a new gossip event.

- If this step is reached we have an ordinary gossip event. First the new topics, if any, are extracted and tried inserted into the hint set. The hints already known are just ignored. Then the deleted topics are traversed and the ones present in the hint set are removed. All gossip events also contain the hint set of the node initiating the gossip. We thus traverse this set and update the local hint set accordingly.

- The TTL is then decremented and if the resulting value equals zero, the event is discarded and the module awaits a new gossip event. In the case where the value is greater than zero the same action as in the update profile module is taken. The entire neighbour set is traversed and for each neighbour the gossip event is addressed and enqueued with a defined probability. When done, the module again awaits a new gossip event.

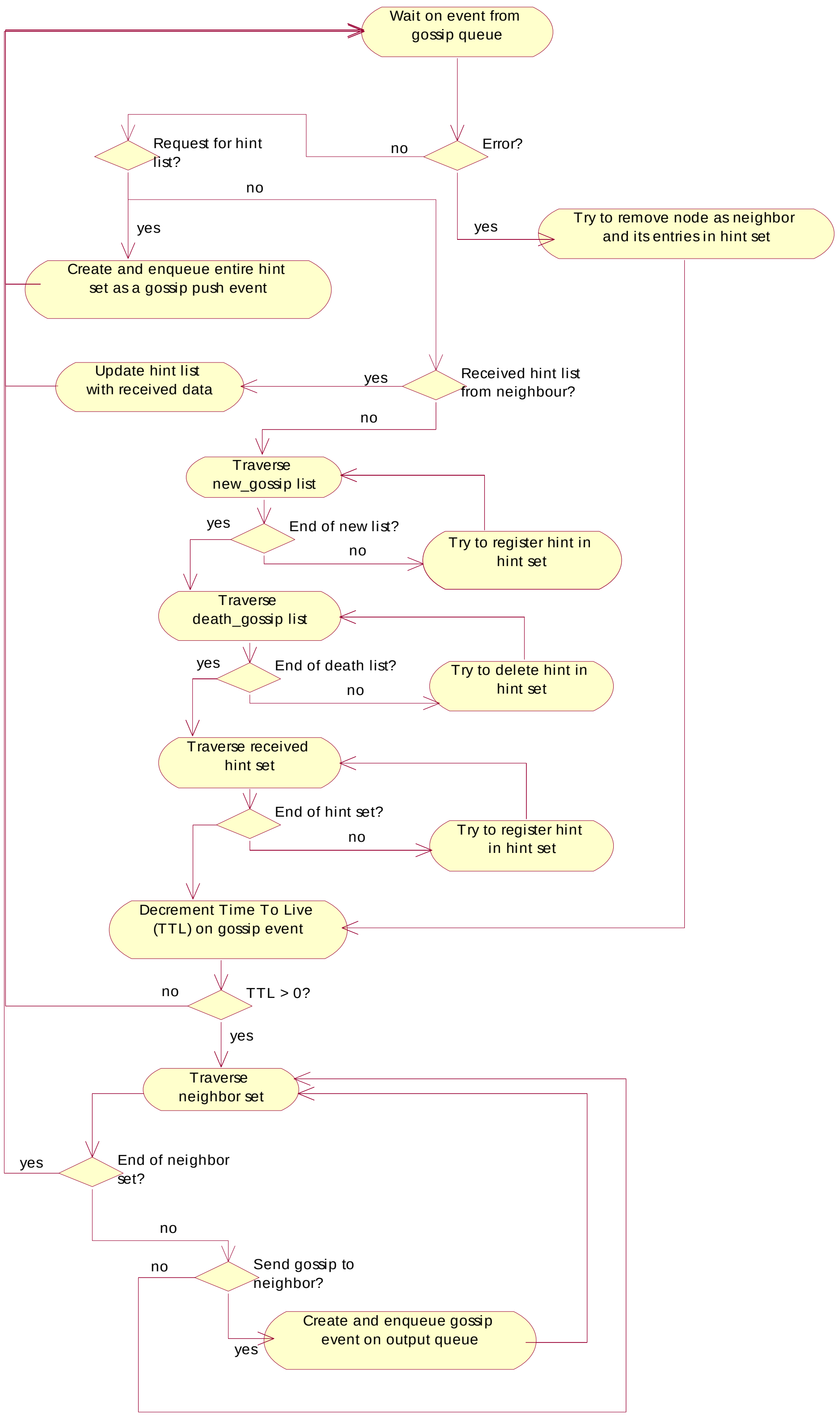

Wait on event from gossip queue
Error?
no
yes
Try to remove node as neighbor and its entries in hint set
Request for hint list?
no
yes
Create and enqueue entire hint set as a gossip push event
Received hint list from neighbour?
yes
Update hint list with received data
no
Traverse new_gossip list
End of new list?
yes
no
Try to register hint in hint set
Traverse death_gossip list
End of death list?
yes
no
Try to delete hint in hint set
Traverse received hint set
End of hint set?
no
Try to register hint in hint set
Decrement Time To Live (TTL) on gossip event
TTL > 0?
no
yes
Traverse neighbor set
End of neighbor set?
yes
no
Send gossip to neighbor?
no
yes
Create and enqueue gossip event on output queue

Figure 14 – Processing the gossip event

6.6 The search event module

This module implements the actual search mechanism. We start out as usual with the activity diagram of the module depict in figure 15.

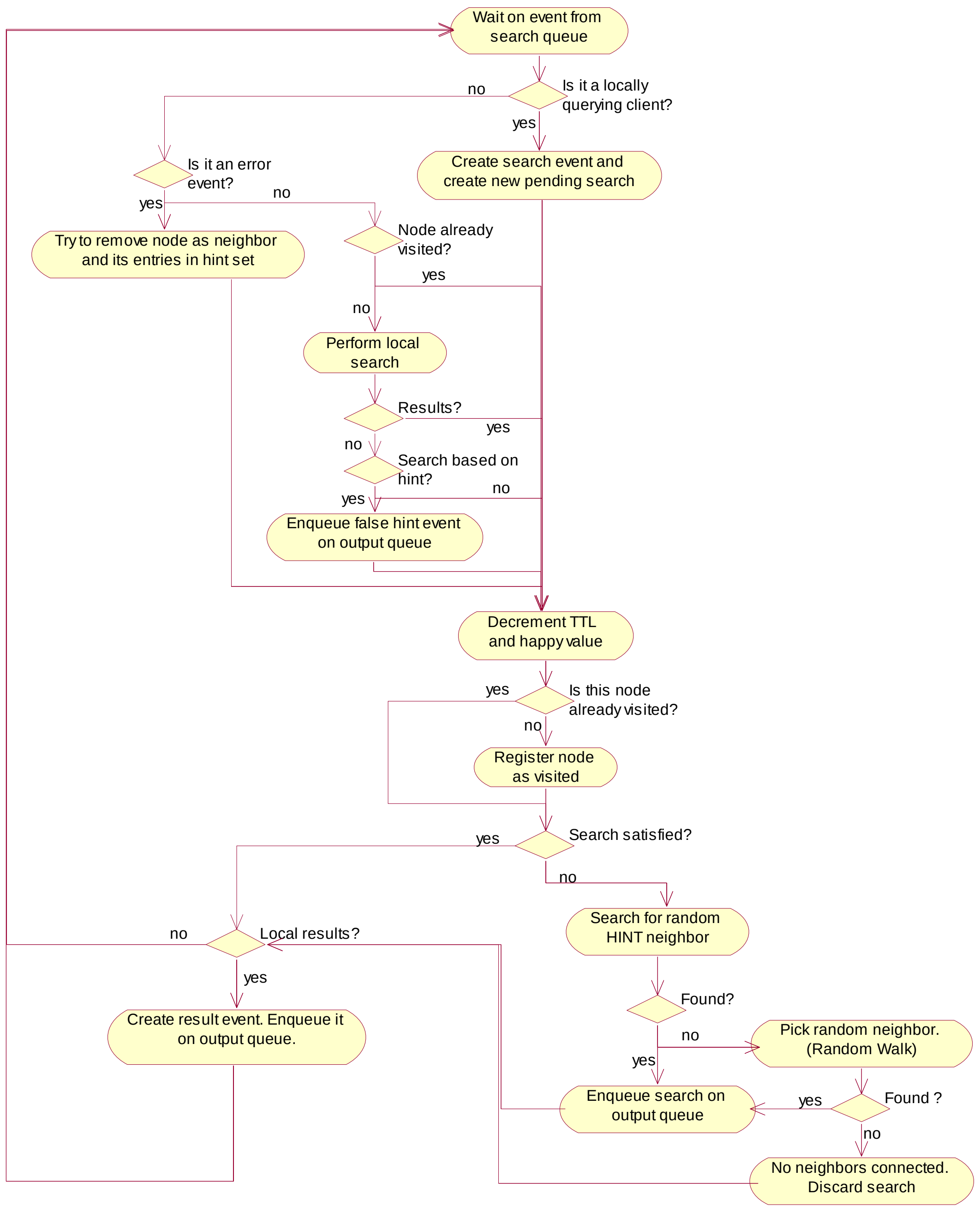

Wait on event from search queue
Is it a locally querying client?
no
yes
Create search event and create new pending search
Is it an error event?
yes
no
Try to remove node as neighbor and its entries in hint set
Node already visited?
yes
no
Perform local search
Results?
yes
no
Search based on hint?
no
yes
Enqueue false hint event on output queue
Decrement TTL and happy value
Is this node already visited?
yes
no
Register node as visited
Search satisfied?
yes
no
Search for random HINT neighbor
Local results?
no
yes
Create result event. Enqueue it on output queue.
Found?
no
yes
Pick random neighbor. (Random Walk)
Enqueue search on output queue
Found ?
yes
no
No neighbors connected. Discard search

Figure 15 – Processing the search event

The processing of the search event is designed and implemented as the following nine steps:

1. When a search event is received we first check if the querying node is local to the current WAIF Recommender System's (WRS) super node. This is the case if this super-node is the first to receive the query from a WRS client. If it's a local querying node we create a pending search, add some extra information to the search event and proceed to step 6. We shall not perform a local search when the query originates from a local client. The WRS system ensures that all WRS clients connected to the same super-node already knows of each other. This is accomplished when a WRS client registers to a WRS super-node. The registering client then receives all topics which the local WRS super-node's clients publish on. In addition a WRS client can pull the WRS super-node for this information if it is needed later. The extra information added to the search event is listed below.

    - The TTL is set to the defined value.
    - A visited list is created, so that the nodes visited by this query can be recorded.
    - The search event's error field is set to false.
    - The search event's hint field is set to false to indicate that this event is not relayed to this node based on a hint.
    - The p-SARS node where the search originated is recorded in the event's from field. We need this information because this is the node in charge of handling the search results for this query.

2. If it's not a local querying node we test if it's an error event. This means that the previous relaying of a search event has failed. When this happens we assume that the unreachable client is dead. The action taken is to remove it from the neighbour set, if present, and to remove any hints registered on this node. Then the processing proceeds to step 6.

3. If it's not an error event, we test if this node is already visited by this search. If this is the case there's no reason we should perform a local search once more on this node. Therefore the algorithm proceeds directly to step 6. If it is the first time we visit this node, we perform a local search through the topic list to see if any clients publishes on the topic in question. In our current implementation we have not implemented pattern search, only exact matches are defined as a hit.

4. In this step we test if the local search performed was successful, that is if the search found any objects that satisfied the query. If the search was successful we proceed to step 6, if not we proceed to the next step.

5. Since the local search was not successful we need to test if this search event was relayed to this node based on a hint. If the event was relayed to this node based on a hint the hint is clearly stale, because no items were found locally on this node. A hint-death event is therefore created and enqueued on the output queue addressed to the node were we received the query from. This

hint-death event is an ordinary gossip event; the only extra ordinary with it is that it is sent with a 100% probability to only this neighbour in the first step. The propagation from the second node and on is as explained in the gossip mechanism above. This gossip event also includes the node's entire hint set.

6. The TTL is decremented by one and the happy value is decremented with the number of hits found locally on this node. If it's the first time this node is visited by this search this node is recorded in the visited list.

7. If both the TTL and happy value still is grater than 0 we need to relay the search to another node. If this is not the case we proceed to step 9.

8. First we search for hints in the hint set to find nodes not already visited that may have local clients publishing on the topic in question. If a hint is found the search is relayed to the node registered on this hint. If more than one node is found we chose randomly among them. If no hints are found, or they are all already visited by this query, we perform random walk. A node is fetched at random among the neighbours not already visited. If all neighbours are visited, we chose randomly among them all. The query is then relayed to the chosen neighbour. In the case where the node is completely disconnected and has no neighbours, the search event is discarded.

9. If some results where found during the search through the topic set of this node, we create a result event. This event is addressed to the original querying p-SARS node and enqueued on the output queue.

6.7 The result event module

The search event module, described above, is in charge of where the searches themselves propagate and how they are processed on each node. When it comes to the result handling though, the result event module is in charge. The activity diagram of this module is presented in figure 16.

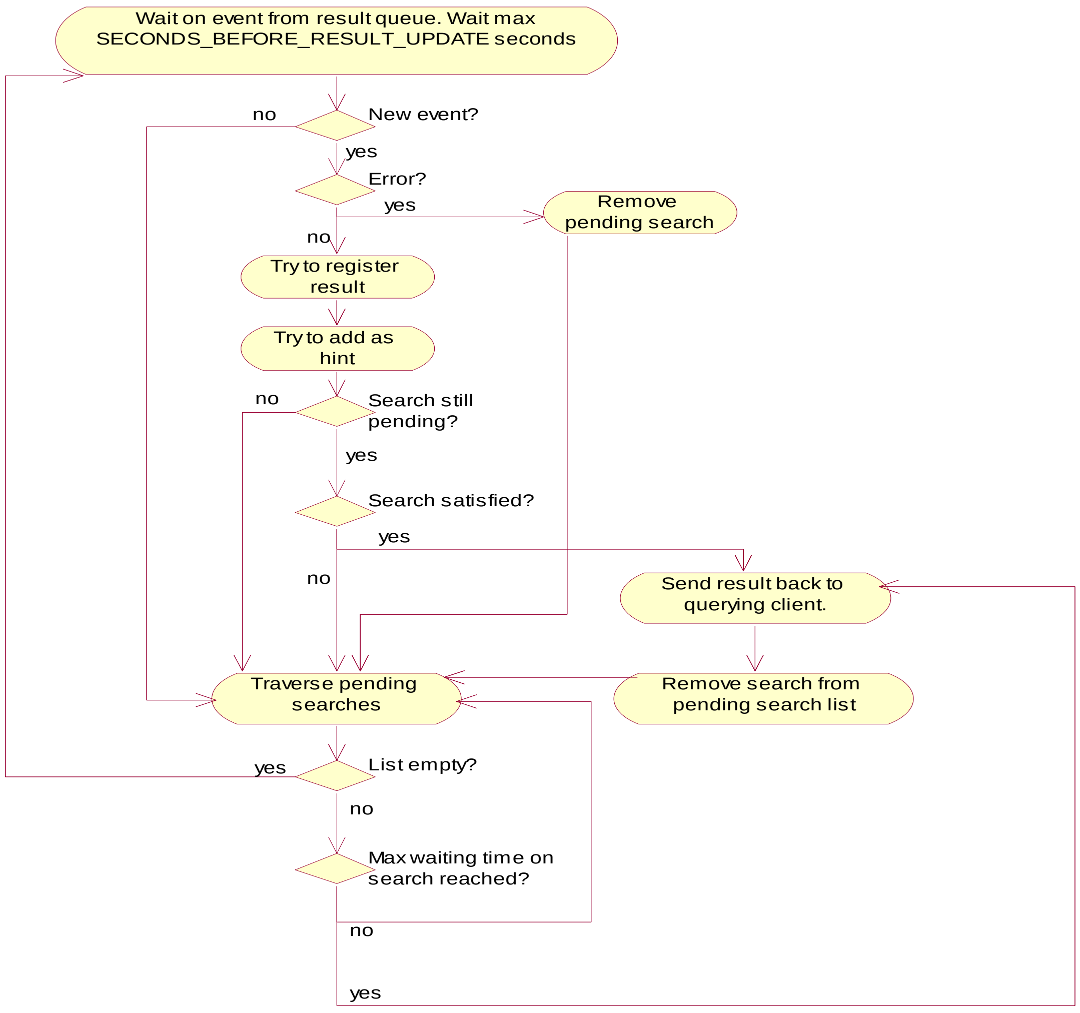

Wait on event from result queue. Wait max SECONDS_BEFORE_RESULT_UPDATE seconds
no
New event?
yes
Error?
yes
Remove pending search
no
Try to register result
Try to add as hint
no
Search still pending?
yes
Search satisfied?
yes
no
Send result back to querying client.
Traverse pending searches
Remove search from pending search list
yes
List empty?
no
Max waiting time on search reached?
no
yes

Figure 16 – Processing the result event

When an event is received or we have waited a defined number of seconds the following steps are taken:

1. If it is a timeout on the event queue we proceed to step 5.

2. If it's an error event, we try to remove the pending search described in the event and proceed to step 5. An error event is received here when the result

couldn’t be delivered to the querying WRS client or the originating p-SARS super-node. Regardless of the cause the search results are discarded.[8]

3. If this step is reached we have received results from a p-SARS node processing a query on our behalf. First we register the results on the corresponding pending search, if it’s still pending, and then we extract hints from the results. The hints are added to the hint list if they’re not already known. If the search is not pending the results are discarded and we proceed to step 5.

4. We test if the pending search now is satisfied. If it is satisfied, send results back to the querying WAIF Recommender System (WRS) client and remove the search as pending.

5. Traverse all pending searches to see if any of them has timed out. If any has, remove them from the list of pending searches and return the result, empty or not, back to the clients.

[8] We have already extracted hints from the search results before we tried to relay the result event the first time.

6.8 Summary

The system is divided and implemented as independent modules. Each module presented is implemented as a thread. This is to ease system maintenance and if we do have to make minor changes to the design we do not have to re-implement the whole system only the module influenced by the change. There is one exception though, if the format on the events propagating through the system are changed all modules processing these events must be changed.

Our system is mostly event based. That is, almost every module awaits an event and when one appears it is processed and a response, if any, is sent onwards to the right module. All events within a p-SARS node flow through queues. The queues are implemented as multi-producer, multi-consumer FIFO queues and when a module tries to fetch an event from a queue it is suspended until an event arrives if not specified otherwise. All communication into and out of a p-SARS node is sent over the TCP/IP protocol.

Chapter 7

The Simulator

To shield us from the tedious work of setting up a real network we decided to design and implement a simulator. Another reason is that we actually don't have the resources in this project to set up a potentially large test-bed to gather the needed test results. The simulator essentially has two purposes; to discover how many messages are sent in the network and the success rate of our search mechanism. However, the simulator is not in any way used for timing tests.

The simulation results will reflect the real system behaviour and let us investigate issues like:

- Will the p-SARS system scale?
- Is the gossip mechanism effective?
- Is the search mechanism effective?
- How the system will react if we change some of the constants like e.g. TTL-search, TTL-gossip and the probability for gossiping itself. How about the number of neighbours each super-node has? Should we extract routing information from search results?

In this chapter we will first explain how we built our simulator, and then we will describe the different tuning capabilities. The results from the simulations will be presented and discussed in chapter 8, Testing. The reader may refer to appendix B for a full source code listing of our simulator.

7.1 Design / Implementation

The simulator, as the real p-SARS system, is built in its entirety in Python. The reason for this is that this high-level programming language saves us for much coding. We were not able to reuse much code from the p-SARS prototype though. The only code reused is the random functions used in both the simulator and p-SARS to fetch a random neighbour and to decide to whom, if any, we should gossip.

We have divided the simulator essentially in four stages. The first stage is to initialize the network; secondly we have the gossip stage; third we have the warm-up stage and last we have the search stage. We will discuss each stage separately in detail below, except for the warm-up stage which essentially is the same as the search stage. We will also discuss an alternative to our gossip mechanism which we have called Random Walk Gossip (RW-G).

7.1.1 Network initialization

The simulated network is held in its entirety in a list. Each element in this list is an object instance and represents a p-SARS node in the network. This object, called node_t from now on, holds three sets of data. The sets held are those already identified in the p-SARS design; the neighbour set, the topic set and the hint set. Each node is assigned an id, and this is what the simulator searches for. To see the relation to p-SARS we think of this id as a topic on which the node publishes.

When the simulator starts it creates a list of node_t instances and initializes the neighbour sets by calling each instance telling it how many neighbours it shall fetch. These neighbours are fetched at random from the list of nodes already created[9]. There are two ways for a node_t instance to fetch neighbours. Either we tell it to fetch exactly X neighbours or we tell it to fetch between 1 and X neighbours. The reason for this is to analyze what effect it has on the overlay network whether or not all neighbours are fully connected. A node is said to be fully connected if it has reached the maximum number of neighbours defined. When the network is initialized we start the gossiping stage.

7.1.2 Gossip

The gossip protocol is implemented, in our simulation, as a recursive depth first function. Based on the TTL of the gossip event and the probability defined, the gossip spreads in the network from the node where it is started just like it does in p-SARS. The gossip event contains only the node id of the node that started the gossip, and this id is saved in the hint set of nodes visited by the gossip. Just like the p-SARS system saves hints passing by in gossip events.

With a probability set to 100%, this algorithm works just like a flooding algorithm with no constraints. That is, a node can be visited more than once, and each node receiving a gossip event will relay this event to all of its neighbours as long as the TTL is grater than zero. The gossip algorithm implemented in both p-SARS and the simulator is presented in figure 17. The probability parameter in our implementation can take on a value between 0 and 100 %.

```
for neighbour in neighbour_set:
        if should_I_send_gossip(probability):
                send(gossip, neighbour)
```

Figure 17 - Pseudo code for the gossip algorithm

Like in the p-SARS system this gossip protocol is implemented to speed up the search algorithm. By populating the hint set on the individual nodes, searches can be directed to the right node if an entry in the hint set exists. If we decide to not do the gossip, the search algorithm will degenerate to plain random walk, making it possible to estimate the effect of the gossip mechanism.

[9] This may actually result in islands in the network which we will discuss below in the search section.

7.1.3 Random Walk Gossip (RW-G)

As another solution to populate the hint sets of our p-SARS nodes we propose Random Walk Gossip (RW-G). We use random walks as it is described in the theoretical framework chapter, but for gossiping instead. That is, a walker carries along with it a node's topic set and uses this set to populate the hint sets of the nodes it encounters on its walk.

7.1.4 Search

The search protocol is implemented, in our simulator, as a recursive function. If a hint is found in the hint set, the search is relayed to the right node. If no hint is found, we perform random walk. When relaying the search, neighbours not already visited have priority over neighbours already visited. This is just like in the p-SARS system, and will help us explore the nodes in our network more efficiently.

Because the graph generated most likely will contain islands we need to reduce the effect an isolated node will have on the simulation results. A search is therefore initiated from a random selected node and the node id we search for is also selected at random. This will not prevent that the existence of islands will affect our simulation results but it will make the extreme case less influential. The extreme case is when a node we either search for or the node where we initiated the search is situated on a small island disconnected from the rest of the network. Of course if the number of neighbours each node have is high it is less likely that islands will exist in our graph than if each node have very few neighbours.

To further enhance our searching protocol we implement the possibility for learning from future searches, just like it's done in p-SARS. Since it is a recursive function we also have the possibility to not only increase the knowledge of the querying node, but also on all nodes on the path to the item we search for by using tail recursion. The latter is not implemented in p-SARS. The reason for this is that in p-SARS we have chosen to send the results directly back to the querying node. Not back through the intermediate nodes on the search path. When we perform what we call a warm-up in our simulations we essentially just perform a defined number of searches with one or both of the learning capabilities turned on.

7.2 Performance tuning

There are several variables that can be tuned in the p-SARS system to achieve optimal performance. These variables are described in the OWN_constants file listed in appendix A. We have decided to implement some of them and a few extra as tuneable variables in our simulator. The extra parameters not available in the real p-SARS prototype are explicitly marked with an (X). These are the tuneable parameters in our simulator:

- Network generation
    - Number of nodes in the network.
    - (X) Whether or not all nodes should fetch exactly Y random neighbours or if they should fetch between 1 – Y random neighbours.
    - Maximum neighbours to fetch defined by the value Y.

- Gossip
    - (X) If gossip should be performed.
    - The TTL on a gossip event.
    - The gossip probability in percent.

- (X) Random Walk Gossip (RW-G)
    - If RW-G should be performed.
    - The TTL on the RW-G event.

- (X) Warm-up
    - Whether or not warm-up is wanted.
    - The TTL on warm-up searches.
    - Number of warm-up searches to perform.

- Search
    - (X) If search should be performed.
    - (X) Whether or not the searches should help populate the hint set.
        - Only the querying node extracts routing information from search results.
        - Both the querying node and all nodes on the path to the item in query extracts routing information from search results.
    - The TTL on searches.
    - (X) Number of searches to perform.

7.3 Describing the simulations

The simulator starts out reading in the test cases from a file called sim.txt. These test cases are then run sequentially. Because we want to limit the impact a badly generated network topology has on the simulation results each run starts with the reinitializing of the overlay network. We also reinitialize the hint set on each node to prevent that previously run tests will affect the test results.

Example input file (sim.txt)[10] containing one test case:

```
# Test 1
#1: Num_nodes
1000
#2: Num_neighbours
9
#3: Fetch neighbours at random?
1
#4: Should we gossip?
0
#5: TTL on gossip
7
#6: Gossip probability
30
#7: Should we perform search?
1
#8: TTL on search
500
#9: Number of searches to perform
1000
#10: Should the originating node learn from searches?
1
#11: Should all intermediate nodes also learn?
0
#12: Should we perform a warm-up?
0
#13: TTL on warm-up searches
500
#14: How many warm-up searches should we perform?
10000
#15: Should we perform Random Walk gossiping (RW-G)?
1
#16: TTL on RW-G
500
#
# Test 2 ...
```

We define a test tuple as the tuple of integers holding the different test parameters. The test tuple for the above test case would look like this:

(1000,9,1,0,7,30,1,500,1000,1,0,0,500,10000,1,500)

All results from the simulator are loged in the file output.txt but also displayed to screen.

[10] In the example input file we have that : 0 equals false and 1 equals true

7.4 Summary

The reason we decided to build a simulator was to shield us from the tedious work of setting up a potentially large test-bed to gather the needed test results of our system. Our simulator essentially has two purposes. By tuning different parameters we firstly want to discover how many messages are sent in the network and secondly we want to investigate the effectiveness of our search mechanism. It’s important to notice however that our simulator is not used for timing purposes.

Chapter 8

Testing

Our testing consists of two parts. First we perform both a conformance test and a throughput test on the p-SARS prototype. Secondly we simulate the system's behaviour using the simulator described in chapter 7 to i.a. investigate how effective our search mechanism is. In the end we present a discussion of the system's scalability.

It's important to notice that we will only present a summary of our results here. All simulation results gathered can be viewed as a whole in appendix D.

8.1 Testing the p-SARS prototype

During the implementation of our system we continuously performed white-box testing to ensure that new additions to the system functioned correctly. This was done by inserting constructed events into one node and observing the path the event took inside the node. We also performed various regression tests to ensure that when a new module was incorporated into our existing system it did not negatively affect the modules already present.

8.1.1 Parameters and factors

Before we start testing our prototype we have to identify the parameters that are not the object of our study, but may affect our test results. When these parameters have been identified it's important that we either eliminate their influence or hold them constant. If this is not done we can end up with test results that aren't correct and cannot be compared.

The parameters that we try to eliminate or keep constant during our tests are:

- *Hardware:* We hold this parameter constant by always running the tests on the same type of machines. For testing we used between one to six Hewlett Packard Kayak XU machines with the following important characteristics:
    - CPU
        - 2 x 300MHz Pentium II
    - Memory
        - 256 MB RAM
    - Network
        - 100 Mb/s FastEther

- *OS:* During our tests we hold this parameter constant by running all performance tests on the same type of operating system. In our case all the machines in the network run Red Hat 9 with the 2.4.20-24.9smp kernel.

- *Processor and network load:* Our experiments are performed in a multi-user environment. Therefore both the network and processor resources available to us will most likely vary during our experiments. To reduce the influence of these parameters we conduct our experiments at night.

- *Memory:* We have limited the workload of our tests so that the tasks of a p-SARS node can be held in its entirety in main memory (RAM). This is done so that our system will not outgrow the physical memory and be swapped to disc. If this happened it would lead to a significant impact on our measurements.

Next we need to identify the parameters we will vary between experiments. These are the factors to be studied. The factors, or variable parameters, in the throughput testing are as follows:

- *Queue size:* The maximum number of elements a queue in our system can contain.

- *Number of queries:* The number of search events to insert into the p-SARS node.

8.1.2 Conformance testing

Conformance testing is used to determine whether the implementation of a system meets the standards or specifications it was designed to meet. In our case this means that we will test our requirements on the finished p-SARS prototype to see if they hold. The tests are performed by starting up six p-SARS nodes, situated on the same physical node, and inserting constructed events. The test cases are constructed based on the requirement’s fit criteria. By observing the path the events take both inside and externally between other nodes of the p-SARS system we conclude if the test is a success or not.

We also monitor the results from the processing of the different events. This is done by printing debug messages, but also by using our debug module to log debug events from all six nodes. In other words we perform both white-box testing and black-box testing of the real system. We are also simultaneously performing an interoperability test between p-SARS and the WAIF Recommender System (WRS). This is because we are connected to the WRS system during our tests and populate our topic sets from a node’s corresponding WRS topic server.

Our results will be presented in the same sequence as the requirements where stated in chapter 4.

Testing our Functional requirements

Requirement 1: The system was able to bootstrap because of the bootstrap-server. Each node contacted the bootstrap-server and fetched the list of already connected nodes. The list was then used to extract neighbours. In our tests all p-SARS nodes ended up with all other five nodes in their corresponding neighbour set after running for a while.

The first node contacting the bootstrap-server will however always get an empty list returned. This is because the server does not yet know of any other nodes, but since we have implemented a system update function that periodically pulls the server for new neighbours if needed, all nodes in our tests finally ends up with five other nodes in their respective neighbour set. See also requirement 10.

Requirement 3: When a search query was inserted into the system the response received always described exactly the same results. This means that the system not only supports searching but also that the algorithm is stable and produces the same result every time. If the time to live (TTL) was set so low that we could not guarantee that the query visited all nodes, the number of query hits fluctuated. The reason for this is that the query then visited a different set of nodes each time.

The pending search technique described in the architectural design chapter also worked. That is, if a query was not satisfied before the timeout, the results already received were sent to the WRS client. If a query was satisfied before timeout it was sent right away. We also observed that when the p-SARS node holding the pending search crashes, the WRS client will not receive the results. This is according to our design and it’s up to the WRS client to find a new super-node to connect to and reissue the query.

Requirement 4: The interaction between p-SARS and WRS functioned correctly. In fact any system that adheres to the event format may communicate with p-SARS. All events passing into, inside and out of the p-SARS system is described in appendix C.

Requirement 7 and 8: When running the system we printed the entire topic set after each system update. We could then observe that each p-SARS node fetched its corresponding topic set and cached it locally. Correct updates where also performed periodically.

Requirement 10: When running the system we printed the entire neighbour set on each system update. We noticed that the p-SARS nodes dynamically removed neighbours that went down and re-inserted them when they came back online. They successfully fetched new neighbours both from the ping-pong events and by contacting the bootstrap-server.

Testing our Non-functional requirements

Requirement 5: When the topic sets where updated and either some new topics and/or some deleted topics were discovered, the node performing the update initiated a gossip event containing this information. We also observed that the gossip behaved as described in the architectural design chapter. Both propagation and termination functioned properly.

Requirement 6: When running the system we printed the entire hint set on each system update. We observed that the system successfully extracted hints from both gossip events and search results. In addition the white-box testing also showed us that the propagation of the queries used this information efficiently to route the queries. There are however no limitations on how big a hint set might be so in the future we will have to adopt a cache policy like e.g. Least Recently Used (LRU) or Least Frequently Used (LFU).

Requirement 9: After bootstrap each client asked each of its neighbours to send it their hint set. They all answered the request and also received the hint sets of their neighbours. Hints where then extracted accordingly.

Requirement 2: When a p-SARS node went down this node was removed from the neighbour sets and the system still was able to process queries. The impact of a node failing is that the WRS clients local to this node will have to find a new super-node to connect to. They also lose all searches they have pending on this node. In addition all other queries currently queued or under processing at the failing node will be lost. Therefore a failing p-SARS node may also influence other, not local, WRS clients because their searches can be prematurely terminated. The search engine as a whole also loses the possibility to recommend these local WRS clients to other querying WRS clients because there, as of now, is no super-node advertising the topics they publish on.

Requirement 11: The system is designed as modules, and when we performed changes we only needed to re-implement parts of a module. There were a few exceptions though, if we changed the format on an event we needed to change all modules processing this event.

Requirement 12: The system was successfully tested in both Windows and Linux. One important thing we discovered though is that the p-SARS system only executes correctly on Python 2.3 or newer. This is because we in our implementation uses a module called Queue that has been enhanced with some new capabilities in this version.

8.1.3 Throughput testing

The throughput tests are performed both with and without the TCP/IP traffic. We want to see how many queries per second a node can process in real life, but also how fast it processes the queries internally. The first test is constructed by creating one new external module called the input module. This module pushes queries to the p-SARS node through TCP/IP, and by starting a timer in the system's pusher module when the first event is received and stopping the timer when the last event is received we can measure how many queries a p-SARS node can process each second when

TCP/IP traffic is included. The reason we stop the propagation of the event in the pusher module is depicted in figure 18.

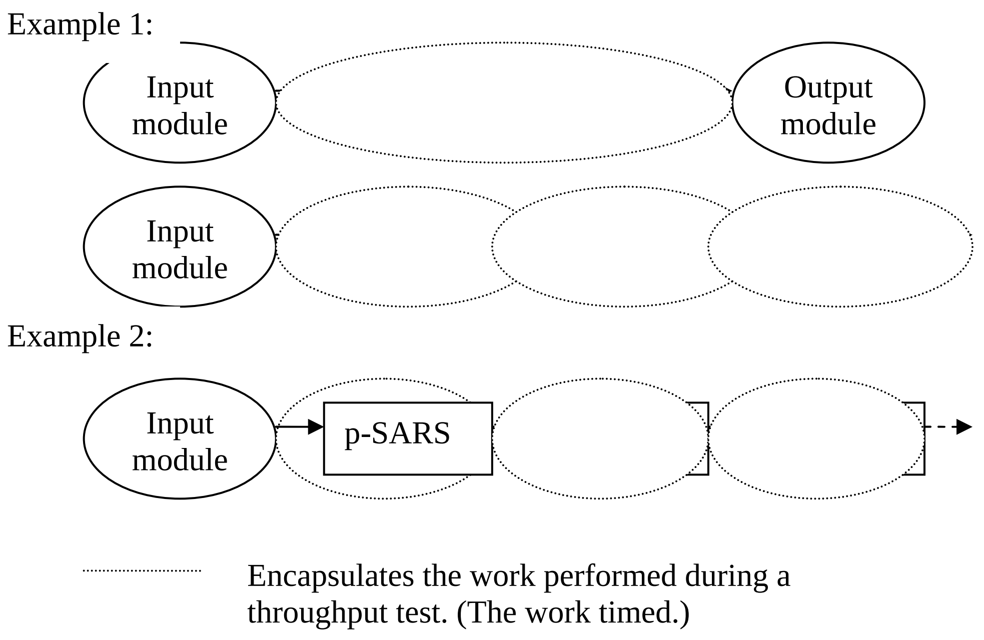

Example 1:
Input module
Output module
Input module
Example 2:
Input module
p-SARS
Encapsulates the work performed during a throughput test. (The work timed.)

Figure 18 - External throughput testing

As we can see in example 1 if we propagate the events to an output module in our throughput testing and perform the timing there we will actually end up timing some of the system's work twice. This is avoided by stopping the event at the pusher module. This is shown in example 2 and as we can see there is now no intersection between the work done and the work timed.

We also want to put our system through an internal throughput test so that we can identify how the TCP/IP traffic influences a node's query processing capability. This is done by modifying the listen module and the pusher module of our system. We also have to modify the search event itself to include a timestamp telling us when the test started. It all works as follows:

- The new listen module creates an internal search event, containing the timestamp, and starts enqueuing this event on the input queue a defined number of times.

- The search events are then processed by the search event processing module. The query is created so that no match will be found on this p-SARS node nor will any hints be available. The reason we have chosen to do this is to ensure that the search event takes the longest internal path and therefore our results will reflect the worst case scenario.

- The pusher module is modified to wait until the predefined number of queries is received, create a new timestamp and compare it to the one received with the events. This will then tell us how much time the p-SARS node needs to process a predefined number of events.

Throughput test results

Table 1 shows a summarization of the results of our internal throughput testing. While throughput testing the p-SARS node we changed the size of the event queues. As we can see the optimal size was found to be from one to about thousand elements for the internal throughput test.

Queue size	# Searches	Errors sending	Time	Search per Second
1	100 000	0	128	781.3
10	100 000	0	129	775.2
100	100 000	0	129	775.2
1000	100 000	0	136	735.3
10 000	100 000	0	177	565.0
100 000	100 000	0	231	432.9

Table 1 - Internal throughput testing

We have not yet for certain identified the reason why our system performs better with a smaller queue size, but we surmise it's because of the Python's garbage collection algorithm. That is, we think that the garbage collection consumes much more resources when the queues grow larger and the number of referenced objects increases. To be able to accurately identify the reason we will most likely have to look into the implementation of the Python Queue module and how Python handles the garbage collection.

When including the TCP/IP traffic we can see in table 2 that the throughput drops dramatically and that the size of the event queues in practice has less impact on the performance than the internal throughput test suggested. Since the best result in the internal throughput test is a factor of 14.7 better than the best result in the external throughput test it's safe to assume that the TCP/IP traffic completely dominates the results. Therefore the size of the queues has little influence here.

Queue size	# Searches	Error sending	Time	Search per Second	# failed open socket
1	5 000	0	99	50.5	0
10	5 000	0	94	53.2	0
100	5 000	0	97	51.5	0
1 000	5 000	0	97	51.5	0
10 000	5 000	0	94	53.2	0

Table 2 - External throughput testing

As we may read from the results the TCP/IP traffic constitute the bottleneck in our system. Possible solutions to increase throughput is the creation of thread pools that awaits incoming connections to a p-SARS node or to hold persistent TCP/IP connections to our neighbours. Another solution to get better performance is of course to re-implement the system in C or C++. We did not choose Python because of its performance characteristics but because it's the ideal language for prototyping. For performance critical software C or C++ is far more effective than Python.

To test what effect persistent TCP/IP connections actually could have on our system we re-implemented the listen module so that it didn't shut down the connection after receiving a search event but instead waited for another one. We also configured 5 other nodes to continuously push search events into the p-SARS node. The results gathered are presented in table 3.

TCP/IP?	Queue size	# Searches	Error sending	Time	Search per Second
TRUE	100	10 000	0	61	163.9
TRUE	100	10 000	0	51	196.0
TRUE	100	10 000	0	57	175.4
TRUE	100	10 000	0	62	161.3
TRUE	100	10 000	0	51	196.0
TRUE	100	10 000	0	50	200.0

Table 3 - Throughput with persistent TCP/IP connections

As we can see the number of events processed increases substantially. The best result is 3.8 times better than the best result measured when we do not keep persistent connections. The reason for this is that the setting up and tearing down of TCP/IP connections is avoided. We have also saved us form creating a thread each time we receive a packet. The lesson learned here is therefore that we will increase throughput if we keep persistent connections to our neighbours.

8.2 Simulating p-SARS

We designed a simulator to help us discover what effect the different techniques incorporated into our design had on the search efficiency without having to set up a potentially large test-bed. The simulator is thoroughly discussed in chapter 7 and will therefore not be discussed here.

8.2.1 Parameters and factors

We have not identified any parameters that will affect our results without our consent. The reason for this is that we do not perform any time critical tests. Therefore the only impact e.g. a saturated node will have in this case is to increase the time until the simulation completes. This is of no importance though since the simulation results themselves will not be affected.

The factors to be studied in our simulations are the parameters in the test tuple. These 16 parameters are already presented in the end of chapter 7 but for readability we will also here present an example. The test tuple: (1000,9,1,0,0,0,1,500,1000,1,0,0,0,0,1,500) describes field by field the following to our simulator:

1. Create a network of 1000 nodes.
2. Maximum nodes of neighbours to fetch are 9.
3. Fetch randomly between 1-9 neighbours per node.
4. We should not perform gossip.
5. This field describes the TTL on a gossip.
6. This field describes the gossip probability.
7. We should perform a search.
8. TTL on search is set to 500.
9. We should perform 1000 searches.
10. The querying node should learn from the results.
11. All intermediate nodes on the path of the search event should not learn from the results.
12. We are not going to perform a warm-up.
13. This field describes the TTL on warm-up searches.
14. This field describes the number of warm-up searches to perform.
15. We are going to perform Random Walk gossiping.
16. The TTL on a Random Walk gossip is set to 500.

8.2.2 Simulations

When designing the simulations we basically just tune the factors in the test tuple to see what effect the different techniques incorporated has on our search mechanism. On each test we i.a. record the success rate and how many nodes a query on average has to visit before a hit is found. There are mainly three questions we try to answer using the simulator:

- How effective is our search mechanism?
- How effective is our gossip mechanism?
- How many messages do the different techniques generate?

We start out with our best coverage results presented in table 4.

Test tuple	Percentage of success	Avg. depth on success	# Gossip sent	# RW-G sent
Searching without hint cache (Pure Random Walk)				
(1000,4,0,0,0,0,1,500,1000,0,0,0,0,0,0,0)	52.4	250.3	-	-
Searching performed after warm-up				
(1000,4,0,0,0,0,1,500,1000,1,1,1,500,1000,0,0)	69.7	87.3	-	-
(1000,4,0,0,0,0,1,500,1000,1,0,1,500,1000,0,0)	65.6	212.5	-	-
Searching performed after gossip				
(1000,4,0,1,30,30,1,500,1000,0,0,0,0,0,0,0)	86	91.9	1,333,670	-
Searching performed after RW-G				
(1000,4,0,0,0,0,1,500,1000,1,0,0,0,0,1,500)	100	2.7	-	500,000
(10000,9,1,0,0,0,1,500,1000,1,0,0,0,0,1,500)	100	19.6	-	5,000,000
(100000,4,0,0,0,0,1,500,1000,1,0,0,0,0,1,500)	92	153.4		50,000,000

Table 4 – Best coverage results

As we can see the Random Walk Gossip (RW-G) strategy we proposed in chapter 7 performs the best with regards to both how many percent of our searches is a success and how many nodes these successful searches in average must visit[11]. We also notice that the average number of messages generated in the overlay network by the RW-G approach is far less than the average number of messages generated by the p-SARS gossip algorithm.

If we compare the RW-G strategy with the warm-up strategy there are some differences. While a RW-G runs until completion every time, i.e. until the time to live (TTL) value reaches zero, a warm-up search functions just like a search and propagates only until a hit is found and then it's terminated. In other words a RW-G uses on average more messages and covers more nodes than the warm-up strategy. In our simulations we also observed that a RW-G covered almost only unique nodes. This is important because if this was not the case the burden on the network on a RW-G would be more or less wasted. Another difference between RW-G and warm-up is that a warm-up search carries with it the search results to populate intermediate nodes while the RW-G carries with it the topic set of the node where it was initiated.

The reason we simulate warm-up is to identify whether or not it's a good idea to extract hint information from the search results in our system. This technique is therefore not proposed as a substitute for e.g. the RW-G approach but as a supplement. The results show that after performing 1000 warm-up searches we get a noticeable increase in the hit rate of our searches compared to the pure random walk

[11] Average depth on success.

approach. This indicates that it's important to utilize the information already flowing through the system regardless of what other additional technique we use to populate the nodes individual hint set.

To see the effect of our designed gossip mechanism we compare it to the test case where the hint cache is disabled. Our designed gossip algorithm implemented in the p-SARS prototype increases the coverage from 52.4 percent to 86 percent and reduces the average number of nodes visited by a search event by 158.4. Although our gossip mechanism increases the effectiveness of our search algorithm it doesn't perform as good as the proposed RW-G approach.

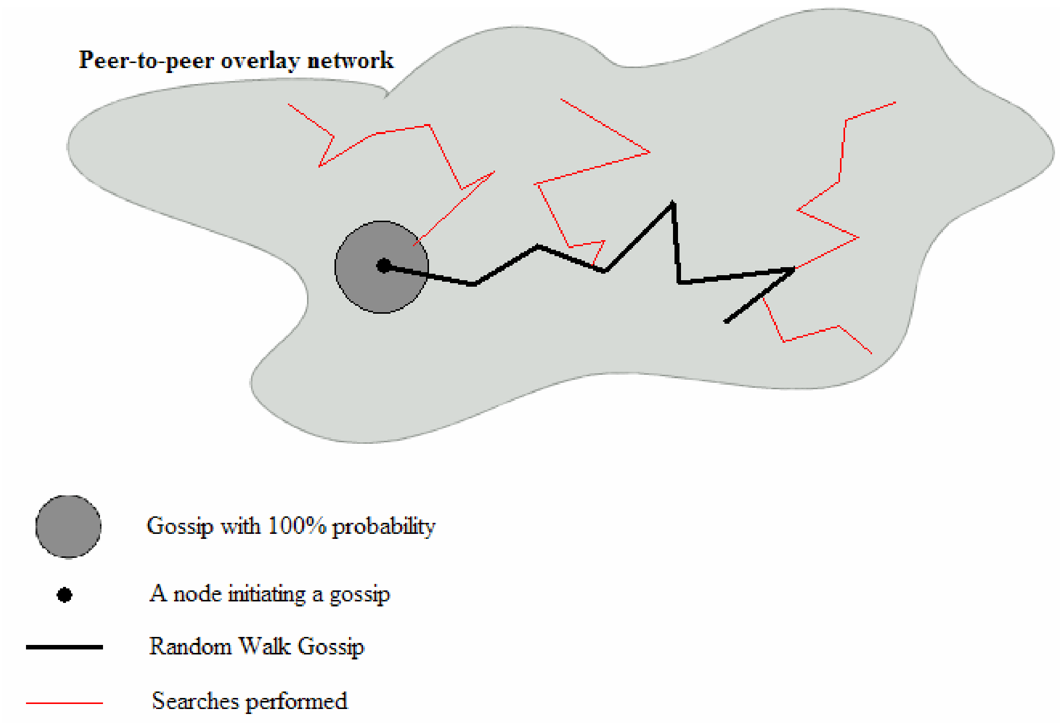

Peer-to-peer overlay network
Gossip with 100% probability
A node initiating a gossip
Random Walk Gossip
Searches performed

Figure 19 – Random Walk gossip vs. designed gossip mechanism

In figure 19 we can see a simplified picture of how the two different gossip mechanisms behave in an overlay network. A gossip with a probability set to 100% behaves as the flooding algorithm described in the theoretical framework chapter. This results in that every neighbour around the gossip initiating node receives the event. The draw back compared to the RW-G approach is that this gossip event only populates the hint caches of relatively close nodes in the overlay network. The RW-G approach on the other hand populate the hint cashes of the nodes encountered on a Random Walk. Therefore it potentially reaches deeper into the overlay network as depict in figure 19. The reason the RW-G approach on average performs better is because it distributes the hints more evenly in the network than the p-SARS gossip mechanism. This results in that the chances that we will find a hint on a search becomes more or less independent on from which node the search is initiated. In our p-SARS gossip mechanism on the other hand we expect that a searching node would benefit from being as close to the gossip initiating node as possible, and that distant nodes will be penalized.

The lessoned learned here is therefore that the branching factor on our gossip algorithm should be small, but in our design we can only reduce the branching factor by reducing the probability for gossiping. The result is that even though reducing the probability for gossiping will make our mechanism more like the RW-G approach it has the side effect that the probability that the gossip event is propagated from a specific node to at least one of its neighbours decreases. That is, if we decrease the gossip probability this will reduce the number of steps the gossip event will propagate.

It's important to notice that although the simulator presents a close to real life picture of the search traffic, it does not when it comes to the gossip traffic or the warm-up traffic. To be able to simulate how effective our search mechanism is we first have to warm up the network. If we e.g. want to build up a network of 1000 nodes we add these 1000 nodes simultaneously, and ask them one by one to initiate e.g. a gossip. This results in a huge amount of messages being sent almost simultaneously. The real world however is somewhat nicer. Here we would not expect all nodes to join simultaneously, but that the network will be created over time.

We also argue that the gossip traffic will not represent the major part of the overall traffic. There are two reasons for this. Firstly we have that the topics on a p-SARS node will not change rapidly because the topics represent human interests. Secondly we have designed and implemented a periodic update of a p-SARS node's topic set. This means that although e.g. 1000 WRS clients are connected to this p-SARS node and they all change their interests between two periodic updates of the cached topic set, this will only generate one gossip event. So if we e.g. update the topic set every thirty minutes, one p-SARS node will generate a maximum of 48 gossip events a day regardless of how many WRS clients are connected to this p-SARS node. If we name the number of messages one gossip produces in the overlay network X we have that each p-SARS node must process: (48*X)/86,400 gossip events per second in the worst case scenario. The same reasoning also holds for the RW-G approach.

If we uses table 4 we can calculate some numbers that are independent on how many nodes that participate in the overlay network:

- We can see that our proposed p-SARS gossip mechanism with a TTL set to 30 and a gossip probability of 30% generates 1,333,670 messages in the overlay network when performing 1000 gossips. Each gossip initiated therefore generates an average of: 1,333,670/1000 = 1333.7 messages. The worst case scenario is then that each p-SARS node must process 48*1333.7/86,400 = <u>0.7</u> gossip events per second.

- The RW-G mechanism is somewhat more effective. Each RW-G generates 500 messages as is defined by the TTL value set. Therefore each p-SARS node must process 48*500/86,400 = <u>0.3</u> gossip events per second in the worst case scenario.

8.3 p-SARS scalability

In this section we will compare the test results with the simulation results to identify how scalable our solution is. We will look at three different network configurations and their scalability and then in the end draw an overall conclusion based on the calculations presented.

8.3.1 Calculations

Our calculations are presented in table 5 and here we will explain the rationale behind them by walking through the first two calculations. These are presented in row one and two in the table.

In our example calculation we have a network configuration of one thousand WAIF Recommender System (WRS) clients per p-SARS node and a total of one thousand p-SARS nodes participating in the overlay peer-to-peer network. This means that the network as a whole supports 1 million WRS clients.

Worst case scenario (WCS):

To estimate the worst case scenario we assume that none of the searches are satisfied before the time to live (TTL) value runs out and that the system uses the Random Walk Gossiping (RW-G) approach. If we look back on the external throughput test we see that the p-SARS prototype is able to process 53.2 search events per second. But since we also have to process 0.3 gossip messages per second per node we need to subtract these. That is, each p-SARS node is able to process 53.2 – 0.3 = 52.9 search events per second. Every search has a time to live (TTL) value set to 500 and this tells us that each search in the worst case scenario produces a workload of 500 messages for the overlay network as a whole. Therefore we have that each p-SARS node may initiate 52.9/500 = 0.1058 searches per second. This means that each WRS client connected to a p-SARS node may initiate 0.1058/1000 = 0.0001058 searches per second. That is, all WRS client may simultaneously initiate one search per 1/ 0.0001058 = 9451.8 seconds without saturating the system.

Best case scenario (BCS):

To estimate the best case scenario we assume a populated hint cache and that the system uses the Random Walk Gossiping (RW-G) approach. The average depth on success for this configuration is measured to be 2.7. That means that each p-SARS node can initiate 52.9/2.7 = 19.6 searches per second without saturating the system. Each WRS client can therefore initiate 19.6/1000 = 0.0196 searches per second. That is, all WRS client may simultaneously initiate one search per 1/0.0196 = 51.0 seconds without saturating the system.

	# search processed per second	# gossip per second	# p-SARS nodes	# WRS clients per p-SARS node	Avg. depth on success	# number of seconds between each search a WRS client can issue
The prototype with Random Walk Gossip (RW-G)						
WCS	53.2	0.3	1000	1000	500	9451.8
BCS	53.2	0.3	1000	1000	2.7	51.0
WCS	53.2	0.3	10 000	100	500	945.2
BCS	53.2	0.3	10 000	100	19.6	37.0
WCS	53.2	0.3	100 000	10	500	94.5
BCS	53.2	0.3	100 000	10	153.4	29.0
The prototype with the implemented gossip mechanism						
WCS	53.2	0.7	1000	1000	500	9523.8
BCS	53.2	0.7	1000	1000	91.9	1750.5
Keeping persistent TCP/IP connections						
WCS	200	0.3	100 000	10	500	25.0
BCS	200	0.3	100 000	10	153.4	7.7
Assuming web-server performance (Based on SEDA Gnutella packet router)						
WCS	1000	6.7	100 000	10	500	94.5
BCS	1000	6.7	100 000	10	153.4	1.5

Table 5 – 1 000 000 WRS clients, scalability calculations

From our calculations in table 5 we can see that the more WRS clients that participate as p-SARS nodes in the overlay network the better our system scales. The reason for this is that, although the number of p-SARS nodes increases with a factor of ten the average depth on success only increases with a factor between 7.2 and 7.8 in our simulations. That is, the overall load increases slower then the additional processing capability when we include more p-SARS nodes in the overlay network. Therefore it is advantageous to have more nodes cooperatly handling the load. We have only simulated networks up to one hundred thousand nodes however. Therefore we cannot tell if this trend continues to hold for network sizes over one hundred thousand p-SARS nodes.

As mentioned before we assume that the rate of change in the topics will be slow. This is because the topics represent the interests of humans and these do not change frequently. This will also most likely affect the rate at which new queries are initiated. That is, because we expect the topics to change slowly there will also be a slow change in the searchable data. Therefore we assume that a WRS client on average also initiates new queries at a slow rate. We think it is reasonable that a WRS client, on average, issues a new query as slowly as one per hour.

From the results in table 5 we can see that our prototype re-implemented with the RW-G approach is capable of offering a much higher query rate than this. Each WRS client can on average issue a query every 29 seconds. And these calculations are based on our prototype implemented in python. If we had re-implemented the system in e.g. C and fine tuned the architecture for performance we can see no reason why we shouldn't come close to the throughput accomplished in [M. Welsh et al. 2001]. In this paper they present a Gnutella packet router, based on the SEDAN architecture, capable of processing 1000 packets per second.

There are basically two reasons why we can make this assumption. First our architecture is surprisingly similar to that of the SEDA architecture and therefore it is reasonable to believe that we could be able to reach the same throughput. Both systems have separated concerns by dividing the application into stages. Also the stages are connected by queues and each stage is event driven just as in our architecture. The only difference is that the SEDA architecture uses resource controllers to dynamically allocate more threads to stages which are saturated. This way they manage to increase overall throughput on high workloads. Because every processing stage in our p-SARS system is asynchronous and event driven we are also able to initiate more than one thread simultaneously to perform the work of a module. Therefore the only thing missing is to incorporate the resource controllers.

Secondly we have that the WRS system selects the best capable nodes to run as p-SARS nodes. That is, we will only use the best WRS clients as nodes in the p-SARS overlay network. Therefore it is also reasonable to assume that the hardware of our p-SARS nodes at least is nearly as good as the hardware which is used in [M. Welsh et al. 2001] to test the packet router.

As we can se from table 5, if we manage to reach the throughput of the SEDA packet router each WRS client will be able to issue a query every 1.5 seconds. This is such a high query rate that the chances that our system will be saturated are almost eliminated.

As an overall conclusion we argue that based on the throughput tests and simulations the designed system with the use of RW-G scales to at least one million WRS clients. Even our implemented prototype which uses the p-SARS gossip algorithm will scale, if the query rate assumption holds. However, based on the results we will probably exchange the current gossip mechanism with RW-G.

8.4 Summary

In this chapter we have shown that our prototype of the p-SARS system satisfies the requirements set in chapter 4. Furthermore we have identified the peak capacity of our system by presenting throughput results.

We discovered during the throughput tests that the TCP/IP traffic has a substantial impact on the throughput and two approaches to remedy this problem were discussed. Persistent TCP/IP connections and thread pools. We even presented measurements to show that the persistent TCP/IP approach works in practice.

With our simulation results we showed that our proposed Random Walk Gossip (RW-G) approach outperforms the p-SARS gossip implementation both with regards to the traffic generated and the resulting search hit ratio. We therefore concluded that our gossip event processing module should be re-implemented to support the RW-G approach instead.

In the end we presented a comparison between our throughput tests and our simulations and concluded that our current p-SARS design with the use of RW-G is capable of scaling to at least one million WRS clients.

Chapter 9

Discussion

In this chapter we will discuss the design and implementation of the p-SARS prototype. We will divide our discussion into the four system mechanisms identified in the design chapter and also add an extra section which we have called miscellaneous.

9.1 The topic update mechanism

This mechanism updates the local topic cache on a p-SARS node. Here we discuss one improvement to this mechanism.

9.1.1 Including the subscribe set

Each p-SARS node caches the topics which the local WAIF Recommender System (WRS) clients publish on. But there is still information available that could further improve the efficiency of our search mechanism. We remember from the design chapter that each WRS client is both a publisher and a subscriber. This means that each WRS client also holds information about other, possibly remote, WRS clients that publish on a given topic. That is, it knows which WRS clients it subscribes to and therefore also some of the topics these nodes offer. This information is actually already available in the event we receive from the WRS topic server and can easily be extracted an added to our topic set.

The reason we cannot add this information to the hint set where it actually belongs is because we do not know the addresses of the super-nodes supporting these remote WRS clients. For this to be possible we would have to augment the WRS clients so that when asked they could tell which super-node they currently are connected to. It’s important to notice here that it’s possible for a local WRS client to subscribe to another local WRS client. These subscriptions are not of any importance though because they only reflect what already is known in the local topic set. It’s only the remote WRS clients and what they publish on that are of interest to us here.

By implementing this enhancement we introduce the possibility for duplicates to occur in the search results. This may happen because there are now possibly several p-SARS nodes advertising the topics a WRS client publishes on. The effect of duplicates and how they can be handled is discussed in the search mechanism section of this chapter.

9.2 The membership mechanism

The bootstrap mechanism is in charge of populating a node’s individual neighbour set with other p-SARS nodes. These neighbours are essential to our system because they are needed for both the search mechanism and the gossip mechanism to function properly. Here we will propose several changes to our current bootstrap-server and finally we also discuss our current ping-pong scheme.

9.2.1 The bootstrap-server

Our project is not focused around the bootstrap problem so we have designed a simple solution that actually just sends the whole list of registered nodes back to the requesting node. The list is then used to populate the neighbour set. A better solution is, as we discussed in the theoretical framework chapter, to use random walks as described in [A. J. Ganesh et al. 2003] after we initially have found a node connected to the overlay network. This will, with a high probability, ensure that the resulting overlay graph stays well connected. A simpler solution but still better than our current design is to randomize the results from the bootstrap server so that not every node chooses the same set of neighbours.

In the theoretical framework chapter we also mentioned the technique of caching nodes encountered during previous runs. This will potentially offload the boot server because we will be able to use this information for later bootstraps. There are however no guarantee that these nodes still are up and running, so this technique can only be seen as a supplement to the bootstrap-server technique. If it fails we need to have a bootstrap-server available.

9.2.2 Ping and pong

In our current design we have a defined maximum on how many neighbours a node may have in its neighbour set. What we don’t have is a limitation on how many nodes that may reckon a specific node as its neighbour. This means that it’s possible that we can end up with a popular node X which is included into every node’s neighbour set except its own. That is, if we have a network of one thousand p-SARS nodes, node X could be forced to answer ping messages from 999 other nodes. This can potentially lead to a message implosion at node X.

One solution is to maintain the set of neighbours which reckon this node as its neighbour and set an upper limit on the size of this set. When the limit is reached, we simply instruct node X to avoid answering the ping messages from the nodes not in this set. This will trick any pinging node, not currently in this set, to think that node X is dead and thus remove it from its own neighbour set. One disadvantage though is that the tricked nodes will remove the hints pointing to node X from their local hint sets.

Another solution is to implement some sort of handshake between a node and the node it wants to incorporate into its neighbour set. If a node realizes that too many nodes currently reckon it as a neighbour it just have to decline any future request until the situation improves.

Gnutella [Gnutella] uses ping and pong messages not as heartbeat messages as we do in p-SARS but as a way of discovering other nodes. A node floods the ping message into the overlay network and receives pong messages from the nodes receiving the ping message. This way the system extracts neighbours from both ping and pong messages.

We could potentially borrow some ideas from this approach and e.g. initiate a random walk carrying a ping event. The event would propagate through the net just like a random walk search and record the nodes it encountered during propagation. At the end it would be sent back to the initiating node. Every node participating could also extract neighbours from the ping event as it’s passing by.

Another approach discussed in the design chapter is to utilize the events already flowing through the network. Especially the search event is suitable because it already carries a list of visited nodes which can be seen as potential neighbours.

9.3 The gossip mechanism

The gossip mechanism is designed and implemented to make the searches more efficient. Here we discuss and propose changes to this mechanism.

9.3.1 The algorithm

During simulations we discovered that our proposed Random Walk gossip (RW-G) strategy actually outperforms our implemented gossip mechanism. The reason is that the RW-G on average more evenly distributes the hints in the overlay network. The p-SARS gossip mechanism could be tuned to have a low branching factor and actually behave similar to the RW-G approach. The disadvantage we discovered was that the probability that our gossip would terminate early would increase when decreasing the branching factor. So if we have to choose between the two approaches now we would choose the RW-G approach.

More simulations have to be performed to identify which gossip approach we should implement in the future, but an interesting idea is to extract some of both techniques. E.g. we could initiate a RW-G and on each node it visits we can, defined by a probability, branch of one or several RW-G events and send them off in different directions in the overlay network. The newly spawned RW-G's would inherit the TTL of it's parent so that RW-G's spawned in early stages would propagate further than the RW-G's spawned late in the propagation of its parent. Optimal branching factor and if this indeed it the way to go is left to investigate. How this would look in the overlay network is depict in figure 20.

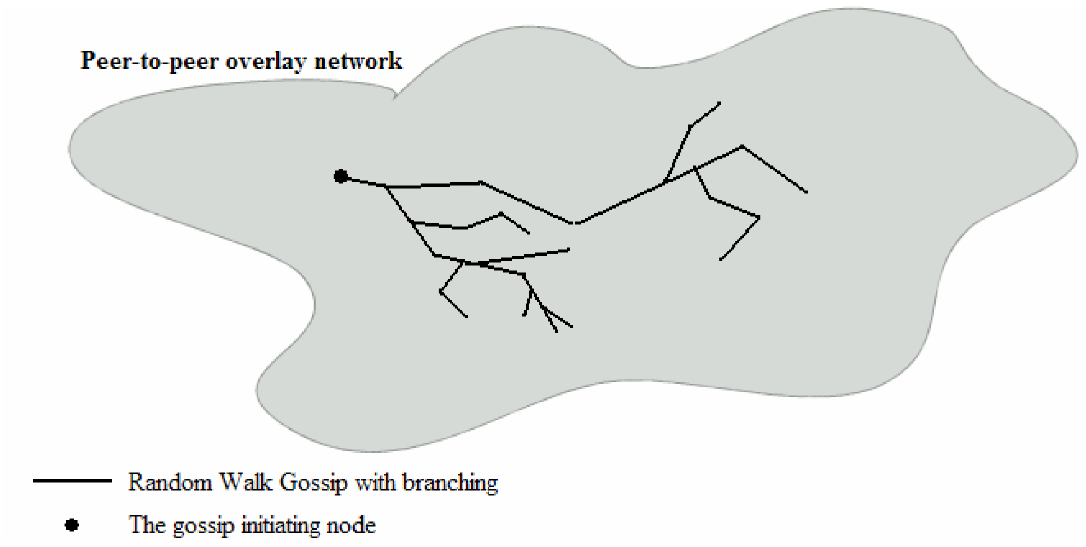

Peer-to-peer overlay network
Random Walk Gossip with branching
The gossip initiating node

Figure 20 - Random Walk Gossip with branching

9.3.2 Achieving global view

Original gossip protocols have the property that they eventually converge to a global view [A. Demers et al. 1987]. We argue that our implemented algorithm also has this property. If we take a look at our gossip algorithm we initiate a gossip on two occasions. First on a topic update if the topic set has changed, and secondly if a node receives a search event relayed on a hint which we discover to be stale. Regardless of the reason we initiate a gossip with this node’s entire hint set. As the nodes receiving this gossip eventually will initiate gossip events on their own the data will eventually propagate throughout the overlay network. What we do not argue is the fact that this propagation presumably will be very slow. The reason for this is that as long as the interests of the WRS clients do not change, no gossip will be initiated. And as we suggested in the design chapter we expect this rate to be low.

However, demanding that a node eventually should have a global view in its cache would, as more clients connect, lead to an un-scalable solution. A solution to this problem is to impose a policy for dropping items when the hint cache reaches a defined maximum size. Widely known cache policies that can be used here are Least Recently Used (LRU) and Least Frequently Used (LFU). This will however have an impact on the global view property. If we e.g. choose LRU as a cache policy the algorithm will converge to a global view of the objects most recently used. As of now we have not implemented a cache policy.

Another important question dealing with gossip algorithms is to decide on how often we should initiate a gossip. The answer to this question is actually straight forward. As long as the system as a whole is capable of handling the extra load we should gossip as often as possible. The more we gossip the more efficient our search mechanism will be.

9.3.3 The gossip simulation

Our simulator’s implemented gossip algorithms actually deviate from the real gossip algorithms described. The fault is that none of the two gossip mechanisms implemented in the simulator disseminates a node’s hint set. Only the topic set is disseminated. We argue however that the results we have obtained in chapter 8 still are valid although probably to pessimistic. The reason for this is that if we also had disseminated the hint sets in our simulations, more hints would have been available on the individual nodes, and this would again have increased the efficiency of our search algorithm. That is, better percentage of success on searches and a smaller average propagation depth of the search events. We also argue that our conclusion on Random Walk Gossip (RW-G) versus the implemented p-SARS algorithm still holds. That is, if we include the hint sets in our gossips we still believe that the more evenly distribution of the hints in the RW-G approach will ensure that the RW-G algorithm still outperforms the p-SARS gossip algorithm.

9.4 Search mechanism

In this section we will discuss the search mechanism and some enhancements.

9.4.1 Search implementation

We chose to design and implement the system as a pure peer-to-peer overlay network. One reason for this was to be able to support pattern searching which is not possible in the distributed hash table (DHT) technique. In our current implementation though we haven't yet incorporated a pattern search. The locally implemented search is in fact a lookup mechanism. That is, the WAIF Recommender System (WRS) client will only get a hit if the query is matched exactly to a topic.

To implement a search mechanism that supports pattern search we do not need to change our overall system design. Only one function need be changed. This is the read function of the topics class situated in the OWN_system.py file. Example of a simple search algorithm in python that supports case sensitive search for a sub-string within a topic is shown below. If we e.g. search for 'ball', we could get WRS clients publishing on both 'football' and 'basketball' returned.

```
def read(self, topic):
        # Set lock
        self.spin_lock()

        # List holding the WRS clients
        # matching the topic
        list = []

        # List holding the matching
        # topics
        matching_topics = []

        # Simple pattern search algorithm, Case sensitive
        topics_available = self.topics.keys()

        for element in topics_available:
            if topic in element:
                matching_topics.append(element)

        # Do uique id --> IP address mapping
        for item in matching_topics:
            # Test if mapping is available
            if item in self.append(self.mapping[item]):
                list.append(self.mapping[item])

        # unlock mutex
        self.unlock()

        # Test if results are found
        if len(list) > 0:
            return list
        else:
            return false
```

Another property of our search implementation is that a WRS client may get more answers returned than the happy[12] value suggests. This will happen if more

[12] The happy value describes how many hits are required to satisfy the query.

results than the happy value describes is found on one node. As an example let’s say that the happy value equals one. The node processing the search event traverse its topic set and finds five hits on this query. Instead of choosing four of these results to discard, all results are returned before the propagation of the search event is terminated. We have chosen this approach because when we have discovered the hits we may as well return them. However, if our algorithm finds many hits on one node this will result in a large result event being sent over the network. To prevent that a result event can saturate the network we should enforce an upper limit on the number of results returned.

9.4.2 Returning the results

In our design we decided to return the answers directly to the p-SARS node we label as the querying node[13]. The alternative would be to route the answer back the way the query came. Our choice has two advantages but also one disadvantage. The first advantage is that the answer arrives at the querying p-SARS node much faster than if we need to route it through the intermediate nodes. The user perceived delay will therefore potentially be lower[14]. The second advantage is that the number of search events traversing the overlay network will be less then in the proposed alternative.

The disadvantage however is that the intermediate nodes will never observe that a p-SARS node has answered the query. If other nodes where able to notice when a node answered a query they could use this information to update their hint sets. In our design the only node that benefits from this is the querying p-SARS node which extracts hints from the results received.

Routing the answer back the same path the query took should be fairly easy to implement. The search event already contains the list of the nodes visited by the search event. This list is also sorted so that the last element in it actually is the last node visited.

9.4.3 Duplications in results

In our current implementation duplicates will not occur in a query’s result set. This is because a WRS client connects to only one p-SARS node, and that the search mechanism ensures that any given p-SARS node is only searched once by the same query. However if we incorporate the change discussed above in the section labelled topic update mechanism, duplicates may occur. This could with little effort be handled by the pending search mechanism. We just have to ensure that the result is not already enqueued on this query, if it is just discard the duplicate.

Our search event carries a happy value that describes when a query is satisfied. Based on the happy value and the time to live (TTL) field we decide if the

[13] The node that initially received the query from the WRS client
[14] If the query isn’t satisfied the user must always wait until the pending search times out

search should be terminated or sent onwards to another p-SARS node. If duplicates are encountered another problem therefore arises. The happy value will be wrongly decremented. The reason for this is that the search event carries no information about previously discovered results so that we cannot decide locally if a hit is a duplicate or not. The premature termination problem could be solved by making the searching node check back with the querying p-SARS node to see if the query is satisfied instead of relying on the search event's happy value as we do now. This will work because the querying p-SARS node is able to check for duplicates and will therefore not wrongly decrement the query's happy value. The extra cost will be the message traffic back and forth and the delaying of the query results themselves.

9.4.4 When to terminate search

The time to live (TTL) value combined with the happy value decides if the search event should be terminated or if it should be forwarded to another p-SARS node. The problem here is to figure out how many nodes participate in the overlay network, because as more nodes join the larger the TTL value has to be. One of the solutions discussed in [C. Law, K. Y. Siu. 2003] is to use long living random walks to estimate the size of the overlay network. The approach taken is to mark the nodes that the walker has counted and to also notify each node it visits with its current count.

Another solution is to always have a large TTL so that the happy value is in charge of when the search should be terminated. The only function the TTL value has is to prevent a loop. This approach is similar to the "Checking" approach discussed in [Q. Lv et al. 2002] but in our solution we propagate the satisfaction value with the search event. We therefore don't need to check back with the querying node every few steps to see if the query is satisfied. This is however necessary if we want to increase the number of walkers issued for one search and also if duplicates may arise.

9.4.5 Anonymity

Gnutella [Gnutella] offers, to some extent, anonymity on behalf of the searching node. This is because the hits on a query are not sent directly back to the querying node. The hits are instead routed back the same path the request came so if the querying node is more than one hop away from the answering node its anonymity is secured.

In our system everybody knows which node issued the query because this information is available in the search event. This is in fact not such a big drawback compared to the Gnutella solution. It's only the search activity that to some extent is anonymous in Gnutella. A client must reveal its identity as soon as it chooses to download a file, just like it has to do in WRS when it chooses to subscribe from another WRS client.

9.5 Miscellaneous

Here we will first discuss that the data we search for in p-SARS as in contrast to a regular file sharing service, is nearly persistent. In the end we will discuss how to gracefully terminate a p-SARS node.

9.5.1 Persistent data

Because the data we search for reflects the interests of specific humans we assume that the rate of change is low. This is reflected in the periodic updates of the topic set which in our implementation currently is set to once every thirty minutes. We believe that our search mechanism could also be used as part of a regular file sharing service. But the scalability results presented in chapter 8 would have to be adjusted. This is because the locally cached topic set would have to be updated at a much higher rate and therefore the gossip traffic would consume more resources. This will lead to a reduced scalability.

9.5.2 Terminating the program

The termination of a p-SARS node is not yet implemented. There are however two solutions to this problem. Firstly we may update a shared variable periodically tested by the threads running. When this termination variable is set to false all threads ends execution. This approach has the disadvantage that it does not flush the system event queues before terminating the program and that it may take a while before the thread actually tests this variable.

To solve both these problems we may introduce a locally issued termination event. When we want to terminate the system we just insert this termination event into the system input queue. This event will then propagate through the event queues and finally reach all modules in our implementation except two. These are the system module and the update topic module and cannot be reached because they are not connected to any queues. Therefore to incorporate this solution into our current implementation we would have to create two new queues, one for each of these modules, with the soul purpose of delivering this termination event. The additional change on the existing code would be minimal. We just have to add the following test after a module receives an event:

```
if event['type'] == 'termination':
    # Possibly do some clean up work and
    # send the event towards next module
    # if any.
    ...
    ...
    # Break out of while loop
    break
```

When the module has propagated the termination event onwards it may itself terminate. There are two exceptions though. Firstly the result event processing module must terminate all pending searches and enqueue the results for the corresponding WAIF Recommender System (WRS) client before it propagates the termination event onwards. Secondly the pusher module will have to ensure that all four event processing modules have sent it the termination event. This is to ensure that all the event queues are flushed and that no additional events should be sent out of the node through TCP/IP.

9.6 Summary

In this chapter we have mainly discussed the four system mechanisms which we identified during design. We have discussed important issues like:

- How we should handle the bootstrap problem.
- The global view property of our gossip mechanism.
- How to choose the correct time to live (TTL) value.
- If we should return the results directly or send them back the way the search event came as it's done in e.g. Gnutella.
- Avoiding message implosion because of ping events.

We have also proposed several enhancements to or current design where the more important once were:

- Including the subscribe set.
    - Avoiding duplicates and the premature termination problem.
- Caching policies like LRU or LFU.
- A simple search algorithm capable of matching sub-strings within a topic.
- A locally issued termination event so that we are able to flush the event queues of our system before shutting down.

The correctness of our simulation results where also discussed. We discovered that the simulator was faulty and that it did not gossip a node's hint set. Only the topic sets of the nodes where disseminated. However, we have argued that our simulation results still are valid although probably to pessimistic with regard to the success ratio and also the average depth on success.

Chapter 10

Conclusion

This chapter concludes the thesis. Here we will present an evaluation and a critique to our work. In the end future work will be discussed.

10.1 Evaluation

The main goal of our project was to design and implement a distributed search engine for the WAIF Recommender System (WRS). We have accomplished both these goals. Firstly we have designed a fault resistant, scalable, self administrative and distributed search mechanism. Secondly we have implemented a prototype of this design. All this is made possible by augmenting a sub-set of the WRS clients with search capabilities and including these nodes in the p-SARS overlay network.

Our choice of an overall distributed architecture was basically twofold. Either we chose the unstructured peer-to-peer architecture or we chose the structured peer-to-peer architecture. If we view p-SARS and WRS as one system the overall architecture is structured peer-to-peer. That is, the WRS system chooses a sub-set of its clients to run as super-nodes. These super-nodes are then enhanced with the p-SARS search mechanism and included in the p-SARS overlay network. Because the WRS system already has selected the most capable nodes to run as p-SARS nodes we decided that it was not worth the added complexity to also introduce a structured architecture between the p-SARS nodes. Therefore, if we look isolated on the p-SARS system, the chosen architecture is unstructured peer-to-peer.

With the help of simulations we have shown that our search mechanism has good coverage. In fact, in a network of 10 000 p-SARS nodes we have a success rate of 100 percent on our searches when using the Random Walk Gossip (RW-G) algorithm to populate the hint caches.

Finally we have compared throughput tests and simulations to decide how scalable our system is. We argue that both our design and our current prototype are capable of supporting in excess of one million WRS clients. The simulations also indicate that the overall load increases slower than the additional processing capability when we include more p-SARS nodes in the overlay network. That is, our solution seems to scale better the more WRS clients are included into the p-SARS overlay network.

We have also shown several enhancements to make our prototype scale even better. This includes i.a. re-implementing the gossip event processing module, so that our prototype uses RW-G, and keeping persistent TCP/IP connections between neighbours. We also assume that we would obtain additional performance if we re-implemented the prototype in C or C++.

10.2 A critique to our work

The biggest error was of course that we in our simulator actually implemented replication techniques instead of the gossip techniques proposed. This happened when we failed to include the hint set into our gossip events, turning the gossip mechanisms into replication mechanisms. Although this is a minor thing to fix in the simulator, it could have corrupted many hours of work. We argue though that the only influence this has on our simulation results is that they are to pessimistic.

We have repeatedly argued throughout this thesis that we wanted a solution that supported pattern searches and therefore the distributed hash table (DHT) technique was discarded. As it turned out we have not implemented pattern search in our prototype. We feel that this should have been done. When that is said, we only need to design a pattern search mechanism and modify one function in the p-SARS prototype for this to be supported.

10.3 Future work

First of all we plan to design and implement an efficient local search algorithm that supports pattern searching.

Next we will try to increase the scalability of our prototype. There are several issues we would like to investigate here and the once we think are the most promising are:

- Keep persistent TCP/IP connections to our neighbours.
- Keep thread pools.
- Re-implement the gossip event processing module so that it supports the Random Walk Gossip (RW-G) mechanism.
- Implement the prototype, or parts of it, in a different language like e.g. C or C++ to see if we can get an increase in performance.

We will also have to look into the security problem of our system. There are two major challenges here:

- First everybody that knows the message format of our system may fake messages and e.g. corrupting the hint caches leading to poor search efficiency.
- Another problem is that the interests of the WRS clients are distributed onto different nodes in the p-SARS overlay network. There is no security mechanism ensuring that these interests cannot be revealed.

Chapter 11

Bibliography

[A. Demers et al. 1987] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry. *Epidemic Algorithms For Replicated Database Maintenance.* In Proc. of the Sixth Symposium on Principles of Distributed Computing, pages 1-12, Vancouver, British Columbia, Canada, August 1987

[A. J. Ganesh et al. 2003] A. J. Ganesh, A. M. Kermarrec, L. Massoulié. *Peer-to-Peer Membership Management for Gossip-Based Protocols.* IEEE Transactions on computers, Vol. 52, No. 2, February 2003

[A. Macro. 1990] A. Macro. *Software Engineering: concepts and management.* Prentice Hall International, pages 49-53, United Kingdom, February 1990

[A. Oram. 2001] A. Oram. *PEER-TO-PEER: Harnessing the Power of Disruptive Technologies.* O'Reilly & Associates, Inc., March 2001.

[A. Rowstron, P. Druschel. 2001] A. Rowstron, P. Druschel. *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems.* In Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany, November 2001

[A. Singla, C. Rohrs. 2002] A. Singla, C. Rohrs. *Ultrapeers: Another Step Towards Gnutella Scalability.* Whitepaper, December 2002

[BareShare] *BareShare*, Homepage (2003). http://www.bearshare.com/

[B. Yang, H. G. Molina. 2001] B. Yang, H. Garcia-Molina. *Comparing Hybrid Peer-to-Peer Systems.* In Proc. of the 27th Int. Conference on Very Large Data Bases, Roma, Italy, September 2001

[B. Yang, H. G. Molina. 2002] B. Yang, H. Garcia-Molina. *Efficient search in peer-to-peer networks.* In Proc. of the 22nd IEEE International Conference on Distributred Computing Systems (ICDCS), Vienna, Austria, July 2002.

[C. Falaoutsos et al. 1994] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, W. Equitz. *Efficient and Effective Querying by Image Content*. Journal of Intelligent Information Systems, vol 3, pages 231 - 262, July 1994

[C. Gkantsidis et al. 2004] C. Gkantsidis, M. Mihail, A. Saberi. *Random Walks in Peer-to-Peer Networks.* To appear in INFOCOM 2004.

[C. Law, K. Y. Siu. 2003] C. Law, K. Y. Siu. *Distributed Construction of Random Expander Networks.* In Proc. of Infocom. IEEE, San Francisco, CA, USA, April 2003

[C. Tang et al. 2003] C. Tang, Z. Xu, S. Dwarkadas. *Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks.* Technical report, HP Labs, November 2002

[Cure] *Intel philanthropic peer-to-peer program.* Homepage (2003). http://www.intel.com/cure/

[D. Piscitello. 2002] D. Piscitello. *Security And Peer-To-Peer Applications.* Business Communications Review, pages 45 – 51, October 2002

[E. Cohen et al. 2003] E. Cohen, A. Fiat, H. Kaplan. *A Case for Associative Peer-to-Peer Overlays.* In Proc. of Workshop on Hot Topics in Networks, Princeton , New Jersey , USA October 2002

[E. Cohen, S. Shenker. 2002] E.Cohen, S. Shenker. *Replication Strategies in Unstructured Peer-to-Peer Networks.* In Proc. of ACM SIGCOMM '02, pages 177-190, Pittsburgh, PA, USA, August 2002

[J. Postel, J. Reynolds. 1983] J. Postel, J. Reynolds. *Telnet Protocol Specification.* RFC 854, May 1983

[J. Postel, J. Reynolds. 1985] J. Postel, J. Reynolds. *File Transfer Protocol (FTP).* RFC 0959, October 1985

[J. Ritter. 2001] J. Ritter. *Why Gnutella Can't Scale. No, Really.* http://www.darkridge.com/~jpr5/doc/gnutella.html February 2001

[F. v. Lohmann. 2003] F.v.Lohmann. *IAAL-: Peer-to-Peer File Sharing and Copyright Law after Napster.* Whitepaper, January, 2003

[G. Hartvigsen. 1998] G. Hartvigsen. *The researcher's Handbook.* Norwegian Academic Press (In Norwegian), Kristiansand, 1998

[Gnutella] *Gnutella*, Homepage (2003). http://www.gnutella.com

[Grokster] *Grokster,* Homepage (2003). http://www.grokster.com/

[H. Balakrishnan et al. 2003] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica. *Looking Up Data in P2P Systems.* Communication of the ACM. Vol. 46, no. 2 February 2003

[H. Hildrum et al. 2002] H. Hildrum, J. D. Kubiatowicz, S. Rao, B. Y. Zhao. *Distributed Object Location in a Dynamic Network.* In Proc. of ACM Symposium on Parallel Algorithms and Architectures, Winnipeg, Canada, August 2002

[I. Stoica et al. 2001] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, H. Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* In Proc. of the SIGCOMM '01, San Diego, California, USA, August 2001.

[KaZaA] *KaZaA*, Homepage (2003). http://www.kazaa.com/us/index.htm

[Konspire] *Konspire*, Homepage (2003). http://konspire.sourceforge.net

[Morpheus] *Morpheus*, Homepage (2003). http://www.morpheus.com/

[Napster] *Napster,* Homepage (2003). http://www.napster.com/

[Newsmonster] *Newsmonster*, Homepage (2003). http://www.newsmonster.org

[Oxygen] *Oxygen*, Homepage (2003). http://oxygen.lcs.mit.edu/

[P. Golle et al. 2001] P. Golle, K. L. Brown, I. Mironov. *Incentives for Sharing in Peer-to-Peer Networks.* In Proc. of the Third ACM Conference on Electronic Commerce, Tampa, Florida, USA, October 2001

[P. J. Denning. 1989] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, P. R. Young. *COMPUTING AS A DISCIPLINE*. Communications of the ACM, Vol. 32, No. 1, pages 9-23, January 1989

[Python] *Python*, Homepage (2003). http://www.Python.org

[Q. Lv et al. 2002] Q. Lv, P. Cao, E. Choen, K. Li, S. Shenker. *Search and Replication in Unstructured Peer-to-Peer Networks.* In Proc. 16th Annual ACM Int'l Conference on Supercomputing, New York, USA, June 2002

[RIAA] *Recording Industry Association of America*, Homepage (2003). http://www.riaa.com/

[Screensaver - Lifesaver] *Screensaver-Lifesaver*. Homepage (2003). http://www.chem.ox.ac.uk/curecancer.html

[SETI@home] *SETI@home*, Homepage (2003). http://setiathome.ssl.berkeley.edu/

[S. Ratnasamy et al. 2001] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. *A scalable content addressable network.* In Proc. of the 2001 ACM SIGCOMM Conference, San Diego, California, USA, August 2001

[S. Robertson, J. Robertson. 1999] S. Robertson, J. Robertson. *Mastering the Requirements Process.* Addison Wesley, pages 136-164, Great Britain, August 1999

[Stumbleupon] *Stumbleupon*, Homepage (2003). http://www.stumbleupon.com

[WAIF] WAIF (Wide Area Information Filtering), Homepage (2003). http://www.waif.cs.uit.no/

[WinMX] *WinMX*, Homepage (2003). http://www.winmx.com/

Appendix A

Appendix A: p-SARS, code listing

The p-SARS code consists of a total of 3078 lines where 1219 of these are non-commenting source statements (NCSS).

<p-SARS/OWN_constants.py>

```
# The p-SARS System (peer-to-peer search engine)
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: OWN_constants.py

# File to define numerical global constants
# to be used by our system

# RETURN VALUES
false = 0
true = 1
duplicate = 2
new_topic = 3
del_topic = 4

# EVENT TYPES
TYPE_SEARCH = 'search'
TYPE_DEBUG = 'debug'
TYPE_GOSSIP = 'gossip'
TYPE_RESULT = 'result'
TYPE_SYSTEM = 'system'
TYPE_BOOTSTRAP_REG = 'reg'
TYPE_FIND_NEIGHBOURS = 'find'
TYPE_FETCH_PROFILE = 'get_profiles'

###### DEFINING THE CONSTANTS ######

# Max size of queues meassured in items
QUEUE_MAXSIZE = 1

# Max time to wait on a TCP connection (socket)
MAX_WAIT = 100

# The address to the central bootstrap server
BOOTSTRAP_SERVER_IP = '129.242.13.22'

# The port where the bootstrap server listens
BOOTSTRAP_SERVER_PORT = 1212

# The maximum nodes kept at the bootstrap server
MAX_NUMBER_OF_NODES_IN_BOOTSTRAP_LIST = 100

# Buffer size on tcp connections
TCP_BUFF_SIZE = 4000

# Number of connections accepted on tcp server socket. Queue!
TCP_NUM_CONNECTIONS = 10000

# If DEBUG = 1, send debug info to debug module.
DEBUG = 0
# The address to the debug module
DEBUG_ADDRESS = ('129.242.13.22',9865)

# The number of nodes each node should strive to be connected too
# The neighbourhood set!!
NUMBER_OF_NEIGHBOURS_WANTED = 9
```

```
# The number of seconds before running system update. (System thread)
SECONDS_BEFORE_SYSTEM_UPDATE = 20

# The number of seconds between each ping/pong message! This will be tested only each
# system_update
SECONDS_BEFORE_PING = 40

# The number of seconds to wait on
# results before sending response to WRS client!
SECONDS_BEFORE_RESULT_UPDATE = 10

# The number of seconds to wait for a search to become satisfied.
SECONDS_TO_WAIT_ON_RESULTS = 20

# The number of seconds between each topic update!
SECONDS_BEFORE_TOPIC_UPDATE = 60*30

# The probability that a neighbor should be sendt gossip.
PROBABILITY_SEND_GOSSIP = 35

# TTL for gossips
TTL_GOSSIP = 5

# TTL for search
TTL_SEARCH = 40

# The number of times we should try to send an event over TCP/IP
#, before deeming the node dead!
NUMBER_OF_REPEAT = 20

# Number of tests to run on a throughput test
THROUGHPUT_TESTING = 5000
```

<p-SARS/OWN_math.py>

```
# The p-SARS System (peer-to-peer search engine)
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: OWN_math.py

#import own libs
from OWN_constants import *

#import standard libs
import random

# The class containing math utils
class math_utils:

    # The init function
    def __init__(self):
        pass

    # Function to deside if system should continue gossip.
    # Args:
    # probability = Integer from 0 to 100, where the int represents the probability in
%
    # Returns true if we should continue gossip, false otherwise
    # Note: I'm not sure how good python's random generator is.
    def should_i_gossip(self, probability):

        # Fetch a random number N where a <= N >= b
        # and a = 0, b = 99
        num = random.randint(0,99)

        # Decide if we should continue gossip based on supplemented
        # probability!
        if num < probability:
            # Continue gossip
            #print 'Gossip'
            return true

        else:
            # Stop gossiping
            #print 'no gossip'
            return false

    # Function to generate a random number 1 - max_num, including
    # 1 and max_num
    # Args:
    # max_num = The maximal number generated
    # Returns the random number generated
    def generate_random(self, max_num):
        # Return random number
        return random.randint(1,max_num)
```

<p-SARS/OWN_network.py>

```
# The p-SARS System (peer-to-peer search engine)
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: OWN_network.py

# import own libs
from OWN_constants import *
import OWN_math

# import standard libs
from socket import *
import pickle
import select
import threading
import time

# Class containing network utils
class network_utils:

    # The init function
    def __init__(self):
        pass

    # Function to fetch the ip address of local node
    # Returns the ip address
    def get_local_ip(self):
        return gethostbyname(gethostname())

    # Open client TCP/IP connection
    # Args:
    # address = A tuple : ('address',port)
    # Returns the connection, or false if error occured
    def open_client_tcp(self, address):
        try:
            c_soc = socket(AF_INET, SOCK_STREAM)
            #soc.setblocking(0)
            c_soc.connect(address)

        except Exception,why:
            # Exception, return false
            return false

        # Return client socket
        return c_soc


    # Open server TCP/IP connection
    # Args:
    # address = A tuple : ('address',port)
    # bufsize = size of msg buffer
    # num_conn = alloved number of connections waiting (simultaniously)
    # Returns server
    def open_server_tcp(self,address,bufsize,num_conn):
        try:
            s_soc = socket(AF_INET,SOCK_STREAM)
            s_soc.bind(address)
            s_soc.listen(num_conn)
        except Exception,why:
            # Exception, return false
            print 'Error creating server socket. Could not bind to address'
            return false

        # Return server socket
        return s_soc

    # Send via TCP/IP
    # Args:
    # msg = The message to send
    # conn = TCP connection
    # timeout = time to wait on socket until it gets writeable
    #           Not implemented yet!
    # Returns true if ok, false otherwise
    def send_tcp(self, msg, conn, timeout):

        # Test to see if socket is writeable
        # (rd,wr,ex) = select.select([conn],[],[],timeout)

        #if len(wr) != 0:
        # Socket is writeable
        try:
            # Send message
            conn.send(msg)

        except Exception,why:
            # Exception, return false
            return false

        #All ok
        return true


    # Receive via TCP/IP
    # Args:
    # conn = TCP connection
    # buffer = Size of package to receive
    # timeout = How long to wait on data from host
    # Returns the message received, or false if error occured
    def receive_tcp(self, conn, buffer, timeout):

        #test to see if socket is readable
        #Use the select call
        (rd,wr,ex) = select.select([conn],[],[],timeout)

        if len(rd) != 0:
            #socket is readable
            try:
                msg = conn.recv(buffer)
                return msg

            except Exception,why:
                print 'Error receiving : ',why

        # Socket not readable
        print 'Timeout on socket.'
        return false


    # Close Connection
    # Args:
    # conn = connection
    # Returns true if ok, false otherwise
    def close_conn(self,conn):
        try:
            conn.close()
        except Exception, why:
            # Exception, return false
            return false

        # All ok
        return true

    # Pack message before sending
    # Args:
```

```
    # msg = message to pack
    # Returns the packed message or an error (false)
    def pack(self,msg):
        try:
            # unpack message
            message = pickle.dumps(msg)
        except Exception, why:
            # Failed packing, return false
            return false

        # Return packed message
        return message

    # Unpack message received
    # Args:
    # msg = message to unpack
    # Retruns the unpacked message or an error (false)
    def unpack(self,msg):
        try:
            message = pickle.loads(msg)
        except Exception, why:
            # Failed unpacking, return false
            return false

        # Return unpacked message
        return message


# Class containing xml_rpc functions!
# Not implemented yet
class xml_rpc_utils:

    # The init function
    def __init__(self):
        pass


# Thread for sending tcp ip messages
# If error in sending event NUMBER_OF_REPEAT times the event is reinserted
# into the input queue with the error field set to true
class send_tcp_t(threading.Thread):

    # The init function
    # Args:
    # event_data = The event to be sendt
    # queues = The set of system queues
    def __init__(self, event_data, queues):
        # init thread
        threading.Thread.__init__(self)
        # the event to be sent
        self.event = event_data
        # Create instance of network class
        self.net = network_utils()
        # The input queue
        self.queues = queues


    def run(self):

        # We set sent to false
        sent = false

        # Repeate the sending procedure until it succeeds or
        # we have tried NUMBER_OF_REPEAT times.
        for element in range(NUMBER_OF_REPEAT):

            # Create client socket
            c_soc = self.net.open_client_tcp((self.event['to'][0],
self.event['to'][1]))

            if c_soc == false:
                # If no socket returned, start all over
                # Error connecting to client
                continue
            else:
                # pack message
                pack = self.net.pack(self.event)

                if not pack == false:

                    # send message
                    if not self.net.send_tcp(pack, c_soc, MAX_WAIT):
                        # Failed sending, try again
                        self.net.close_conn(c_soc)
                        continue

                    else:
                        # succeded sending
                        sent = true
                        self.net.close_conn(c_soc)
                        # end for loop
                        break

                else:
                    # Packing failed, Try all over
                    self.net.close_conn(c_soc)
                    continue

        if not sent:
            # Error occured. Set error, and enqueue message to input queue
            self.event['error'] = true
            self.queues.input_q.put(self.event)
            print 'MESSAGE SENDING FAILED!!!!'


# Thread for sending xml-rpc messages.
# Not implemented yet
class send_xml_rpc_t(threading.Thread):

    # The init function
    # Args:
    # event_data = the event to send
    # queues = The system queues
    def __init__(self, event_data, queues):
        # init thread
        threading.Thread.__init__(self)
        # the event to be sent
        self.event = event_data
        # Create instance of network class
        self.net = xml_rpc_utils()
        # The set of queues
        self.queues = queues

    def run(self):
        while true:
            time.sleep(10)
            print 'XML_SENDING Thread started'


# The thread listening for incomming data! (TCP connection)
class listen_t(threading.Thread):

    # Init thread
    # Args:
    # addr = The listen modules address
    # queues: The set of system queues
    def __init__(self, addr, queues):
        # init thread
        threading.Thread.__init__(self)

        # Create instance of the network utils class
        self.net = network_utils()

        # Init some network variables
        self.NUM_CONN = TCP_NUM_CONNECTIONS
        self.ADDR = addr

        # Hold on to the system queues
        self.queues = queues
    # The body of the thread!
    def run(self):

        print 'Event listen thread started..'
        # Fetch server socket
        s_soc = self.net.open_server_tcp(self.ADDR, TCP_BUFF_SIZE, self.NUM_CONN)

        if s_soc == false:
            print 'no socket returned'

        else:
            while true:
                # Accept incomming connection
                c_soc, addr = s_soc.accept()

                # Spawn off thread to handle this
                process_thread = process_incomming_data(c_soc, self.queues)
                process_thread.start()


# Thread to receive, unpack and enqueue the event
class process_incomming_data(threading.Thread):

    # The init functions
    # Args:
    # c_soc = The socket connecting us to another p-SARS node
    # queues = The system queues
    # Returns nothing
    def __init__(self,c_soc, queues):
        # init thread
        threading.Thread.__init__(self)

        # Keep reference to queues
        self.queues = queues
        # Keep reference to socket
        self.c_soc = c_soc

        # Create instance of network utils
        self.net = network_utils()

    def run(self):
        # Fetch data from client
        data = self.net.receive_tcp(self.c_soc,TCP_BUFF_SIZE,MAX_WAIT)

        # If data, enqueue it!
        if data != false:

            #Unpack data (pickle)
            data = self.net.unpack(data)

            # if unpack ok, enqueue data. If not ok, its just discarded
            if data != false:
                # Insert in input queue. This call will block if
                # the queue is full!
                self.queues.input_q.put(data)

        # Try to close socket. If error, print error
        if self.net.close_conn(self.c_soc) != true:
            print 'Failed closing socket'


# Thread to receive, unpack and enqueue the event
# A modified version above that holds persistant tcp/ip
# connections. (Until the connection times out defined by the
# MAX_WAIT parameter!)
class process_incomming_data_tmp(threading.Thread):

    # The init functions
    # Args:
    # c_soc = The socket connecting us to another p-SARS node
    # queues = The system queues
    # Returns nothing
    def __init__(self,c_soc, queues):
        # init thread
        threading.Thread.__init__(self)

        # Keep reference to queues
        self.queues = queues
        # Keep reference to socket
        self.c_soc = c_soc

        # Create instance of network utils
        self.net = network_utils()

    def run(self):
        first = false
        while(true):
            # Fetch data from client
            data = self.net.receive_tcp(self.c_soc,TCP_BUFF_SIZE,MAX_WAIT)

            # If data, enqueue it!
            if data != false:

                #Unpack data (pickle)
                data = self.net.unpack(data)

                # if unpack ok, enqueue data. If not ok, its just discarded
                if data != false:
                    if first:
                        first = false
                        data['time'] = int(time.time())

                    # Insert in input queue. This call will block if
                    # the queue is full!
                    self.queues.input_q.put(data)

            else:
                break

        # Try to close socket. If error, print error
        if self.net.close_conn(self.c_soc) != true:
            print 'Failed closing socket'

        print 'Persistant connection dropped on time_out!!!'



# A modified version of the listen module used for
# throughput testing
class listen_tmp(threading.Thread):

    # The init function
    # Args:
    # addr = The listen modules address
    # queues = The system queues
    def __init__(self, addr, queues):
        # init thread
        threading.Thread.__init__(self)
        # Init nerwork stuff
        self.net = network_utils()
        self.NUM_CONN = TCP_NUM_CONNECTIONS
        self.ADDR = addr
        self.BUFSIZE = TCP_BUFF_SIZE
        self.queues = queues



    # The body of the thread!
    def run(self):
        print 'Started spaming node :) %d ' % int(time.time())
        time.sleep(5)
        start = int(time.time())



        # A for loop just pumping search events into the
        # system input queue. This is to test how many
        # search event a node can process pr. second
        for i in range(THROUGHPUT_TESTING):
            # Insert in queue
            #print event
            # Create internal search event
            event = {}
            event['to'] = ('129.242.13.22',9090)
            event['type'] = TYPE_SEARCH
            event['topic'] = 'gibberish'
            event['reply_addr'] = ('129.242.13.22',9034)
            event['happy'] = 20
            # Perform local search
            event['local'] = false
            event['error'] = false
            event['ttl'] = 2
            event['visited'] = []
            event['hint'] = false
            event['time'] = start
            # The supernode the query originated
            event['from'] = ('129.242.13.22',9094)

            self.queues.input_q.put(event)

        print 'Finito spaming!'


# Receive incomming Asynchron XML RPC.
# Parse data, create event and put it in inputqueue
# Not implemented yet
class xmlrpc_listener_t(threading.Thread):

    # The init function
    # Args:
    # addr_o = The set of system addresses
    # queues = The set of system queues
    def __init__(self, addr_o, queues):
        # init thread
        threading.Thread.__init__(self)
        # Make arguments accessable
        self.queues = queues

        self.addr_o = addr_o

    def run(self):
        print 'XML-RPC listen thread started'
        while true:
            # Fetch rpc's and create events..
            time.sleep(1000)
            pass


# Pusher thread. Thread maintaining the output queue and
# starts the right thread to push event out (based on it's content)
class pusher_t(threading.Thread):

    # The init function
    # Args:
    # queues = The event queues
    # addr_o = The set of system addresses
    def __init__(self, queues, addr_o):
        # init thread
        threading.Thread.__init__(self)

        # Set of queues
        self.queues = queues
        # Make instance of network utils
        self.net = network_utils()

    def run(self):
        num = 0

        while true:
            # Fetch outgoing event
            event = self.queues.output_q.get()

            # Start thread to send the event
            send_thread = send_tcp_t(event,self.queues)
            send_thread.start()


# A modified pusher module used in throughput testing.
class pusher_tmp(threading.Thread):

    # The init function
    # Args:
    # queues = The event queues
    # addr_o = The set of system addresses
    def __init__(self, queues, addr_o):
        # init thread
        threading.Thread.__init__(self)

        # Set of queues
        self.queues = queues
        # Make instance of network utils
        self.net = network_utils()

    def run(self):
        num = 0
        s = 0
        print 'Modified pusher created!'
        while true:
            # Fetch outgoing event
            event = self.queues.output_q.get()

            if 'time' in event:
                num += 1
                if num == 1:
                    s = int(time.time())

                if num == THROUGHPUT_TESTING:
        print 'Number: %d, time: %d' % (num, int(time.time()))
```

<p-SARS/bootstrap_server.py>

```
# The p-SARS System (peer-to-peer search engine)
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: bootstrap_server.py

# import own libs
from OWN_constants import *
import OWN_network

# The class containing the bootstrap server
class boot_server:

    # The init function
    def __init__(self):
        pass

    # The main method for the bootstrap server.
    def main(self):
        print 'Starting bootstrap server!'

        # Create instance of the network_utils class
        net = OWN_network.network_utils()

        # The address where the bootstrap server will run
        ADDR = (BOOTSTRAP_SERVER_IP,BOOTSTRAP_SERVER_PORT)

        # Create server socket
        s_soc = net.open_server_tcp(ADDR,1024,100)

        # Create list containing the last nodes that did a bootstrap.
        list = []

        # Needed for throughput testing to trick the p-SARS
        # node to think it has a neighbour so that all events
        # are propagated to this node
        # list.append(('129.242.13.187',9080))

        # Test if we indeed got the server socket
        if s_soc == 0:
            # Failes setting up socket
            print 'no socket returned'

        else:
            # Start bootstrap server
            while 1:

                # Wait on client
                c_soc, query_addr = s_soc.accept()

                # Fetch data from client
                data = net.receive_tcp(c_soc,1024,MAX_WAIT)

                # Test if data is received!
                if data != false:

                    # Unpack data (pickle)
                    data = net.unpack(data)

                    # Test if unpacked data is ok
                    if data != false:
                        # Process the client request
                        # If reg, register node
                        if data['type'] == TYPE_BOOTSTRAP_REG:
                            # If duplicate do not insert in list
                            if data['addr'] not in list:
                                list.append(data['addr'])

                            # Test if list is to long. If it is,
                            # remove the oldest element
                            if len(list) > MAX_NUMBER_OF_NODES_IN_BOOTSTRAP_LIST:
                                tmp = list[0]
                                list.remove(tmp)


                        # Create response to client
                        res = {}
                        res['nodes'] = list
                        # pack it
                        res = net.pack(res)
                        # Send replay to the requesting p-SARS node
                        # The replay contains the node list
                        if not net.send_tcp(res,c_soc,100):
                            # Sending failed
                            print 'Message could not be sendt'

                # Try to close socket. If error, print error
                if net.close_conn(c_soc) != true:
                    print 'Failed closing socket'

# Start the bootstrap server by calling main!
if __name__ == '__main__':
    # Make instance of boot_server
    boot = boot_server()
    boot.main()
```

<p-SARS/start.py>

```
# The p-SARS System (peer-to-peer search engine)
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: start.py

# Import own libs
import OWN_system
import OWN_network

# import standard libs
import sys
```

```
# A class working as a container for important system
# addresses
class addr:
    # Init method
    # Args:
    # ip_tcp = The IP address where the system listens for incomming events
    # port_tcp = The port where the system listens for incomming events
    # ip_rpc = The IP address where the system listens for RPC communication
    # port_rpc = The port where the system listens for RPC communication
    # ip_profile = The IP address for the topic server
    # port_profile = The port for the topic server
    def __init__(self, ip_tcp, port_tcp, ip_rpc, port_rpc, ip_profile, port_profile):
        self.ip_tcp = ip_tcp
        self.port_tcp = int(port_tcp)
        self.ip_rpc = ip_rpc
        self.port_rpc = int(port_rpc)
        self.ip_profile = ip_profile
        self.port_profile = int(port_profile)


    # Function to read all addresses
    # Returns all addresses as a tuple
    def read(self):
        return (self.ip_tcp, self.port_tcp, self.ip_rpc, self.port_rpc,
self.ip_profile, self.port_profile)

    # Function to fetch the TCP/IP listen address
    # Returns only the TCP/IP listen address
    def read_listen_tcp(self):
        return (self.ip_tcp,self.port_tcp)

    # Function to fetch the XML-RPC listen address
    # Returns only the XML-RPC listen address
    def read_listen_rpc(self):
        return (self.ip_rpc,self.port_rpc)

    # Function to fetch the profile server address
    # Returns only the profile server address
    def read_profile_server_address(self):
        return (self.ip_profile,self.port_profile)


# This is the main method. It fetches the needed
# arguments and starts up the system
def main():

    if len(sys.argv) != 5:
        # To few arguments given. Print usage
        print 'USAGE : python start.py [port_tcp] [port_rpc] [ip_profile]
[port_profile]'

    else:
        # Create instance of network utils
        net = OWN_network.network_utils()
        local_ip = net.get_local_ip()

        # Create instance of address object
        addr_o = addr(local_ip, sys.argv[1], local_ip, sys.argv[2], sys.argv[3],
sys.argv[4])

        # Start system thread. The system thread initiates bootstrap and continously
        # maintains the neighbour list. It also initiates event processing!
        search_mechanism_t = OWN_system.system_t(addr_o)
        search_mechanism_t.start()

# Starting the program by calling main!
if __name__ == '__main__':
   # Call main
   main()
```

<p-SARS/OWN_system.py>

```
# The p-SARS System (peer-to-peer search engine)
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: OWN_system.py

# import own libs
from OWN_constants import *
import OWN_math
import OWN_network
import OWN_profiles
import OWN_event_handler

# import standard libs
import mutex
import time
import Queue
import copy
import threading


# The bootstrap class.
# Contains the bootstrap method.
class bootstrap:

    # The init function
    # Args:
    # addr_o = The set of addresses needed by the system
    def __init__(self, addr_o):
        # Make instance of OWN_network
        self.net = OWN_network.network_utils()

        # Fetch the address to the listen module
        self.ADDR = addr_o.read_listen_tcp()

    # Function to do bootstrap or only fetch neighbours.
    # This is done by contacting the bootstrap server
    # Args:
    # neighbours_o = the neighbor set
    # do_reg = A boolean telling us if we should register or only
    #          fetch neighbours
    # Returns number of additional nodes connected if any..
    def bootstrap(self, neighbours_o, do_reg):
        num_connected = 0

        if do_reg:
            # Contact central server and register as supernode
            # create request
            req = {}
            req['type'] = TYPE_BOOTSTRAP_REG
            # The address where the listen thread listens, and where the events
            # to this node should be sent
            req['addr'] = self.ADDR

        else:
            # Only request the nodes registered at the bootstrap server
            req = {}
            req['type'] = TYPE_FIND_NEIGHBOURS



        # Pack request (pickle)
        req = self.net.pack(req)

        # create client socket
        c_soc = self.net.open_client_tcp((BOOTSTRAP_SERVER_IP, BOOTSTRAP_SERVER_PORT))

        if c_soc != false:
            # send message
            self.net.send_tcp(req, c_soc, MAX_WAIT)

            # In response of register get the nodes registered!
            res = self.net.receive_tcp(c_soc, TCP_BUFF_SIZE, MAX_WAIT)

            # Close connection
            self.net.close_conn(c_soc)

            # unpack data
            res = self.net.unpack(res)

            # Get current time as a time stamp. (An integer)
            timestamp = int(time.time())

            # Test if data is unpacked ok
            if res != false:
                # Traverse list and try to insert the node
                for node in res['nodes']:

                    # If the desired number of neighbours is reached. Break out
                    # of for loop
                    if num_connected >= NUMBER_OF_NEIGHBOURS_WANTED:
                        break

                    # Try to insert node, but not if it is our selves
                    if node != self.ADDR:
                        tmp = neighbours_o.append(node)
                        if tmp == true:
                            # node was inserted!
                            num_connected = num_connected + 1

            else:
                # Could not unpack data. No nodes fetched
                print 'Unpacking event from bootserver failed!'
                return false

        else:
            # Could not create client socket.
            print 'Could not connect to bootserver. Bootstrap FAILED'
            return false

        # Return number of nodes inserted into the neighbour set
        return num_connected


# The system thread. Keeps track on the neighborhood set.
# Removes persumed dead nodes and pings nodes when they should be pinged
class system_t(threading.Thread):

    # The init function
    # Args:
    # addr_o = The system address set
    def __init__(self, addr_o):
        # init thread
        threading.Thread.__init__(self)

        # Hold on to the addresses
        self.addr_o = addr_o

        # Create the sets needed. Neighbour set, topic set, hint set and the
        # pending search set.
        # (Encapsulated in one object called sets.)
        self.sets = sets()

        # Make instance of bootstrap class
        self.bootstrap = bootstrap(addr_o)

        # Create the set of all event queues!
        # (Encapsulated in one object called queues.)
        self.queues = event_queues()

    def run(self):
        # Do bootstrap
        connected_nodes = self.bootstrap.bootstrap(self.sets.neighbours_o, true)

        print 'nodes connected during bootstrap: %d' % connected_nodes

        # Start the process of periodically updating the topic set
        update_topic_set_t = OWN_profiles.update_topic_set_t(self.queues, self.sets,
self.addr_o)
        update_topic_set_t.start()
        # Create and start incomming event handling thread
        t = OWN_event_handler.event_handler_t(self.queues, self.sets, self.addr_o)
        t.start()

        # To initialize the local hint list with data from neighbours, we
        # need to request their hint sets and topic sets!
        # Create event
        event = {}
        event['type'] = TYPE_GOSSIP
        event['from'] = self.addr_o.read_listen_tcp()
        # This is a get event!
        event['get'] = true
        # To support error on sending
        event['ttl'] = 1
        event['push'] = false
        event['error'] = false
        # Send event to all neighbours of this node!
        # If all neighbours are started at once, like we do during testing
        # we should put a delay here or else the gossip events wont have time
        # to propagate through the net, populating the hint sets.
        # time.sleep(10)
        self.sets.neighbours_o.send_event_to_all_neighbours(event,self.queues)


        # Continue to maintain neighbor set
        while true:
            # Print all sets on each system update for debugging!!
            # hint_set = self.sets.hint_o.read_all()
            # topic_set = self.sets.topic_o.read_all()
            # neighbour_set, not_important = self.sets.neighbours_o.read()

            #print '############ HINT #############'
            #print hint_set
            #print '################################'
            # print '\n############ TOPIC ############'
            # print topic_set
            # print '################################'
            # print '\n######### NEIGHBOUR ###########'
            # print neighbour_set
            # print '################################'


            # Sleep a while before updating system
            time.sleep(SECONDS_BEFORE_SYSTEM_UPDATE)

            # Fetch neighbour list.
            list, dict = self.sets.neighbours_o.read()

            # Test if some nodes should be removed. If no pong message has been
received
            # since last system-update --> remove (pinged == true)
            del_list = []
            for element in list:
                pinged = dict[element][2]

                if pinged == true:
                    # Record which nodes to delete. If we delete them directly, the
                    # traversal of the list will be incorrect
                    del_list.append(element)

            # Remove node from neighbor set, and remove hints associated with
            # this node
            for element in del_list:
                # Remove neighbour
                if self.sets.neighbours_o.remove(element):
                    # Debug
                    # print 'Node removed from neighborhood set.'
                    # print element
                    pass

                # Remove hints
                if self.sets.hint_o.remove(element):
                    # Debug
                    # print 'Hints removed from hint set'
                    # print element
                    pass

            # Fetch current time
            timestamp = int(time.time())
            # Refresh neighbour list.
            list, dict = self.sets.neighbours_o.read()

            for element in list:
                last_ping = dict[element][1]

                # Test if we should send a ping message
                if timestamp - last_ping > SECONDS_BEFORE_PING and dict[element][2] ==
false:
                    # We must ping node, create message
                    ping = {}
                    ping['to'] = dict[element][0]
                    ping['from'] = self.addr_o.read_listen_tcp()
                    ping['type'] = TYPE_SYSTEM
                    ping['topic'] = 'ping'
                    ping['error'] = false

                    # Update node as pinged
                    if self.sets.neighbours_o.update_as_pinged(element, true) == true:
                        # print 'Neighbor updated as pinged'
                        # Enqueue message only if we could updated node as pinged
                        self.queues.output_q.put(ping)

            # If not enough neighbors, try to fetch more by asking the
            # bootstrap server
            if len(list) < NUMBER_OF_NEIGHBOURS_WANTED:
                # Fetch more neighbours
                num = self.bootstrap.bootstrap(self.sets.neighbours_o, false)

                # Print the number of neighbours added, if any
                if num > 0:
                    print 'NEW NEIGHBOURS FOUND %d' % num



# Helper functions to protect shared variables
# The locking mechanism is currently spin-locks.
##########################################################

# A class to create and hold the event queues needed by
# the system
class event_queues:

    # Init queues
    def __init__(self):
        # Create output queue
        self.output_q = Queue.Queue(QUEUE_MAXSIZE)
        # Create input queue
        self.input_q = Queue.Queue(QUEUE_MAXSIZE)
        # Create event queues
        self.search_q = Queue.Queue(QUEUE_MAXSIZE) # The queue for incomming search
events
        self.gossip_q = Queue.Queue(QUEUE_MAXSIZE) # The queue for incomming gossip
events
        self.result_q = Queue.Queue(QUEUE_MAXSIZE) # The queue for incomming result
events
        self.system_q = Queue.Queue(QUEUE_MAXSIZE) # The queue for incomming system
events

# A class to hold the instances of the sets needed in the program
class sets:

    # Init sets
    def __init__(self):
        # Create instance of neigbour set utils
        self.neighbours_o = neighbours()
        # Create instance of topic set utils
        self.topic_o = topics()
        # Create instance of hint set utils
        self.hint_o = hints()
        # Create instance of pending_search utils
        self.pending_o = pending_search()

# Class to controll the access to the neighbour set
class neighbours:

    # The init function
    def __init__(self):

        # The neighbour set
        self.neighbours = []
        # The corresponding ping time
        self.ping_time = {}

        # Initialize the mutex
        self.m = mutex.mutex()
        # Initialize math utils
        self.math = OWN_math.math_utils()

    # Function to accquire mutex
    def spin_lock(self):
        while not self.m.testandset():
            pass

    # Function to relase mutex
    def unlock(self):
        self.m.unlock()


    # Set update time!
    # Args:
    # node = The node connected to this ping-time
    # time = The timestamp for last ping
    # returns true when value set, false if node not in neighbour set
    def set_ping_time(self, node, time):
        # Set lock
        self.spin_lock()

        # Test if node exists. If it does, insert timestamp
        if node in self.neighbours:
            self.ping_time[node] = [node, time, false]
            # unlock mutex
            self.unlock()
            return true

        else:
            # node does not exist
            # unlock mutex
            self.unlock()
            return false

    # Set node as pinged or unpinged!
    # Args:
    # node = the node to be updated in ping_time dictionary
    # pinged = true if node is pinged, false if node doesn't have pending ping message
    # returns true when done
    def update_as_pinged(self, node, pinged):

        if pinged:
            # Set node as pinged
            self.ping_time[node][2] = true
            return true

        else:
            # set node as not pinged
            self.ping_time[node][2] = false
            return true

    # Function to fetch the neighbour set, and the ping times
    # Returns the whole set
    def read(self):
        # Set lock
        self.spin_lock()

        # Return copies of the sets in question!!
        res = (copy.deepcopy(self.neighbours), self.ping_time.copy())

        # unlock mutex
        self.unlock()
        return res

    # Function to add a new neighbor. Neighbor is added if
    # its not in the neighbour set already, and if
    # its not already full.
    # Args:
    # node = the new super-node to add to set
    # Returns duplicate if exists, true if inserted and false if
    # we already have enough neighbors
    def append(self, node):
        # Set lock
        self.spin_lock()

        # Test if we have enough neighbors already
        num = len(self.neighbours)

        if num <= NUMBER_OF_NEIGHBOURS_WANTED:

            # Only add node if its not already added
            if node not in self.neighbours:
                # Add as neighbour
                self.neighbours.append(node)
                # Set ping-time
                self.ping_time[node] = [node,int(time.time()),false]

                # Unlock mutex
                self.unlock()
                return true

            else:
                # We have a duplicate
                # unlock mutex
                self.unlock()
                return duplicate

        else:
            # We have enough neighbors
            # unlock mutex
            self.unlock()
            return false

    # Function to remove a node from the neighbourhood set.
    # Also remove the entry in the ping_time dictionary
    # Args:
    # node = the node to remove
    # Returns true if ok, false otherwise
    def remove(self, node):
        # Set lock
        self.spin_lock()

        try:
            # Remove it from neighbor set
            self.neighbours.remove(node)
            # Remove the ping time entry
            del self.ping_time[node]
        except Exception, why:

            # unlock mutex
            self.unlock()
            return false

        # unlock mutex
        self.unlock()
        return true


    # Function to count number of neighbours in set
    # Returns number of nodes in set
    def count(self):
        # Set lock
        self.spin_lock()
        num = len(self.neighbours)

        # unlock mutex
        self.unlock()
        return num

    # Function to pick random neighbor to be used in Random Walk
    # Try to find a neighbour not visited, but if that fails --> pick
    # random among them all
    # Returns a random neighbour, or false if no neighbour is found
    def pick_random_neighbour(self, visited):
        # Set lock
        self.spin_lock()

        # Fetch number of neighbours
        num = len(self.neighbours)

        if num == 0:
            # unlock mutex
            self.unlock()
            return false

        else:
            # Create list to hold neighbours not already
            # visited by the query
            list = []
            # Traverse neighbours. Fetch those not already
            # visited
            for element in self.neighbours:
                if element not in visited:
                    # Node not visited. We add it to the list
                    list.append(element)

            length = len(list)

            if length > 0:
                # There is still neighbours not visited
                # generate random number between 1 and length
                random_int = self.math.generate_random(length)

                # Fetch neighbour and return it
                neighbour = self.neighbours[random_int - 1]

                # unlock mutex
                self.unlock()
                return neighbour

            # All neighbours are visited.
            # Pick random node between all of them
            else:
                length = len(self.neighbours)

                # generate random number between 1 and num
                random_int = self.math.generate_random(length)

                # Fetch neighbour and return it
                neighbour = self.neighbours[random_int - 1]

                # unlock mutex
                self.unlock()
                return neighbour


    # Function to send a event to all neighbours
    # Args:
    # event = The event to send
    # queues = The system event queues
    # Returns true when done
    def send_event_to_all_neighbours(self, event, queues):
        # Set lock
        self.spin_lock()

        # Traverse all neighbours!
        for address in self.neighbours:
            tmp = event.copy()
            tmp['to'] = address
            # Enqueue event!
            queues.output_q.put(tmp)

        # Unlock
        self.unlock()
        return true


# Class to controll access to the topic set
class topics:

    # The init function
    def __init__(self):

        # Dictionary to hold existing topics and the publishers on each
        # topic (topic set)
        self.topics = {}

        # Dictionary to map the clients unique id (e-mail address) to it's
        # current ip/port!
        self.mapping = {}

        # Initializing the mutex
        self.m = mutex.mutex()

    # Function to accquire mutex
    def spin_lock(self):
        while not self.m.testandset():
            pass

    # Function to relase mutex
    def unlock(self):
        self.m.unlock()

    # Function to fetch the whole topic set
    # Returns the topic set
    def read_all(self):
        self.spin_lock()

        ret = self.topics.copy()

        self.unlock()
        return ret



    # Function to read who has a specific topic.
    # Args:
    # topic = The topic we are searching for
    # Returns the publishers if any, false if no one
    def read(self, topic):
        # Set lock
        self.spin_lock()

        list = []

        # See if the topic is registered.
        if topic in self.topics:
            # The topic is found. Return list with addresses
            # This means that we have to map id to e-mail addresses
            # before we return the list

            for item in self.topics[topic]:
                # Test if mapping is available
                if item in self.mapping:
                    list.append(self.mapping[item])

            # unlock mutex
            self.unlock()
            return list

        # Topic is not registered!
        else:
            # unlock mutex
            self.unlock()
            return false


    # Function to update the topics the clients of the WRS super-node
    # publishes on.
    # Args:
    # list = The list of publishers! Also contains what they publish on!
    # Returns two lists : new_topic_list and del_topic_list
    def update(self, list):

        # Set spin-lock
        self.spin_lock()

        # Create a few lists
        del_list = []
        publisher_list = []
        new_topic_list = []
        del_topic_list = []
        del_publisher_list = []


        # Traverse the publishers!
        for element in list:
            # Create replay address
            address = (element['return_address'],element['return_port'])
            id = element['email']

            # Save id's so we can find out if any has to be removed. Do we need
this???
            publisher_list.append(id)

            # create mapping if it does not exist
            if id not in self.mapping:
                self.mapping[id] = address

            # If mapping exist check that it hasn't changed.
            # If mapping changed, update with new value!
            else:
                if self.mapping[id] != address:
                    # update
                    self.mapping[id] = address

            # Traverse topics
            for topic in element['interests']:

                # Test if it is a new topic
                if topic not in self.topics.keys():
                    # We have a new topic. Record it
                    new_topic_list.append(topic)
                    # Create new entry in dictionary
                    self.topics[topic] = []
                    # Add node
                    self.topics[topic].append(id)

                # It's not a new topic
                else:
                    # Add node if it's not already added
                    if id not in self.topics[topic]:
                        self.topics[topic].append(id)


            # Traverse exsisting topics to see if this publisher
            # should be removed from one of them!
            for topic in self.topics.keys():
                if topic not in element['interests']:
                    # The node is not producing on this topic
                    # Check if the node is registered. If it is, remove it
                    if id in self.topics[topic]:
                        # Register it for removal afterwards!
                        # But ensure that it's not already registered for removal.
                        # This may occure if the same person is registered twice in
the
                        # WRS system
                    if (topic,id) not in del_list:
                        del_list.append((topic,id))


        # Traverse del_list and remove nodes that no longer produces on a topic!
        for element in del_list:
            # Remove item
            self.topics[element[0]].remove(element[1])
            # Is there still someone producing on topic, or should we remove
            # the topic as a whole??
            if not len(self.topics[element[0]]) > 0:
                # Remove topic
                del self.topics[element[0]]
                # Add it to the del_topic_list
                del_topic_list.append(element[0])

        # Traverse publisher list, and remove publishers that no longer exists
        for element in self.mapping:
            if element not in publisher_list:
                # Record for deleting
                del_publisher_list.append(self.mapping[element])
        # Delete
        for element in del_publisher_list:
            del self.mapping[element]


        # Debug, print topic set after update
        print '################ Topic set ##################'
        print self.topics
        print '#############################################'


        # unlock mutex
        self.unlock()
        return (new_topic_list, del_topic_list)


    # Function to map between id and address
    # Args:
    # id = the unique client id (e-mail address)
    # Returns the address if available, false otherwise
    def map_id_to_address(self, id):
        # Set lock
        self.spin_lock()

        # Test if mapping exist
        if id in self.mapping:

            # Fetch address
            address = self.mapping[id]

            # unlock mutex
            self.unlock()
            return address

        # unlock mutex
        self.unlock()
        # No mapping found!
        return false

    # Functions to count number of topics
    # Returns number of topics
    def count_topics(self):
        # Set lock
        self.spin_lock()

        num = len(self.topics)

        # unlock mutex
        self.unlock()

        return num


# Class to controll access to the hint set
```

```
class hints:

    # The init function
    def __init__(self):

        # The hint set
        self.hints = {}
        # Initializing the mutex
        self.m = mutex.mutex()
        # Initializing math utils
        self.math = OWN_math.math_utils()

    # Function to accquire mutex
    def spin_lock(self):
        while not self.m.testandset():
            pass

    # Function to relase mutex
    def unlock(self):
        self.m.unlock()

    # Function to read who might have a specific topic.
    # Args:
    # topic = The topic we are searching for
    # Returns the hints if any, false if no one
    def read(self, topic):
        # Set lock
        self.spin_lock()

        # See if the topic is registered.
        if topic in self.hints:
            # The topic is found. Return list
            list = copy.deepcopy(self.hints[topic])

            # unlock mutex
            self.unlock()
            return list

        # Topic is not registered!
        else:
            # unlock mutex
            self.unlock()
            return false

    # Function to read all hints saved locally
    # Returns entire hint dictionary!
    def read_all(self):
        # Set lock
        self.spin_lock()

        # Copy dictionary
        ret = self.hints.copy()

        # Unlock
        self.unlock()
        return ret

    # Function to add hints to hint set
    # Args:
    # topic = the topic where the hint should be registered under
    # node = The super-node to register
    # Returns duplicate if already registered, true otherwise
    def append(self, topic, node):
        # Set lock
        self.spin_lock()

        # Test if topic exists
        if topic in self.hints:

            # The topic exists, append node
            # But we need to test if the hint is already known
            if node in self.hints[topic]:
                # Gossip already known
                # unlock mutex
                self.unlock()
                return duplicate

            else:
                # Gossip not known
                self.hints[topic].append(node)
                # unlock mutex
                self.unlock()
                return true

        else:
            # The hint is under a new topic
            # Create list and append node
            self.hints[topic] = []
            self.hints[topic].append(node)

            # unlock mutex
            self.unlock()

            return true

    # Function to remove a node from hint set
    # Args:
    # node = The super-node to remove
    # Returns true when done
    def remove(self, node):
        # Set lock
        self.spin_lock()

        # Traverse all topics, if node there delete it!
        # Fetch all topics in dictionary
        topics = self.hints.keys()

        for element in topics:
            if node in self.hints[element]:
                # Node is present. We delete it
                self.hints[element].remove(node)

        # unlock mutex
        self.unlock()

        return true

    # Function to remove super-node from specific hint
    # Args:
    # topic = topic under which the super-node should be removed
    # node = The super node
    # Returns true when done or false if it does not exist
    def remove_one(self, topic, node):
        # Set lock
        self.spin_lock()

        # Test if topic exists
        topic_list = self.hints.keys()

        # if it really is there, remove it
        if topic in topic_list:

            if node in self.hints[topic]:
                # node found under topic, remove it
                self.hints[topic].remove(node)

            else:
                # node not found under topic
                # unlock mutex
                self.unlock()
                return false

        # If topic does not exist
        else:
            # unlock mutex
            self.unlock()
            return false

        # unlock mutex
        self.unlock()
        return true


    # Function to read the whole hint set
    # Returns the whole hint set
    def read_all(self):
        self.spin_lock()

        # Create copy of the hint set
        hint_copy = self.hints.copy()

        self.unlock()
        return hint_copy

    # Function to count the number of topics which we have
    # hints for
    # Returns number of topics
    def count(self):
        # Set lock
        self.spin_lock()

        num = len(self.hints)

        # unlock mutex
        self.unlock()

    # Function to pick random hint node to be used in Hint search, but
    # exclude nodes already searched by the query
    # Args:
    # topic = The topic we are searching on
    # visited = list of visited nodes
    # Returns a random node that has the topic in question if any, false otherwise
    def pick_random_hint_node(self, topic, visited):

        # Set lock
        self.spin_lock()

        # If topic exists, traverse hints and find those not already
        # visited
        if topic in self.hints:
            list = []
            for element in self.hints[topic]:
                if element not in visited:
                    # element not already visited! Add it to
                    # potential visit list!
                    list.append(element)

            length = len(list)

            if length > 0:
                # We still have hints not visited. Pick one
                # random and return it!
                random = self.math.generate_random(length)
                # Fetch node
                node = list[random - 1]

                # Unlock and return node
                self.unlock()
                return node

            else:
                # Unlock, and return false
                # All hints are already visited
                self.unlock()
                return false

        else:
            # No hint found on topic. Unlock and return false!
            self.unlock()
            return false


# Class to controll access for the pending search set
class pending_search:
    # The init function
    def __init__(self):

        # init dictionary over pending searches
        self.pending = {}
        # Create mutex
        self.m = mutex.mutex()


    # Function to accquire mutex
    def spin_lock(self):
        while not self.m.testandset():
            pass

    # Function to relase mutex
    def unlock(self):
        self.m.unlock()

    # Function to test if search is still pending
    # Args:
    # client = The search is registered under this client
    # topic = The topic of the search
    # Returns true if search is pending, false if not
    def search_pending(self, client, topic):
        # lock mutex
        self.spin_lock()

        if client in self.pending:
            if topic in self.pending[client]:
                # Search is pending
                # unlock mutex
                self.unlock()
                return true

        # unlock mutex
        self.unlock()
        return false

    # Function to create pending search
    # Args:
    # client = The client where the search should be registered under
    # topic = The topic of the search
    # happy = The number of hits needed before this search is satisfied
    # reply_addr = The address where the search should be returned when finished
    # Return true when done. Returns false if search already is pending!
    def create_pending_search(self, client, topic, happy, reply_addr):
        # lock mutex
        self.spin_lock()

         # Test if we have a pending search on this client
        if not client in self.pending:
            # Create it
            self.pending[client] = {}

        # Test if topic is registered on client. If not register
        if not topic in self.pending[client]:
            # Create it
            self.pending[client][topic]= {}
            # Create list to put results
            self.pending[client][topic]['results'] = []
            # Create time stamp
            self.pending[client][topic]['timestamp'] = int(time.time())
            # Create happy entry
            self.pending[client][topic]['happy'] = happy

        # Search is already pending!
        else:
            # unlock mutex
            self.unlock()
            return false

        # unlock mutex
        self.unlock()
        return true
    # Function to append results to a pending search
    # Args:
    # client = the search is registered under this client
    # topic = the search topic
    # results = the results to be added
    # Returns true when done, or false if search not pending!
    def append(self, client, topic, results):

        # lock mutex
        self.spin_lock()

        # Test if we have a pending search on this client
        if client in self.pending:
            if topic in self.pending[client]:
                # The search is active. Add results
                self.pending[client][topic]['results'] =
self.pending[client][topic]['results'] + results

                # Debug
                # print self.pending[client][topic]['results']

                # unlock mutex
                self.unlock()
                return true

        # Search does not exist anymore!
        # unlock mutex
        self.unlock()
        return false

    # Function to discover if search is satisfied
    # Args:
    # client = the search is registered under this client
    # topic = the topic on search
    # Returns true if satisfeid, false otherwise
    def happy(self, client, topic):

        # lock mutex
        self.spin_lock()

        # Test if we have a pending search on this client
        if client in self.pending:
            if topic in self.pending[client]:
                # The search is active. Count results
                num = len(self.pending[client][topic]['results'])

                if num >= self.pending[client][topic]['happy']:
                    #unlock mutex
                    self.unlock()
                    return true

        #unlock mutex
        self.unlock()
        return false


    # Function to remove pending search
    # Args:
    # client = The client
    # topic = The topic on search to be removed
    # Returns true if search removed, false otherwise
    def remove(self, client, topic):

        # lock mutex
        self.spin_lock()

        # Test if it exists!
        if client in self.pending:
            if topic in self.pending[client]:
                # Search exists, delete it
                del self.pending[client][topic]
                # Test if we should remove client to.
                # This is done as long as the client does not have any
                # other pending searches!
                if not len(self.pending[client]) > 0:
                    del self.pending[client]

                # Unlock mutex
                self.unlock()
                return true

            # The search topic did not exist.
            else:
                # Test to see if we should remove client!
                if not len(self.pending[client]) > 0:
                    del self.pending[client]

        # Search did not exist
        # Unlock mutex
        self.unlock()
        return false

    # Function to remove all pending searches on a node.
    # Used when querying p-SARS node is persumed dead!
    # Args:
    # node = Remove all pending searches on this node
    # Returns true when done
    def remove_all(self, node):

        # lock mutex
        self.spin_lock()

        # Test if it exists!
        if node in self.pending:
            # Delete
            del self.pending[node]

        # Search did not exist
        # Unlock mutex
        self.unlock()
        return true

    # Function to read results from a pending search
    # Args:
    # client = The node where the search is registered under
    # topic = The topic of the search
    # Retruns results or false if the search does not exist
    def read(self, client, topic):
        # lock mutex
        self.spin_lock()

        # Test if it exists!
        if client in self.pending:
            if topic in self.pending[client]:
                # Fetch results
                res = self.pending[client][topic]['results']
                # unlock mutex
                self.unlock()
                return res

        # Search not pending
        # unlock mutex
        self.unlock()
        return false



    # Function to read all pending searches
    # Returns entire dictionary
    def read_all(self):
        # lock mutex
        self.spin_lock()

        # Create copy of the pending search set
        ret = self.pending.copy()

        #unlock mutex
        self.unlock()
        return ret
    # Function to process pending searches and figure out which one's
    # to terminate
    # Returns the list of searches terminated if any, empty list if none
    def process_pending(self):
        # lock mutex
        self.spin_lock()

        # Create list of terminated searches
        terminated = []

        # Get time
        current_time = int(time.time())

        tmp = self.pending.copy()

        # Iterate through pending searches
        for client in tmp:
            for topic in tmp[client]:
                # Test if search has timed out
                if (current_time - tmp[client][topic]['timestamp']) >
SECONDS_TO_WAIT_ON_RESULTS:
                    # append search to terminated list
                    terminated.append((client, topic, tmp[client][topic]['results']))

        # Delete terminated searches
        for element in terminated:
            del self.pending[element[0]][element[1]]
            # See if there are more pending searches on this client.
            # if not delete it
            if not len(self.pending[element[0]]) > 0:
                del self.pending[element[0]]



        # unlock mutex
        self.unlock()
        return terminated
```

<p-SARS/OWN_event_handler.py>

```
# The p-SARS System (peer-to-peer search engine)
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: OWN_event_handler.py

# import own libs
from OWN_constants import *
import OWN_network
import OWN_system
import OWN_math

# import standard libs
import Queue
import threading
import time

# The class containing function for event handling
class event_handler:

    # Init function
    # Args:
    # sets = The system sets
    # queues = The event queues
    # addr_o = The system addresses
    def __init__(self, sets, queues, addr_o):

        # Hold on to sets
        self.sets = sets
        # hold on to queues
        self.queues = queues
        # Hold on to addresses
        self.addr_o = addr_o


        # Start the threads processing these queues
        # Start search event processing
        search_thread = process_search_t(self.queues, self.sets, self.addr_o)
        search_thread.start()

        # Start gossip event processing
        gossip_thread = process_gossip_t(self.queues, self.sets, self.addr_o)
        gossip_thread.start()

        # Start result event processing
        result_thread = process_result_t(self.queues, self.sets, self.addr_o)
        result_thread.start()

        # Start system event processing
        system_event_thread = process_system_t(self.queues, self.sets, self.addr_o)
        system_event_thread.start()

    # Function to identify event and enqueue it
    # on the right event queue
    # Args:
    # event = incomming event
    # Return true, if all ok, false if error
    def identify_event_enqueue(self, event):

        if event['type'] == TYPE_SEARCH:
            # We have a search, enqueue it
            # print 'WE HAVE A SEARCH EVENT'
            self.queues.search_q.put(event)

        elif event['type'] == TYPE_GOSSIP:
            # We have a rumor about a publisher, enqueue it
            # print 'WE HAVE A GOSSIP EVENT'
            self.queues.gossip_q.put(event)

        elif event['type'] == TYPE_RESULT:
            # We have a result, enqueue it
            # print 'WE HAVE A RESULT EVENT'
            self.queues.result_q.put(event)

        elif event['type'] == TYPE_DEBUG:
            # We have a debug event, print it
            # print 'WE HAVE A DEBUG EVENT'
            print event

        elif event['type'] == TYPE_SYSTEM:
            # We have a system event, enqueue it
            # print 'WE HAVE A SYSTEM EVENT'
            self.queues.system_q.put(event)

        else:
            print 'UNKNOWN event. No action taken on : ' + event[heading]
            return false

        # All ok
        return true


# The event handler thread.
# Initiates the event_handler class
# and starts the listen and pusher thread
class event_handler_t(threading.Thread):

    # The init function
    # Args:
    # queues = The event queues
    # sets = The system sets
    # addr_o = The system addresses
    def __init__(self, queues, sets, addr_o):
        # init thread
        threading.Thread.__init__(self)

        # Hold on to queues
        self.queues = queues
        # Hold on to sets
        self.sets = sets
        # Hold on to addresses
        self.addr_o = addr_o


    def run(self):
        # Init OWN_event_handler
        self.event = event_handler(self.sets, self.queues, self.addr_o)

        # Create and start up a listen thread
        listen_thread = OWN_network.listen_t(self.addr_o.read_listen_tcp(),
self.queues)
        listen_thread.start()

        # Create and start up RPC listening thread!!
        #xmlrpc_thread = OWN_network.xmlrpc_listener_t(self.addr_o.read_listen_rpc(),
self.queues)
        #xmlrpc_thread.start()

        # Create and start up pusher thread.
        # This thread processes the system-output queue
        pusher_thread = OWN_network.pusher_t(self.queues,self.addr_o)
        pusher_thread.start()

        # Receive incomming message, identify it and place it in right queue
        # for processing.
        while true:
            # Fetch from queue when data ready
            event = self.queues.input_q.get()

            # process event, and enqueue it in right queue
            self.event.identify_event_enqueue(event)


####################################
# Threads to process event queues! #
####################################

# The thread processing incomming search events
class process_search_t(threading.Thread):

    # Args:
    # queues = The system event queues
    # sets = The system sets
    #       (Neighbour set, Topic set, Hint set and pending search set)
    # addr_o = The set of system addresses. (ip,port) tuples
    def __init__(self, queues, sets, addr_o):
        # init thread
        threading.Thread.__init__(self)

        # The queues
        self.queues = queues
        # The sets
        self.sets = sets
        # Hold on to the addresses
        self.addr_o = addr_o

        # Fetch the listen module address
        self.ADDR = addr_o.read_listen_tcp()

        # Create instance of the network utils class
        self.net = OWN_network.network_utils()

    def run(self):

        while true:
            # Fetch incomming search event
            data = self.queues.search_q.get()

            # If search originates from local client. Add it as pending!
            # And add a few entries in the dictionary
            if data['local']:
                # Set pending
                self.sets.pending_o.create_pending_search(data['reply_addr'],
data['topic'], data['happy'], data['reply_addr'])
                # Set error to false
                data['error'] = false
                # Set ttl
                data['ttl'] = TTL_SEARCH
                # Create visited list
                data['visited'] = []
                # This search is not based on hint
                data['hint'] = false
                # Set from address. This is the super-node where the query originated.
                # The reply_addr is the super-nodes client that issued the request
                data['from'] = self.ADDR


            # Create debug event
            debug = {}
            debug['to'] = DEBUG_ADDRESS
            debug['type'] = TYPE_DEBUG
            debug['from'] = self.ADDR
            debug['debug'] = 'Search topic: %s. TTL = %d' %
(data['topic'],data['ttl'])


            # Test if it is an error from previous sent search message
            # If it is, update neighbor list and hint list
            if data['error']:
                # Remove node as neighbour. BUT the message may just have failed
                # because the neighobur is currently satturated.
                self.sets.neighbours_o.remove(data['to'])

                # Remove hints recorded on that node.
                self.sets.hint_o.remove(data['to'])

                # DEBUG
                debug['debug'] += ' (ERROR) '
                #print 'ERROR RELAYING SEARCH: %s %d' % (data['to'][0], data['to'][1])
                #print 'Neighbour removed from neighbour set'


            # Fetch topic
            topic = data['topic']

            # Decrement TTL, even if the event failed to be sent. (Error = true)
            # This is to prevent looping indefinity when the node is disconnected
from all
            # others! We even decrement the TTL first time around!
            data['ttl'] = data['ttl'] - 1

            # Performe local search. But do not perform local search if error,
            # the search is originating from this node or this node already is
visited!!
            result_local = false
            local_search = false
            if not (data['error'] or data['local'] or (self.ADDR in data['visited'])):
                # Search locally on topic, and get result.
                result_local = self.sets.topic_o.read(topic)
                local_search = true
                # DEBUG
                debug['debug'] += ' (LSearch) '


            # Add this node as visited, but not if already added
            if not (self.ADDR in data['visited']):
                data['visited'].append(self.ADDR)
                # DEBUG
                debug['debug'] += ' (+Visited) '


            # Calculate number of hits
            if result_local != false:
                num_result_local = len(result_local)

            else:
                # We have not performed a local search. No results
                # are therefore available
                num_result_local = 0

                # Test if the search was based on a hint. If so we must send a message
back to
                # the last node and tell him that the hint is not true. This should
not be done if a local
                # search has not been performed!
                if data['hint'] and local_search:
                    # Last visited node is the receiver of this message
                    last_visited_index = len(data['visited']) - 1

                    # Ensure that we do not get a list index out of range!
                    if last_visited_index >= 0:
                        # Create the event
                        gossip = {}
                        gossip['type'] = TYPE_GOSSIP
                        gossip['to'] = data['visited'][last_visited_index]
                        gossip['from'] = self.ADDR
                        # No new entries are to be added in hint list
                        gossip['new_gossip'] = []
                        # The item to remove from hint lists!
                        gossip['death_gossip'] = []
                        gossip['death_gossip'].append(data['topic'])
                        # Add local hint set
                        gossip['gossip_dict'] = self.sets.hint_o.read_all()
                        # Set ttl
                        gossip['ttl'] = TTL_GOSSIP
                        # Enqueue message
                        self.queues.output_q.put(gossip)
                        # DEBUG
                        debug['debug'] += ' (False hint) '
                        print 'FALSE HINT, we do not have the topic "%s" locally' %
data['topic']

            # Update HAPPY value. When happy value is 0, search is satisfied
            data['happy'] = data['happy'] - num_result_local

            # Search hint table if search not yet satisfied and TTL != 0
            if data['ttl'] > 0 and data['happy'] > 0:
                # Search local hints, and get random node returned. Do not include
                # already searched nodes!
                random_node = self.sets.hint_o.pick_random_hint_node(topic,
data['visited'])

                # If random hint node is found, relay search
                if random_node != false:
                    # Set address to whom search is headed for
                    data['to'] = random_node
                    data['hint'] = true
                    data['local'] = false
                    data['error'] = false
                    # Enqueue it on output queue
                    self.queues.output_q.put(data)
                    # DEBUG
                    debug['debug'] += ' (Hint found) '

                else:
                    # No hint is found. Search is degenerated to plain Random Walk
                    # Fetch random neighbour for random Walk.
                    random_node =
self.sets.neighbours_o.pick_random_neighbour(data['visited'])

                    if random_node != false:
                        # Neighbour found, address the event to it
                        data['to'] = random_node
                        data['hint'] = false
                        data['local'] = false
                        data['error'] = false
                        # Enqueue it on output queue
                        self.queues.output_q.put(data)
                        # DEBUG
                        debug['debug'] += ' (R_Walk) '

                    else:
                        # All has failed. Node has no neighbours!
                        # just discard search. Results, if any, will be returned
downbelow
                        print 'NO NEIGHBOURS CONNECTED'
                        # DEBUG
                        debug['debug'] += ' (No neighbours connected!) '


            # If results found on this node. Send them directly to querying node
            if num_result_local > 0:
                # Prepare result message
                res = {}
                res['to'] = data['from']
                res['from'] = self.ADDR
                res['type'] = TYPE_RESULT
                res['topic'] = data['topic']
                res['result'] = result_local
                res['reply_addr'] = data['reply_addr']
                res['error'] = false

                # Enqueue result message
                self.queues.output_q.put(res)

                # DEBUG
                debug['debug'] += ' (Results sent) '
            else:
                # DEBUG
                debug['debug'] += ' (No local results) '

            # Test if we should send debug event to debug module
            if DEBUG:
                self.queues.output_q.put(debug)


# The thread processing incomming gossip events
class process_gossip_t(threading.Thread):

    # Args:
    # queues = The system event queues
    # sets = The system sets
    #       (Neighbour set, Topic set, Hint set and pending search set)
    # addr_o = The set of system addresses. (ip,port) tuples
    def __init__(self, queues, sets, addr_o):
        # init thread
        threading.Thread.__init__(self)

        # The queues
        self.queues = queues
        # The sets
        self.sets = sets
        # The addresses
        self.addr_o = addr_o
        # Fetch the address of the listen module
        self.ADDR = self.addr_o.read_listen_tcp()

        # instance of network
        self.net = OWN_network.network_utils()
        # instance of math
        self.math = OWN_math.math_utils()

    def run(self):
        while true:
            # Fetch incomming gossip event
            data = self.queues.gossip_q.get()

            # The sending of the gossip message failed. Node is persumed down. Remove
it.
            if data['error']:
                print 'GOSSIP ERROR! Removing all hints on this node, and removing it
as a neighbor'
                # Remove it from neighbor set
                self.sets.neighbours_o.remove(data['to'])
                # Remove it from hint list
                self.sets.hint_o.remove(data['to'])
```

```
                    # We must relay gossip if the TTL is still positive
                    data['error'] = false

                    # We shall not wait for a new event. Must try to relay the one
                    # we failed to relay

                # Neighbour requesting our gossip list!
                elif data['get']:
                    # Create push gossip
                    # Send it to querying node. (Found in the from field)
                    event = {}
                    event['type'] = TYPE_GOSSIP
                    event['to'] = data['from']
                    event['from'] = self.ADDR
                    # Set TTL, because the sending of this message
                    # can fail
                    event['ttl'] = 1
                    # Include our entire hint set!
                    event['gossip_dict'] = self.sets.hint_o.read_all()

                    # Also add all entires in the local topic set to this
                    # gossip.
                    # Extract all topics from topic set
                    topic_set = self.sets.topic_o.read_all()
                    topics = topic_set.keys()

                    # Add them with this p-SARS nodes address
                    for element in topics:
                        # If the entry does not already exist, create it
                        if element not in event['gossip_dict']:
                            event['gossip_dict'][element] = []

                        # Ensure that we do not introduce a duplicate
                        # even though a p-SARS node should never be
                        # present in its own hint set
                        if self.ADDR not in event['gossip_dict'][element]:
                            # Not present, add it
                            event['gossip_dict'][element].append(self.ADDR)


                    event['error'] = false
                    # This is not a request!
                    event['get'] = false
                    # This is a push event!
                    event['push'] = true

                    # Enqueue event
                    self.queues.output_q.put(event)
                    print 'Hint list is sent to neighbour'

                    # Wait for new event
                    continue

                # Getting the gossip list from a neighbour!
                elif data['push']:
                    # Fetch dictionary, and insert hints from neighbours
                    # hint list
                    print 'Hint set is received from neighbour!'
                    for topic in data['gossip_dict']:
                        for node in data['gossip_dict'][topic]:
                            # Add node to hint list on the right topic, but ensure that we
don't
                            # add hints pointing to this node.
                            if node != self.ADDR:
                                if self.sets.hint_o.append(topic,node) == true:
                                    print 'Hint added from neighbours hint list!!!'

                    # Wait for new event
                    continue

                # We have an ordinary gossip event
                else:
                    # Only register gossip if it's not from this node!!
                    if data['from'] != data['to']:
                        # Traverse new gossips
                        for element in data['new_gossip']:
                            # Register gossip
                            res = self.sets.hint_o.append(element,data['from'])

                            if res == duplicate:
                                # DEBUG
                                # print 'already known'
                                pass
                            else:
                                # DEBUG
                                print 'hint is added!'

                        # Traverse death gossips
                        for element in data['death_gossip']:
                            # Remove hint from hint set
                            print 'DEATH GOSSIP RECEIVED!!!!!'
                            res = self.sets.hint_o.remove_one(element, data['from'])
                            if res:
                                # DEBUG
                                print 'Hint removed'
                                pass
                            else:
                                # DEBUGlocal_ip
                                # print 'Hint does not exist'
                                pass

                        # Travers hint set receivd
                        for topic in data['gossip_dict']:
                            for node in data['gossip_dict'][topic]:
                                # Add node to hint list on the right topic, but ensure
    that we don't
                                # add hints pointing to this node.
                                if node != self.ADDR:
                                    if self.sets.hint_o.append(topic,node) == true:
                                        print 'Hint added from received hint list!!!'

                    else:
                        # DEBUG
                        # print 'Gossip originates from this node!!'
                        pass

                # PREPARE TO RELAY GOSSIP
                # Decrement ttl on gossip
                data['ttl'] = data['ttl'] - 1

                # Test to see if we should continue gossip
                if data['ttl'] > 0:
                    # Fetch neighbors
                    neighbor_set, not_important = self.sets.neighbours_o.read()

                    # Send gossip to neighbor with a probability of
    PROBABILITY_SEND_GOSSIP
                    # Traverse all neighbors
                    for element in neighbor_set:
                        if self.math.should_i_gossip(PROBABILITY_SEND_GOSSIP):
                            print 'send gossip'
                            # Gossip to this neighbor!
                            # Create event
                            data['to'] = element

                            # enqueue message
                            self.queues.output_q.put(data)

                            # DEBUG
                            # enqueue debug event
                            # debug = {}
                            # debug['to'] = DEBUG_ADDRESS
                            # debug['from'] = self.addr_o.read_listen_tcp()
                            # debug['debug'] = 'Gossip sendt on topic. to %s %d TTL = %d'
    % (element[0],element[1],event['ttl'])
                            #self.queues.output_q.put(debug)
                        else:
                            # No gossip sent
                            print 'NO GOSSIP'
```

```
# The thread processing incomming results
class process_result_t(threading.Thread):

    # The init function
    # Args:
    # queues = The system event queues
    # sets = The system sets
    #       (Neighbour set, Topic set, Hint set and pending search set)
    # addr_o = The set of system addresses. (ip,port) tuples
    def __init__(self, queues, sets, addr_o):
        # init thread
        threading.Thread.__init__(self)

        # The queues
        self.queues = queues
        # The sets
        self.sets = sets
        # The addresses
        self.addr_o = addr_o


    def run(self):
        # DEBUG values
        num_res = 0
        num_dead = 0

        while true:
            # Fetch incomming result. But only wait on queue for a defined number of
seconds
            empty = false
            try:
                data = self.queues.result_q.get(true,SECONDS_BEFORE_RESULT_UPDATE)
            except Queue.Empty:
                # Nothing in queue
                empty = true

            # Test if we have a result event
            if not empty:
                # Test if it is an error. In that case, try to remove pending search!
                if data['error']:
                    # Remove pending searches on client!
                    if 'reply_addr' in data:
                        self.sets.pending_o.remove_all(data['reply_addr'])

                    # Either we have a WRS client down or the querying p-SARS node is
down
                    # If it is a WRS client the topics the client publishes on will
not be
                    # removed here. But in profile update, when it's discovered that
the
                    # client is dead. If it's the p-SARS node that has died, this node
has no
                    # pending search and the removal will fail.

                    # DEBUG
                    num_dead += 1

                else:
                    # Add result to pending search!
                    res = self.sets.pending_o.append(data['reply_addr'],
data['topic'], data['result'])

                    # Try to add hint in hint set
                    if self.sets.hint_o.append(data['topic'], data['from']) == true:
                        print 'HINT ADDED FROM SEARCH RESULT!'

                    # Test if search is still pending
                    if res:
                        # See if search is satisfied
                        if self.sets.pending_o.happy(data['reply_addr'],
data['topic']):
                            print 'SEARCH SATISFIED %d' % num_res

                            # DEBUG
                            num_res += 1

                            # Search is satisfied before timeout. Return it to
querying
                            # WRS client!
                            event = {}
                            event['type'] = TYPE_RESULT
                            event['topic'] = data['topic']
                            # Fetch results. Search is there because we already testet
for that
                            event['results'] =
self.sets.pending_o.read(data['reply_addr'], data['topic'])
                            event['to'] = data['reply_addr']
                            # Enqueue answer to client
                            self.queues.output_q.put(event)
                            # Remove search
                            self.sets.pending_o.remove(data['reply_addr'],
data['topic'])


            # Process pending searches. See if any of them should be removed!
            # Fetch timed out searches
            res = self.sets.pending_o.process_pending()

            # Create answers to the WRS clients and enqueue them
            for element in res:
                num_res += 1
                event = {}
                event['type'] = TYPE_RESULT
                event['to'] = element[0]
                event['topic'] = element[1]
                event['results'] = element[2]
                print 'SEARCH HAS TIMED OUT %d , %d' % (num_res,num_dead)
                # Enqueue element
                self.queues.output_q.put(event)


# The thread processing incomming system events
class process_system_t(threading.Thread):

    # The init function
    # Args:
    # queues = The system event queues
    # sets = The system sets
    #        (Neighbour set, Topic set, Hint set and pending search set)
    # addr_o = The set of system addresses. (ip,port) tuples
    def __init__(self, queues, sets, addr_o):
        # init thread
        threading.Thread.__init__(self)

        # The queues
        self.queues = queues
        # The sets
        self.sets = sets
        # The addresses
        self.addr_o = addr_o

    def run(self):
        while true:
            # Fetch incomming result.
            data = self.queues.system_q.get()

            # Test if error
            if data['error']:
                # For now we just drop the message
                # Neighbour will be removed on next system update when pong has not
                # been received
                pass

            # Test if ping
            elif data['topic'] == 'ping':
                # We have a ping message. Create a pong message
                tmp = data['to']
                data['to'] = data['from']
                data['from'] = tmp
                data['topic'] = 'pong'
                # enqueue pong message
                self.queues.output_q.put(data)

                # Fetch timestamp
                timestamp = int(time.time())

                # Try to set ping time
                # Test if time was set!
                if self.sets.neighbours_o.set_ping_time(data['to'], timestamp) ==
true:
                    # DEBUG
                    # print 'Timestamp set on node:'
                    # print data['to']
                    pass

                else:
                    # Try to insert node as neighbor
                    if self.sets.neighbours_o.append(data['to']) == true:
                        print 'Pinging node added as neighbor!'


            # Test if pong
            elif data['topic'] == 'pong':
                # We have a pong message.
                # Create timestamp
                timestamp = int(time.time())

                # Try to set ping time and test if time was set!
                if self.sets.neighbours_o.set_ping_time(data['from'], timestamp) ==
true:
                    pass

                else:
                    # Neighbor is to late with pong message
                    # we should insert him again if room!
                    if self.sets.neighbours_o.append(data['from']) == true:
                        print 'Pong to late. But neighbor is reinserted!'

            # It's an unknown event!
            else:
                print 'System event not recognized!!'
```

<p-SARS/OWN_profiles.py>

```
# The p-SARS System (peer-to-peer search engine)
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: OWN_profiles.py

#import own libs
from OWN_constants import *
import OWN_network
import OWN_math

# import standard libs
import time
import threading

# Class that contains utils for managing profiles.
# Currently only topic management
class profile_utils:

    # The init function
    def __init__(self):
        # init network utils
        self.net = OWN_network.network_utils()

    # Function to fetch profiles
    # Args:
    # host_addr = The address tuple for the profile server
    # Returns the profiles received from profile server, or false if nothing returned
    def fetch_topics(self, host_addr):
        #create network connection
        soc = self.net.open_client_tcp(host_addr)


        if soc != false:
            # Get local socket address
            my_addr = soc.getsockname()

            #create request package and pack it
            msg = {'type':TYPE_FETCH_PROFILE,'address':my_addr}
            msg =  self.net.pack(msg)

            #send package
            self.net.send_tcp(msg, soc, MAX_WAIT)

            #fetch and return answere
            msg = self.net.receive_tcp(soc, TCP_BUFF_SIZE, MAX_WAIT)

            if len(msg) > 0:
                # Unpack (unmarshall) message
                msg = self.net.unpack(msg)
                # Return it
                return msg

            else:
                # Empty message returned from topic server
                return false

        else:
            # Could not open connection to topic server
            return false



    # Function to save profiles to file
    # Args:
    # Profile_list = The list of interests and ip
    # filename = Name of the file containing profiles
    # Returns true if ok, false otherwise
    def save_profiles(self, profile_list, filename):

        # Pack message
        ok = self.net.pack(profile_list)

        # Test for error
        if not ok:
            return false

        else:
            # If not error
            # Open file for writing bytes
            fp = open(filename,'wb')
            # Write to file
            fp.write(msg)
            # Close file
            fp.close()

            # Return true
            return true


    # Function to load profiles from file
    # Args:
    # filename = name of file where profiles can be found
    # Returns profile list, or 0 if error
    def load_profiles(self, filename):
        # Open file & read content
        fp = open(filename,'rb')

        # unpack profile dictionary
        ok = self.net.unpack(fp.read())

        # Test for error
        if not ok:
            # We have an error
            return false

        else:
            # Return unpacked data
            return msg


# The thread updating the profiles of the local publishers
# Args:
# queues = The system queues
# sets = The system sets
# addr_o = The system addresses
class update_topic_set_t(threading.Thread):

    def __init__(self, queues, sets, addr_o):
        # init thread
        threading.Thread.__init__(self)

        # Keep queues
        self.queues = queues
        # Keep sets
        self.sets = sets

        # Fetch the listen module address, and profile server address
        self.ADDR = addr_o.read_listen_tcp()
        self.PROFILE_SERVER_ADDR = addr_o.read_profile_server_address()

        # instance of the math_utils class
        self.math = OWN_math.math_utils()
        # instance of the profile_utils class
        self.profile = profile_utils()

    def run(self):

        # If all nodes are started at once, we must wait until their neighbour sets
are
        # initialized... (If not there will be no gossiping because there are no
neighbours yet!)
        # Should just be used during testing..
        # time.sleep(5)

        while true:
            print 'Updating topic list with information from topic server! :'
            print self.PROFILE_SERVER_ADDR
            # Fetch profiles
            result = self.profile.fetch_topics(self.PROFILE_SERVER_ADDR)


            if result != false:

                # Update local publishing list!!
                new_topic_list, del_topic_list = self.sets.topic_o.update(result)

                #print '############# TOPICS Discovered ########################'
                #print new_topic_list
                #print '_____________________________________________'
                #print del_topic_list
                #print '##############################################'

                if len(new_topic_list) > 0 or len(del_topic_list) > 0:

                    # prepare to gossip
                    new_gossip = []
                    death_gossip = []
                    # populate new_gossip list
                    if len(new_topic_list) > 0:
                        # Traverse new_topic_list
                        for element in new_topic_list:
                            new_gossip.append(element)

                    # populate death_gossip list
                    if len(del_topic_list) > 0:
                        # Traverse del_topic_list
                        for element in del_topic_list:
                            death_gossip.append(element)

                    # Fetch hint set
                    hint_set = self.sets.hint_o.read_all()

                    # Create gossip event
                    event = {}
                    event['type'] = TYPE_GOSSIP
                    event['from'] = self.ADDR
                    event['ttl'] = TTL_GOSSIP
                    # Add new and deleted topics
                    event['new_gossip'] = new_gossip
                    event['death_gossip'] = death_gossip
                    # Add local hint set
                    event['gossip_dict'] = hint_set
                    event['error'] = false
                    event['get'] = false
                    event['push'] = false

                    # Send gossip to neighbor with a probability of
PROBABILITY_SEND_GOSSIP
                    # Fetch neighbors
                    neighbor_set, not_important = self.sets.neighbours_o.read()

                    # Traverse all neighbors
                    for element in neighbor_set:
                        if self.math.should_i_gossip(PROBABILITY_SEND_GOSSIP):
                            # Gossip to this neighbor!

                            # Address gossip event
                            event['to'] = element

                            # enqueue event
                            self.queues.output_q.put(event)
                            print 'Gossip initiated after topic update'

                            # enqueue debug event
                            #debug = {}
                            #debug['to'] = DEBUG_ADDRESS
                            #debug['from'] = self.ADDR
                            #debug['debug'] = 'Gossip sendt on topic. to %s %d TTL =
%d' % (element[0],element[1],event['ttl'])
                            #self.queues.output_q.put(debug)

                        else:
                            print 'NO RANDOM GOSSIP'

            else:
                print 'Failed connecting to topic server!'


            # Sleep a little while before updating the topic set!
            # This wait is placed at the bottom to create a do-while structure..
            time.sleep(SECONDS_BEFORE_TOPIC_UPDATE)
```

<p-SARS/debug.py>

```
# The p-SARS System (peer-to-peer search engine)
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: debug.py

# Import own libs
from OWN_constants import *
import OWN_network

# The main method
def main():
    # Create instance of OWN_network
    net = OWN_network.network_utils()

    # create server tcp
    s_soc = net.open_server_tcp(DEBUG_ADDRESS, TCP_BUFF_SIZE, 10000)
    if s_soc != false:

        # Open debug file!
        fp = open('debug.txt','a')
        fp.write('######## START LOG #########\n\n')

        run = 1
        while run:
            print 'Debug logger started'
            c_soc, addr = s_soc.accept()
            data = net.receive_tcp(c_soc, TCP_BUFF_SIZE, MAX_WAIT)

            # unpack and save data to file
            data = net.unpack(data)

            if data != false:
                # Stop debug??
                if data['debug'] == 'end':
                    fp.write('######### END LOG ##########\n\n')
                    # close file
                    fp.close()
                    # close connection
                    c_soc.close()
                    s_soc.close()
                    # end while loop
                    run = 0
                    print '\n\nbye'
                else:
                    address = '%s : %d' % (data['from'][0],data['from'][1])
                    # Write debug to file
                    fp.write(address + ' : ' + data['debug']+'\n')
                    # close client connection
                    c_soc.close()
    else:
        # Failed creating tcp/ip socket
        print 'Debug event collector not started!!!'

# Starting the program by calling main!
if __name__ == '__main__':
   # Call main
   main()
```

<p-SARS/end_debug.py>

```
# The p-SARS System (peer-to-peer search engine)
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: end_debug.py

# Import own libs
from OWN_constants import *
import OWN_network

# Make instance of network utils
net = OWN_network.network_utils()

# Create terminate debug event
data = {}
data['debug'] = 'end'

# pickle event
data = net.pack(data)

if data != false:
    # open client connection
    c_soc = net.open_client_tcp(DEBUG_ADDRESS)

    if c_soc != false:
        # send event
        net.send_tcp(data, c_soc, MAX_WAIT)
        # close connection
        if not net.close_conn(c_soc):
            print 'Failed closing socket after debug termination event wass sent'

    else:
        print 'Sending debug event failed'
```

Appendix B

Appendix B: Simulator, code listing

The simulator code consists of a total of 686 lines where 306 of these are non-commenting source statements (NCSS).

<simulator/sim.py>

```
# The p-SARS Simulator
# Designed and implemented by Rune Devik
# 2003, 15 August - 15 December
# Master of Engineering Thesis
# File: sim.py

# Import modules
import time
import math
import random
import copy
import string

# Defining true/false
true = 1
false = 0

# Container class for the defined tuning variables!
# ARGS:
# list = The list of variables defining one simulation
# Returns: Nothing
class tuning_variables:
    def __init__(self,list):

        # SET NETWORK VARIABLES
        # Number of nodes in network
        self.num_nodes = list[0]
        # Maximum number of neighbours
        self.max_neighbours = list[1]
        # Fetch random number of neighbours?
        self.random_number_of_neighbours = list[2]

        # SET GOSSIP VARIABLES
        # Should we gossip?
        self.gossip = list[3]
        # TTL on gossip
        self.TTL = list[4]
        # Probability of gossip
        self.probability = list[5]

        # SET RANDOM WALK REPLICATION VARIABLES
        # Should we perform RW replication
        self.RW_replication = list[14]
        # TTL on RW replication
        self.RW_replication_TTL = list[15]

        # SET SEARCH VARIABLES
        # MAX 994, MAXIMUM RECURSION DEPTH IN PYTHON.
        # Should we perform search?
        self.search = list[6]
        # TTL on search
        self.TTL_search = list[7]
        # Number of searches to perform
        self.num_searches = list[8]
        # Should the hint set be populated from search results?
        self.learning = list[9]
        # Should intermediate nodes also fetch hints from search
        # results?
        self.learning_plus = list[10]

        # SET WARM - UP VARIABLES
        # Should we do warm-up?
        self.warmup = list[11]
        # TTL on warm-up searches
        self.TTL_warmup = list[12]
        # Number of searches to perform on warm-up
        self.num_warmup = list[13]


# The class simulating a node in the p-SARS system
class node_t:

    # The init function
    # Args:
    # id = The node id!
    # Returns: Nothing
    def __init__(self, id):

        # Init neigbourhood set, hint_set and topic_set
        self.neighbours = []
        self.pingtime = int(time.time())
        self.id = id
        self.hint_set = []
        self.topic_set = []
        self.topic_set.append(self.id)

    # Function to populate the neighbour set of a node
    # ARGS:
    # ran = An integer deciding how many nodes are to
    #       be added in the neighbour set
    # num_nodes = The number of nodes in the network
    # Returns: Nothing
    def insert_new_neighbours(self, ran, num_nodes):
        # fetch random ran numbers of neighbours between 0 and num_nodes
        for element in range(ran):
            while 1:
                ran_neighbour = random.randint(0,num_nodes - 1)
                # not include this node as its own neighbour!
                if ran_neighbour != self.id:
                    if ran_neighbour not in self.neighbours:
                        # We have found a new node not already added
                        self.neighbours.append(ran_neighbour)
                        break

    # Function to count number of hints in hint_set
    # Returns: number of hints
    def count_hints(self):
        return len(self.hint_set)


# The class that implements the simulation operations
class sim:

    # Initializes the network
    # Args:
    # num_nodes = The number of nodes in the network
    # Returns: Nothing
    def __init__(self, num_nodes):
        self.nodes = []
        for i in range(num_nodes):
            self.nodes.append(node_t(i))
            self.num_nodes = num_nodes


    # Function to start gossip
    # Args:
    # node = The node where the gossip should start
    # probability = The probability of gossiping to each individual node
    # ttl = Time to live on gossip
    # Returns number of reached nodes (may have duplicates)
    #          and the depth reached on this gossip
    def gossip(self, node, probability, ttl):
        # Traverse neighbours
        num = 0
        depth_of_gossip = 0
        depth = []
        # Gossip based on probability
        for element in self.nodes[node].neighbours:
            if self.should_i_gossip(probability):
                tmp = self.relay_gossip(element, self.nodes[node].topic_set,
probability, ttl, depth)

                num = num + tmp

                # Record gossip depth
                depth_of_gossip = len(depth)

        return (num,depth_of_gossip)

    # Function to relay gossip (Recursive function)
    # Args:
    # node = The node where the gossip should start
    # hints = The hints gossiped
    # probability = The probability of gossiping to each individual node
    # ttl = Time to live on gossip
    # depth = A list of the depths this gossip has reached.
    # Returns number of nodes reached including itself!
    def relay_gossip(self, node, hints, probability, ttl, depth):
        # Add gossip to hint set
        num = 0
        # Traverse hints received
        for element in hints:
            # If they originate from this node, don't add them
            if element != self.nodes[node].id:
                # Test if hint already is known
                if element not in self.nodes[node].hint_set:
                    self.nodes[node].hint_set.append(element)

        # Decrement TTL
        ttl = ttl - 1

        # If new depth is reached, add it!
        if ttl not in depth:
            depth.append(ttl)

        # Should we continue gossiping
        if ttl > 0:
            # Traverse all neighbours and send to them
            # based on the given probability
            for element in self.nodes[node].neighbours:
                if self.should_i_gossip(probability):
                    tmp = self.relay_gossip(element, hints, probability, ttl, depth)
                    num = num + tmp

            num = num + 1
            return num

        else:
            # Discard gossip
            return 1


    # Function to start search
    # Args:
    # node = The node where the search should start
    # Topic = The node_id to search for!
    # ttl = Time to live for search event
    # learning = True if the searching node should learn from the
    #            search results!
    # learning_plus = True if the intermediate nodes should learn
    #                 from the search results
    # Returns: If topic is found and the depth reached on search
    def search(self, node, topic, ttl, learning, learning_plus):
        # Create visited list.
        visited = []

        # See in hint_set
        if topic in self.nodes[node].hint_set:
            # Topic found in hint set
            # Forward search to this node
            id = topic # Important that id == topic in this simulation
            #print 'Yes hint found. Search relayd on hint from node : %d' % self.id
            found, depth = self.relay_search(id, topic, ttl, visited, learning_plus)
        else:
            # Topic not found in hint set
            # Pick random neighbour and relay search
            num = len(self.nodes[node].neighbours)
            if num == 0:
                # No neighbours connected. Search failed
                print 'No neighbours connected!!!!!!!!!!!!'
                return (false,0)
            else:
                # Pick random neighbour
                neighbour = random.randint(0, num - 1)
                # Relay search
                found, depth =
self.relay_search(self.nodes[node].neighbours[neighbour], topic, ttl, visited,
learning_plus)

        # Extract hint from results if it is a learning search
        if found and learning:
            # Add node id in hint set if it's not already there!
            if topic not in self.nodes[node].hint_set:
                self.nodes[node].hint_set.append(topic)

        return (found, depth)


    # Function to relay search
    # Args:
    # node = The node the search is currently on
    # Topic = The node_id to search for!
    # ttl = Time to live for search event
    # visited = The list of already visited nodes
    # learning = True if all intermediate nodes on the search path
    #             also should learn from the search results
    # Returns: If topic is found and the depth reached on search
    def relay_search(self, node, topic, ttl, visited, learning):

        # Decrement TTL
        ttl -= 1

        # Search locally
        if topic in self.nodes[node].topic_set:
            # Bingo, we are on the right node
            found = true
            depth = ttl
            #print 'topic %d found on node %d' % (topic, self.id)
            return (found, depth)

        if ttl > 0:
            # Add node in visited list, if it's not already there!
            if self.nodes[node].id not in visited:
                visited.append(self.nodes[node].id)

            # Look for hints
            if topic in self.nodes[node].hint_set:
                # Relay search based on hint
                id = topic
                found, depth = self.relay_search(id, topic, ttl, visited, learning)

                # Add as hint in hint list
                if found and learning:
                    if topic not in self.nodes[node].hint_set:
                        self.nodes[node].hint_set.append(topic)

                return (found,depth)
            else:
                num = len(self.nodes[node].neighbours)
                if num == 0:
                    # No neighbours connected. Search failed
                    print 'No neighbours !!!!!!!!!!!!!'
                    return (false, ttl)
                else:
                    # copy neighbour list
                    tmp = copy.deepcopy(self.nodes[node].neighbours)
                    # Only the neighbours not already visited should be
                    # chosen first. Remove already visited nodes
                    for element in visited:
                        if element in tmp:
                            # Node visited earlier
                            tmp.remove(element)

                    num = len(tmp)

                    if num > 0:
                        #print 'Neighbours left'
                        # Pick random neighbour
                        neighbour = random.randint(0, num - 1)
                        found, depth = self.relay_search(tmp[neighbour], topic, ttl,
visited, learning)

                    else:
                        #print 'No neighbours left'
                        # all neighbours visited. Pick one random among all neighbours
                        num = len(self.nodes[node].neighbours)
                        neighbour = random.randint(0, num - 1)

                        # Relay search
                        found, depth =
self.relay_search(self.nodes[node].neighbours[neighbour], topic, ttl, visited,
learning)


                    # Add as hint in hint list
                    if found and learning:
                        if topic not in self.nodes[node].hint_set:
                            self.nodes[node].hint_set.append(topic)

                    return (found,depth)
        else:
            return (false,ttl)


    # Function to initialize the neighbourhood set of all nodes
    # in the simulated network
    # Args:
    # max_num = Maximum number of neighbours to fetch
    # random_num = Boolean, tells if each node should fetch
    #              either exactly (false) X neighbours
    #              or between 1 - X neighbours (true)
    # Returns: Nothing
    def fetch_neighbours(self, max_num, random_num):
        for element in self.nodes:
            # Fetch random amount of neighbours between
            # 1 and max_num if random_num is true
            # else always fetch max_num
            if random_num:
                ran = random.randint(1,max_num)
            else:
                ran = max_num

            element.insert_new_neighbours(ran, self.num_nodes)

    # Function to print the three sets of a node
    # Neighbour set, topic set and hint set
    # Used for debugging only
    # Returns: Nothing
    def print_lists(self):
        for element in self.nodes:
            print 'Neighbours: '
            print element.neighbours
            print 'Topic set'
            print element.topic_set
            print 'Hint set: '
            print element.hint_set
            print ''
    # Simulating gossip
    # Args:
    # probability = probability of gossiping to individual neighbour
    # ttl = TTL on gossip event
    # Returns: Number of messages and av depth of all gossips
    def sim_gossip(self, probability, ttl):
        # Traverse all nodes, make them gossip!!
        num = 0
        depth_of_gossip = 0

        for node in range(len(self.nodes)):
            if node % 100 == 0:
                print 'Gossiping from node %d' % node
            # Perform gossip
            tmp, depth = self.gossip(node, probability, ttl)
            num += tmp
            depth_of_gossip += depth

        return (num, depth_of_gossip)


    # Simulate search
    # Args:
    # topic = The node id/topic to search for
    # ttl = TTL on search event
    # learning = If search results should be incorporated into the hint set
    #            of initiating node
    # learning_plus = If search results should be incorporated into the hint set
    #            of all nodes participating in the search
    # Returns: If the search is successfull, depth of search,
    #          and node where the search was initiated
    def sim_search(self, topic, ttl, learning, learning_plus):
        # Find random start node
        ran = random.randint(0,self.num_nodes - 1)

        found, depth = self.search(ran, topic, ttl, learning, learning_plus)

        #print 'Search started on node: %d' % ran
        #if found:
        #    print 'Results found on depth %d' % depth
        #else:
        #    print 'search failed'
        return (found, depth, ran)


    # Count average number of hints per node in the network
    # Print result
    # Returns: Nothing
    def count_average_num_hints(self):
        num = 0
        # Traverse all nodes and ask them how many hints they have
        for element in range(self.num_nodes):
            num += self.nodes[element].count_hints()

        # Print result
        return 'Average number of hints: %d/%d = %d\n' %
(num,self.num_nodes,num/self.num_nodes)


    # Function deciding if we should gossip based on a given probability
    # Args:
    # probability =  The probability of gossiping to each individual node
    # Returns true if we should gossip, false otherwise
    def should_i_gossip(self, probability):
        # Fetch a random number N where a <= N >= b
        # and a = 0, b = 99
        num = random.randint(0,99)

        # Decide if we should continue gossip based on supplemented
        # probability!
        if num < probability:
            # Continue gossip
            #print 'Gossip'
            return true
        else:
            # Stop gossiping
            #print 'no gossip'
            return false

    # Simulate RW replication
    # Args:
    # TTL = The TTL on the RW replication
    # Returns the number of messages used on replication and the
    # total number of unique nodes visited
    def random_walk_replication_sim(self, TTL):
        num = 0
        # Traverse all nodes and perform random walk
        # replication from them
        for node in range(len(self.nodes)):
            visited = []
            hints = self.nodes[node].topic_set
            # Start simulation and fetch number of unique nodes visited by each RW
            num += self.random_walk_replication(node, TTL, visited,
self.nodes[node].topic_set)
            if node % 100 == 0:
                print 'RW replication from node %d' % node

        # Calculate number of messages sent in simulation
        messages = len(self.nodes) * TTL
        # Return number of messages used on replication
        return (messages, num)


    # The recursive function performing the work
    # Args:
    # node = The node we currently are on
    # TTL = The current TTL on the RW replication
    # visited = The list of already visited nodes
    # hints = the set of hints to distribute
    # returns nothing
    def random_walk_replication(self, node, TTL, visited, hints):
        num_visited = 0
        # Add hints to hint set
        for hint in hints:
            if not self.nodes[node].id == hint:
                if not hint in self.nodes[node].hint_set:
                    # Append hint
                    self.nodes[node].hint_set.append(hint)

        # Add this node as visited
        if node not in visited:
            visited.append(node)

        # Decrement TTL
        TTL = TTL - 1

        if TTL > 0:
            # Fetch neighbour list
            tmp = copy.deepcopy(self.nodes[node].neighbours)

            # Travers visited nodes
            for element in visited:
                # Remove all visited nodes from neighbour set
                if element in tmp:
                    # Node visited earlier
                    tmp.remove(element)

            # Is there any neighbours left
            num = len(tmp)

            if num > 0:
                # There are still neighbours not visited
                # Pick random neighbour
                neighbour = random.randint(0, num - 1)

            else:
                # There are no neighbours left
                # Pick one random among them all
                num = len(self.nodes[node].neighbours)
                neighbour = random.randint(0, num - 1)
            # Continue recursion
            num_visited = 
self.random_walk_replication(self.nodes[node].neighbours[neighbour], TTL, visited, 
hints)

            return num_visited

        else:
            # TTL is 0, return number of unique neighbours visited
            return len(visited)

# Function to fetch simulation data from file.
# Returns the simulation data found
def fetch_simulation_data():
    # Fetch tests and put them in a list
    tests = []
    # open file
    fp = open('sim.txt','r')

    data = true

    while data:
        tmp = []
        counter = 0
        # read and record contense
        while true:
            line = fp.readline()
            if counter == 16:
                # We have fetched an entire test case
                break

            # Test for end of file
            if line == "":
                # No more data in file
                data = false
                # Break out of for loop
                break
            else:
                #There is still data
                if not line.startswith('#'):
                    tmp.append(int(line))
                    counter += 1

        if len(tmp) == 16:
            # Simulation fetched, add it to test list
            tests.append(tmp)

    # Close file
    fp.close()

    print '%d simualtions added.' % len(tests)

    return tests

# Function to write simulation results to file
# ARGS:
# output = The data to write to file
# Returns nothing
def write_simulation_results(output):
    # Open file
    fp = open('sim_results.txt','a')
    # Write
    fp.write(output)
    # Close
    fp.close()

# The main function
def main():

    # Fetch simulation data
    sim_data = fetch_simulation_data()
    # Run the simulations fetched one by one
    for element in sim_data:
        print 'Simulation started'
        print element
        output =  '\n###### Simulation started ######\n'
        output += 'Test case:\n'
        for item in element:
            output += '%d,' % item
        output += '\n\n'
        # Fetch Tuning variables
        var = tuning_variables(element)

        # Begin timing
        s = int(time.time())

        # Initialize network
        simulator = sim(var.num_nodes)
        simulator.fetch_neighbours(var.max_neighbours,var.random_number_of_neighbours)

        output += 'Network initialized with %d nodes\n' % var.num_nodes
        if var.random_number_of_neighbours:
            output += 'All nodes has between 1 and %d neighbours\n' %
var.max_neighbours
        else:
            output += 'All nodes has %d neighbours\n' % var.max_neighbours

        # Warm up the network with a few searches
        # but only if both warm-up and learning is wanted
        if var.learning and var.warmup:
            print 'Warming up the network'
            output += 'Warming up the network\n'
            num_msg = 0
            for i in range(var.num_warmup):
                # Calculate random node to search for
                random_node = random.randint(0,var.num_nodes - 1)
                # perform search
                found, depth, ran =
simulator.sim_search(random_node,var.TTL_warmup,var.learning, var.learning_plus)

                num_msg += var.TTL_warmup - depth

            output += 'Num messages for warm-up\n: %d' % num_msg
            # Print average number of hints in hint_sets
            output += simulator.count_average_num_hints()

        # Test if we should perform gossip
        if var.gossip:
            print 'Performing gossip!'
            output += '\nSimulating gossip with a probability = %d, and a TTL = %d\n'
% (var.probability,var.TTL)
            num_gossip, depth_of_gossip =
simulator.sim_gossip(var.probability,var.TTL)

            # Print average number of hints in hint_sets
            output += simulator.count_average_num_hints()

            output += 'Number of messages sent : %d\n' % num_gossip
            output += 'Average per node %d/%d : %d\n' % (num_gossip, var.num_nodes,
num_gossip/var.num_nodes)
            output += 'Average depth per node %d/%d : %d\n' % (depth_of_gossip,
var.num_nodes, depth_of_gossip/var.num_nodes)

        # Test if we should perform random_walk_replication_sim
        if var.RW_replication:
            print 'RW replication started'
            num , num_visited =
simulator.random_walk_replication_sim(var.RW_replication_TTL)
            output += '\nRW Replication: %d \n' % num
            output += '\nUnique node visited in average %d/%d \n' % (num_visited,
var.num_nodes)

        # Test if we should perform search
        if var.search:
            print 'Performing search.'
            # Perform search in net!
            num_found = 0
            sum_depth = 0
            num_msg = 0
            output += '\nSearching... \n'

            for i in range(var.num_searches):
                if i % 100 == 0:
                    print 'Search num %d' % i

                # Calculate random node to search for
                random_node = random.randint(0,var.num_nodes - 1)
                # Do the search
                found, depth, ran = 
simulator.sim_search(random_node,var.TTL_search,var.learning, var.learning_plus)
                #print 'Search #%d started on: %d' % (i,ran)
                if found:
                    # Topic found during search
                    num_found += 1
                    sum_depth += var.TTL_search - depth

                num_msg += var.TTL_search - depth

            # Print search summary
            output += 'Number of messages sendt: %d\n' % num_msg
            output += 'Number of successes: %d\n' % num_found
            if num_found > 0:
                output += 'Average depth on success: %d/%d = %d\n' % (sum_depth, 
num_found, sum_depth/num_found)
                output += 'Percentage of success : %d/%d = %d\n' % (num_found, 
var.num_searches, num_found/var.num_searches)
            else:
                output += '%d\n' % var.TTL_search - sum_depth

        # Print average number of hints in hint_sets
        #simulator.count_average_num_hints()

        # Print time used on simulation
        e = int(time.time())
        output += 'Simulation time = %d\n' % (e - s)

        # Write results to file
        write_simulation_results(output)

        print output

# Start the program by calling main!
if __name__ == '__main__':
   # Call main
  main()
```

Appendix C

Appendix C: Describing the events

In this appendix we will present the structure of every event flowing into, through and out of our p-SARS system. External events are events flowing into and out of a p-SARS node. Internal events are events flowing between p-SARS nodes.

The search event

The external search event is used by the WAIF Recommender System (WRS) to ask the p-SARS system to perform a query on its behalf. This event is therefore the search interface p-SARS presents to other systems.

External event:

type	topic	reply_addr	happy	local
'search'	'topic'	('ip',port)	5	true

The local field tells the p-SARS node that it has received a query from a WRS client and that it has to create a pending search on this query and modify this search event into an internal event before propagating it onto the next p-SARS node. The reply_addr field describes the address where the WRS client wants the answer returned. Happy describes how many hits are necessary to satisfy this query and topic is the topic we should search for.

Internal event:

to	type	topic	reply_addr	happy	local	error	ttl	visited	hint	from
('ip',port)	'search'	'topic'	('ip',port)	5	false	false	125	[('ip',port)....]	true	('ip',port)

More information is needed when propagating the query between the p-SARS nodes. These extra fields are already described in the implementation chapter, the search event module section, and will not be revisited here.

The gossip event

The gossip event is an internal event used to populate the hint sets of other nodes in the p-SARS system.

Internal event:

to	type	from	ttl	error	new_gossip	death_gossip	get	push	gossip_dict
('ip',port)	'gossip'	('ip',port)	20	false	[topicX,...]	[topicY,...]	false	false	{'topic1':[('ip',port),..],..}

It has three different tasks based on the values set:

1. If the get field is true, there is another p-SARS node requesting the hint set of this node.

2. If the push field is true, there is a p-SARS node sending this node its entire hint set.

3. If neither get nor push is true we have an ordinary gossip event. That is, a node has discovered either a new or deleted topic in its topic list and therefore has initiated a gossip to help other nodes update their hint sets.

Important fields not already mentioned are:

- The new_gossip field which describes the new topics discovered during a topic update.

- The death_gossip field which describes the removed topics after a topic update.

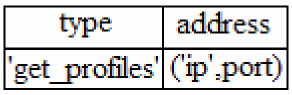
- The gossip_dict field containing a nodes entire hint list.

The topic update event

The topic set is updated by requesting an update from the corresponding WRS topic server. There are therefore two external events here. The p-SARS request event and the WRS reply event.

External event:

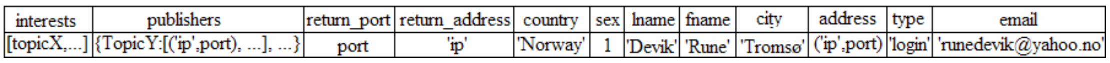
Request:



type	address
'get_profiles'	('ip',port)

Reply:

The replay event is in fact a list of events where each event contains different information about a WRS clients connected to this WRS super-node.



interests	publishers	return_port	return_address	country	sex	lname	fname	city	address	type	email
[topicX,...]	{TopicY:[('ip',port), ...], ...}	port	'ip'	'Norway'	1	'Devik'	'Rune'	'Tromsø'	('ip',port)	'login'	'runedevik@yahoo.no'

For use in the p-SARS system we only extract the interests, the IP address, and the e-mail address of the individual WRS users.

The result event

When a p-SARS node processing a query finds one or more publishers it needs to relay its findings to the p-SARS node in charge of gathering the results from this query. This is done with the internal result event. When the pending search on the p-SARS node in charge of the query either times out or gets satisfied, this node must relay the gathered results to the WRS client where the query initially originated. For this we use what we call the external result event.

Internal event:

to	type	from	error	topic	reply_addr	result
('ip',port)	'result'	('ip',port)	false	'topic'	('ip',port)	[('ip',port), ...]

External event:

to	type	topic	result
('ip',port)	'result'	'topic'	[('ip',port), ...]

The Debug event

The debug event is an internal p-SARS event used to send debug messages to the centralized debug module.

Internal event:

to	type	from	debug
('ip',port)	'debug'	('ip',port)	'debug info'

The debug field contains debug information.

The system event

The system event is either a ping or a pong event i.e. if the topic field is set to pong this is a pong event.

Internal event:

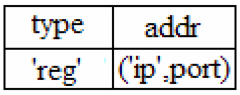
to	type	from	error	topic
('ip',port)	'system'	('ip',port)	false	'ping'

The bootstrap server event

To populate the neighbour set we use different techniques and one of them is to contact the centralized bootstrap-server.

External event:

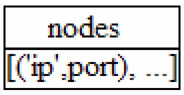
Request:



type	addr
'reg'	('ip',port)

The type field is either set to “reg” or “find”. “Reg” means that this event is of type register. The bootstrap-server therefore adds this node to the list of p-SARS nodes currently running in the overlay network. If the type field is set to find, the p-SARS node is not registered.

Reply:



nodes
[('ip',port), ...]

The bootstrap-server responds with the list of p-SARS nodes already connected to the overlay network regardless of what value the request event’s type field has.

Appendix D

Appendix D: CD-ROM

We have chosen to include test results, test code and also the code presented in appendix A and B on a CD-ROM. The data on the disc is organized into these folders:

- *p-SARS*: The full source code for our p-SARS prototype.
- *simulator*: The full source code for our simulator.
- *test-code:* Code implemented for testing purposes.
- *test-results:* All our test results presented in a Microsoft Excel worksheet.

The readme file included on the CD-rom describes how to start up the p-SARS system and what is needed.