TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Master's Thesis

Distributed Tuple Space

carried out at the

Information Systems Institute
Distributed Systems Group
Technical University of Vienna

under the guidance of
Univ. Prof. Dr. Schahram Dustdar
and
D.I. Martin Treiber

by

Stefan Spiegel
St. Antoniusstrasse 21a
A - 6890 Lustenau
Matr.No. 9825354

Vienna, August 31, 2007 _____

Abstract

This thesis illustrates issues and solutions to build a decentralized and distributed tuple space (DTSpace) on top of a Peer-to-Peer (P2P) network. The tuple space (TS) model was first described by David Gelernter in 1985. The TS provides communication and synchronization capabilities for distributed processes and applications. During the last two decades, many approaches have been developed for the TS. Most of them, for example, JavaSpaces from Sun or TSpaces from IBM, are centrally managed. An example for a distributed TS is SwarmLinda, which imitates the behavior of a swarm in nature to store and retrieve data items called tuples (e.g. like an ant colony).

The DTSpace approach in this thesis is divided into three parts: (1) TS, (2) P2P and (3) distributed mutual exclusion, which are examined in this thesis. The DTSpace approach provides the three operations *out*, *in* and *read*, which are originally described by David Gelernter. A distributed mutual exclusion algorithm approach is used to provide the mutual access to tuples.

Kurzfassung

Diese Diplomarbeit stellt Probleme und Lösungen vor, um einen verteilten und dezentralisierten Tupel Space (DTSpace), basierend auf einem Peer-to-Peer (P2P) Netzwerk, zu entwickeln. Das Tupel Space (TS) Modell wurde erstmals von David Gelernter im Jahre 1985 beschrieben. Der TS stellt Kommunikations- und Synchronisationsmechanismen für verteilte Applikationen zur Verfügung. Während der letzten zwei Jahrzehnte wurden viele Ansätze für den TS entwickelt. Viele davon haben einen zentralisierten Ansatz, wie z.B. JavaSpaces von Sun und TSpaces von IBM. Ein Beispiel für einen verteilten TS ist SwarmLinda. Dieser Ansatz imitiert das Verhalten eines natürlichen Schwarms, um Datensätze, so genannte Tupel, zu speichern und wieder zu finden (z.B. wie eine Ameisenkolonie).

Der DTSpace Ansatz in dieser Diplomarbeit ist in drei Bereiche unterteilt: (1) TS, (2) P2P und (3) verteilter Mutual Exclusion (wechselseitiger Ausschluss). Diese werden in der Diplomarbeit detailliert behandelt. Die drei Operationen *out*, *in* und *read*, welche ursprünglich von David Gelernter beschrieben wurden, werden vom DTSpace Ansatz unterstützt. Ein verteilter Mutual Exclusion Algorithmus wird verwendet, um zu gewährleisten, dass nur ein Prozess zur gleichen Zeit auf ein Tupel zugreifen kann.

Acknowledgements

First I want to thank my advisor Univ. Prof. Dr. Schahram Dustdar who supported me during this thesis and during the practical training before. Many thanks belong to Martin Treiber who helped me with valuable hints and with his experience in scientific work – we had a lot of fun. Hong-Linh Troug helped me with the application scenario and I want to thank him for his dedication. I also want to thank Roman Schmidt, Surenderreddy Yerva and especially Renault John from Ecole Polytechnique Fédérale de Lausanne (Switzerland), who gave me support for the P-Grid implementation.

I want to thank especially my friends Philip Köb, Nina Pichler, Karin Puchegger, Nurgül Sarikaya and Philipp Metzler who supported me during the difficult time of writing this thesis.

Last but not least particular thanks belong to my beloved brother Christian who was there, when I needed him and my parents, who made it possible for me to study and who supported me during my studies.

Table of Contents

1. Introduction

In this chapter I shortly introduce what a tuple space is. A tuple space can be viewed as a blackboard where, for example, people, can put notes on it, read or remove them. Consider, for example, a blackboard as a platform to search for or offer flats. The demands and offers for flats are written on little paper snippets (notes). The notes are pinned on to the blackboard. In tuple space terminology the notes are tuples and the blackboard is the tuple space. The people who are interacting with the blackboard, i.e. read, remove or put notes onto the blackboard, are called processes or nodes. There are three interactions with the blackboard: (1) pin a note onto the blackboard, (2) remove a note and (3) read a note. The tuple space offers these three operations: (1) *out* to write a tuple into the tuple space, (2) *in* to remove a tuple and (3) *read* to read a tuple.

In the tuple space it is possible to wait for tuples, which does not exist in the tuple space yet. This would be the demanding notes on the blackboard. For example, if someone is looking for a flat, he or she puts a note on the blackboard and waits until someone, who offers a suitable flat, notifies the looking person. The looking person has to sit tight until a notify message is made (e.g. someone calls, that he or she is offering a suitable flat, for which the looking person searches for). In tuple space terminology this is called blocking. If a requested tuple does not exist, the demanding node waits until a suitable (also called matching) tuple is inserted into the tuple space.

It is possible that demanding and offering notes for a suitable flat co-exist on the blackboard. For example, the offering person and the demanding person do not look for suitable demands or offers. Therefore both the offering and demanding person are waiting indefinitely, if no one does anything further. In contrast to the blackboard the tuple space guarantees that, if a node is requesting a tuple and a matching tuple is inserted into the tuple space, the requesting node gets the tuple.

The blackboard example is an abstraction of a tuple space. A tuple space is a virtual space for notes, i.e. tuples. A tuple space can be used for communication

and synchronization between distributed processes and applications. How this communication works in detail is discussed in Chapter 3.1. This thesis works out an approach for a distributed tuple space.

The remaining thesis is organized as follows. Chapter 2 discusses the motivation, the problem discussed in this thesis and an application scenario. Chapter 3 gives an overview of different tuple space implementations; distributed hash table (DHT) networks and discusses distributed mutual exclusion approaches for Peer-to-Peer (P2P) networks. In Chapter 4 the distributed tuple space (DTSpace) approach is discussed in detail. Chapter 6 evaluates the DTSpace prototype implementation in different test scenarios. Future work and novel approaches are discussed in Chapter 6. A summary is found in Chapter 7 and a conclusion is presented in Chapter 8. The Appendix contains a short discussion of the DTSpace prototype implementation (architecture, configuration and usage).

2. Motivation

The motivation of building a distributed tuple space on top of a P2P network is based on the need of distributed applications. In the last decade, distributed networking and applications became very popular. One reason is that more and more computers are connected with each other through networks and are able to collaborate with each other. The biggest network at the moment is the Internet with millions of participating computers. Another reason is that the home computers have become very powerful and are able to share their resources. One of the classic examples is Napster – a file sharing P2P network. During the late 90's, Napster attracted millions of users to share their files. The success of Napster is its simplicity in usage and powerful collaboration of resources. The distributed tuple space also contains these two qualities. The distributed tuple space is simple to use, since there are only three operations. However, the distributed tuple space provides powerful features like synchronization and mutual exclusion capabilities. The additional benefits of a distributed tuple space are that it is dynamic, scalable and can be used in heterogeneous networks. Besides the distributed approach for communication like a distributed tuple space there is a classical approach for communication between network nodes – the client-server approach.

The client-server approach is not very scalable if the number of clients is large, since the server has to handle every client and thus may become a bottleneck. The disadvantage of using a centralized approach for a tuple space in a distributed system is that it scales not very well, since the centralized components may become a bottleneck on heavy load. Another benefit of using a distributed tuple space is fault tolerance, since centralized approaches present a single point of failure, if no special precautions are taken (i.e. redundancy).

Another reason to build a distributed tuple space is that most tuple space implementations are centrally managed (e.g. IBM TSpaces, JavaSpaces from Sun). There are as well some distributed approaches (e.g. SwarmLinda). As discussed in [5], the development of SwarmLinda is in an early stage.

Since the P2P networks have become quite sophisticated, it is a close choice to use a P2P network (i.e. DHT) as underlying organization and communication platform to implement the distributed tuple space. One reason to use an already existing P2P network implementation is that it does not make sense to invent new routing or storage strategies if they already work. Another reason is that the focus can be set on the tuple space and not on P2P networking. The approach in this thesis is a proof of concept for a dynamic distributed tuple space using a P2P network (i.e. DHT). However, it is important to assess the performance of a distributed tuple space.

2.1 Application Scenario

Multiple organizations and commercial companies are involved to coordinate and organize big sport events like the European Football Championship in 2008. The championship is distributed among several European cities in two countries. During the event, associated organizations need to share heterogeneous information for communication/coordination purposes. Examples include schedules of the soccer games, the transportation of the teams from one accommodation to another, coordination of volunteers, etc. In addition, sudden changes of schedules, like flight delays when moving a soccer team from one city to another, must also be coordinated among different organizations (an airport can inform the hotel and the transportation organization, which waits for the soccer team to transport it to the hotel, etc.). Figure 1 illustrates the application scenario.

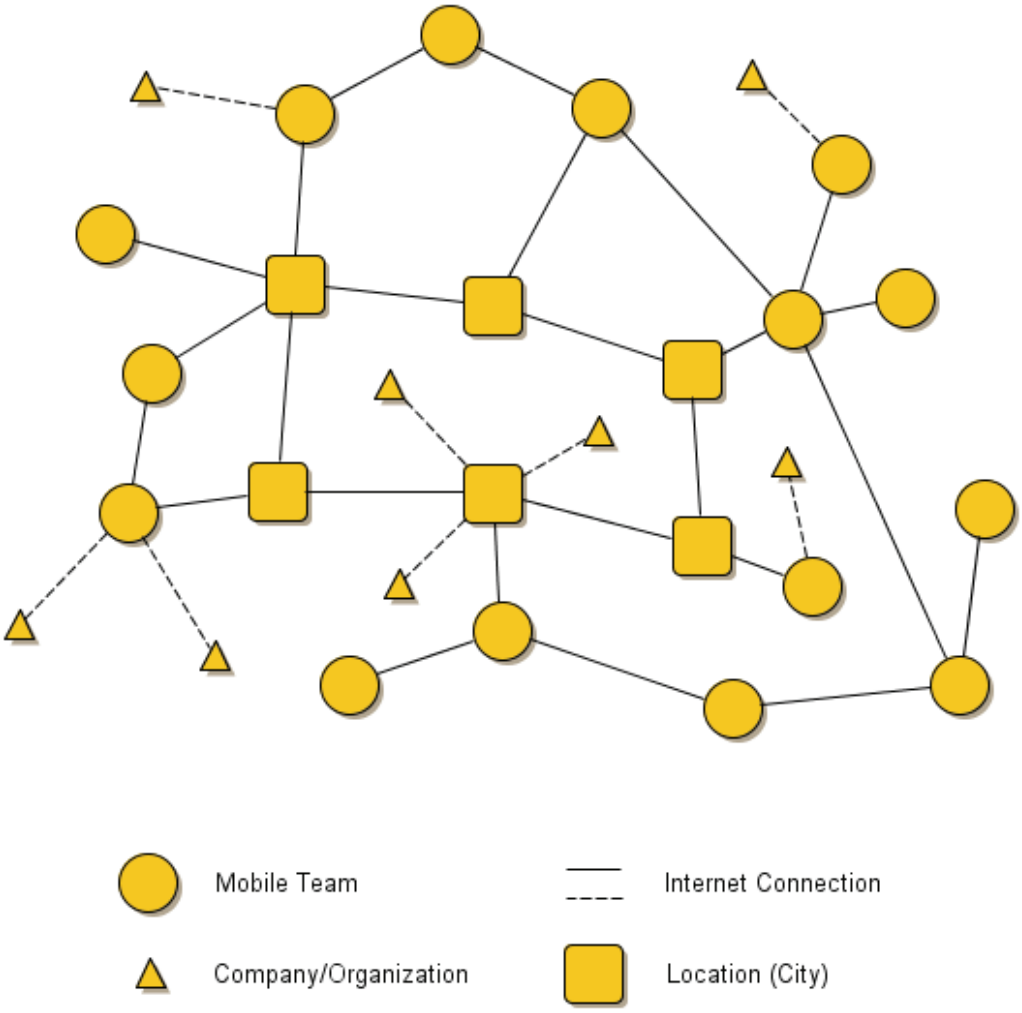

Figure 1: Application scenario overview. Note that a mix of ad hoc/mobile peers and a stable backend, i.e., the cities, are considered.

In the scenario, the DTSpace middleware architecture is based on P2P technology, which is fully decentralized, self organized, fault tolerant and scalable. The participants share responsibilities, because every participant adds resources, while they participate. Therefore, no bottleneck emerges and this approach provides a dynamic communication and coordination network. The middleware itself provides a set of basic operations, which are used to interact with participants of the middleware.

The scenario foresees front end applications, i.e., applications that utilize DTSpace middleware but do not provide itself middleware functionality, for mobile teams or

individuals with mobile devices. These devices are connected to the network using available hot spot base at the different locations, e.g. football stadiums, city centers, etc. The organizations use their Internet connection to communicate with their mobile teams or other organizations. For example, volunteers give feedback, if they notice traffic jam. The traffic coordination then diverts the traffic accordingly.

3. Related Work

In this chapter I introduce the three main topics, into which the DTSpace approach is divided. The first part contains a discussion about tuple spaces. The second part gives an overview over P2P networks with the focus on distributed hash tables (DHTs). A tuple space and DHT network implementations are also discussed in this chapter. I discuss mutual exclusion for distributed systems in the last part of this chapter.

3.1 Tuple Space Model

As discussed in Chapter 1 the tuple space model (TS) can be viewed as a blackboard for notes. The distributed programming language Linda [1] is an example for a TS, described by David Gelernter in 1985. The TS communication is based on the so-called generative communication. Generative means, that if processes need to communicate, they generate a data object (i.e. tuple, note) and put it into a shared space called TS [6]. The tuple exists in the TS until it is explicitly deleted from the TS. A tuple is an ordered vector consisting of value-type pairs, also called fields. It can contain executable code (active tuple) or data values (passive tuple) [1]. Any process can access every tuple in an equivalent way [1]. In this thesis the TS is referred to the Linda model.

A distributed programming language like Linda, in general, enables message passing mechanisms and code execution for distributed systems [7]. In most distributed languages the message passing (i.e. communication) between processes is partially uncoupled in space and is not uncoupled in time [1]. The Linda model provides both. The communication in the Linda model is fully uncoupled in space and in time. For example, if a process A wants to communicate with a process B, it generates a tuple and writes it into the TS. The tuple exists in the TS until a process removes it from the TS (uncoupled in time) [1]. The processes A and B need not to run at the same time in order to be able to communicate with each other. If processes communicate over the TS the communication is anonymous, because the

tuples do not refer to their originating process and the receiver does not notify the originator of the tuple, if it removes a tuple. For example, process A does not know which process removes the tuple, which it has written into the TS. Process B does not know which process has written a tuple into the TS. Uncoupled in space means, that for example, k distributed processes running in k different address spaces can communicate over a common TS. The tuples written by different processes (thus coming from different address spaces) may be tagged with the same name. Every process can manage the tuples although they come from different address space [1].

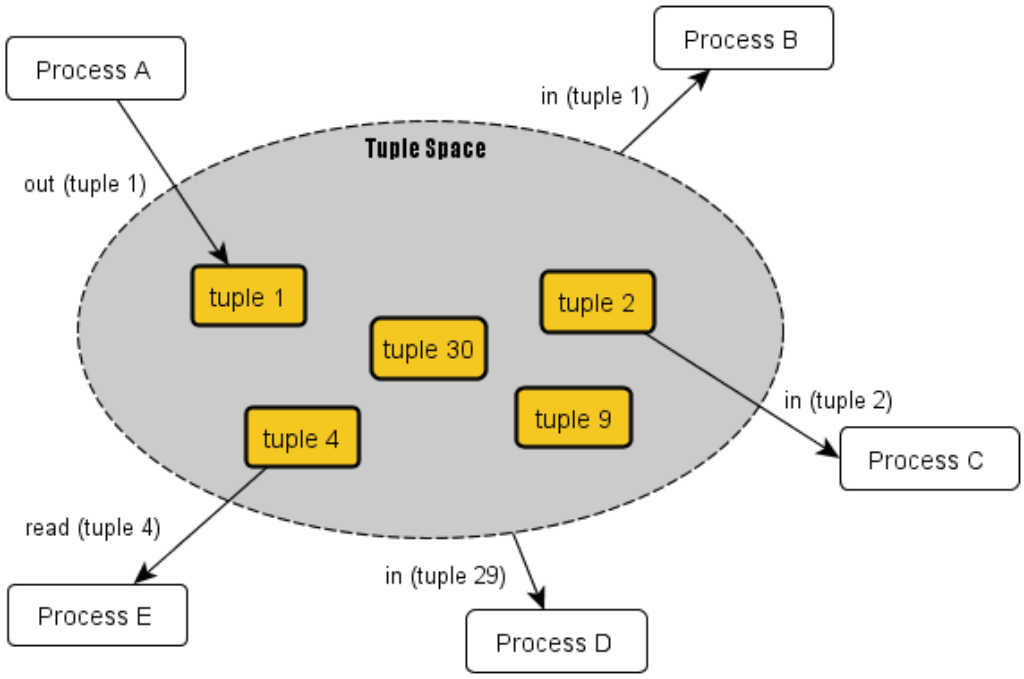

Figure 2: Example of a tuple space

There are three operations defined in the TS, which are (1) *out*, (2) *read* and (3) *in* [1]. The *out*-operation writes a tuple into the TS. The *read*-operation reads a tuple and the *in*-operation removes a tuple from the TS [1]. The *read*- and *in*-operations are blocking operations. To search for a tuple a so-called template is defined for the *read*- and *in*-operation. A template contains values for the fields and wildcards (also called formals [1]). Wildcards are placeholders in templates, if a value of a field is not important for a match. Consider a process, which wants to read a tuple

from the TS. The process first defines a template, which is then matched among the tuples in the TS. If a matching tuple in the TS exists, it is returned. If no matching tuple exists in the TS, the *read*-operation blocks until a matching tuple is available in the TS [1]. If more than one tuple match the template, an arbitrary tuple from the result set is returned.

Figure 2 illustrates a TS with 5 participating processes. It is possible, that different applications use the same TS without disturbing each other. Therefore not every process communicates with every other process participating in the TS. For example, process A writes a tuple into the TS and process B removes it. Process C removes an existing tuple from the TS. Process D is blocking and waits for a tuple, which does not exist in the TS yet. Process E reads a tuple from the TS, but does not change it.

The blocking operations *read* and *in* can be used to synchronize processes. For example, the *read*- and *in*-operations can be used for asynchronous message passing [7]. Consider a process A only writes tuples into the TS and a process B only removes them. Process B has to wait for tuples, which process A writes into the TS. Therefore process B is dependant on process A. Process A is able to process faster than process B, since process A is able to write tuples into the TS without waiting for process B to remove them. In addition synchronized message passing can be implemented on the application layer using TS. For example, every time process A writes a tuple into the TS, it waits (blocks) for a tuple from process B to read it, e.g. a tuple which says “I am ready – now continue”, written by process B. After process B has written such a tuple into the TS, process A is allowed to continue processing.

The Linda model is a simple coordination tool for distributed applications. There are also some criticisms with reference to its performance, because a TS hides its complexity like data sharing and the required communication. According to [3], this may result into unpredictable performance.

Many approaches have been made to implement the TS. There are several approaches to implement a TS [3, 8]:

- **Centralized:** Use a centralized server to provide the TS. This may be effective to match tuples, but may become a bottleneck to handle a big number of processes.

- **Hashing:** The placement of tuples is dependent on the content. The TS is partitioned among several processors (i.e. servers).

- **Full replication:** All servers hold a copy of all tuples in the TS.

- **Fully distributed:** The tuples may reside on every participating processor without any restrictions.

3.1.1 Tuple Space Implementations

In this section, I discuss the following TS implementations:

- **JavaSpaces** (by Sun Microsystems): As the name suggests JavaSpaces is a TS specification and implementation, written in Java. It is based on other systems from Sun, like Java RMI (Remote Method Invocation) and JINI [3]. The tuples, stored in JavaSpaces are Java Objects [2]. JavaSpaces supports transactions, which is an extension of the original Linda model [9]. JavaSpaces is a client-server approach [2].

- **GigaSpaces**: GigaSpaces [10] is a commercial implementation of the JavaSpaces specification. JavaSpaces is built into a product named GigaSpaces Enterprise Edition. A feature of GigaSpaces is that non-Java clients can access GigaSpaces through the SOAP protocol [3].

- **The Blitz Project**: Blitz [11] is an open-source implementation of JavaSpaces. The implementation is also based on JINI to use its services. It provides scripts to install and configure the TS easily. Blitz also provides monitoring functions. The monitoring tool is called Dashboard and has a simple GUI [9].

- **IBM TSpaces**: TSpaces is a commercial and centralized Java implementation of TS. "TSpaces provides a large number of operations over and above the basic Linda operations" [3, p.1007]. For example, it provides an event notification mechanism, which can notify registered nodes, if a tuple is written or deleted from the TS [3]. TSpaces also provides matching mechanisms on XML data stored in the tuple [3].

- **XMLSpaces**: XMLSpaces [8] is based on the TSpaces implementation of IBM. The system provides a flexible matching mechanism for XML data. Instead of a centralized server XMLSpaces provides a distributed and partial replication approach with several servers on different locations. The distributed servers provide one logical TS. The XMLSpaces uses Java RMI (Remote Method Invocation) for communication.

- **SwarmLinda**: SwarmLinda [4] is a distributed TS implementation in Java. The positioning and retrieval mechanism of tuples is based on swarms in nature, e.g. like in an ant colony. The system is completely distributed and self-organized. The development of the Java of the SwarmLinda implementation is in an early stage [5].

- **LighTS**: LighTS [12] is a centralized lightweight open-source Java implementation of the Linda model. LighTS extends the Linda model with more operations than the basic operations *out*, *read* and *in*. In contrast to IBM's TSpaces LighTS does not provide functionality like security or transactions, etc. LighTS supports Java RMI for remote access and is extendable.

The centralized JavaSpaces and TSpaces implementations may become a performance bottleneck with a large number of participating processes [3].

3.2 Peer-to-Peer Networks

This thesis discusses a decentralized approach, which uses a Peer-to-Peer (P2P) network for communication between processes and storage for the tuples.

A P2P system is defined as follows:

"Peer-to-peer systems are distributed systems that operate without centralized organization or control." [13, p.79]

Peer-to-Peer (P2P) networks belong to the classification of overlay networks. In the last decade P2P networks have become more and more popular. Especially file-sharing tools [14] like Napster, Gnutella [15] and BitTorrent [16]. These systems use the P2P technology to share all kind of data. The term P2P originally was used to describe the communication between two peers [15]. It is also a "point-to-point communication between two equal participants" [15, p.24].

In this thesis I follow the definition of overlay networks:

"An overlay network is a virtual network of nodes and logical links that is built on top of an existing network with the purpose to implement a network service that is not available in the existing network." [17]

As the term overlay network is very general, there are many sub classifications. For example, a related sub class is Grid Computing [15]. This thesis discusses the P2P approach, as it would go beyond the scope of this thesis.

P2P networks can be sub classed into unstructured and structured P2P networks as discussed later in detail. In unstructured P2P networks the connections between the

nodes in the network is randomly, the routing of messages is uncoordinated (e.g. broadcasts) and the data placement is not organized (similar data can reside in other parts of the network). This strategy offers a complete decentralized approach of a network. The structured P2P networks do coordinate routing of messages and the data placement is structured (e.g. a datum is uniquely mapped into a binary tree, where it can be found deterministically). The structured P2P network defines a new layer on top of the unstructured P2P network – a logical layer, which organizes and maintains routing tables and data items. This logical layer means, that more maintenance has to be done (e.g. keep the structure, like routing tables and binary trees up to date).

The two definitions of the terms P2P and overlay network have some things in common – they build a network or system, which collaborate with each other. There are two main ideas behind these collaborations of computers or systems:

- Share and merge resources
- Abstraction

P2P networks have been used to **share** resources since the beginning of the Internet. The ARPANET, for example, was originally used to share computing resources [15].

In 1999, Napster is one of the first P2P networks. Its purpose is to share and swap files [18]. With the popularity of Napster the term P2P nowadays is strongly related to file-sharing tools. However P2P networks can be used to share other resources than files. For example, to combine and share unused resources like CPU-power, storage place, network bandwidth [19]. Personal home computers are getting faster and more powerful recently and may have unused resources. This situation, for example, uses the SETI@home project to compute radio signals to search for extraterrestrial intelligence [19]. It distributes the collected radio signals among the participating computers, which for themselves analyze them. The result of

SETI@home is that it is faster than ASCI White, a supercomputer from IBM with 12.3 TFLOPS [19].

Another range of application of a P2P network is **abstraction**. Nowadays there are many different devices like PDAs, mobile phones and home computers with different communication protocols. The communication protocols may be Bluetooth, HTTP, TCP/IP and many others. One way to communicate with each other is to implement a P2P network, which puts a virtual layer on top of the different devices to use the same protocol [15] and also able to communicate with each other.

The classical and widely spread approach for communication is the client-server interaction. Usually the information is concentrated centrally on a server. The Internet is an example for the client-server approach. Web pages are hosted on servers, which are requested by client applications like web browsers [20]. Clients are also able to send data to the server, for example, if filling out forms or order books on an online store. The role of the participants in the communication, client and server, is rigid. In a P2P network, the participants (peers) have equal roles. Every peer acts as a client and a server at the same time. Thus a peer in a P2P network is also called servent. Another difference to the client-server approach is that the resources of the peers, e.g. storage place, CPU power, network bandwidth and others, can be shared among the peers in the P2P network. This results in to a huge resource pool, which can be used for distributed calculations, e.g. SETI@home as mentioned above or for distributed databases like OceanStore [21].

All P2P approaches have some common aspects and goals:

- Scalability
- Robustness
- Fault-tolerance

Scalability is one of the key aspects in P2P networks. The client-server approach has a bottleneck – the server. The computing power and bandwidth of a server is limited and if many clients query the server, it may become a bottleneck. There are many approaches to solve these problems (e.g. server farms) but there is always a limited amount of resources to handle the clients. The benefit of P2P networks is that every node in the network acts as a client and as a server at the same time. Therefore the computing power and bandwidth is distributed all over the participating nodes, instead of centralizing all resources at one place, where it may become a bottleneck. Another benefit is that with every participating node, the resources, available in the network increases, e.g. storage place or CPU power. Every node can add its resources to the P2P network. So, P2P networks are able to automatically adapt the resources while growing and keep being scalable to a theoretically infinite number or participating nodes (also called peers).

Another aspect of P2P networks is **fault-tolerance**. The P2P network is under constant change if peers join or leave the network. This change in the network is called churn. Participants of the P2P network are allowed to join and leave without warning. The benefit of the P2P network is that the resources are distributed and a failure or leave of a participant does not impact the whole system. Even if many nodes fail, every node in the network maintains several connections to different nodes [15], making it **robust**, for example, against single connection failures.

To find data items in distributed networks, more effort has to be done, than in centralized systems, because there is usually no global view over all existing data items at any node and at any time in the P2P network [22]. Consider for instance a centralized database with 10 data items. The worst case is to check all of these 10 items during a search process, but if the data item exists, it can be found. In a P2P network consisting of hundreds or even millions of nodes, it cannot be guaranteed, that a data item is found, even if it exists. One reason is that P2P networks are, by nature, unreliable and distributed. Another reason is that search algorithms are limited, as discussed later. A benefit of a centralized database is that the number of data items in the database is known (e.g. check the size of the data table).

In this thesis, I focus on the following approaches to manage a search in a distributed environment:

- Centralized
- Flooding
- Random walks
- Distributed Hash Tables (DHTs)

I do not consider hybrid approaches, since it is out of the scope of this thesis.

Napster has implemented a **centralized** approach to search for data items. Every node in the network sends a list of content (e.g. file names, keywords), which it shares, to a centralized server when joining the network. The server maintains the list of content (index) for every node in the network. Therefore the server has a global view over all data items, which are available in the P2P network. If a node searches for a keyword it sends a request to the server, which searches for a match in its index and returns the result. The result contains the node, which shares the requested content. Then the searching node contacts the node directly and downloads the data item from this node. Only keywords, file names and on which node the datum resides are maintained centrally. The actual datum remains among the participating nodes and the actual download is managed directly between the requesting and the datum-sharing node. One major drawback of this approach is that the index server is a single point of failure and search results can be filtered (e.g. censored).

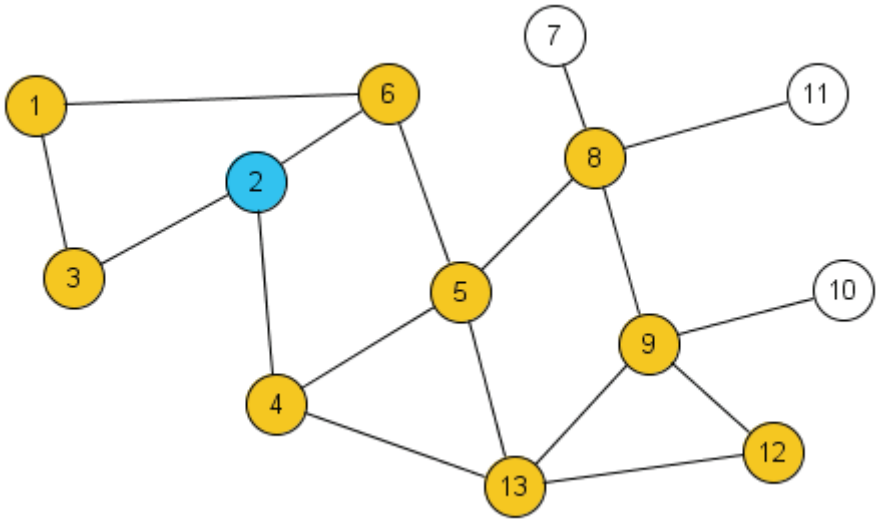

Figure 3: Example of a P2P network with the flooding approach and TTL = 3. All nodes except number 7, 10 and 11 are inside the Gnutella horizon, if the request message is sent from node 2.

A different search approach is **flooding** [13]. For example, Gnutella [15] uses this kind of search algorithm. The search request is sent to all neighbors of the requesting node. The neighbors themselves look locally for a match. If a node matches the request, it returns the result. Otherwise the node forwards the request to all its neighbors (except to the one, from which it got the request) and so on. To limit the traffic during a search process, the request message gets a time to live (TTL) stamp, which is decreased every time it is forwarded. If the TTL reaches 0, the request message is deleted. For example, if the TTL is 5, the message can be forwarded only 5 times. "The TTL is also known as the Gnutella horizon" [15, p.104]. There are two major drawbacks in this flooding approach. First, it causes much traffic overhead, since the request is sent to all nodes in the Gnutella horizon. Second, if the P2P network is broader than the Gnutella horizon, the request message does not reach every node in the network. If the match for the request is outside the Gnutella horizon, the request does not deliver a result, although the datum exists. Figure 3 illustrates a situation, where the Gnutella horizon is too small to find a requested datum, although it exists. The blue node with the number 2 sends a search message with TTL = 3. The message only reaches the yellow marked nodes (1, 3, 4, 5, 6, 8, 9, 12 and 13). The nodes with the numbers 7, 10 and

11 are not reached, because the message is deleted before it is forwarded to them. If the requested datum only exists on one of the nodes with the numbers 7, 10 and 11, the requesting node with the number 2 does not get a result for the search message.

The **random walk** approach "forwards a query message to a randomly chosen neighbor at each step until the object is found" [23, p.89]. The improvement is a significant reduction of the traffic during the search process. The random walk approach increases the waiting time for the result, because the nodes are only queried one after another. In the flooding approach many nodes are queried in every step, thus reaching many nodes in a relatively short time. In the random walk approach only one node is queried in every step. The search message is called "walker" [23]. To minimize the delay time more than one "walker" can be sent, but at the cost of more traffic. The drawback of using multiple "walkers" is that the "walkers" have to be stopped, if a search result has been found. The "walkers" walk forever, if they don't find a result. Consider for example, a network that starts 10 "walkers" during a search. One "walker" finds the datum and terminates. But the other 9 "walkers" still search for a result although one result has been found. The question arises how they can be stopped. Limiting the time of the "walkers" with a TTL has the same problem as discussed in the flooding approach, which is illustrated in Figure 3.

The **DHT** approach follows a complete different approach to search for data items. The search algorithms flooding and random walk take the structure of the P2P network as it is and tries to search with best effort for data items. The DHT approach is more than a search algorithm – it structures the P2P network and "use precise placement algorithms and specific routing protocols to make searching efficient" [23]. The DHT is discussed in detail in Chapter 1.3.

Much research effort has been put into the optimization of the search process in P2P networks [23, 24, 25, 13]. For example, [26] discusses an aggregation/broadcast algorithm, which is able "to collect and disseminate information efficiently on a global scale" [26, p.81]. The algorithm needs

additional storage place, but causes relatively small overhead and is robust against node failures [26].

Since the resources in a P2P network are distributed over the participating nodes, these distributed resources have to be managed, so that they are available, when needed. “In order to resolve the problem of resource management in P2P systems, many solutions have been suggested recently by research institutions and industries” [20, p.36]. There are two main classes in decentralized P2P networks, which themselves use different management approaches - unstructured and structured P2P networks.

3.2.1 Unstructured Peer-to-Peer Networks

Unstructured P2P networks do not maintain information about resources of other peers and therefore message forwarding cannot be routed intelligently [27]. Also “no rule exists that defines where data is stored and which nodes are neighbors” [20, p.37]. Therefore the data is distributed unorganized and a data retrieval turn out to be expensive or even impossible, as discussed in the previous section. Gnutella is an example for a completely unstructured P2P network. It is also called a “pure” P2P network, because it does not have any centralized mechanisms like Napster. For example, when Napster has to shut down its index servers in 2001, because of copyright problems with the content, which was shared in the Napster network [18], the Napster did not work any more without these index servers. It is more difficult to shut down the Gnutella network, because there is no centralized point to shut down in order to disable the whole network. To connect to the Napster network every node has to connect to one of the index servers. In the Gnutella network, every node can connect to an arbitrary node, which already is connected to the Gnutella network in order to participate. If a node in the Gnutella network fails, the connectivity of the network remains, since every node maintains several connections to its neighbors, thus is making it robust. The drawback of such an unstructured P2P network is that it is not very scalable. As discussed in the previous section, the search is limited by the so-called Gnutella horizon. Even

when using a random walker to search for data, it may not be found, because of the random approach. For example, the request message may be sent into the wrong direction, since the direction is chosen randomly and not by considering the content of the message.

One major drawback of an unstructured P2P network is the unorganized distribution of the resources. In order to find a resource, the node, which holds the resource, has to be found. The search algorithms have to deal with the unorganized topology of the network. Consider two equal resources, each residing on different parts of the network. To find both resources the query has to be routed into two different directions. Therefore a broadcast or random search is nearly inevitable. To make operations in the P2P network more efficient, the network has to be structured. But with more structure comes more complexity and new issues to solve, as discussed in the next sections.

3.2.2 Structured Peer-to-Peer Networks

To solve the scalability and search problems in unstructured P2P networks, many different approaches have been proposed. They form the class of structured P2P networks.

There are two architectures of structured, decentralized P2P networks [23]:

- Loosely structured
- Tightly structured

In **loosely structured** P2P networks the topology and the routing is controlled based on local decisions. In unstructured and loosely structured P2P networks the datum and its index reside on the same node or nearby in a physical sense (e.g. one-hop replication as discussed later). In **tightly structured** networks the index datum placement is also controlled based on local decisions [23, 27], but the datum

storage and the index storage are uncoupled and may reside separately. The datum storage is still arbitrarily organized, but the index storage of the data is organized to make the search process more efficient.

The data-index is a pointer to the node, where the actual datum is stored. The index is also used as keyword for which can be searched to find the correspondent datum in the P2P network.

Loosely structured P2P networks improve the way search messages are routed. There are simple, but effective ways to do this. For example, in an improved Gnutella version every node gets a capacity indicator [25]. High-capacity nodes can handle more traffic than low-capacity nodes. If a message is forwarded, it is sent to a neighbor with high capacity instead to a randomly chosen node. Therefore the traffic is routed more efficiently with focus on traffic handling, because it is routed along nodes with enough power to process it. A drawback, when using only the capacity approach is that if the queried datum resides on a low-capacity node, the datum may not be found, because the message is routed only to high-capacity nodes. An extension of this approach is the so-called one-hop replication, where "all nodes maintain pointers to the content offered by their immediate neighbors" [25, p.409]. This increases the possibility to find a datum, because the index resides on more than one node. This approach also reduces the traffic, since the message can be answered before it reaches the content holding node.

Freenet, for example, uses an adaptive routing scheme [27]. Every node examines the messages (e.g. search requests, data insertions), which are passed to the node, to learn where to route the messages more efficiently. For example, when examining an answer message, the node stores from where this answer comes from. Later, when a new request for this datum is passed to this node, it searches the routing table and finds the location of the datum. The message then is forwarded towards the location of the datum. Freenet also replicates popular data "to peers where they are more likely to be found" [28, p.8].

Tightly structured P2P networks go one step further. They also structure the data-index itself. The goals are to "enable more deterministic resource location operations with shortest lookup path, fault-tolerant operation and high availability" [20, p.46]. The storage space for data-index is addressed uniquely and every node is responsible for one part of the address (ID) space.

In tightly structured P2P networks there are two types of neighborhoods for nodes: physical and logical neighbors. A node keeps direct connections to other nodes (e.g. TCP/IP connections) for sending and routing messages. These directly connected nodes are called physical neighbors.

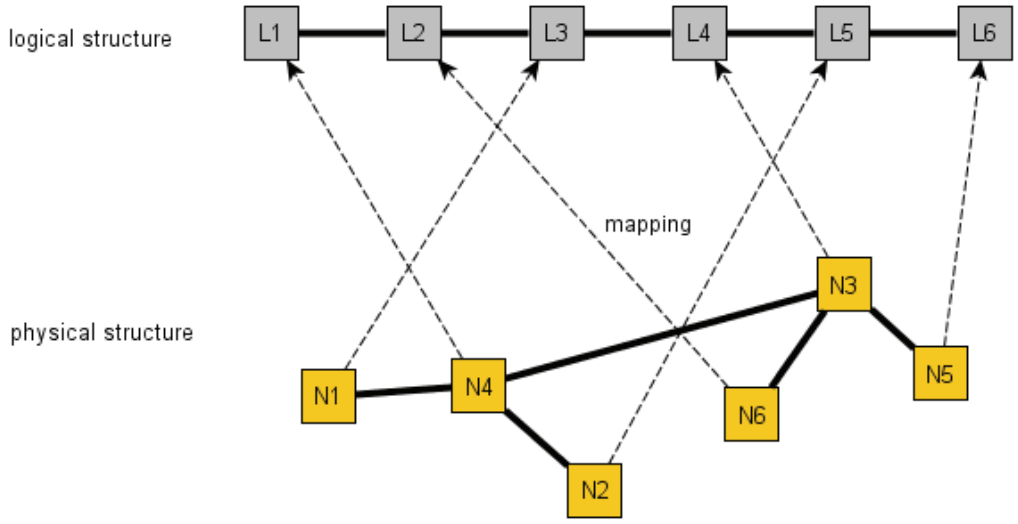

Figure 4: Example for logical and physical neighborhoods in tightly structured P2P networks. The solid lines are direct connections between the nodes. Dashed arrows show the mapping of a physical node to the logical node (e.g. the physical node N1 is mapped to the logical node L3).

The tightly structured P2P network builds up a logical structure over the physical connected network. In the logical structure every node gets a logical place (via mapping). For example, every node gets an ascending number as they arrive in the network. The direct neighbors of a node in the logical structure are called logical neighbors. A physical neighbor of a node needs not to be a logical neighbor. Figure 4 illustrates an example where the nodes are mapped to a logical structure. For example, the node N3 has three physical neighbors, namely N4, N5 and N6. N3

maintains actual connections to these physical neighbors and is mapped to the logical place L4 in the logical structure. The logical neighbors of node N3 (logical place = L4) are L3 and L5, which are the physical nodes N1 and N2. The nodes N1 and N2 are physically not connected to the node N3. The logical structure is used to search for data, since it is efficiently structured in the logical level. For example, if the node N4, which is mapped to L1, wants to send a message to its direct logical neighbor L2, the node N4 only needs one hop at the logical level. In the physical layer, this message has to be routed along the following physical path: N4 – N3 – N6. N6 is the physical node, which is logically mapped to L2.

The benefit of a tightly structured P2P network is that the logical structure can be organized independently. For example, independently of the physical structure, the logical structure can be organized to efficiently find data. The most popular approach for a tightly structured P2P network is the Distributed Hash Table (DHT) [20]. P2P networks, which use DHTs, are discussed in detail in the next section.

3.3 Distributed Hash Tables

DHTs play a central role in this thesis, because the DTSpace prototype implementation is built on top of P-Grid [29]. P-Grid is a P2P network implementation, which uses a DHT. DHTs are tightly structured P2P networks, which uncouples the index location from the resource location as discussed in the previous section.

As discussed before the P2P networks have problems with finding resources, because of the distributed nature of P2P networks. The approach of DHT maps every resource to a unique identifier (ID), also called index. The ID can be calculated with a hash function from the resource value (e.g. file name). Every ID is taken from a so-called ID space, which is a range of IDs to identify resources. For example, the integers range from 2^0 to 2^{128} can be an ID space. Also hash functions can be used to calculate IDs for the resources. How the resources are

mapped into the ID space “has an important impact on the load-balancing properties of the” P2P network [30, p.13]. For example, Chord uses a uniform hash function, which “implicitly provides load-balancing” [30, p.13]. The drawback of this uniform distribution of the IDs is that “range queries will be expensive to process” [30, p.13]. For example, the hash function may distribute similar values to nodes, which reside on different parts of the P2P network, because they get complete different IDs. Therefore the similar values with complete different IDs have to be searched in different parts of the network. However, if the hash function clusters similar values, the load balancing has to be done explicitly, like in P-Grid [30].

The ID space is divided into ranges. Every node in the P2P network gets a range for which it is responsible. If a new resource is inserted into the P2P network, its ID is calculated with the hash function. The ID, together with a reference to the actual resource location is stored on the node, which is responsible for the ID. The resource itself (e.g. file) remains on the original node.

Maintenance strategies [30] take care that the DHT structure remains consistent and functional during churns. To maintain fault-tolerance, replication of the IDs is a common used technique. Routing strategies are used to route queries to their destination [30] and also support the maintenance strategies by routing maintenance messages. Also redundant routing entries for the same destination are maintained to be able to route a message on an alternative route, if a routing entry fails.

There exist different approaches for the logical structure of a DHT. For example, a ring structure, which is used in Chord [31] or a tree structure like in P-Grid [32]. Figure 5 illustrates two possible structures. The yellow marked rectangles in Figure 5 are participating nodes in the P2P network. Figure 5 (a) shows a ring structure, as used in Chord [20]. As shown in Figure 5 (a) every node has a successor node and also maintains a predecessor node. The ID space is successively assigned along the successor relations. For example, node 1 is assigned the first possible ID. Node 2

holds the following ID and so on. Also wide range relations exist, which minimizes the needed hops of a query by leaving out nodes in certain cases as discussed later in detail (refer to Chapter 3.2.2). In Figure 5 (b) a tree structure, used by P-Grid is illustrated. P-Grid uses a prefix ID space. For example, node 1 and 2 are responsible for the IDs, which start with 00, node 5 is responsible for IDs with prefix 01 and so on. P-Grid is discussed in more detail in Chapter 3.2.1.

A main characteristic of a DHT is that a query is routed towards its destination on every hop, until it reaches its destination. In contrast to random walk algorithms the query is routed to a random node, independent if it is the direction towards the destination. This is not the case in a DHT.

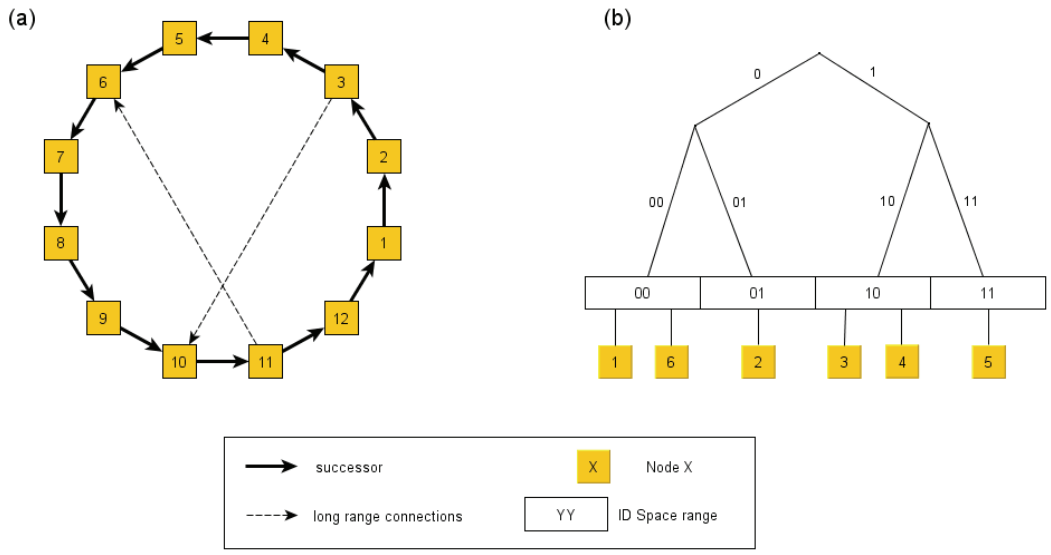

Figure 5: Examples for a ring structure of a DHT (a) and tree structure (b). Solid arrows indicate successor nodes and dashed arrows indicate long range connections.

Some DHT implementations use the small-world phenomenon as model to build up the connection structure of the ID space (e.g. Chord, P-Grid). Stanley Milgram described what he called the small-world phenomenon in 1960's [33]. He made a social experiment in which he sent a few letters to arbitrary people in Nebraska to deliver the letter to a person in Massachusetts. Every person should forward the letter to people they know, based on a few information likes name, address and

occupation of the target person. The letters are then subsequently forwarded until it reaches its target. Milgram found out, that the letters had an average number of hop count (intermediate people) of six [33]. His hypothesis was, “that everyone in the world can be reached through a short chain of social acquaintances” [20, p.63]. The small-world graph of the social network, which people have in the real world, consists of many short-range acquaintances and a few long-range acquaintances. Milgram had the theory that every person in the USA can be found within an average of six hop acquaintances [20]. This is still an open question [20], but as discussed in Chapter 3.2 this approach scales good and has good search capabilities.

The following two sections discuss different implementations of DHT networks. The focus in this thesis is on the search, route and replication capabilities of DHT networks.

3.3.1 P-Grid

According to [32, p.30], “P-Grid is a peer-to-peer lookup system based on a virtual distributed search tree, similarly structured as standard distributed hash tables.” P-Grid is data oriented and has a special focus on range queries. A bit string (i.e. key or index) is calculated with a hash function for every datum. The virtual distributed search tree in P-Grid is a binary prefix tree illustrated in Figure 7.

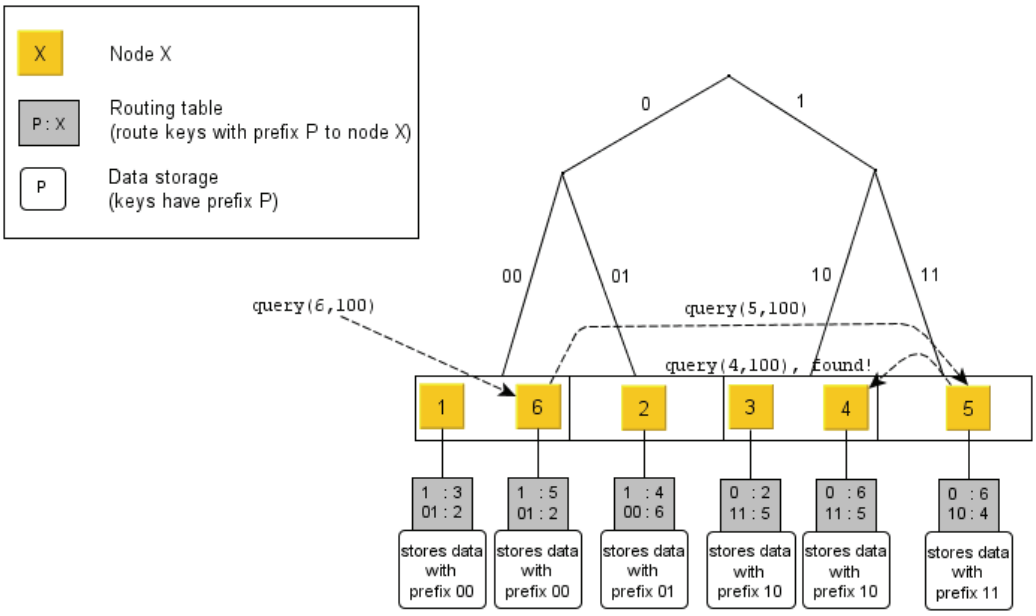

Figure 7: Example of a P-Grid network

Every node is responsible for a path in the binary search tree. A path is the route from the root of the binary search tree to a leaf. The path is generated from a virtual binary tree as illustrated in Figure 7. For example, the nodes 1 and 6 are responsible for the path "00". As illustrated in Figure 7, the binary search tree represents the complete ID space. The ID space consists of binary bit strings [22]. For every path in the binary search tree, there is at least one node responsible for it. It is to say, that every leaf (i.e. path) has at least one node hold responsible for it. In Chord for example, it is possible, that no node is responsible for a path (i.e. ID). The structure in Chord is a ring instead of a binary tree. Every node in the ring is assigned an ID from the ID space. It is not necessary to assign the complete ID space to the nodes in the Chord network (as is the case in P-Grid). Consider for example, a node requests a datum with an ID, which is not assigned to a node in the Chord network. The datum with such an ID is stored on the successor node in the ring structure. Therefore the complete ID space can be used, although IDs are not assigned explicitly to nodes. The successor node is then responsible for the ID. For example, the nodes with IDs = 2, 4 and 7 exist in the Chord ring network and the ID space ranges from 1 to 7. The IDs 1, 3, 5 and 6 are not assigned to any node. If a

datum with ID = 5 is written into the Chord network, the successor node with ID = 7 is responsible for data with ID = 5.

For every bit (level) in the path (binary search tree), the node stores at least one reference to a node on the other part of the binary search tree at the same level [22]. For example, the node 1 in Figure 7 is responsible for the path "00". Therefore the routing table contains at least one node, which is responsible for the path, which begins with "1". In addition at least one node, which is responsible for the path prefix "01", is maintained in node 1's routing table. These are, for example, node 3 for the path prefix "1" and node 2 for the path prefix "10" in Figure 7. The routing table holds multiple entries per path level to be fault-tolerant, if an entry does not work.

A routing example is illustrated in Figure 7 as well. A query for the key "100" is sent to node 6. Node 6 is only responsible for keys with prefix "00". Therefore node 6 looks up a node in its routing table, which is responsible for a key, which starts with "1". Node 5 is responsible for a key, which starts with "1" and node 6 forwards it to node 5. The first digit of the key is equal to node 5s path, but the node is not responsible since the 2nd digit of the key is "0". Node 5 looks up a node in its routing table, which is responsible for the prefix "10" and thus forwards the query to node 4. Node 4 is responsible for this query and answers it. The query is successively routed to a "closer" node in every routing step (i.e. hop). P-Grids' search and routing algorithm needs $O(\log(n))$ steps on an average to route a query to a responsible node (n is the number of leaves in the binary search tree).

Keys (i.e. IDs, indices) can have different sizes and also paths can have different lengths in P-Grid. The ID space is divided recursively, so that every node gets about the same number of data to store [34]. For example, if much data with prefix "00" is stored at the nodes 1 and 6, the path is divided recursively to balance the amount of data stored at every node. Therefore the single path "00" is divided into the paths "001" and "000" to balance the number of data at the nodes. This specialization may result into an unbalanced tree structure, if one path is

specialized several times, but other paths are not. The drawback in an unbalanced binary search trees is that the routing table grows up to linear size, compared to the network size [34].

To increase the fault-tolerance, P-Grid associates more than one node for every path (i.e. partition replication) [34]. As in Figure 7 illustrated, the nodes 1 and 6 are associated with the same path “00” and the nodes 3 and 4 are responsible for the path “10”.

Besides partition replication, data replication is done in the P-Grid network for every datum. Other P2P implementations like Freenet replicate only popular data [28]. P-Grid uses the tree structure to choose the location for the replications [22]. Figure 8 illustrates the replication mechanism. The datum is first copied to its logical direct neighbor. After that, it is copied to the next 2, 4, 8, etc. logical neighbors by masking the key (i.e. ID) of the datum. This is done until the number of copies of the asset “000”, i.e. data with key “000”, is approximately the same than other assets [22]. The approximate replication factor is set in the configuration of P-Grid. Every replica knows a few locations of other replicas [35]. Therefore, if one replica is found other replicas can be found.

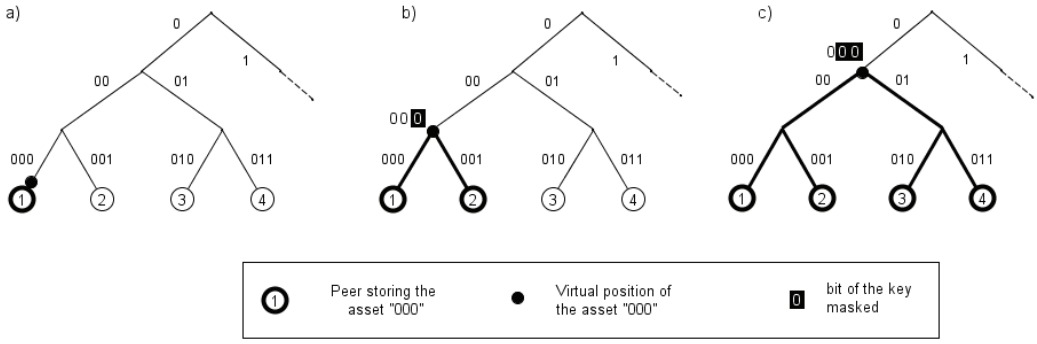

Figure 8: The replication mechanism in P-Grid

P-Grid supports updates of data items, which is based on rumor spreading [35]. The update algorithm provides probabilistic guarantees for consistency [22]. [36] discusses the rumor spreading algorithm in detail. An update may not reach every

replica existing in the network. The cost to reach the few replicas, which are not updated, may be very high. It is not necessary to keep all replicas updated. It is only necessary to deliver the correct (updated) data. Replicas, which are found during an update, also are more likely to be found during a query [37].

When nodes start up a new P-Grid network, the communication is based on broadcast messages. Since P-Grid's functionality (e.g. search algorithm, replication and update algorithm) depends on the binary search tree the goal is to build up a functional binary search tree as fast as possible. So the first step of a P-Grid network is to build up the binary search tree. This phase is called bootstrapping [34].

The partitioning of the ID space is made as follows (for $p \leq \frac{1}{2}$, where p is a probabilistic size):

1. Every node, which has not decided for a path yet, contacts arbitrary chosen nodes until a decision has been made.

2. If the contacted node has not decided yet, these two nodes make a balanced partition decision (e.g. the one node takes path "0" and the other one takes "1") with probability $\alpha(p)$ and exchange their routing table references.

3. If the contacted node already has path "1", the node, which initiated the contact decides with probability $\beta(p)$ to take "0" and with probability $1-\beta(p)$ to take "1". If the node takes "0", it takes the reference to the contacted node into its routing table. If the contacting node takes "1", it asks the contacted node to get a reference from the other part of the search tree, i.e., a node, which's path starts with "0".

This algorithm is modeled as Markov-Process. β, for example, can be calculated from the equation $p = 1 - \beta^1(1 - 2^\beta)$ [34]. The decisions in the bootstrapping phase are based on random and probabilistic approaches, which are discussed in [34] in more detail.

The partitioning of the ID space is continued after the bootstrapping phase [37]. For example, if the data, stored on a certain node (path) exceeds a certain threshold, the path is partitioned with a certain probability into a more specialized path. This causes load balancing, because the number of data previously stored on one node is now distributed on two nodes. Also merging specialized paths is possible. Since the P-Grid network is under continuous churn, maintenance operations like path partitioning are done periodically.

To repair inconsistencies in routing tables, P-Grid uses the approach of correction on use. This approach only repairs a faulty entry, if it is used [34]. The drawback is that a query may take longer, if a faulty entry is encountered during routing a query. The benefit is that since nodes in a P-Grid network are often available only temporarily, faulty routing table entries only cause maintenance traffic if they are used. Otherwise they stay in the routing table (causing no correction traffic) and can be used, if the node is available again [34]. To make the routing table fault-tolerant, P-Grid maintains redundant references in the routing table (as discussed previously). [34] gives a detailed analysis of the approaches correction on use and correction on failure. The approach correction on failure repairs failures, when they appear.

3.3.2 Other Distributed Hash Table Implementations

In this section I discuss a few DHT-based P2P network implementations:

- **Chord**

  As discussed before, Chord builds up the ID space in a form of a ring. In Chord every node has a unique decimal ID. The maximal ID is predefined and presents the maximal number of possible participating nodes in the Chord network. Every node maintains the successor node (i.e. direct following neighbor) in its finger table. "The finger table of a node with ID x is formed of the nodes $y = x + 2^i$ with $0 \leq i < \log_2 N$, N being the maximum possible number of nodes that form the network" [20, p.48]. The long-range connections have logarithmic distances. If a node with ID $y = x + 2^i$ does not exist, the entry in the finger table is replaced with an existing node, which has the next higher ID.

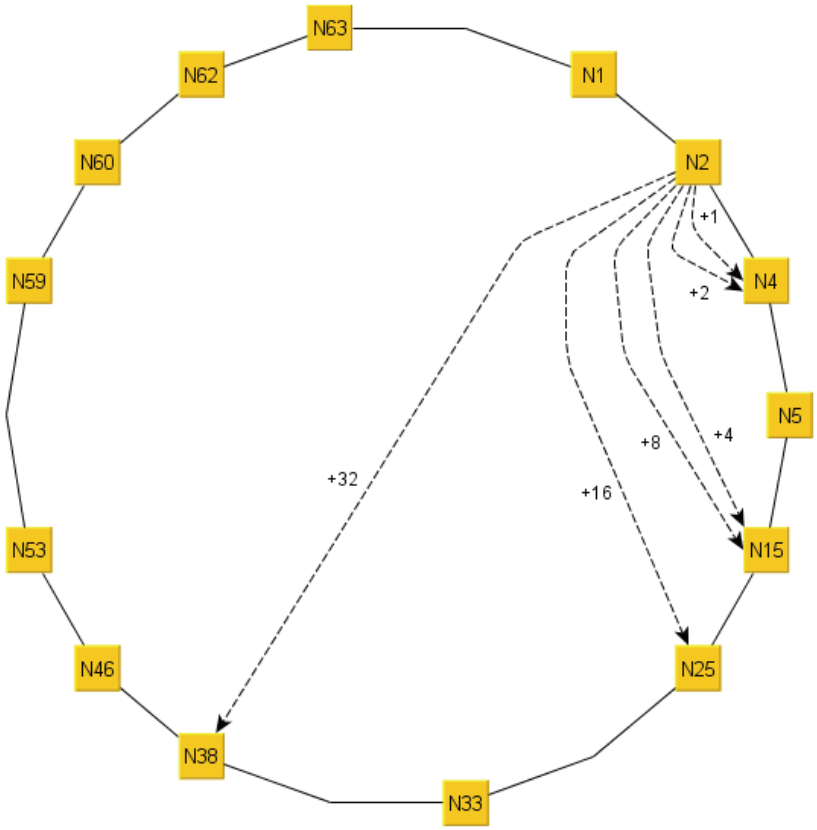

Figure 9: Example of a Chord P2P network

(a) Finger Table for N2	
N2 + 1	N4
N2 + 2	N4
N2 + 4	N15
N2 + 8	N15
N2 + 16	N25
N2 + 32	N38

(b) Finger Table for N4	
N4+1	N5
N4+2	N15
N4+4	N15
N4+8	N15
N4+16	N25
N4+32	N38

Table 1: Finger table for the node N2 (a) and N4 (b) in Figure 9

Figure 9 depicts an example of a Chord network with $N = 64$. Table 1 shows the finger table for nodes N2 and N4. In this example the resources in the network are hashed into an ID between 1 and 64. The index of the resource is maintained on the node, which is responsible for the index ID. If the node with the ID does not exist, it is stored on an existing node with the next higher ID. For example, the index for the resource with ID = 30 is stored on node N33.

If a node looks for a query with key k, it looks up a node in its finger table, which has the largest ID smaller than the key k and forwards the query towards this node. For example, if node N2 is looking for a datum with the key 38, it checks the finger table. The key 38 has an entry in N2's finger table , illustrated in Table 1 (a). The query can be forwarded directly to N38. If N2 wants to look up a datum with the key 5, the closest entry in N2's finger table is N4. N2 forwards the query to N4. N4 looks up the key in its finger table, illustrated in Table 1 (b). The direct successor of N4 is found as a match for the key 5. The query is forwarded to N5, which is responsible for the query. According to [38] a lookup of a key requires only $O(\log N)$ messages.

To maintain the finger table entries Chord periodically checks them by sending messages into the network. This causes a lot of overhead traffic, which affects the performance of Chord [20]. The ring structure is rigid, since the entries in the finger table are rigid. For example, the entry N2 + 8 in the finger table (Table 1 (a)) must be N15 unless there are other nodes, which are between N2 + 8 and N15 (e.g. N8 to N14). If there are other nodes in the range N2 + 8 and N15 the entry in the finger table must be the next bigger node after N2 + 8.

Every "Chord node maintains a 'successor-list' of its r nearest successors on the Chord ring" [39, p.156]. This list guarantees that in case of a node failure, the ring keeps closed (for routing messages). For example, if no successor-list exists and every node only knows its closest successor node (neighbor) – besides long distance entries in the finger table. If a node fails, the predecessor of the failing node loses its successor node and thus the Chord ring is open. Routing may fail, if a query is destined to the failed node, because there is no alternative. Also long distance entries in the finger table referring to the failed node get invalid. For example, if node N38 from Figure 9 fails, the predecessor node N33 looses its successor node. The node N2 looses a long distance entry in its finger table (Table 1 (a)), since the entry N2+32 points to the failed node N38. In such cases, invalid finger table entries are corrected. During this time a message is routed to other nodes in the finger table to circumvent the missing nodes. For example, N2 forwards a message to the next shorter reference in the finger table, i.e. to N25. The consequence is that data items which are stored on the failed node are not accessible any more and messages are routed over more hops to their destination.

Replication of data items in Chord is not available implicitly, but it can be implemented by a higher application [39]. Since the Chord nodes maintain a successor-list, a potential replication strategy could be that every k successors of a node also stores this data item.

- **Pastry**

In contrast to Chord, Pastry [40] has not a rigid structure like a ring. The structure of the Pastry network is completely random. Pastry assigns a random ID to every node in the P2P network. The ID space ranges from 0 to $2^{128}-1$ and every ID is 128 bit long. "Pastry can route to the numerically closest node to a given key in less than $\left\lfloor \log_{2^b} N \right\rfloor$ steps" [40, p.331]. N represents the number of peers in the P2P network and b is a configuration parameter (typically $b = 4$).

The routing table of Pastry is divided into three parts:

- Routing table
- Neighborhood set
- Leaf set

Node ID 1023102			
Leaf set			
Smaller		Larger	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232
Routing table			
-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32-102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	
Neighborhood set			
13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Table 2: Example of a routing table in Pastry: The node ID is 1023102. The first row of the routing table is row 0; the second row is row 1 and so on. The grey cells in the routing table show the corresponding digit of the present node's node ID. The node IDs are split into three areas: (1) the common prefix – (2) the next digit – (3) rest of the node ID [40]

The **routing table** contains IDs of nodes with different prefix matches. The routing table holds $\left\lfloor \log_{2^b} N \right\rfloor$ rows. Every row holds $2^b - 1$ entries. In every row n, the $2^b - 1$ entries (other node IDs) match the first n digits of the current node ID, but differ in the remaining digits. For example, the current node ID is 10233102. The node holds the entry 10233001 in its 5th row of the routing table. This node ID shares the first 5 digits (i.e. **10233**001) with that of the current node ID.

The **neighborhood set** contains IDs and IP addresses of physically connected nodes. Physically means, that an actual connection can be built to these nodes with the underlying network (e.g. TCP/IP, http). The neighborhood set is used to route messages.

The **leaf set** contains nodes with numerically closest larger and smaller node IDs than the current node ID. The neighborhood set and leaf set usually contain 2^b or $2x2^b$ entries. Numerically close means that the numerical distance between two nodes is small (e.g. nodes with a distance of 10 are assumed to be close). For example, a node with ID = 1023102 has nodes in the ID range 1023101 to 1023092 as closest smaller nodes and nodes with ID range 1023103 to 1023112 as closest larger nodes in the leaf set.

To route a message to a key, the node first checks its leaf set, if the key numerically lies within the range of the leaf set. If the key does lie in the range of the leaf set, the key is forwarded to the node with the numerically smallest distance between the node ID and the key. If the key is outside the range of the leaf set, the routing table is used to find a node ID, which has a common prefix with the key. The length of the common prefix between the node ID in the routing table and the key has to be at least equal to the common prefix between the key and the current node. If the prefix of the key is equal to that of the receiving node ID, the receiving node ID has to have a smaller numerical distance to the key than the current node ID. According to [40, p.334] “this simple routing procedure always converges, because each step takes the message to a node that either shares a longer prefix with the key than the local node, or shares as long a prefix, but is numerically closer to the key than the local node”. Table 2 illustrates a routing table for a Pastry node.

During a join of a node, the joining node sends its randomly generated ID to an already connected node. The already connected node sends the new ID to a node which’s ID has a numerically small distance to the new ID. The return message contains all state information of nodes, which encountered the

message during routing. The new node analyses these state information to create its own state. If needed, the new node requests additional state information from other nodes. After that it informs any node, which has to be aware of the newly joined node. The benefit of this procedure is that a new node initializes a proper state at the beginning and updates the state of already connected nodes of its arrival [40]. Routing entries, which are out of date, are repaired, when they are needed [40]. Therefore this repairing strategy only causes overhead, if a faulty entry in the routing table is used (i.e. correction on use).

Pastry is more dynamic in choosing the entries in the routing table than Chord. There can be arbitrary entries in the routing table which only have to meet the restrictions, as discussed above. For example, the first 2 digits of the node ID in the routing table have to be equal to the current node ID in the 2^{nd} row. Also the routing process is not so strict, than in P-Grid or Chord, because Pastry routes a message with a key k to the node with the numerically closest ID, i.e., numerically smallest distance between key and node ID [40]. Therefore the target node ID needs not to be the same key in the message.

A benefit of routing a key to the closest node ID is that this behavior is used for data replication [40]. The query is routed along nodes with a numerically closer ID in every routing step. Replicas are places near the closest node ID compared to their key. Since the replicas are placed on nodes near their key and the query is routed along the numerical distance to the key, the query comes by the replicas. If a replica fails the query finds the next replica, closer to the key. The query may approach closer to the key to find the next replica.

- **Tapestry** [41]

This DHT overlay network is based on the Plaxton mechanism, discussed in [42]. The Pastry network [40] is also based on the Plaxton mechanism. Every node maintains a neighbor map (i.e. routing table). The neighbor map is organized into levels. For every level *n* (bit position in the ID) several nodes are

maintained, which have the nearest distance and match the first *n* bits of the ID. At every level at least three nodes are maintained for fault-tolerance. If a message is routed to a node, which is not responsive the node is marked as invalid in the neighborhood map and another entry is used to route the message. A message is routed again to the invalid marked node with a certain probability. If the node is still not responsive, the entry in the neighborhood node is removed. If the node is responsive, the entry is marked as valid. However, a node, which is marked as invalid, is probed for a second time before it is deleted from the neighborhood map. To maintain the neighborhood entries, periodic heartbeats are sent to the direct neighbors of a node [41].

Tapestry maintains a root node for every datum in the P2P network. To find a datum, the message is routed to the root node, which is unique in the whole network. To calculate this root node for a datum, a message with the key of the datum is tried to route to the node with the same ID than the key. Since it is unlikely that a node with the same ID exists (because the ID space is very big), the message is routed to a node with the numerically smallest distance between the ID and the key. This node then acts as the root node for the datum. This mechanism is called surrogate routing [41]. Replications of a datum is maintained by creating multiple root nodes for the datum.

Every datum in Tapestry gets a timeout, which has to be refreshed on periodically terms. Therefore every storage server has to republish its data in order to keep them in Tapestry. If a datum gets inaccessible it simply timeouts and therefore is deleted in Tapestry.

As Tapestry has similar approaches to the Pastry network, the joining mechanism of a new node to the Tapestry network is similar. A new node sends a message to the node, which has the closest ID to the new node ID. The message gathers routing information, which is sent back to the joining node. With this information the new node populates its neighbor map and also informs nodes of its appearance.

3.4 Distributed Mutual Exclusion

In the previous sections communication in distributed environments using TSs and P2P networks have been discussed. For distributed processes, which closely work together, communication is not the whole story [43].

The Linda model provides exclusive access to tuples. That is, no concurrent processes can access the same tuple at the same time. For example, if process A wants to remove a tuple from the TS and a concurrent process B wants to read the same tuple at the same time, the access to the tuple has to be serialized. If process A gets access to the tuple first, process A removes the tuple from the TS. If process B accesses the same tuple with a *read*-operation afterwards, it does not exist any more. Process B then blocks until another matching tuple is written into the TS. However, if process B is able to read the tuple before process A, both processes are satisfied, since the *read*-operation does not change the tuple and it can be accessed by process A afterwards. For example, if no mutual access is guaranteed in the TS, the operations *read* and *in* concurrently access the same tuple at the same time. This may cause inconsistency in the flowing of the distributed application, in which both processes participate. The worst case of non-controlled concurrency in the TS is that if two processes want to remove the same tuple at the same time and both processes get a copy of the tuple. The correct procedure is that only one process gets the tuple and the second process blocks until another matching tuple is written into the TS. However, when two concurrent *read*-operations take place at the same time for the same tuple, the outcome may not be as problematic as for two *in*-operations. Two concurrent *read*-operations may cause no difference when they occur at the same time instead of serializing the access, because they just read the same tuple without changing it.

There are different approaches for mutual exclusion algorithms for so-called critical sections. The critical section in the TS is a tuple. If more than one process wants access the same tuple, the access has to be controlled (e.g. serialized). To control access to critical sections in single-processor systems centralized

algorithms like semaphores and monitors can be used [43]. For multi-processor systems like P2P networks, other approaches have to be considered.

The different mutual exclusion algorithms for multi-processor systems can be classified into three groups [43, 44]:

- Centralized algorithms
- Token-based algorithms
- Permission-based (also called quorum-based [45]) algorithms

Centralized distributed mutual exclusion algorithms are based on single-processor mutual exclusion algorithms [43]. A distributed system uses a single coordinator, which has a global overview over all critical sections and maintains request queues for them [43, 44]. If a process wants to access a critical section, it asks the coordinator for permission. The drawback is that the coordinator is a single point of failure and may be a performance bottleneck in large systems. Figure 6 illustrates a system with a centralized coordinator. [43] The coordinator can be chosen from any participating process (node). Additional information can be found in [43].

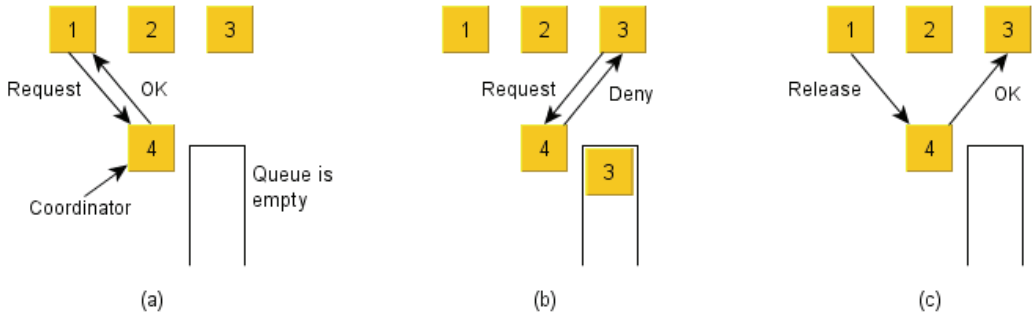

Figure 6: **(a)** Process 1 asks the coordinator for permission to enter the critical section and is granted access. **(b)** Process 3 also asks for permission for the same critical section and is denied, since process 1 is in charge. The coordinator queues the request of process 3. **(c)** When process 1 releases the critical section the queued process 3 gets an ok as a grant to access the critical section.

The processes in a **token-based distributed mutual exclusion algorithm** are logically organized as a closed directed ring [44]. This structure is like the Chord ring, discussed in Chapter 3.2.2. Every process knows its next process in line. A unique object, called token, circulates around this ring structure. The token is the permission to access a certain critical section, since the token is unique among all processes. If a process owns the token, it is permitted to access the critical section. If a process does not need the critical section and owns the token, it just passes the token to the next process. If no process needs the critical section the token circulates around the ring.

There are some problems in this token-based approach. If the token is lost no one is permitted to access the critical section any more. To create a new token is difficult, because it is difficult to get prove, if the token is really lost. According to [43], "the fact that the token has not been spotted for an hour does not mean that it has been lost: somebody may still use it" [43, p.270]. If a process, currently holding the token, fails, the token is lost. However problems can occur, if duplicated tokens for the same critical section exist [44]. Another problem is that if a process fails, the ring is open, so the token cannot circulate anymore. To repair the hole in the ring every process has to maintain a successor list, as the Chord P2P network does (Chapter 3.2.2). Since the token is passed to every process, the token-based approach may become a performance bottleneck if the number of processes becomes large. Although one hop of a token between two processes may cause little overhead, passing the token through thousands or millions of processes, take unacceptably amount of overhead. If there are many critical sections to be managed, there is a token for every critical section, which circulates through every process. The tokens cause traffic even if the critical sections are not used [44]. Descriptions of different token-based algorithms can be found in [44].

In **permission-based** or **quorum-based distributed mutual exclusion algorithms** the access to critical sections is controlled by getting permissions from a set of processes (nodes) in the system, also called quorum [44]. A quorum is defined as a number of nodes, which are able to make decisions (e.g. grant access to a tuple). A

node set (i.e. quorum) contains nodes, participating in the system. The node set is able to coordinate permissions for any kind of resource (critical section), regardless whether the node in the node set holds the resource or not. If a node wants to access the critical section it sends requests to all nodes in the set. The nodes in the node set return the permissions. Only one permission per node in the node set and critical section is allowed. If the requesting node has the majority (quorum) of permissions it is granted access to the critical section. The problem is to find the minimal set of nodes needed to grant access to a critical section [44], i.e., the ratio of the number of nodes in the node set and permissions. Many algorithms have been developed to manage distributed mutual exclusion [44], but "it is not possible to directly adapt the proposed solutions into the P2P domain" [46, p.296]. Chapter 3.4.1 discusses two protocols in more detail. These protocols are permission-based distributed mutual exclusion algorithms.

3.4.1 Distributed Mutual Exclusion Protocols

In the last few years, three permission-based distributed mutual exclusion algorithms have been proposed for the use in P2P networks. The protocols are able to work in highly distributed and churn prone environments:

- Sigma protocol [47]
- End-to-End mutual exclusion protocol [46, 45]
- Non-End-to-End mutual exclusion protocol [46, 45]

The three protocols use replicas instead of node sets. Every resource in a P2P network is a critical section. Since replicas can be used for fault-tolerance (see Chapter 1.2), these replicas are also used for permitting requests to the critical sections.

The **Sigma protocol** is based on the Strawman protocol [47]. Every node, which wants access a critical section (a resource) sends a request to all replicas of the

resource and waits for responses. A replica returns a grant vote, if the replica is not owned by another node (i.e. has voted for another node before). If it is owned by another node the replica returns a vote in which it states who the current owner of the replica is. The node, which gets *m* out of *n* replica votes ($m > \frac{n}{2}$) is the winner and is able to access the critical section. The other nodes, which have not gained enough votes, release their votes and try again later. [47]

As P2P networks are in constant change, failures of nodes occur unexpected. Therefore replicas can fail and thus the mutual exclusion of a critical section can be broken. To make the algorithm robust against violations of the critical section attention has to be paid for the ratio of $\frac{n}{m}$. [47] discusses, that "even with n = 32 and $\frac{n}{m}$ = 0.75 (i.e. m = 24)" … "the chance of breaking the exclusivity is 10^{-40}" [47, p.14]. The Sigma protocol architecture is illustrated in Figure 10 and works as follows:

A node, which wants access a critical section, sends a request message to every replica (SendRequest). The replicas sends a response message back (SendResponse) with either a grant vote or the current owner of the replica, if this replica already has voted. The requesting node is queued at the replica, if there is another current owner. There are three different outcomes after the requests are sent to every replica:

- The requesting node is the winner of the critical section and is allowed to access it.

- Another concurrent node wins the vote. The requesting node does nothing, but waits until it is notified when the concurrent node leaves the critical section.

- No requesting node has gained enough votes to access the critical section. Therefore the node sends a yield message (SendYield) to every replica, which reorders the request queue and votes again. This is repeated until a winner is chosen.

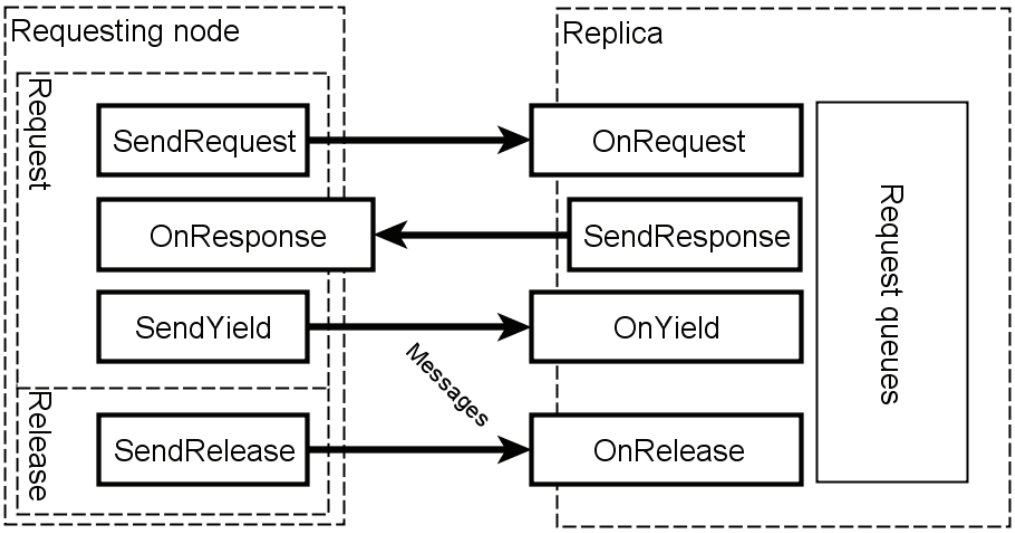

Figure 10: Architecture of the Sigma protocol.

When the requesting node is finished with the critical section it sends a release message (SendRelease) to every replica and another node is allowed to enter the critical section.

[47] discusses approaches to cope with failures. For example, if a replica fails, the replaced replica rebuilds its queue state from the queues at the other working replicas (informed backoff [47]). Consider, for example, if a node, which is currently the owner of the critical section fails. The access to the critical section will be blocked forever, because it is not released. Therefore the Sigma protocol proposes, that the replicas grant renewable leases for the critical sections. When the lease expires the next node is granted access to the critical section. [47]

The distributed mutual exclusion protocols **End-to-End** and **Non-End-to-End** are similar. They are based on the Sigma protocol. In the End-to-End protocol the

current owner of a critical section holds the request queue, instead of having the request queue on every replica (as in the Sigma protocol). When the owner of the critical section leaves the critical section, the owner sends the request queue to the next requester in the queue.

Every replica has a quorum set, which is a group of nodes. A quorum set consists of the nodes, which are in the routing path of a request message. The routing path goes from the requesting node to a replica of the resource (critical section). The path is dependent on which overlay network is used (e.g. Chord). If a node requests the critical section, the quorum set of the replicas forwards the request to the current owner, which puts it into the request queue. The quorum set periodically contacts the current owner of the critical section to check if it still is alive to avoid a deadlock. [48, 45]

The Non-End-to-End protocol distributes the request queue among the nodes in the quorum set. This protocol is more distributed and more fault-tolerant, since the request queue is not stored on a single node (i.e. current owner of the critical section) [48]. The message overhead is relatively low [45]. The performance in terms of time ranges from 3 to 7.5 seconds to gain access to a critical section [45].

The idea behind the End-to-End and Non-End-to-End protocols is load balancing, since the replicas in the Sigma protocol are directly contacted for every vote and therefore may become a bottleneck. In the quorum set a request can be forwarded to the owner of a critical section before it reaches the replica. Every node of the quorum set is able to forward a request. The maintenance cost of the quorum set is not mentioned in [45, 46], but since a quorum set updates its content, it seems to have a maintenance impact. The DTSpace approach uses a hybrid of the two protocols Sigma and End-to-End.

This thesis discusses a novel approach of completely distributed, dynamic and reliable TS. In comparison to many centralized TS implementations like TSpaces or LighTS the DTSpace approach does not have any centralized approaches. The

storage of tuples and interactions between the nodes in the network are distributed. Fault-tolerance is implemented with a replication mechanism, provided by the P2P network P-Grid [29].

4. The Distributed Tuple Space Approach

As discussed before I use a P2P network for the distributed tuple space (DTSpace) approach. Therefore I have to choose which P2P network implementation is suitable for the DTSpace. There are two main requirements for the P2P network. First, the P2P network implementation has to be written in Java programming language and second the implementation has to be available as open-source. The main reason why the P2P network implementation should be open-source is that the DTSpace prototype implementation is open-source and thus free available for further research and development.

There are different approaches for P2P networks, as discussed in Chapter 3.2. It turns out, that structured P2P networks have some advantages in contrast to unstructured approaches. The advantages are (1) better scalability, (2) deterministic searches (no enclosed search like in Gnutella with the Gnutella horizon) and (3) less communication overhead. Structured P2P network approaches for DHTs build up an organized but dynamic search and placement structure. These capabilities are suitable for the DTSpace. DTSpace uses the index of the DHT to store the data (e.g. tuple, field). The index is normally used to store references of data. For example, if files are shared the actual files are kept on the nodes and references to the location of the files are stored in the index of the DHT. The index can be logically organized as tree or ring structure (refer to Chapter 3.2.2). The indices are used to search efficiently for data. In the DTSpace this index structure holds the actual data (e.g. tuple, field). One reason for storing the tuples in the index structure of the DHT is that the datum in the tuples is not very big. The traffic in the DHT therefore is manageable. Another reason is that the replication mechanism of the DHT can be used to replicate the data.

The actual choice falls on P-Grid [29], because of the following reasons:

- Available as open-source (GNU license)
- Written in Java

- Extendable and adaptable framework (e.g. one can add routing-strategies, new message types)
- P-Grid is well documented
- P-Grid uses proved approaches (e.g. gossiping for update strategies)
- P-Grid is capable of range queries, which may be helpful for future developments of DTSpace
- Support from the developers of P-Grid

In the remaining chapter I discuss a novel approach of the DTSpace. The DTSpace approach and its prototype implementation can be used for other P2P networks than P-Grid. For example, the distributed mutual exclusion (DMutex) module can be fully reused. Only some methods of an abstract class must be implemented (refer to the Appendix for more detail).

4.1 The Linda Model in DTSpace

The DTSpace approach has the same capabilities like the original Linda model, introduced by David Gelernter in 1985 [1]. The DTSpace approach supports the basic operations (1) *in*, (2) *out* and (3) *read* (see Chapter 3.1).

There is one big difference to the other approaches of the Linda model like TSpaces or JavaSpaces: It is completely distributed over a network of nodes (i.e. DHT), which automatically organize themselves. The DHT network, used in DTSpace, is P-Grid. In contrast to SwarmLinda [4], which organizes the distributed network based on nature swarms; P-Grid is organized more deterministic (see Chapter 3.3.1).

The major benefits of this approach are:

- Scalability for an unlimited number of participants, who can use the same tuple space.

- Robustness - it is difficult to loose data, since they are replicated.

- Fault-tolerance – the tuple space is still operable if, for example, a part of the network fails.

- Every node can choose to participate and leave without any precautions. Normal churns of nodes does not affect the performance and reliability of the tuple space. For example, locally stored data need not to be pushed into the network when disconnecting to avoid data loss. The data is replicated among the remaining network, if necessary.

The drawbacks are discussed in Chapter 3.3.

The DTSpace has to guarantee mutual access to a tuple (see Chapter 3.1). Therefore an adapted voting algorithm, which is discussed in [47] and [45], is used for mutual exclusion. The voting algorithm, which is used in DTSpace, is discussed in detail in Chapter 4.8. Centralized approaches for mutual exclusion usually have to be adapted in a distributed environment, as discussed in [43] and Chapter 3.4.

4.2 Architecture of the DTSpace

The DTSpace architecture consists of three layers: (1) the DTSpace module and the factory module, (2) the wrapper layer and the (3) overlay network. Figure 11 illustrates the DTSpace architecture.

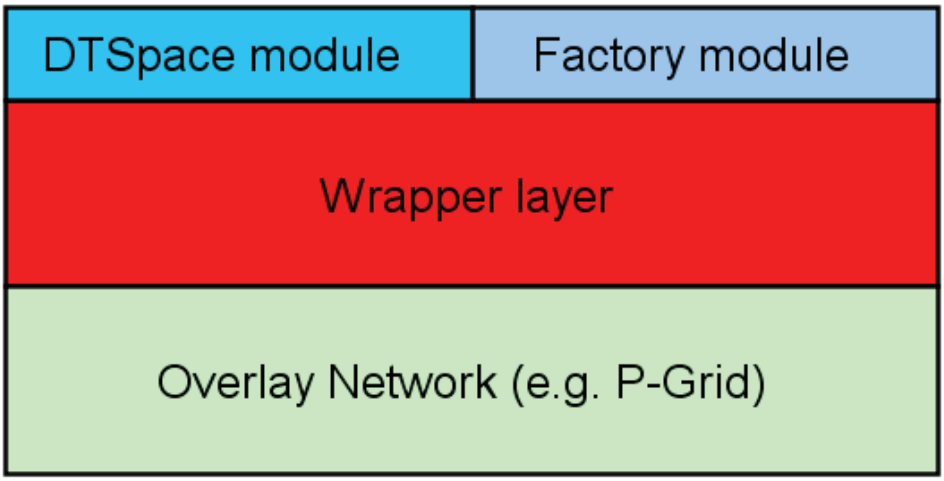

Figure 11: DTSpace architecture

The upper layer of the DTSpace is the DTSpace module, which provides the three operations (1) *out*, (2) *read* and (3) *in*. The Factory module generates the DTSpace module, configures the Overlay Network and the Wrapper layer as well. The Factory also generates instances of objects, which are used to interact with the DTSpace (e.g. tuples, templates, configuration objects). For example, if a new tuple is inserted into the DTSpace the Factory generates the new tuple. Also template objects are generated in the Factory module to search for tuples. Processes, which use the DTSpace, only have to interact with the DTSpace module and the Factory module. The Wrapper layer provides tuple space functionalities on top of the Overlay Network (e.g. mutual exclusive access to a tuple). The second layer (Wrapper layer) has several purposes. Consider using P-Grid as an overlay network. P-Grid does provide replication for fault-tolerance, but does not provide mutual exclusion. Therefore the Wrapper layer wraps the *out*-operation of the DTSpace to the insert-operation of P-Grid. The DTSpace also can be viewed as an overlay network, providing new functionalities to the overlay network P-Grid. For example, if a *read*- or *in*-operation is invoked the Wrapper layer provides mutual exclusion, which is not implemented in the overlay network P-Grid. At the bottom of the DTSpace architecture the overlay network provides connectivity, search and

storage capabilities for the DTSpace. For the prototype implementation of DTSpace P-Grid, a DHT network (see Chapter 3.3), is used.

The DTSpace hides all its complexity and functionalities behind the top layer. The underlying overlay network can be exchanged without affecting the operations of the DTSpace. One benefit of the prototype implementation of the DTSpace is that some classes (e.g. DMutex, tuples, templates) can be reused if another overlay network than P-Grid is used.

4.3 DTSpace features

As mentioned before the DTSpace approach provides the three basic tuple space operations and is fully distributed. Some features, which are hidden from processes, have to be implemented in the DTSpace as well in order to work as a tuple space (e.g. exclusive access to a tuple). The following list shows different features, which the DTSpace provides:

- Basic tuple space operations (*out*, *read* and *in*)
- Blocking for the *read*- and *in*-operations
- Exclusive access to a tuple for concurrent processes (realized with distributed mutual exclusion)
- Scalability
- Fault-tolerance (realized with replication, provided by P-Grid)
- Flexible search granularity (refer to Chapter 4.6)

Some of the features depend on the used overlay network (e.g. scalability). The overlay network P-Grid, which is used in the prototype implementation of the DTSpace for example, provides scalability and fault-tolerance. All tuple space operations, blocking and exclusive access to a tuple are implemented in the Wrapper layer of the DTSpace. A detailed discussion of the prototype implementation can be found in the Appendix.

4.4 Data Storage in DTSpace

To operate with a tuple space three main data structures namely (1) tuple, (2) field and (3) template are needed (see Chapter 3.1). Besides these three data structures the DTSpace uses so-called blocking fields. They are inserted if a process (node) does not find a matching tuple. The blocking fields contain a reference to the blocking node and the field value, for which the node is looking for. The search process is discussed in detail in Chapter 4.7.

Every data object (e.g. tuple, field) is replicated among the nodes in the DTSpace network. The replication rate defines how many copies of every data object are maintained in the DTSpace. The benefit of replication is fault-tolerance, scalability and robustness. Even if a part of the DTSpace network fails, the probability of a data loss is very low, because the physical and logical distribution of the nodes is different (see Chapter 3.2.2). If a logical neighbor of a node fails, then it is unlikely that it is also a physical neighbor (for a detailed discussion, refer to Chapter 3.2.2).

The storage method, i.e., the way how tuples are persisted in a DTSpace has a great impact on the search granularity and storage usage. The search granularity defines the granularity of the search keys in the DTSpace. Consider a field value with "Steven Green" and a search granularity of a field. To get a result, the complete value of the field has to be defined as a search key in order to retrieve the field. It is not possible to search for the forename "Steven", because the granularity is too coarse. To be able to search for a key, which is finer than the field value, the granularity has to be refined. For example, refine the granularity to the precision of a word. Then it is possible to search for the key "Steven" although the complete field value is "Steven Green".

For every tuple, which exists in the DTSpace, the following data objects are stored: Every field of the tuple by itself and the tuple as it is. The number of data objects, which are stored in the DTSpace, can be calculated from the Equation 1.

Equation 1: data objects = (number of tuples + number of fields) * replication rate

For example, a tuple with 4 fields and a replication rate of 3 results into 15 data objects. Besides blocking fields, tuples and fields are the basic objects, which are replicated in the DTSpace.

4.5 Data Mapping in DTSpace

Data objects (e.g. tuple, field) are mapped to the search structure (index) in the DTSpace. Every data object in the DTSpace has a unique ID, an index string and an index (the hashed value of the index string). The index string is a key to search for a value in the DTSpace. The index string is generated from the data object itself. The index is the hash value of the index string and is used to distribute the data objects equally among the nodes in the DTSpace for load balancing (see Chapter 3.3). Since P-Grid is used as overlay network in DTSpace, the index structure of P-Grid is used.

The main question is how the index string should look like to make the data retrieval efficient. For example, if the index string is equal for different tuples the search result for a specific tuple is big, because the index string does not distinguish between the different tuples. The following example illustrates how to make the index string efficient. Table 3 defines 6 tuples stored in the DTSpace.

ID	Tuple	Number of fields
1	(“Diego”:string, “student” :string, “Columbia” :string, “25” :string)	4
2	(“Maria” :string, “student” :string, “Austria” :string, “26” :string)	4
3	(“Peter” :string, “worker” :string, “Germany” :string, “30” :string)	4
4	(“14” :string, “June” :string, “3” :string, “25” :string, “p.m.” :string, “conference” :string, “Security” :string)	7
5	(“25” :string, ”April” :string, “4” :string, “10” :string, “p.m.” :string, “appointment” :string, “Mr. Miller” :string)	7
6	(“meeting with” :string, “Mr. X” :string, “Grove Street” :string, 25:integer)	4

Table 3: 6 Tuples. The type for a field is defined after the sign “:”.

1. Straightforward approach

A straightforward approach is to take the value of the field/tuple to generate the index string:

Index string = [value of the field]

The tuple with the ID=1 has the index strings “Diego”, “student”, “Columbia”, “25” for the fields and the index string “DiegostudentColumbia25” for the tuple itself.

The drawback of this approach is that every tuple, which has at least one field, which matches a search key, is put into the result set, independent of the position and number of the field in the tuple. For example, search for the template (*, *, *, “25”:string). Independent of the type of the field the result set contains the tuples with the IDs 1, 4, 5 and 6, because every tuple contains a field with the value “25”. The correct tuple has to be filtered out from the resulting set afterwards. The result set causes more traffic than necessary, because the tuples with the IDs 4, 5 and 6 have nothing to do with the template. The tuple with ID 4 and 5 have not the correct number of fields. The tuple with the ID 6 has the wrong field type.

2. Extended approach

Every search process gets a template to search for a tuple. This template contains search keys and structural information about the tuple to search for. For example, the template additionally contains the total number of fields, the positions of the fields and the types of the fields. All these information can be used to generate the index string. To minimize the result set, the index string is extended with the number of fields in the tuple. The index string then has the format:

Index string = [value of the field] [number of fields in the tuple]

The tuple with ID = 1 then has the index strings “Diego4”, “student4”, “Columbia4”, “254” for the fields and the index string “Diego4student4Columbia4254” for the tuple itself. The search for the template (*,

*, *, “25”:string) does improve. The result does not contain the tuple with the IDs 4 and 5, because they have 7 fields. The tuple with ID 6 is still in the result set, although the type of the field does not match with that of the template.

3. Approach, implemented in the DTSpace

I extend the former approach for the index string with the type of the field and the position of the field. The value of the field, the position of the field, the number of fields in the tuple and the type of the fields is information, which is implicitly defined, if a template is created. The index string then looks like:

Index = [value of the field] [type of the field] [number of fields in the tuple] [position of the field]

The tuple with ID = 1 then has the index strings “DiegoSTRING41”, “studentSTRING42”, “ColumbiaSTRING43”, “25STRING44” for the fields and the index string “DiegoSTRING41studentSTRING42ColumbiaSTRING4325STRING44” for the tuple itself. The search for the template (*, *, *, “25”:string) then returns only the tuple with ID = 1 in the result set.

The last approach filters out tuples, which obviously do not match the template, before they get into the result set and cause traffic. For example, the tuple with the ID = 5 is not in the result set, although it contains a field with the value “25”. Still there are some drawbacks for this approach. For example, if the DTSpace contains many tuples with the same number of fields and equal values, the result set is large, causing a lot of traffic, because many tuples matches the template, but only one tuple finally is returned as result. The third approach is used in the DTSpace.

4.6 Correlation between Search Granularity and Storage

The search granularity in the DTSpace is a field, because there is a tradeoff between granularity, complexity and storage usage. The finer the search

granularity, the more storage is used for one tuple. For example, if the granularity is set to one character, every character of a field value is split into an extra field in order to search for a granularity of character. It is then possible to search for a substring in a value. For example, the tuple (“Dublin”, “Ireland”, “Europe”) then looks like (“D”,“u”,“b”,“l”,“i”,“n”,“I”,“r”,“e”,“l”,“a”,“n”,“d”,“E”,“u”,“r”,“o”,“p”,“e”). Every field of this extended tuple contains additional information about the position in the original tuple. This causes much more overhead information (and thus more storage usage). For every field the position and number of fields in the original tuple is stored. Instead of 40 data objects, 200 data objects are used to store the same tuple, but different search granularity. With this search granularity of a character it is possible to search for a substring of the value in the tuple, e.g. a template to search for the substring “Euro” looks like (*, *, *, *, *, *, *, *, *, *, *, *, *, “E”, “u”, “r”, “o”, *, *). The drawback besides the storage usage is that the position of the substring has to be known in order to search for it. Table 4 and Figure 12 give an example of the storage usage, using different granularities. The Equation 1 from Chapter 4.4 is taken to calculate the storage usage.

Granularity	**Tuple**	**Number of needed fields**	**Storage usage (in objects)**
Field	(“Dublin”, “Ireland”, “Europe”)	3	40
Character	(“Dublin”, “Ireland”, “Europe”)	19	200

Table 4: An example for the storage usage, using different search granularities (replication rate = 10).

Although the value of every field in the character-granularity-approach is one character long, it causes an expensive overhead. Every field is stored in an object which contains information like position in the field, position in the tuple and ID of the tuple. Therefore the value of the field is a small part compared to the overhead information. The storage usage also depends on the length of the values in the tuple, because the values are split into one-character fields.

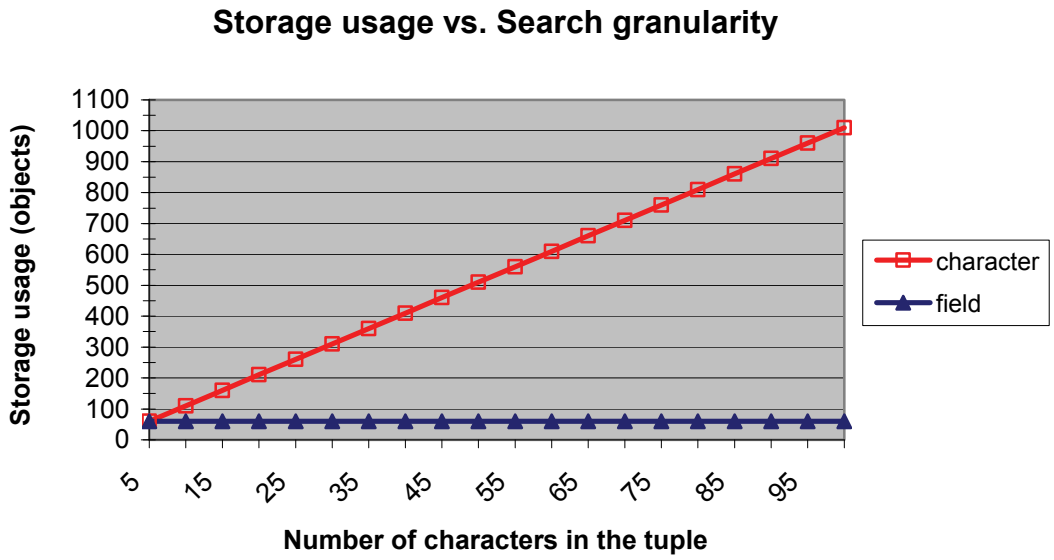

Figure 12: Storage usage of a tuple with 5 fields and different lengths of the value of the tuple. The replication rate is 10. The field-granularity requires only 60 objects independent of the length of the values. The character-granularity requires more objects to store the tuple the longer the value of the tuple is.

The storage usage is counted in objects, because the size of the overhead information is big (e.g. 200 bytes) compared to the size of the value of a field (e.g. 30 bytes for 30 characters). The character-granularity-approach is more expensive than the field-granularity-approach, because every character is stored in an extra object. Figure 12 graphically shows the differences of the storage usage. The x-axis shows the total length of the values in the tuple (number of characters). The tuple has 5 fields in the field-granularity-approach and has to be split into more fields if using the character-granularity-approach. Consider, for example, the total length of the values in the 5 fields of the tuple is 25 (characters). The tuple is split into 25 fields. Every field contains one character. The required objects to store this tuple can be calculated with the Equation 1 as follows: (1 + 25) * 10 = 250 objects, for a replication rate of 10. In the field-granularity approach the number of required objects to store the tuple stays the same, if the length of the values change. In the character-granularity approach the required objects to store a tuple is dependant on the length of the values – the longer the value, the more storage space is required.

Another problem is to search for matching tuples, because if regular expressions are allowed as search keys some cases lead into an infinite number of possible search queries, for which it has to be searched for. For example, the regular expression “A?” where the question mark is a placeholder for one character. The DTSpace generates one search query (i.e. the matching tuple must have two fields, where the first field must have the value “A”) for the expression “A?” which can be handled easily. The regular expression “A*” has an infinite number of possible matches since “*” represents an arbitrary combination of characters. For every possible length of the expression “A*”, a search query has to be sent. If a tuple with a short length (e.g. 5 characters) exists, the search process is finished quickly. But if only a matching tuple with a long value (e.g. 200 characters) exists, the search process has to send 200 queries in the worst case. Consider no matching tuple exists for the expression “A*”. The search process sends an infinite number of queries, since it does not know the length an existing matching tuple. This situation causes an unmanageable amount of search queries. To handle such an amount of queries causes a lot of traffic and may take a lot of time to answer that many queries.

4.7 Search in DTSpace

The SQL language is a sophisticated query language often used in databases [49]. The SQL language can handle regular expressions – even range queries are possible. In a distributed environment the communication overhead to reach the same granularity than SQL is very high. For example, the communication overhead to get a global overview of every stored datum in a distributed network is very high, because every node has to be contacted. Consider a distributed network, existing of hundreds or thousands of nodes, physically distributed all over the world. Contacting every single node in a large network can be time and bandwidth consuming.

As discussed in Chapter 3.2 distributed networks have to manage tasks like robustness, fault-tolerance, etc. In the DTSpace network there is a tradeoff between speed, complexity, storage usage and granularity. As discussed in Chapter 4.6 the

search granularity has a direct impact on the storage usage in the DTSpace network and on the number of search queries to find a matching tuple. The DTSpace approach has a search granularity of one field – either the whole value of a field matches the keyword or does not. The user of the DTSpace is able to increase the granularity by dividing the fields into more fields and therefore able to search for finer keywords.

The search process in the DTSpace approach has three main steps:

1. Search for non-wildcarded fields
2. Search for tuples
3. Access the tuple

Figure 13 illustrates an activity-diagram of the *read*-operation. The *in*-operation is similar to the *read*-operation. In contrast to the *read*-operation, the *in*-operation deletes the tuple and its correspondent fields after it successfully accessed the tuple. Figure 14 illustrates an activity-diagram of the *in*-operation.

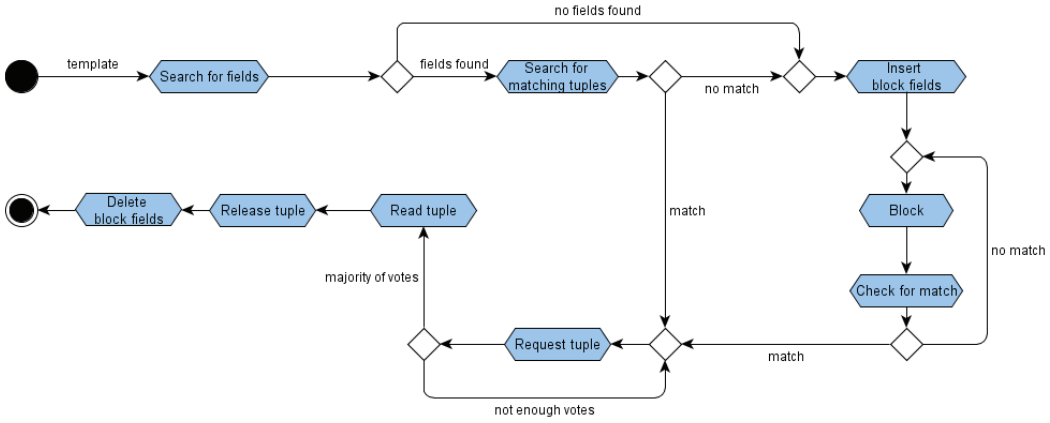

Figure 13: *Read*-operation activity-diagram

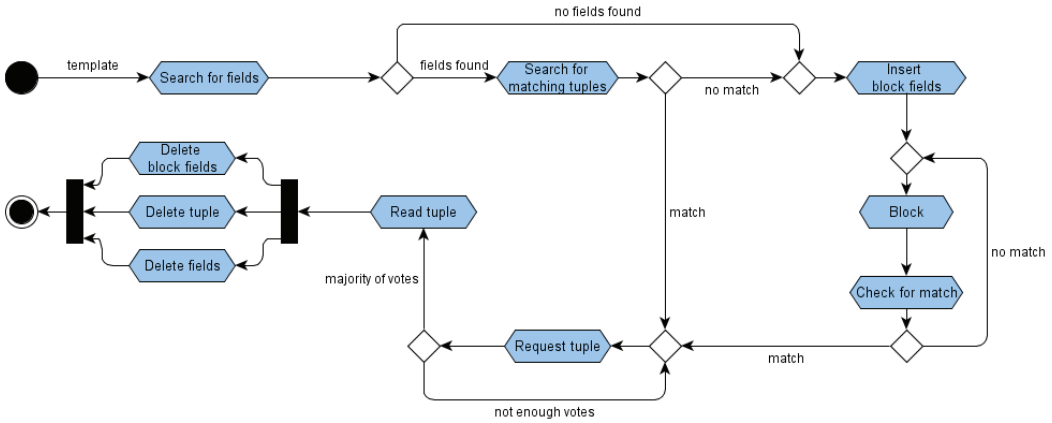

Figure 14: *In*-operation activity-diagram

To search for a tuple, a template needs to be defined first. The template can be considered as a keyword to search for. The template must contain at least one field with a value. Every other field can contain wildcards. The template is handed over to the search process of the local node. The search process searches for every field in the template for which a value has been defined (i.e. non-wildcarded fields). If the node receives a matching field, it reads the tuple reference, stored in the field. The node starts to search for this reference. The reference acts as a pointer to the tuple to which this field belongs. If a tuple has been found, the search process checks if it matches the template. If the tuple matches the template, the distributed voting algorithm is used to get mutual exclusive access to the tuple. The distributed voting algorithm is discussed in detail in Chapter 4.8. If no matching field or tuple is found, the node inserts so-called block fields, which contain a reference to the searching node and the value of the field for which the node searches for. After inserting these block fields, the node blocks (e.g. waits a certain amount of time) and periodically repeats the search for matching fields and tuples.

If another node in the DTSpace inserts a new tuple in the meantime, it searches for block fields, which match one or more fields of the newly inserted tuple. If a matching block field is found, the node sends a notification message to the blocking node that new matching fields has been inserted into the DTSpace network. If the blocking node receives a notification message it immediately ends

blocking (i.e. interrupts the delay period) and begins to search for matching fields and tuples. The block fields are deleted after the search process has ended.

The notification message is sent in a best effort approach in the distributed network. The notification message may not reach its destination. Then the blocking node does not interrupt the delay period and will find the new fields when the next search period starts. The benefit of the notification message is to interrupt the delay period of the blocking node and thus minimizing the overall time to get a matching tuple. The blocking node periodically repeats the search for matching fields and tuples, if the notification message is lost.

4.8 Distributed Mutual Exclusion

If the search process has found a matching tuple, it has to access the tuple in a mutual exclusive way, because the tuple space model provides exclusive access to tuples. Since the tuples are replicated in the overlay network two processes (nodes) may access the same tuple, but different replicas at the same time. The overlay network does not know if the same tuple is accessed at the same time on different replicas (locations), because there is no global view in the overlay network, which can prevent simultaneous access on different replicas. The mutual exclusion is violated, if two nodes access different replicas of the same tuple. To solve this problem, in a distributed network, a so-called voting algorithm is implemented. As discussed in Chapter 3.4 there are several approaches for distributed mutual exclusion algorithms. I combine the two approaches (1) Sigma protocol [47] and (2) End-to-End mutual exclusion protocol [46, 45]. The main features are taken from the Sigma protocol. The End-to-End mutual exclusion protocol maintains a quorum set, which I do not use, because the effort to maintain a quorum set for tuples, which do not reside very long in the DTSpace, is expensive in terms of communication overhead.

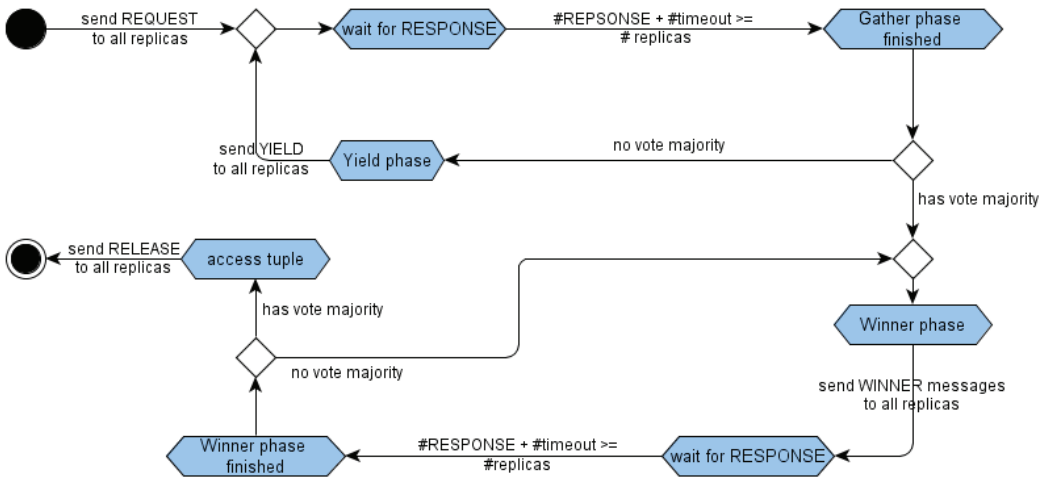

Figure 15: Activity diagram for the distributed mutual exclusion algorithm from the point of view of a requesting node

The voting algorithm approach for the DTSpace has three main phases – (1) a gathering, (2) a winner and (3) a yield phase. The yield phase is only needed, if no majority of votes is reached in the first voting round. Voting is done in the gathering and in the winner phase. Every replica of a tuple acts independently. The replicas maintain a request queue for every tuple. To get access to a tuple, the requesting node has to gather the majority of votes for two times to guarantee mutual exclusion. It is to say, that if a node wants to access a tuple, it sends REQUEST messages to every replica of the tuple in the first phase. If a REQUEST message reaches a replica, the replica stores the request into its request queue. The first node in the request queue is the owner of the replica. Every replica answers with a RESPONSE message (i.e. vote), which contains the current owner of the replica. If a replica already has voted for another node (e.g. other REQUEST messages have been received before), it returns a RESPONSE message, which contains the current owner of the replica. All RESPONSE messages (votes) are counted on the requesting node. There are two different votes for a requesting node: (1) the votes, for which the node itself is the owner of a replica (own vote) and (2) the votes, for which other nodes are the owner of a replica (foreign votes). If the requesting node has got the majority of own votes (i.e. number of

RESPONSE messages, which contain the own node as owner of the replica) the second phase (i.e. winner phase) is started.

In the winner phase the requesting node, which owns the majority of replicas sends WINNER messages to all replica. Every replica, which gets a WINNER message, sets the requesting node as winner of the replica. It is to say, that after the WINNER message is received on a replica, no other node can change the state of the replica – the owner is fixated until the winner node releases the replica. The replica returns a RESPONSE message to the requesting node, if it gets a WINNER message. The RESPONSE message contains the owner (winner) of the replica (e.g. the requesting node). The winner votes are counted again. If the requesting node gets the majority of winner votes (own votes), it is granted access to the tuple.

Consider the situation if no requesting node gets the majority of votes in the first phase. For example, there are 6 replicas for a tuple and three challenging nodes, which want to access the same tuple at the same time. It can happen that every requesting node gets only two votes. So no node has the majority of votes. In this case every requesting node sends a so-called YIELD message to every replica (this is called the yield phase). This is a call for the replicas to reorder their requesting queues and vote again by sending back a RESPONSE message to the sender of a YIELD message. The RESPONSE message contains the new owner of the replica. The problem of the YIELD message is that it is sent the number of requesting nodes times. Every requesting node sends a YIELD message to every replica, because the nodes act independently. Therefore several reorders happen in a short amount of time. To avoid multiple reordering, the order of the request queue on the replica is frozen for a certain amount of time after one reorder is done. Every YIELD message, requesting for a reordering of the request queue is ignored during this time. This solves the problem that a new vote gets invalid as soon as a new YIELD message arrives at the replica.

Consider the case if the request queue of a replica is not frozen. If node A sends a YIELD message and receives a RESPONSE message, in which node A is now the

owner of the replica. Node A counts the RESPONSE message as an own vote. After the replica has sent the RESPONSE message to node A, node B sends a YIELD message. The replica node reorders its request queue again and node B is the owner of the replica now. Node B also receives a RESPONSE message, that node B is the owner of the replica. Both requesting nodes A and B believe that they are the owner of the replica. If this situation happens on several replicas both nodes A and B are able to get the majority of votes. This leads to a conflict, because both nodes send WINNER messages and fixate the owner of the replicas. This race condition has an uncertain outcome. One node may get the majority of votes and thus granted access or no node gets the majority of votes and therefore releases the currently owned replicas and starts over again. This strategy does cost time and message overhead, because the replicas have to be released first before another vote can take place. This can be prevented during the YIELD phase, because no two requesting nodes believe that they are both winners in the first phase. Delaying the reordering process on the replicas leads into a clear situation, in which for example, only one node gets the majority of votes. Therefore only one node sends WINNER messages. There is no guarantee, that the winner node gets the majority of votes in the winner phase (e.g. if nodes fail or the routing does not reach every replica). After the access to a tuple, the owner of the tuple sends a RELEASE message to all replicas to free it and make the tuple accessible for other nodes.

A WINNER message is overrides a REQUEST or a YIELD message. It is to say that if a WINNER message arrives at a replica node, the replica node directly sets the sender of the WINNER message as owner of the replica. This owner cannot be changed until the owner sends a RELEASE message to the replica. Therefore a WINNER message overwrites the current owner, which only has sent a REQUEST or YIELD message. The WINNER message fixates the owner of a replica. If the current owner of a replica has sent a WINNER message and another node accidentally sends a WINNER message, the owner of the replica does not change, since the owner has been fixated with the first WINNER message. Figure 15 illustrates the voting algorithm as activity diagram from the point of view of a requesting node.

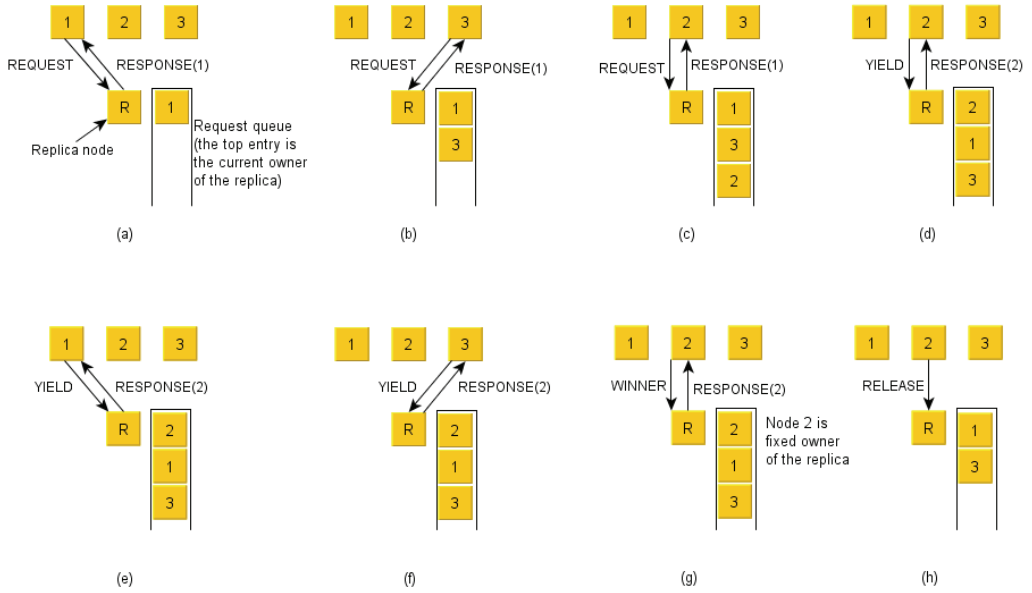

Figure 16: Distributed mutual exclusion process from the point of view of a replica node R.

Figure 16 illustrates an example of a replica node R during a voting process. Three concurrent nodes (1, 2 and 3) want to access the same tuple. The messages illustrated in Figure 16 are sent to all replicas, which hold the tuple in question. Figure 16 illustrates how a replica node acts during concurrent access requests. For example, node 1 requests the tuple on the replica node R and sends a REQUEST message (Figure 16 (a)). The replica node R has not voted before (i.e. the request queue is empty) and therefore takes the request into its request queue. The replica node returns a RESPONSE message, which contains node 1 as owner. Shortly after, the REQUEST message of node 3 arrives at node R (Figure 16 (b)). The replica node R returns a RESPONSE message, which states node 1 as owner of the replica node R, since the REQUEST message of node 1 arrived first on the replica node R. Node R takes the request of node 3 into its request queue at the second place. Also node 2 sends a REQUEST message and receives the same RESPONSE message as node 3 (Figure 16 (c)). The request of node 2 is also put into the request queue of the replica node. Consider the situation that no requesting node gets the majority of own votes and therefore begin to send YIELD messages to all replica nodes (Figure 16 (d)). The YIELD message from node 2 arrives at the replica node R first. This reorders the request queue randomly. Node 2 gets on the first place of

the request queue and is the new owner of the replica node R now. The RESPONSE message for node 2 contains node 2 as the new owner. The request queue is frozen for a certain amount of time, because of the YIELD message. When the YIELD message of node 1 arrives at the replica node, the request queue is still frozen and therefore no reordering of the request queue is done (Figure 16 (e)). Node 1 receives a RESPONSE message with node 2 as owner. In Figure 16 (f) node 3 also sends a YIELD message and receives a RESPONSE, which contains the node 2 as owner, since we consider that the time to freeze the request queue, after a YIELD message has been received, is not over. Consider some time later node 2 got the majority of votes. Node 2 sends a WINNER message to all replicas. For example, if a WINNER message reaches the replica node R, node 2 is selected as owner of the replica. The replica node R returns a RESPONSE message, which contains the final owner of the replica node R (Figure 16 (g)). If node 2 gets the majority of votes for the previously sent WINNER messages, it can access the tuple. If node 2 has finished the access to the tuple, node 2 sends a RELEASE message to every replica (Figure 16 (h)). The current owner is deleted from the request queue of the replicas and the other nodes in the request queue (1 and 3) continue the vote by sending YIELD messages to reorder the request queues on the replica nodes.

4.9 Blocking Operations in DTSpace

The tuple space model described by David Gelernter provides the two operations *read* and *in* as so-called block operations (see Chapter 3.1). If a node does not find a tuple in the DTSpace, it blocks until a matching tuple is inserted. There are two ways to find out if a matching tuple is inserted in the meantime of blocking: (1) periodically repeat the search for a matching tuple and (2) a notification mechanism, if a tuple is inserted into the DTSpace. The DTSpace approach contains both approaches.

If a node does not find a matching tuple, the node inserts block fields for every non-wildcarded field in the template. Every block field contains an address where

to contact the blocking node. Every block field also contains the value of the field for which the node is looking for, since the search process deals with fields in its first phase (refer to Chapter 4.7). The block fields are inserted into the DTSpace. If a node inserts a new tuple it searches for block fields, which match fields of the tuple. A block field matches a field of a tuple, if the index string of a field of the tuple matches the index string of the block field. If matching block fields are found, the node sends a notify message to the blocking node. The blocking node resumes searching for a matching tuple, since there has been inserted a new tuple. The new tuple may not match, because every block field only provides information about one field and the total number of fields in the tuple. If the tuple does not match, the blocking node blocks again (without inserting block fields, because they already have been inserted before). The blocking node waits for a certain amount of time and searches for the tuple again until a match is found.

The purpose of block fields is to notify the blocking node, which resumes searching for a matching tuple before the delay time period (blocking time) runs out. Therefore the notify message shorten the time to wait. The notify mechanism needs not to be reliable, because the blocking node do not rely on a notification (since it periodically repeats to search for matching tuples). If a notify message arrives at the blocking node it increases the performance of the DTSpace (in terms of time), because a matching tuple which has just been inserted is found faster. If a blocking node finds a matching tuple it deletes the previously inserted block fields.

5. Evaluation

In this chapter different test scenarios evaluate the capabilities of the DTSpace prototype implementation. Every test scenario has a different focus (e.g. average duration of the operations).

5.1 Test Environment

One computer is used during the tests. The computer is an AMD Athlon XP 2500+ with 1024 MB RAM. The operating system is Windows XP with SP2. The nodes are started in the environment of Eclipse Europa [50] with Java version 1.6.

5.2 Scenario Setup

Every test scenario uses the same node graph. The initial node graph is illustrated in Figure 17 and contains 10 nodes. The label of every node shows the port number on which the node communicates. The directed arrow shows the bootstrap node of the current node. For example, the node with label “Node@12664” runs at port number 12664 and its bootstrap node is “Node@12663”.

The 10 nodes are started at once. Before starting the test scenarios the nodes is given time to build up a DTSpace network (e.g. building up routing tables and creating paths). After a build up time of 10 minutes the different test scenarios uses the DTSpace network. The first test scenario puts tuples into the DTSpace network with the *out*-operation.

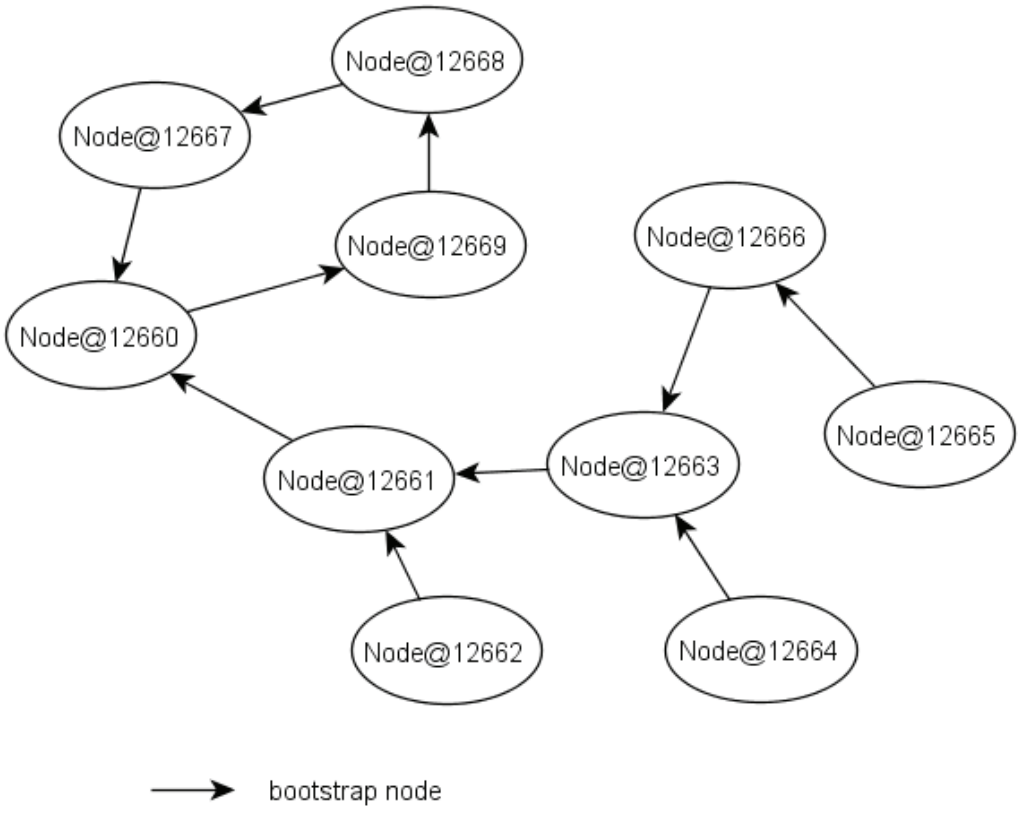

Figure 17: Node graph for the test scenarios (10 nodes)

5.3 DTSpace Operations Test

In these test scenarios the performance of the operations *out*, *in* and *read* is tested. For these three tests, an already built up DTSpace network is used (as discussed in Chapter 5.2). The active node ("Node@12664") does repeat every operation 200 times using different tuples. 200 random tuples are inserted into the DTSpace network.

5.3.1 Out-Operation

In this scenario the performance of the *out*-operation to write tuples into the DTSpace is tested. Therefore the time, which is needed to invoke the *out*-operation, is measured. 200 random tuples are written into the DTSpace network. A tuple has

between 1 and 6 fields and the field values have random character strings. Every tuple is written from the active node with the label "Node@12664" into the DTSpace network. Between two *out*-operations two seconds is waited. The reason of the two second delay is that I have encountered very high CPU usage, if much data is written into P-Grid in a very short time. The delay keeps the CPU usage low and the operation time can be measured without interference of the CPU load. Figure 18 depicts the times, which every *out*-operation needs to write a tuple into the DTSpace network. The red line shows the average time for an *out*-operation. The average time to invoke an *out*-operation for the 200 tuples is 550 ms, as illustrated in Figure 18.

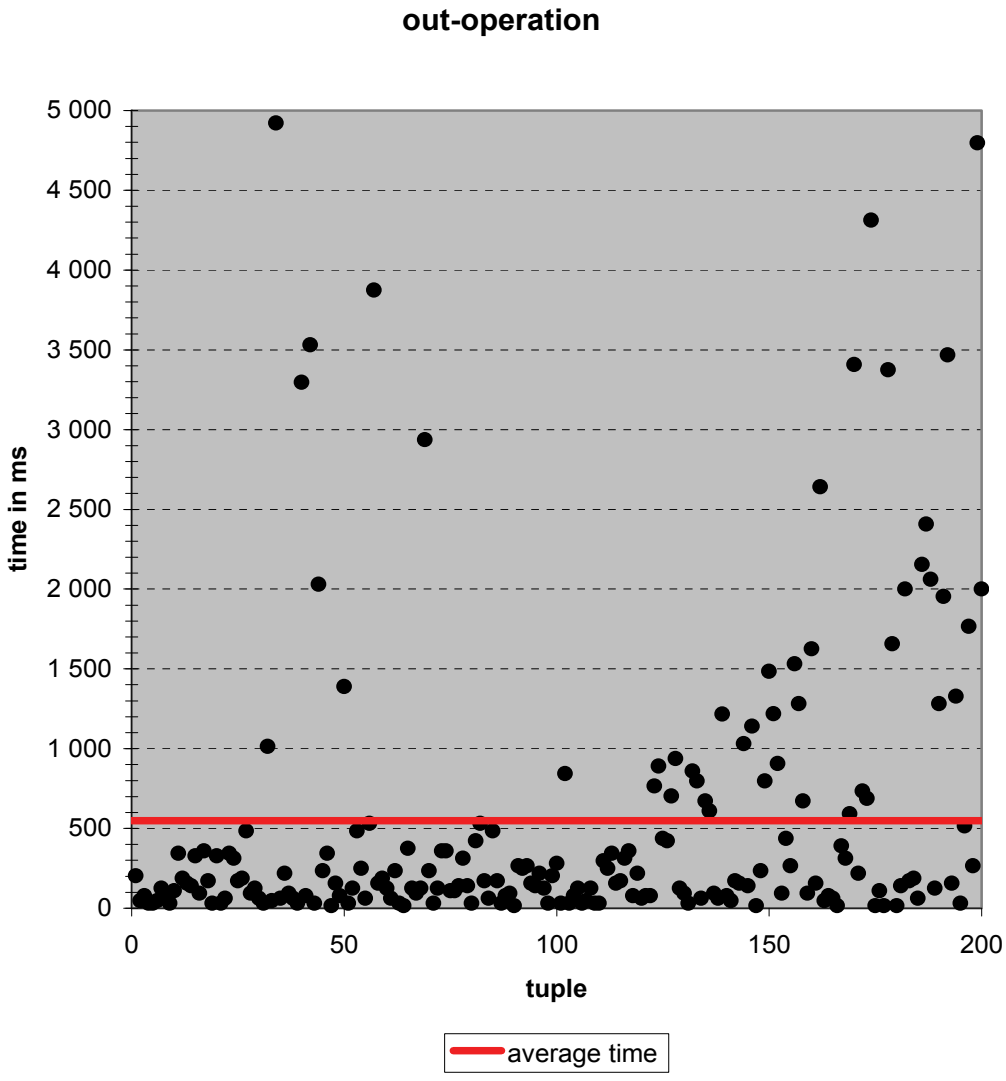

Figure 18: Time to invoke an *out*-operation.

5.3.2 Read-Operation

This scenario tests the performance of the *read*-operation. The test is similar to the *out*-operation test in Chapter 5.3.1. 200 tuples already exist in the DTSpace network. The templates, which are used for the *read*-operation, are randomly generated from the tuples already existing in the DTSpace network. Therefore at least one matching tuple exists for every template. 200 templates are tested. The time, which is needed for every *read*-operation is presented graphically in Figure 19. The red line shows the average time needed to invoke a *read*-operation (i.e. 14.200 ms).

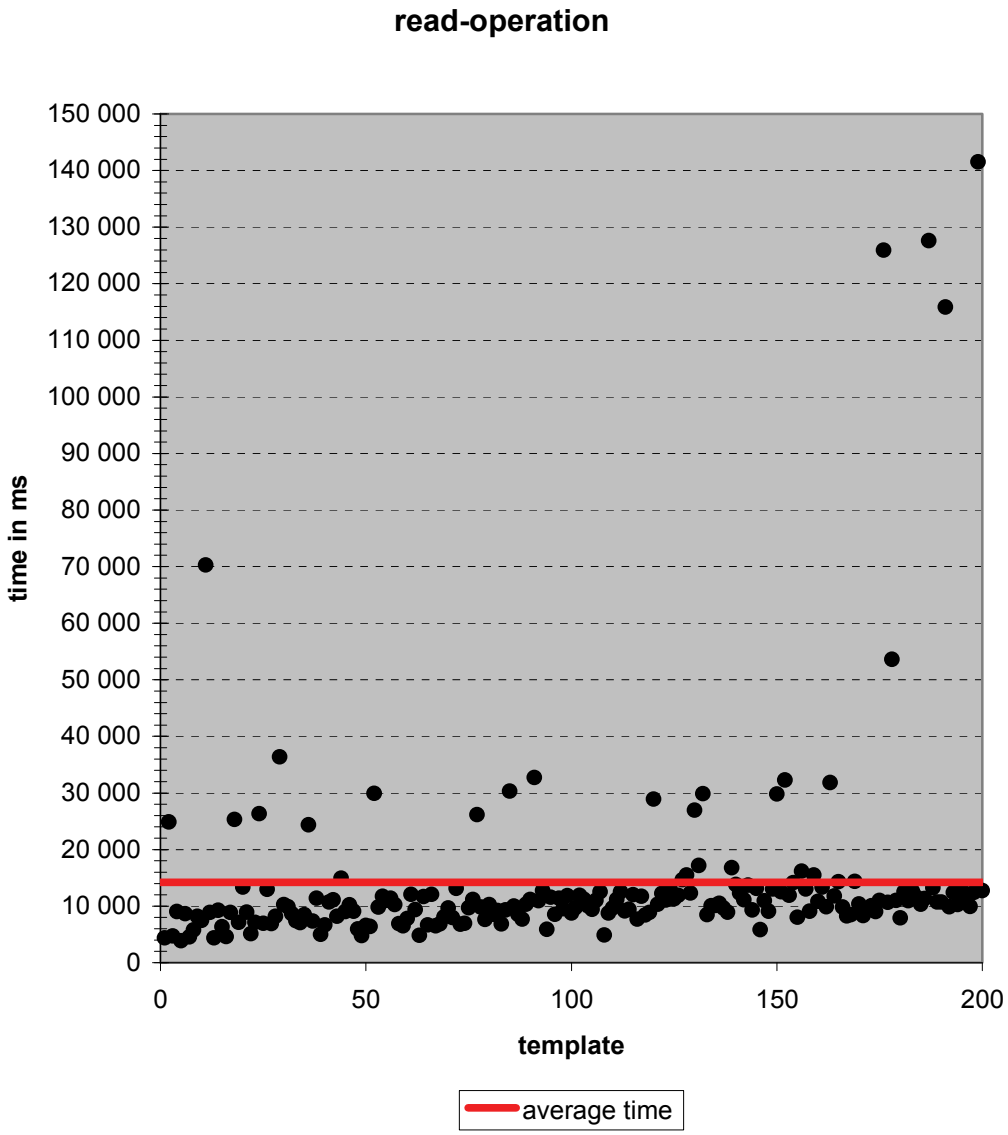

Figure 19: Time needed to invoke a *read*-operation.

5.3.3 In-Operation

The test of the *in*-operation is done like the *read*-operation test in Chapter 5.3.2. The 200 templates from the *read*-operation test are used for the *in*-operation test. The time needed for the invocation of the *in*-operations is graphically illustrated in Figure 20. The red line describes the average time needed to process an *in*-operation (i.e. 5.853 ms).

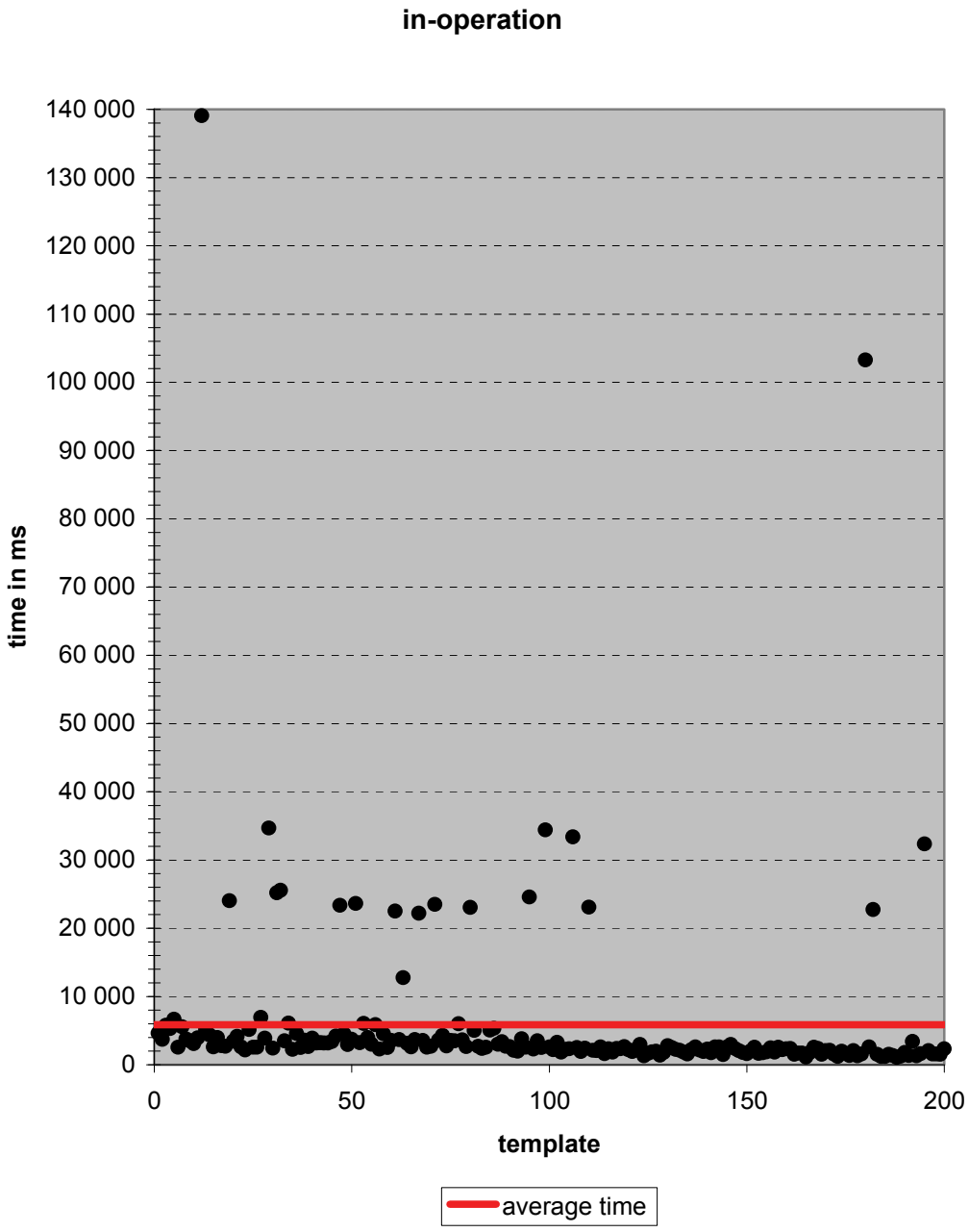

Figure 20: Time to invoke an *in*-operation.

5.4 Concurrency Test

In this test scenario the distributed mutual exclusion of the DTSpace is tested. A modified *read*-operation is invoked by three nodes at the same time for the same tuple to test the distributed mutual exclusion. The DTSpace prototype implementation is modified not to release a tuple after a *read*-operation successfully accessed a tuple, because the original *read*-operation in the DTSpace prototype implementation does release a tuple after the node accessed it. If other nodes try to get access to the same tuple, no other node is allowed to access it. Consider, for example, a node gets granted access to a tuple. In order to test the mutual exclusion algorithm two other nodes try to get access to the same tuple.

The replication rate in the DTSpace network is 6. Because there is no global view of the DTSpace network the replication rate has to be approximated. Therefore it cannot be guaranteed, that there exists exactly 6 copies of a tuple. First, the majority threshold was set to 4, but the result was that none of the three concurrent nodes can get access to the tuple. Therefore I take an optimistic approach and set the majority threshold to 50% (i.e. 3) compared to the replication rate. In the following test results 3 votes out of maximal 6 votes are necessary to get access to a tuple. As the test results state in Table 5 50% threshold is enough during concurrent voting. Either one node or no node is granted access to a tuple. Table 5 illustrates the time to get access to a tuple and which node of the three concurrent nodes gets granted access. There are 6 results, where no node gets access to the tuple. The rows, which are only filled with '-', are results, where no node is granted access to a tuple. This may happen in an unreliable P2P network, but there is no case, where two nodes get granted access to the same tuple at the same time. The distributed mutual exclusion algorithm in the DTSpace prototype implementation is not violated even with an optimistic 50% threshold.

Time in ms	Node 1	Node 2	Node 3
6.250	**x**	**-**	**-**
7.313	**-**	**-**	**x**
-	**-**	**-**	**-**
10.110	**-**	**-**	**x**

5.984	-	**x**	-
-	-	-	-
10.141	-	-	**x**
17.468	-	-	**x**
10.282	-	**x**	-
6.484	**x**	-	-
-	-	-	-
-	-	-	-
6.516	-	-	**x**
6.328	**x**	-	-
-	-	-	-
-	-	-	-
14.313	-	**x**	-
9.719	-	**x**	-
10.250	-	**x**	-
8.765	-	-	**x**
6.578	-	**x**	-
17.515	-	-	**x**
34.672	-	**x**	-
13.281	-	**x**	-
30.375	**x**	-	-

Table 5: The nodes 1-3 try to access the same tuple at the same time. The 'x' illustrates, which node gets granted access. The time in ms in the left column shows the time, which is needed to get access to the tuple. The sign '-' illustrates that the node in this column does not get access to the tuple.

5.5 Conclusion

The measured times of the different operations show that the DTSpace network is an unreliable P2P network. The results of the *out*-operation test states that it takes about 0.5 second in average to write a tuple into the DTSpace network. In the beginning of the test phase I encountered high CPU load, if many *out*-operations are invoked in a short time (e.g. invoke a new *out*-operation as soon as the previous operation is finished). The reason was that P-Grid peaks the CPU load up to 100% if tuples are inserted very fast. To circumvent this I delayed every *out*-operation by two seconds.

The average time in the *in*-operation test is nearly half of that in the *read*-operation test, although the *read*- and *in*-operations are very similar. There are several possible reasons for this improvement. On the one hand in the *in*-operation test the DTSpace network is running longer than for the *read*-operation test, because the same DTSpace network is taken for the two tests. On the other hand the same templates are used to invoke the *in*-operation. This behavior indicates that P-Grid has improved the routing towards the tuples during the *read*-operation test, which was done before the *in*-operation test. Another possible reason could be that the more often a route is used, the more sophisticated the route becomes in P-Grid. However, P-Grid may learn from messages to improve its routing infrastructure and therefore improves the message passing. These are only proposal reasons, why the operations in the second test (i.e. *in*-operation test) are faster. However, further investigations had to be done.

The concurrency test give results about the distributed mutual exclusion algorithm. Some tuples could not be accessed in concurrency, because no concurrent node got the majority of votes. There are a few reasons, why this can happen. First, the message to all replicas, which is sent by every concurrent node, may not reach enough replica nodes, due to the unreliable nature of P2P networks. Another reason could be that there exist too few replicas to get the majority of votes during the voting algorithm. P-Grid maintains the number of replicas, but since P-Grid is a P2P network no node has a global view over the network to know the current number of replicas of a tuple. Therefore the number of replicas is estimated towards the configured replication rate in P-Grid.

6. Future Work

The DTSpace approach has a simple interface for the processes, which uses the DTSpace. An API between the DTSpace and the overlay network would make it easier to use the DTSpace approach with other P2P networks too. The DMutex module (as discussed in Appendix A) is an example to reuse the DTSpace for another P2P network. The benefit to use the DTSpace approach with different P2P networks would be that it can be tested in different environments or the P2P network performance can be tested with the DTSpace API, since the DTSpace is quite demanding (e.g. replication, message passing for votes, many storage objects).

Improving the distributed mutual exclusion approach is another interesting field for further research. The majority threshold and replication rate are predefined in the current approach. To make the majority threshold and the replication rate dynamic (since the majority of votes depends on the replication rate) is a non-trivial challenge, since it has to be guaranteed, that the mutual exclusion must not be broken. Some P2P networks work with randomized attributes (e.g. random walker, discussed in Chapter 3.2). Also the replication rate in P-Grid is estimated around the configured replication rate. Therefore making the majority threshold dynamic is interesting. Consider the situation that some replicas fail and there are too few replicas available to get the majority to access the tuple. Consider the replicas are only replaced during maintenance round, but the time for a new maintenance round has not come yet. Therefore no process can access the tuple during this time. If the majority threshold is dynamically adapted to the number of replicas, currently available, nodes can access tuples, even though replicas are missing.

Another interesting extension of the DTSpace approach would be to make the DTSpace capable for range queries. For example, to match tuples which's field value lies within a certain range (e.g. return a tuple, which's first field has a value between 3 and 9).

7. Summary and Conclusion

This thesis illustrates a novel approach, which uses a P2P network to distribute a tuple space (TS). Beginning with the file-sharing tool Napster in 1999, P2P networks have drawn up attention for user applications (e.g. Gnutella) and for research (e.g. Chord). Therefore a lot of research has been done to make P2P networks (1) robust, (2) fault-tolerant and, most importantly, (3) scalable. In this thesis, P-Grid a distributed hash table (DHT) implementation is used for the distributed TS (DTSpace) prototype implementation. P-Grid provides a P2P network API, which is able to replicate data. P-Grid builds up a binary search tree to efficiently search for data and to efficiently route messages in the P-Grid network.

The original TS model was first introduced by David Gelernter in 1985. The TS provides communication and synchronization capabilities for distributed applications. A TS can be viewed as a blackboard on which someone can put notes, read or remove notes. These actions are equivalent to the three TS operations *out*, *read* and *in*. Following the blackboard analogy notes on the blackboard are tuples, and the blackboard itself is a tuple space. During the last two decades many approaches have been developed for the TS. Most of them, for example, JavaSpaces from Sun or TSpaces from IBM, are centrally managed. An example for a DTSpace is SwarmLinda, which imitates the behavior of a swarm in nature to store and retrieve tuples (e.g. like an ant colony).

This thesis also provides an approach for data mapping in a DTSpace. The purpose of data mapping is to make the search process efficient. For every tuple and for every field a so-called index is generated. The index is used to filter out non-matching tuples in the beginning of the search process. For example, to filter out tuples, which, for example, match the values of a field, but the tuple does not match the number of fields of the template.

The TS operations *read* and *in* are blocking operations. A node, which does not find a matching tuple periodically search for matching tuples. A node, which

inserts a tuple into the DTSpace searches for nodes, which are looking for the new tuple. If the node finds a blocking node, the node notifies the blocking node that a new tuple has been inserted. The blocking node immediately resumes its search process. Synchronization in a TS is achieved with mutual access to a tuple. The DTSpace approach in this thesis uses a hybrid approach of the two distributed mutual exclusion protocols Sigma and End-to-End that grants mutual access to tuples.

I implemented a Java prototype on top of the P2P API P-Grid. The prototype implementation provides the three TS operations *out*, *in* and *read* and an implementation of the distributed mutual exclusion approach. Every tuple is split into its fields. The fields are stored, together with the tuple itself, in the P-Grid index. The fields of a tuple hold a reference to the tuple to which they belong. These fields are used to search for matching tuples of a template. A template can contain wildcards to replace unknown values of the tuple to search for. Therefore the search process in DTSpace first searches for fields for which the value is known. The fields in the search result hold references to matching tuples. The found tuples are matched with the template and one matching tuple is returned.

Finally the evaluation of the DTSpace prototype implementation shows the performance and capability of the approach discussed in this thesis. The TS operations *out*, *read* and *in* were tested in a DTSpace network, which consisted of 10 nodes. The time needed to invoke the operations was measured. The results indicated that the distributed mutual exclusion algorithm scales with the size of the P2P network and the number of tuples. Experiments with the vote majority of the distributed mutual exclusion algorithm showed that an optimistic strategy (only 50% of the votes are needed) is sufficient for mutual exclusive access to tuples.

8. References

[1] David Gelernter: Generative Communication in Linda. ACM Trans. Program. Lang. Syst. 7(1): 80-112 (1985)

[2] Joseph Ottinger: Using JavaSpaces. (http://www.theserverside.com/tt/articles/article.tss?l=UsingJavaSpaces, last access: July 2007), February 2007

[3] George Wells, Alan Chalmers, Peter G. Clayton: Linda implementations in Java for concurrent systems. Concurrency - Practice and Experience 16(10): 1005-1022 (2004)

[4] The SwarmLinda Project. (http://cs.fit.edu/~rmenezes/SwarmLinda/, last access: July 2007)

[5] Ahmed Charles, Rolando Menezes, Robert Tolksdorf: On the Implementation of SwarmLinda – A Linda System Based on Swarm Intelligence (extended version). CS-2004-03: Florida Tech, Computer Science, February 2004

[6] Nicholas Carriero, David Gelernter: Linda in Context. Commun. ACM 32 (4): 444-458 (1989)

[7] Gregory R. Andrews: Distributed programming languages. Proceedings of the ACM ’82 conference: 113-117 (1982)

[8] Robert Tolksdorf, Dirk Glaubitz: Coordinating Web-Based Systems with Documents in XMLSpaces. CoopIS 2001: 356-370

[9] Joseph Ottinger: Software: Blitz JavaSpaces 1.26. (http://www.theserverside.com/reviews/thread.tss?thread_id=42164, last access: July 2007), 2006

[10] GigaSpaces. (http://www.gigaspaces.com/pr_ee.html, last access: July 2007)

[11] The Blitz Project. (http://www.dancres.org/blitz/, last access: July 2007)

[12] LighTS. (http://lights.sourceforge.net/, last access: July 2007)

[13] Diego Doval, Donal O'Mahony: Overlay Networks: A Scalable Alternative for P2P. IEEE Internet Computing 7(4): 79-82 (2003)

[14] P2P networks and programs: The development of P2P networks. (http://www.indypeer.org/doc/p2p_networks.php, last access: July 2007)

[15] Ian J. Taylor: From P2P to Web Services and Grids: Peers in a Client/Server World. Springer-Verlag London Limited (ISBN 1-85233-869-5), 2005

[16] Johan A. Pouwelse, Pawel Garbacki, Dick H. J. Epema, Henk J. Sips: The Bittorrent P2P File-Sharing System: Measurements and Analysis. IPTPS 2005: 205-216

[17] Ion Stoica (UC Berkeley): Overlay Networks (slides). (http://www.cs.virginia.edu/~cs757/slidespdf/757-09-overlay.pdf, last access: July 2007)

[18] Filesharing – Napster – History. (http://wiki.media-culture.org.au/index.php/Napster, last access: July 2007)

[19] Andy Oram: Peer-to-Peer: Harnessing the Power of Disruptive Technologies. O'Reilly Media (ISBN 0-59600-110-X), 2001

[20] Giscard Wepiwé: HiPeer: An Evolutionary Approach to P2P Systems. PHD, Technical University Berlin, Germany, 2006 (http://opus.kobv.de/tuberlin/volltexte/2006/1319/pdf/wepiwe_giscard.pdf, last access July 2007)

[21] John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, Ben Y. Zhao: OceanStore: An Architecture for Global-Scale Persistent Storage. ASPLOS 2000: 190-201

[22] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, Roman Schmidt, Jie Wu: Advanced Peer-to-Peer Networking: The P-Grid System and its Applications. PIK 26(3), 2003

[23] Qin Lv, Pei Cao, Edith Cohen, Kai Li, Scott Shenker: Search and replication in unstructured peer-to-peer networks. ICS 2002: 84-95

[24] Vana Kalogeraki, Dimitrios Gunopulos, Demetrios Zeinalipour-Yazti: A local search mechanism for peer-to-peer networks. CIKM 2002: 300-307

[25] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, Scott Shenker: Making gnutella-like P2P systems scalable. SIGCOMM 2003: 407-418

[26] Ji Li, Karen R. Sollins, Dah-Yoh Lim: Implementing aggregation and broadcast over Distributed Hash Tables. Computer Communication Review 35(1): 81-92 (2004)

[27] Manfred Hauswirth, Schahram Dustdar: Peer-to-Peer: Grundlagen und Architektur. Datenbank-Spektrum 13: 5-13 (2005)

[28] Karl Aberer, Manfred Hauswirth: An Overview on Peer-to-Peer Information Systems. Workshop on Distributed Data and Structures (WDAS-2002), Paris, France, 2002

[29] P-Grid. (www.p-grid.org, last access: July 2007)

[30] Karl Aberer, Luc Onana Alima, Ali Ghodsi, Sarunas Girdzijauskas, Seif Haridi, Manfred Hauswirth: The Essence of P2P: A Reference Architecture for Overlay Networks. Peer-to-Peer Computing 2005: 11-20

[31] David Liben-Nowell, Hari Balakrishnan, David R. Karger: Analysis of the evolution of peer-to-peer systems. PODC 2002: 233-242

[32] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, Roman Schmidt: P-Grid: a self-organizing structured P2P system. SIGMOD Record 32(3): 29-33 (2003)

[33] Jon Kleinberg: The Small-World Phenomenon: An Algorithmic Perspective. Proceedings of the 32nd ACM Symposium on Theory of Computing, 2000

[34] Karl Aberer, Anwitaman Datta, Manfred Hauswirth, Roman Schmidt: The P-Grid overlay network: From a simple idea to a complex system (in German). Datenbank-Spektrum, 13, dpunkt.verlag, 2005

[35] Anwitaman Datta, Manfred Hauswirth, Karl Aberer: Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. ICDCS 2003: 76-85

[36] Richard M. Karp, Christian Schindelhauer, Scott Shenker, Berthold Vöcking: Randomized Rumor Spreading. FOCS 2000: 565-574

[37] Karl Aberer: P-Grid: A Self-Organizing Access Structure for P2P Information Systems. CoopIS 2001: 179-194

[38] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Netw. 11(1): 17-32 (2003)

[39] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, Hari Balakrishnan: Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM 2001: 149-160

[40] Antony I. T. Rowstron, Peter Druschel: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Middleware 2001: 329-350

[41] B. Y. Zhao, J. Kubiatowicz, A. Joseph: Tapestry: An Infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, Computer Science Department, 2001

[42] C. Greg Plaxton, Rajmohan Rajaraman, Andréa W. Richa: Accessing Nearby Copies of Replicated Objects in a Distributed Environment. SPAA 1997: 311-320

[43] Andres S. Tanenbaum, Maarten van Steen: Distributed Systems: Principles and Paradigms. Prentice-Hall Inc. (ISBN 0-13088-893-1), 2002

[44] Martin G. Velazquez: A Survey of Distributed Mutual Exclusion Algorithms. Technical Report CS-93-116, Department of Computer Science Colorado State University, 1993

[45] M. Muhammad: Efficient Mutual Exclusion in Peer-to-Peer Systems (thesis). University of Illinois at Urbana-Champaign, 2005 (http://kepler.cs.uiuc.edu/docs/moosa_thesis/moosa_thesis.pdf, last access: July 2007)

[46] M. Muhammad, A. S. Cheema, and I. Gupta: Efficient Mutual Exclusion in Peer-to Peer Systems. Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing: 296-299 (ISBN: 0-7803-9492-5), 2005

[47] Shi-Ding Lin, Qiao Lian, Ming Chen, Zheng Zhang: A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems. IPTPS 2004: 11-21

[48] M. Muhammad, Adeep S. Cheema, Indranil Gupta: Efficient Mutual Exclusion in Peer-to-Peer Systems. IEEE Grid Computing Workshop 2005: 296-299

[49] Mike Chapple: Structured Query Language (SQL). (http://databases.about.com/od/sql/a/sqlbasics.htm, last access: August 2007)

[50] Eclipse Europa. (http://www.eclipse.org/europa/, last access: August 2007)

Appendix

A The DTSpace Prototype Implementation

The prototype implementation for the DTSpace is built on top of the DHT network P-Grid [29]. Because P-Grid has special abilities the modules of the DTSpace prototype are only partially reusable. For example, the search module (Seeker) has to be newly implemented if another overlay network is used. The distributed mutual exclusion module (DMutex) is mainly reusable. The DTSpace objects like fields, tuples and templates are also reusable.

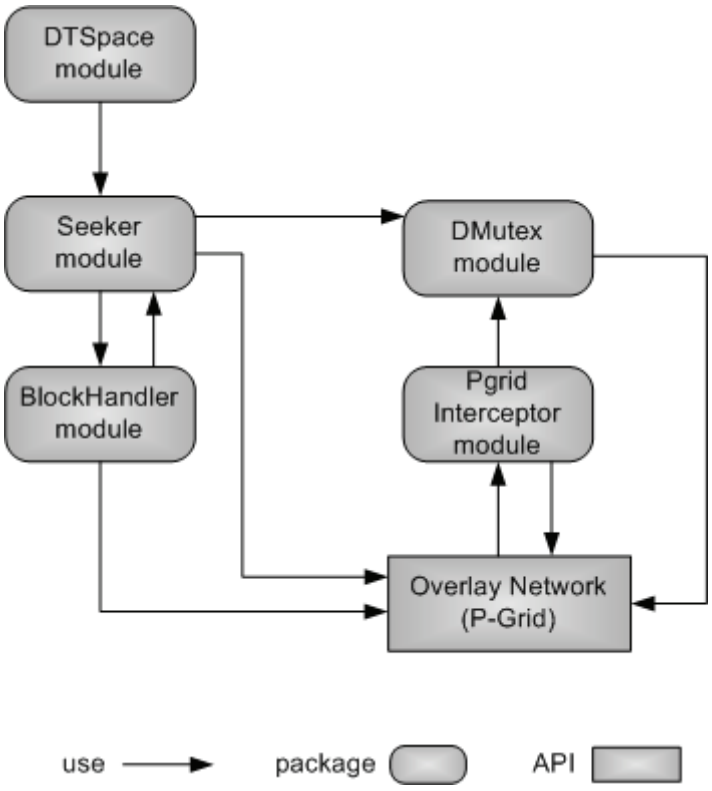

Figure 21: Main modules in the DTSpace prototype implementation

The DTSpace prototype implementation starts three threads (i.e. *dtspace.blockhandler.PGridBlockhandler*, *dtspace.seeker.PGridSeeker* and *dtspace.dmutex.PGridDMutex*) besides instantiating classes (e.g. *dtspace.dmutex.PGridDMutex* for the DMutex module). The threads are started on startup of the DTSpace node. The P-Grid implementation is also started in the

beginning, which itself contains several threads. Figure 21 illustrates a detailed overview of the main modules in the DTSpace prototype and its interactions. The Seeker and BlockHandler modules are responsible for searching and matching tuples. The DMutex module is responsible for the distributed mutual exclusive access of a tuple. The PGrid Interceptor module is built into the P-Grid implementation as discussed later. On top of all modules the DTSpace module is the point of contact to the processes or nodes, which interact with the distributed tuple space. The DTSpace module offers a simple interface, which is illustrated in Figure 23.

The modules, which are illustrated in Figure 21, are discussed briefly as follows:

- **DMutex module** (package *dtspace.dmutex*): This module maintains the request queues for the locally stored tuples. The request queues are stored in a hash table and contain access requests from nodes, which want to access a tuple. The DMutex module incorporates the protocol for distributed mutual exclusion, which is discussed in Chapter 4.8. The modules Seeker and PGrid Interceptor interact with the DMutex module. The Seeker module invokes the DMutex module, if the local node wants to access a tuple. The DMutex module handles the votes for replicas.

- **Seeker module** (package *dtspace.seeker*): The Seeker module is one of the central modules, because it has to coordinate the whole search and match process. If the local node is looking for a tuple the Seeker module sends search queries into the overlay network. If no result is returned or a timeout occurs the Seeker module invokes the BlockHandler module to handle the blocking process as discussed in Chapter 4.7. The BlockHandler module invokes the Seeker module, if a notification message arrives. The Seeker module continues the process, if a notification message is received. After a matching tuple is found the Seeker module also handles the access permission of a tuple by invoking the DMutex module.

- **BlockHandler module** (package *dtspace.blockhander*): This module is invoked if the Seeker module does not find a matching tuple. The BlockHandler module is also invoked, if a new tuple is inserted. The module notifies the nodes, which are waiting for a tuple.

- **PGrid Interceptor module** (package *dtspace.pgrid*): This module handles incoming and outgoing maintenance messages, which are sent in the P-Grid network. The PGrid Interceptor module keeps the request queues in the DMutex module up to date. For example, if a local tuple is replicated to another node, the PGrid Interceptor module adds the current request queue of the tuple to the maintenance message. The request queue is sent along with the new replica. On the receiving node of the replica the PGrid Interceptor module reads out the request queue from the maintenance message and adds it to the other request queues in the DMutex module. If a tuple is deleted the PGrid Interceptor module deletes the correspondent request queue in the DMutex module. The overlay network invokes the PGrid Interceptor module. Therefore the overlay network implementation has to be changed. The changes in the overlay network are discussed in detail in the JavaDocs of the class *dtspace.pgrid.PGridInterceptor*.

- **DTSpace module** (package *dtspace*): This module represents the tuple space interface for the processes and nodes, which are using the DTSpace network. The module invokes the tuple space operations *out*, *read* and *in*.

- **Overlay Network** (P-Grid): This is not a module, written for the DTSpace prototype implementation. It provides an API, which is developed and written by [29]. P-Grid is discussed in Chapter 3.3.1.

Nearly for every class in the DTSpace prototype an interface has been defined. The interfaces can be found in the packages with the prefix *dtspace.interfaces*. Overlay specific (i.e. P-Grid specific) implementations can be found in the packages with the prefix *dtspace.pgrid* or *dtspace.constants.pgrid*, since P-Grid is the overlay

network used in the prototype implementation. Other P-Grid specific classes have the term *PGrid* in their class names.

Constants, which are used in the DTSpace prototype, can be found in the packages with the prefix *dtspace.constants*. The classes in the packages *dtspace.constants* are extendable in an easy way, because they contain a constant, which contains all defined constants in the class. For example, the constant *ALL_PROPERTIES* is an array of strings, which contains all available constants in the class *dtspace.constants.configuration.PropertiesPGrid*. Another example is *ALL_PHASES*, which contains every possible phase existing in the class *dtspace.constants.dmutex.Phases*. The benefit of this structure is that automatic check loops (e.g. for-loops) can easily go through all possible constants and it is easy to get a complete list of possible constant values to see which constants are available. For example, all data types, which are possible for a field, are listed in the array *dtspace.constants.dtspace.FieldTypes.ALL_FIELD_TYPES*. The big benefit for loops, which need to check every constant in a class, is that it is extendible. Consider a new *FieldType* is needed. The new type only needs to be written into the class *dtspace.constants.dtspace.FieldTypes* and must be added to the array *ALL_FIELD_TYPES*. Therefore all loops, which check all available types (i.e. exists for a field) automatically are extended by extending the array *ALL_FIELD_TYPES*, since the checking uses only this array.

The implementations of the interfaces can be found in the packages, which do not have *interface* in their package names. For example, the interfaces in the package *dtspace.interfaces.dtspaceobjects* can be found in the package *dtspace.dtspaceobjects*. Abstract classes are used to be general and reusable for other implementations. For example, the abstract class *dtspace.dmutex.DMutex* is used to implement the distributed mutual exclusion module for the P-Grid overlay network (i.e. class name *dtspace.dmutex.PGridDMutex*). If another overlay network is used, the abstract class *dtspace.dmutex.DMutex* can be used to implement the DTSpace using the new overlay network. Only a few methods have to be implemented instead of the whole interface for the DMutex module. Another

example is the abstract class *dtspace.interfaces.DTSpace*, which also can be partially reused. The modules BlockHandler, which is defined in *dtspace.interfaces.blockhandler.IBlockHandler* and the Seeker module, which is defined in *dtspace.interfaces.seeker.ISeeker* have to be implemented completely new, if another overlay network is used, because these modules have to be adapted very specifically to the used overlay network. The structure of the implemented modules BlockHandler and Seeker for the P-Grid overlay network can be used as a template for a new implementation of the modules. Namely the packages *dtspace.seeker.PgridSeeker* and *dtspace.blockhandler.PGridBlockHandler* can be used as a template for a new implementation of these two modules.

The DTSpace prototype implementation uses the singleton pattern to ensure that specific classes are instantiated only once (e.g. the DMutex module). The singleton pattern also provides access from everywhere in the implementation. It is to say that a reference to a singleton instance does not have to be handed over as parameter to every class, which needs it. I took over this principle from the P-Grid implementation.

The modules Seeker, BlockHandler and DMutex work with phases. For example, if no search is processed the Seeker module is in the phase *IDLE*. Please refer to the JavaDocs of the DTSpace prototype implementation for a detailed description of all available phases for the three modules Seeker, DMutex and BlockHandler (i.e. *dtspace.constants.seeker.Phases*, *dtspace.constants.blockhandler* and *dtspace.constants.dmutex.Phases*).

Figure 22 illustrates a class diagram with the main interfaces (in violet) of the DTSpace objects, their implementations (in green) and references. The dashed lines are implementation relations of interfaces and the continuous lines are generalization relations. The most generalized interface is *IDTSpaceObject*. For example, a template is a special form (implementation) of a tuple. A block field and a wildcard are special fields. These similarities are a benefit for the implementation, because many implemented methods can be reused from the super

class. Another benefit is that working with the objects in the DTSpace is easier to understand, since they are similar. For example, the method *isValid()* is available in every DTSpace object and checks, whether the object contains the correct values or not. The block field is a DTSpace internal object and is not needed at the user level, but since it is a DTSpace object it also belongs to the same package than tuples or fields.

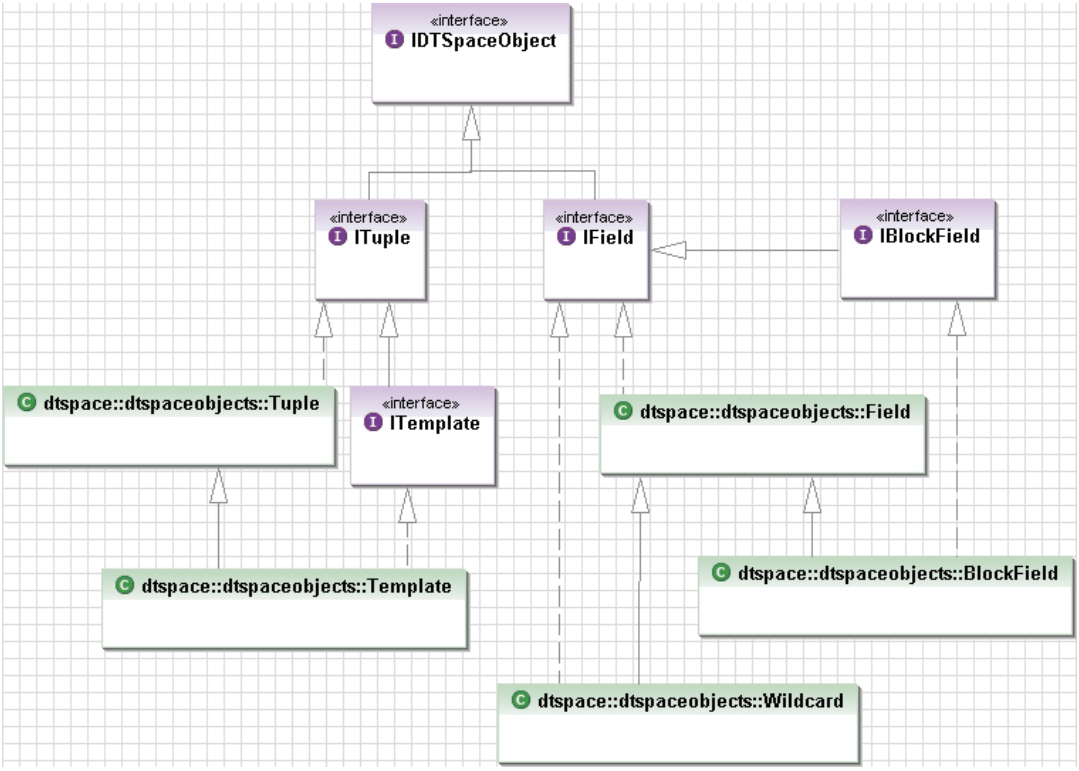

Figure 22: Class diagram of the main DTSpace objects (interfaces are violet and implementations are green)

The DTSpace prototype implementation also contains examples for a DTSpace system in the package *dtspace.test.scenarioA* and *dtspace.test.scenarioB*. These packages contain two classes each, which can be used to test the DTSpace prototype or use them as a template to build an application, which uses the DTSpace.

The class *DTSpaceActive* can be used to interact with the DTSpace network via command line (type *help* to get the possible commands in the command line after

starting the process). The class *DTSpacePassive* is configured to periodically output the locally stored tuples of a node. It is easy to build up a DTSpace network with these two example classes.

Besides the packages and classes discussed in this chapter there are many other packages in the DTSpace prototype implementation. To discuss all of them would go beyond the scope of this thesis. Please refer to the JavaDocs of the DTSpace prototype implementation.

B Configure and Use the DTSpace Prototype Implementation

The DTSpace prototype is written in Java and has a simple interface, which is illustrated in Figure 23.

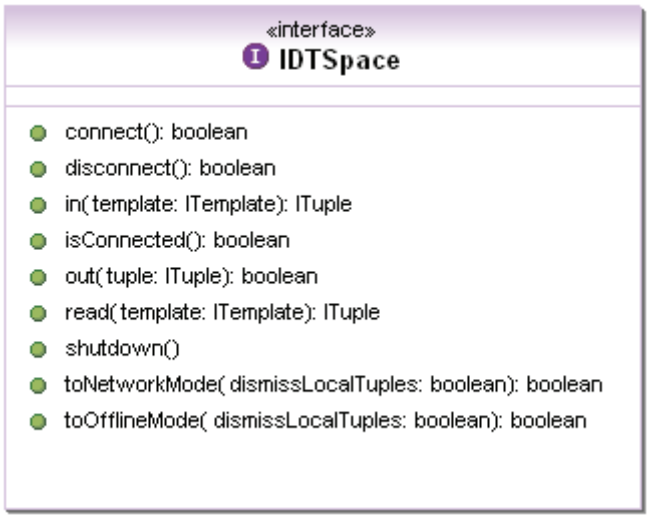

Figure 23: Interface of the DTSpace prototype

The interface for the DTSpace prototype is illustrated in Figure 23. The following methods have been implemented in the DTSpace prototype:

- **connect( ):** This method connects the local node to the DTSpace network.

- **in(template: ITemplate):** Remove a tuple from the DTSpace network, which matches the template.

- **isConnected( ):** Check, whether the local node is connected to the DTSpace network or not.

- **out(tuple: ITuple):** Write a tuple into the DTSpace network.

- **read(template: ITemplate):** This method reads a matching tuple.

- **shutdown( ):** Nicely shut down all processes of the local DTSpace node.

The following methods in the DTSpace interface are not implemented yet (future work):

- **disconnect( ):** Disconnect the local DTSpace node from the DTSpace network. It does not work, because the overlay network P-Grid is not developed to disconnect. P-Grid only leaves the network without special actions (e.g. leave message, transfer locally stored replicas to other nodes before leaving).

- **toNetworkMode(dismissLocalTuples: boolean):** This method connects to the DTSpace network and deletes all locally stored tuples, if the parameter *dismissLocalTuples* = *true*. Locally stored tuples are kept, if the parameter is *false*.

- **toOfflineMode(dismissLocalTuples: boolean):** Disconnect from the DTSpace network and keep all locally stored tuples, if the parameter *dismissLocalTuples* is set to *false*. Otherwise all locally stored tuples are deleted.

The methods *toNetworkMode* and *toOfflineMode* can be used for processes, which work offline from time to time. It is to say that the processes need not to stay online (i.e. connected to the DTSpace network) all the time. For example, a process can prepare tuples to write into the DTSpace network if it is not connected to the DTSpace network (i.e. in offline mode). The tuples are put into the network the next time the process connects to the DTSpace network. Consider the situation where the connection time to the DTSpace network is expensive (i.e. modem connection). The process writes all tuples into the local storage during offline mode. The local process invokes the *out*-operation to write tuples into the DTSpace network. Because the local process is offline the tuples are stored only locally. If the process connects to the DTSpace network by invoking the method *toNetworkMode(false)* the locally stored tuples are put into the DTSpace network. The tuples then are available for every process, which is connected to the DTSpace network.

To configure and work with the DTSpace implementation the following three classes are used:

- **dtspace.DTSpaceFactory:** This class is the factory for every object instance (e.g. DTSpace instance, tuples, templates), which is needed at the user level. For example, the factory is used to create the DTSpace instance at the beginning as well as to create templates to search for tuples, etc. Please refer to the JavaDocs of the DTSpace prototype implementation for further descriptions of the available methods. A detailed example to configure and start a DTSpace node is discussed later in this chapter.

- **dtspace.PGridDTSpace:** This class is the DTSpace instance and is created in the *dtspace.DTSpaceFactory*. This instance provides the three basic TS operations *out*, *read* and *in*. Other methods than these basic operations exist in *dtspace.PGRidDTSpace* (e.g. shutdown). Please refer to the JavaDocs in the DTSpace prototype implementation for a detailed description of the class.

- **dtspace.constants.dtspace.FieldTypes:** This class provides the data types, which must be used in the fields of a tuple or template. For example, if an integer value is stored in a field, the data type can be set to INT. The reason for this additional information for a value is that every value in a field is stored as a string value. For example, an integer value is converted into a string value. If an integer value is converted into a string value the information of the type is lost. Therefore the type has to be declared explicitly. All types are available in the class *dtspace.constants.dtspace.FieldTypes*.

There are two classes to configure the DTSpace implementation on startup:

- **dtspace.constants.configuration.PropertiesDTSpace:** Get a default configuration instance for this class by invoking *dtspace.DTSpaceFactory.createConfigDTSpace()*. All necessary properties are initialized with a default value. This class is used to configure the DTSpace specific properties (e.g. storage directory for the local tuples, blocking delay).

- **dtspace.constants.configuration.PropertiesPGrid:** Get a default configuration instance for this class by invoking *dtspace.DTSpaceFactory.createConfigONetwork()*. The instance, which is returned, contains all necessary properties for the overlay network. The prototype implementation uses P-Grid. Therefore P-Grid specific properties can be set in the class (e.g. directory for the local storage place, hostname and port number of the bootstrap host, replication rate).

How to configure the DTSpace prototype implementation?

First of all, the configuration properties for the DTSpace must be set. As discussed above, the factory generates an instance of the configuration class

(*dtspace.DTSpaceFactory.createConfigDTSpace()*) and initializes it with the default values.

The second part of the configuration is to configure the overlay network. The DTSpace prototype uses P-Grid and therefore P-Grid specific properties are necessary (get a default configuration instance from *dtspace.DTSpaceFactory.createConfigONetwork()*).

For a detailed description of the properties please refer to the JavaDocs of the prototype implementation, since including the JavaDocs goes beyond the scope of this thesis (*dtspace.constants.configuration.PropertiesDTSpace* and *dtspace.constants.configuration.PropertiesPGrid*).

The two configuration classes from above are used to create the DTSpace instance. The code snipped in Figure 24 is an example to configure and create a DTSpace instance.

```
1   /**
2   * Set the arguments...
3   */
4   String localPort = 6655;
5   String dirName = “DTSpacePrototype“;
6   String bootstrapIP = “192.168.0.3”;
7   String bootstrapPort = “1555”;
8
9   /**
10  Get factory instance…
11  */
12  DTSpaceFactory factory = DTSpaceFactory.getInstance();
13
14  /**
15  Create configurations...
16  */
17  IConfigProperties configDTS = factory.createConfigDTSpace();
18  IConfigProperties configPGrid = factory.createConfigONetwork();
19
20  /**
21  Change standard configurations for the overlay network...
22  */
23  configPGrid.setProperty(PropertiesPGrid.ONETWORK_LOCAL_PORT, localPort);
24  configPGrid.setProperty(PropertiesPGrid.BOOTSTRAP_ADDRESS, bootstrapIP);
25  configPGrid.setProperty(PropertiesPGrid.BOOTSTRAP_PORT, bootstrapPort);
26  configPGrid.setProperty(PropertiesPGrid.LOG_LEVEL, "3");
```

```
27      configPGrid.setProperty(PropertiesPGrid.VERBOSE_DEBUG, "false");
28
29      /**
30      Change the standard configurations of DTSpace...
31      */
32      configDTS.setProperty(PropertiesDTSpace.LOG_LEVEL, "2");
33      configDTS.setProperty(PropertiesDTSpace.VERBOSE_DEBUG, "true");
34      configDTS.setProperty(PropertiesDTSpace.BLOCK_HANDLER_DELAY, "2000");
35      configDTS.setProperty(PropertiesDTSpace.PROPERTY_FILE, dirName + ".xml");
36
37      configDTS.setProperty(PropertiesDTSpace.REPLICATION_RATE, "5");
38      configDTS.setProperty(PropertiesDTSpace.DMUTEX_MAJORITY_THRESHOLD, "3");
39
40      String userDirName =
41      configPGrid.getProperty(PropertiesPGrid.ONETWORK_DATA_PATH) +
42      dirName + System.getProperty("file.separator");
43
44      configPGrid.setProperty(PropertiesPGrid.ONETWORK_DATA_PATH, userDirName);
45
46      String dataDir = configDTS.getProperty(PropertiesDTSpace.DTSPACE_DATA_PATH) +
47      dirName +
48      System.getProperty("file.separator");
49      configDTS.setProperty(PropertiesDTSpace.DTSPACE_DATA_PATH, dataDir);
50
51      /**
52      Check, if the configuration instances are valid (ok)...
53      */
54      if !(configDTS.isValidConfiguration())
55      System.exit(-1);
56
57      if !(configPGrid.isValidConfiguration())
58      System.exit(-2);
59
60      /**
61      Create DTSpace...
62      */
63      IDTSpace dts = factory.createDTSpace(configDTS, configPGrid);
64
65      /**
66      Check if DTSpace has been created...
67      */
68      if (dts == null) {
69      System.exit(-3);
70      }
```

Figure 24: Code snippet to configure and create a DTSpace

In the lines 4-7 the user configuration is set like the bootstrap IP address, the local port number and the directory where the configurations and data are stored. The factory to create the DTSpace, tuples, fields and configuration class instances is stored in a variable in line 12 for a more comfortable usage of the factory instance. Line 17 and 18 generate default configuration class instances for the DTSpace. In

the lines 23-27 the default configuration class instance for the overlay network P-Grid is changed. For example, the bootstrap host and the local port, where other DTSpace nodes can contact the local node are set. The lines 32-49 change the DTSpace configuration like the directory where to store the configurations, the replication rate. Refer to the JavaDocs of the DTSpace prototype for a detailed description of the configuration properties. The lines 54-58 are optional. In the lines 54-58 the two configuration instances are checked, whether they are set correctly. For example, it is checked if the log file name is correctly set or the replication rate has a correct value. The line 63 finally configures the DTSpace instance with the properties, which are set in the configuration class instances. After the DTSpace is created it can be used immediately.

The factory for the DTSpace is capable of more than creating the DTSpace instance and its configuration properties. It also creates tuples, fields and templates. Figure 25 illustrates a code snipped to create a tuple. In the lines 9-11 string arrays are defined, which represent the fields of the tuple and its data types. In the line 16 the tuple instance is created, which holds 4 fields which are defined in the line 9 and every field type, which is defined in the lines 10-11. The lines 21 and 22 write the content of the tuple to the standard output as illustrated in Figure 26. The first line in Figure 26 shows the values of the fields of the tuple and the second line holds the data types of the correspondent fields.

```
1       /**
2       * Get the factory...
3       */
4       DTSpaceFactory factory = DTSpaceFactory.getInstance();
5
6       /**
7       * Define the fields in the tuple and data types for every field as array of string...
8       */
9       String[] dataArray = {"University", "Vienna", "Austria", "1040"};
10      String[] dataTypeArray = {FieldTypes.STRING, FieldTypes.STRING,
11      FieldTypes.STRING, FieldTypes.INT};
12
13      /**
14      * Create the tuple...
15      */
16      ITuple tuple = factory.createTuple(dataArray, dataTypeArray);
17
```

```
18  /**
19  * Print the content of the tuple...
20  */
21  System.out.println(tuple.getDataStructure());
22  System.out.println(tuple.getDataTypesStructure());
```

Figure 25: Code snipped to create a tuple

```
1   (University, Vienna, Austria, 1040)
2   (STRING, STRING, STRING, INT)
```

Figure 26: Content of the tuple defined in Figure 25.

Templates are created the same way as illustrated in Figure 27 for tuples. The difference to tuples is that so-called wildcarded fields can be used in templates. A wildcarded field indicates that a field is not important for a match. Figure 27 illustrates a code snipped, which creates a template. Figure 28 shows the content of the template ("*" represents wildcarded fields).

```
1   /**
2   * Get the factory...
3   */
4   DTSpaceFactory factory = DTSpaceFactory.getInstance();
5
6   /**
7   * Define the fields in the template as array of string...
8   */
9   String[] dataArray = {"University", null, null, null};
10  String[] dataTypeArray = {FieldTypes.STRING, null, null, null};
11
12  /**
13  * Create the template...
14  */
15  ITemplate template = factory.createTemplate(dataArray, dataTypeArray);
16
17  /**
18  * Print the content of the template...
19  */
20  System.out.println(template.getDataStructure());
21  System.out.println(template.getDataTypesStructure());
```

Figure 27: Code snippet to create a template

```
1   (University, *, *, *)
2   (STRING, *, *, *)
```

Figure 28: Content of the template defined in Figure 27

Besides these methods to create a tuple or template as illustrated above, there are several other methods in the factory class. They all work similar and offer a simple way to create DTSpace objects (i.e. fields, tuple and templates).

C Abbreviations

API	Application Programming Interface
DHT	Distributed Hash Table
DTSpace	Distributed Tuple Space
HTTP	Hypertext Transfer Protocol
ID	Identifier
P2P	Peer-to-Peer
PDA	Personal Digital Assistant
RMI	Remote Method Invocation
SETI (@home)	Search for Extraterrestrial Intelligence
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol / Internet Protocol
TS	Tuple Space