OPAX - an Open Peer-to-Peer Architecture for XML Message Exchange

eingereicht von

Bernhard Schandl

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magister rerum socialium oeconomicarumque
Magister der Sozial- und Wirtschaftswissenschaften
(Mag. rer. soc. oec.)

Fakultät für Wirtschaftswissenschaften und Informatik
Universität Wien

Fakultät für Informatik
Technische Universität Wien

Studienrichtung Wirtschaftsinformatik

Begutachter:
Univ.Prof. Dr.techn. Wolfgang Klas
Univ.Ass. Dipl.-Inf. Dr.techn. Gerd-Utz Westermann

Wien, im August 2004

Zusammenfassung

Diese Arbeit beschreibt OPAX, eine Nachrichteninfrastruktur auf Basis eines Peer-to-Peer-Netzwerks. OPAX ermöglicht Applikationen die Einrichtung von Netzwerken und die Verteilung von XML-Dokumenten an die teilnehmenden Peers. OPAX verbirgt die administrativen Tätigkeiten des Netzwerks vor der Applikation. Die Netzwerktopologie, also die Anordnung und Verknüpfung von teilnehmenden Peers, kann in OPAX frei gewählt werden, sodass es der Applikation ermglicht wird, die Topologie entsprechend den Bedürfnissen zu wählen und anzupassen.

Ein besonderer Graph, der Hyperwürfel, wird vorgestellt, und seine Eignung als Topologie für ein Peer-to-Peer-Netzwerk wird analysiert. Die Hyperwürfel-Topologie wird in OPAX in zwei Varianten implementiert: ein vollständig symmetrisch-verteilter Ansatz und ein zentralistischer Ansatz werden vorgestellt und diskutiert. Schließlich wird die Referenzimplementation von OPAX und den Hyperwürfel-Topologien beschrieben.

Abstract

This thesis describes OPAX, a Peer-to-Peer messaging infrastructure. With OPAX, applications may create networks and distribute XML documents over the participating peers. OPAX hides the administrative aspects of the network from the application. It provides the possibility to use any topology system to arrange peers and manage their connections, thus giving the opportunity to select and adapt the topology to the application's requirements.

A special graph, the hypercube, is presented and its applicability for a Peer-to-Peer network topology is analyzed. Based on OPAX, both a fully-symmetric distributed and a centralized implementation are presented and discussed. Finally, a reference implementation of OPAX and the hypercube topologies is described.

Contents

Chapter 1

Introduction

1.1 Motivation

The development of computer systems evolved in two major directions since the beginning of the "computer age". The explosion of computing power (by the factor 10^{12} since 1945[32]), together with an enormous decay of prices, and the introduction of high-speed network systems lead to the successful establishment of *distributed systems*.

One key requirement in distributed applications is *messaging*. The fact that parts of the application reside on remote computer systems implies the need for mechanisms that transport information over the underlying network infrastructure to other participants of the distributed system. Without messaging, there is no distributed system.

In this paper, we present an infrastructure that allows the creation of distributed systems without the need to consider low-level aspects of messaging: *OPAX*, an *Open P2P Architecture for XML Message Exchange*, provides functionality to create networks, add and remove participants, and to send and receive messages.

In the context of multimedia, wide fields for the application of distributed systems exist. The increase of network bandwidth allows to distribute multimedia content world-wide. Using a messaging infrastructure, this distribution effort may be reduced in order to only transport the data amount actually required by users and/or applications.

> Events of interest all over the world become available on the Internet as multimedia content like pictures, video and audio streams, or web pages. These events are bound not only to a certain topic but also to their location and time. The question is how recent multimedia events can be made available and "experienceable" to people considering their different personal contexts such as their location and interests. [6]

In such a situation, users subscribe to the system by specifying certain criteria which de-

scribe their fields of interest. Any entity publishing multimedia content sends out "notification telegrams" which contain meta data about the event. Based on their filter criteria, the user's application instance selects events and downloads the multimedia content from the specified location.

> As the Internet's bandwidth increases, *web radio* is getting more and more popular. The large number of radio program providers makes it impossible to keep the view over the bulk of broadcast content. It would be desirable to establish a system where listeners can subscribe to radio programs based on their content.

This is an application similar to the one mentioned before. Based on certain criteria, the software that receives the radio program data selects the broadcast which meets the listener's interests, connects to the appropriate streaming server and plays the program, without any need for the user to search his desired program and switch between different stations.

> An e-learning company wants to establish a blackboard application to enable seminar attendants to discuss their questions and insights, allow teachers to announce hints to problem solutions, and to publish multimedia material to support the instruction. Students subscribe news channels depending on the courses they attend and their personal interests.

A system implementing this scenario may organize the network in several channels, one for each course, one for administrative messages, and so on. The student's application receives messages, filters it, and files it into a news archive which the student may configure according to his/her personal demands.

1.2 Requirements to a Messaging Infrastructure

The following requirements have been defined as the foundation for this work. There may exist other requirements to infrastructures, depending on the application; as for this work, we consider this list as complete.

- *Application Independence* - To allow the re-use of a messaging infrastructure, it is important not to determine parameters which affect the application built on top of the infrastructure. The infrastructure should provide the feasibility to send any kind of data.
- *Scalability* - The scalability of a system may be measured in at least three dimensions[33] - numerical, geographical, and administrative. While the geographical aspect of scalability is not the focus of this work, numerical scalability is crucial for applications that may grow over time. Numerical scalability requires the algorithm to be efficient even when the system has to cope with rapidly increasing participant numbers. Administrative scalability implies the need for a mechanism to ease the administration and maintenance of a growing number of participants.

- *Dynamic* - Akin to scalability, a messaging infrastructure must be able to bear a dynamic number of participants. It must be possible to add or remove participants without to need to reconfigure or restart the whole system. Ideally, such changes in the number of participants should be transparent to the application.
- *Efficiency* - Especially when working with large numbers of participants, the infrastructure must provide algorithms that execute with low effort. Tasks like *broadcast* cause high network consumption if the used algorithms are not designed carefully.
- *Reliability* - A distributed system should be able to handle the sudden loss of participants or links between participants without the need to shutdown the whole system. It should provide mechanisms that make the failure transparent to the application and prevent the loss of data.
- *Transparent Organization* - To make the dynamic and reliability requirements transparent to applications, the infrastructure must be able to organize itself without the need for external intervention.
- *Security* - Although consideration of security aspects is crucial to distributed systems because of the large number of possible vulnerabilities, these aspects are out of the scope of this work.
- *Equality of Participants* - Each participant in the network should have the possibility to send *and* receive messages.

1.3 Peer-to-Peer: An Approach for Messaging Infrastructures?

The requirements stated above lead to the consideration of a *Peer-to-Peer*[1] network to implement a messaging infrastructure. The term "Peer-to-Peer" arose in the beginning days of the Internet and is today one of the keywords of information technology. Many systems, like Napster[3], Gnutella[4], or Freenet[5], have been created and are active to a certain extent today.

Peer-to-peer systems allow the creation of symmetric infrastructures where multiple, equivalent participants(*"peers"*) of an arbitrary number form a network. The combination of the P2P idea with highly ordered topology types, like the *hypercube*, may lead to a scalable, efficient, dynamic, and reliable communication infrastructure. With OPAX, presented in this paper, we implement a useful, yet simple P2P framework that allows to use different topologies, depending on the application's requirements.

Concepts for the publish/subscribe idea in combination with a P2P infrastructure have been presented in [39] and [40], the latter using a content-based publish/subscribe system (Scribe)[41] that is implemented using a distributed hash table.

[1] In this report, also the abbreviated form *P2P* is used.

1.4 Contributions

In this paper, we present OPAX, a Peer-to-Peer-based messaging infrastructure for XML documents. It makes the network operations (opening of, joining, or departure from a network) transparent to the application layer and disburdens it from the administrative aspects of the network. It provides functionality to broadcast application-dependent messages and allows peers to synchronize them upon joining.

OPAX allows the selection of any topology for the network. In this paper, a *hypercube* topology is implemented and presented, as this type of topology fulfills requirements stated before. For the management of the hypercube topology, both a decentralized and a centralized approach are presented and discussed. The paper addresses problems that the usage of the hypercube topology causes and presents approaches to solve these issues. A reference implementation in Java is presented.

1.5 Organization of the Thesis

Chapter 2, "Peer-to-Peer - an Overview", gives a review about P2P networks, as well as past and future evolution of the concept of P2P. Chapter 3 presents the hypercube topology, and its advantages of using it as a P2P topology are depicted. The design principles of the OPAX framework are formulated in chapter 4, while chapter 5 depicts two different approaches to implement a hypercube-based P2P network in OPAX, as well as a comparison of them. Chapter 6, "Example Implementation", documents JIO, an Java-based implementation of the OPAX framework. Chapter 7, "Future Work", specifies which issues the current OPAX specification does not cover, and gives starting points for proceeding activity.

Chapter 2

Peer-to-Peer - an Overview

Today, the term *Peer-to-Peer*, respectively the acronym *P2P* is used in a very vague way. From a social point, a *Peer* is defined as *one that is of equal standing with another*[7]. P2P is seen as a communications model, wherein each participant has the same capabilities as each other.

When talking about P2P from a technical point of view, one usually means Peer-to-Peer-Computing. Wikipedia defines a P2P network as *network that does not have fixed clients and servers, but a number of peer nodes that function as both clients and servers to the other nodes on the network*[8].

Panayiotis Tsaparas defines five characteristics of P2P networks[9]:

1. Clients are also servers and routers
2. Nodes are autonomous (there exists no central administrative authority)
3. The network is dynamic: nodes enter and leave the network "frequently"
4. Nodes collaborate directly with each other (not through well-known servers)
5. Nodes have widely varying capabilities

These criteria implicate certain restrictions to P2P network implementations. The absence of a central managing and coordinating instance implies that a P2P network protocol must cope with unexpected peer or link failures, a requirement that may be hard to fulfill in certain situations, as we will see later. The peers' autonomy requires that the network must be adequately protected against attacks and malignant perpetrators as well as eavesdroppers or message forgers. The direct collaboration of peers requires mechanisms to handle Internet barriers, like firewalls or network address translation facilities.

2.1 Classification of Peer-to-Peer Networks

Classification schemes for P2P networks exist in great number. Lv[34] uses the following classification (alternative denomination taken from [31]):

2.1.1 Centralized (Mediated)

Napster[3] and other similar systems have a constantly-updated directory hosted at central locations (e.g., the Napster web site). Nodes in the P2P network issue queries to the central directory server to find which other nodes hold the desired files. While Napster was extremely successful before its legal troubles, it is clear that such centralized approaches scale poorly and have single points of failure.

2.1.2 Decentralized and Unstructured (Pure P2P)

These are systems in which there is neither a centralized directory nor any precise control over the network topology or file placement. Gnutella[4] is an example for such a design. The network is formed by nodes joining the network following some loose rules. The resulting topology has certain properties, but the placement of files is not based on any knowledge of the topology (as it is in structured designs). To find a file, a node queries its neighbors, which then forward to request to their neighbours, and so on. The most typical query method is flooding, where the query is propagated to all neighbors within a certain radius. These unstructured designs are extremely resilient to nodes entering and leaving the system and to node and link failures. However, the current search mechanisms are extremely unscalable, generating large loads on the network participants.

2.1.3 Decentralized but Structured (Hybrid)

These systems have no central directory server, and so are decentralized, but have a significant amount of structure. By "structure" we mean that the network topology is tightly controlled and that files are placed not at random nodes but at specified locations that will make subsequent queries easier to satisfy. In *loosely structured systems* this placement of files is based on hints; the Freenet P2P network[5] is an example of such a system. In *highly structured systems* both the topology and the placement of files are precisely determined; this tightly controlled structure enables the system to satisfy queries very efficiently.

2.2 Structures Applied to P2P Messaging

For the precise application of constructing a P2P messaging infrastructure, the following considerations can be made about the three categories mentioned above:

1. The centralized topology, as stated above, has the main disadvantage of a single point of failure. The implications of a failure in this central instance are dependent on the responsibilities that this entity has. One can consider a network where the central instance is used to locate and address other peers: In this case, a failure of the central instance causes a sudden breakdown and inhibits communication of any kind. If the central instance is pure topology manager in the sense that it administrates the arrangement and the integration and departure of peers, communication will still be possible in the case of a failure, but it will be impossible to integrate new peers or remove peers that want to leave the network. Appropriate mechanisms have to be integrated into the servers to make them fail-safe and scalable, as needed.

   On the other hand, centralized serves have advantages when the network is desired to perform operations that require *global knowledge* (i.e. knowledge that is distributed across the network), such as searching. Yet, a failure of the central instance makes it impossible to perform those operations. However, with a careful design and disposition of reliabilities, centralized systems may be a practical mechanism for a P2P messaging infrastructure. We will present a centralized structure in section 5.2.

2. A decentralized, unstructured network will have poor performance for operations requiring global knowledge. As the development of such a network's topology may result in a single chain in the worst case, operations like searching or broadcast do not perform well enough to serve high demands. Additionally, as peers may suddenly leave the network, such unstructured networks often can not guarantee that messages be delivered or a search delivers results.

   In [30], several modifications to Gnutella's[4] design are presented that dynamically adapt the topology and the search algorithms to accommodate today's P2P systems' heterogenity.

3. Decentralized, structured networks try to combine the advantages of both approaches. The structured topology of the network allows the implementation of optimized search, broadcast, and routing algorithms. The big disadvantage of this topology type is that the information about the network structure is distributed among peers (as there is no central instance), yet it must be kept in a consistent state because algorithms that are executed locally expect the network to be in such a state. This may cause problems, as we will see in section 5.1, where we present a distributed hypercube topology.

   [27] presents *Chord*, a scalable P2P lookup service based on a key onto node mapping based on a distributed routing table. Kademlia[28] routes queries and locates nodes using a XOR-based metric topology.

   Hildrum and Kubiatowicz[35] introduce an approach that makes P2P systems like Pastry[36, 37] or Tapestry[38] tolerant to certain classes of failures when routing a message through the network.

2.3 History

ARPANET Today's most important and popular network, the *Internet*, has originally been designed as peer-to-peer system. It was the initial goal of ARPANET[1] to share resources, as indicated by the title of ARPA's program plan for ARPANET, *"Resource Sharing Computer Networks"*[20]. ARPANET was connecting independent hosts, distributed over the USA, without establishing master/slave or client/server relations between hosts. As ARPANET was designed as a purely scientific medium, no security barriers had to be installed, allowing unobstructed communications between ARPANET's participants.

The first protocols that were used in ARPANET, Telnet[21] and FTP[22], are asymmetric in the sense that they use a client/server relationship between peers. This fact led to the design of *Network Control Program*[23], a protocol that was also known as "host-to-host" protocol - a terminology very close to today's "peer-to-peer".

Usenet Usenet, also called *News*, can be considered as the first Internet-based file sharing system. Similar to today's P2P systems like Gnutella or Freenet, Usenet does not establish centralized mechanisms of control. Firstly introduced in 1979, Usenet was based on the *Unix-to-Unix Copy Protocol (UUCP)*, which enables UNIX systems to share e-mail, files, or messages. This mechanism, together with a topic hierarchy, and the *Network News Transport Protocol (NNTP)*, led to one of the most important and powerful applications of the Internet.

Usenet is an example for a successful, locally administered system. There exists no central authority; activities like creation of a newsgroup are decided by democratic votes. Every user is allowed to create own newsgroups within the *alt.** hierarchy. Every news host's administrator has the opportunity to select groups that are not available on his host.

These characteristics make Usenet a good example for new P2P applications. Usenet shows how distributed systems may work well, without a centralized controlling instance, presumed the goodwill and mutual valuation of users and administrators, coupled with an simple, even powerful system architecture.

DNS The *Domain Name System* is an example for a system that incorporates P2P technologies as well as hierarchical information structures. Originally, DNS was created to solve a file sharing problem: In early Internet days, the mapping of hostnames to IP addresses was stored statically in a local file. This file was distributed over the network periodically. With the increase of numbers of hosts, this process was no more feasible. To solve the mapping problem, DNS was developed.

One of the major points of interests in the field of P2P is the scalability of DNS: Originally, the system was designed for a few thousand hosts, while today, a hundred millions of hostnames and IP addresses are stored in DNS. Thus, many of the methods used in DNS may be used

[1] Advanced Research Projects Agency Network

in today's P2P application development. These methods include the usage of caching, the forwarding of requests through the DNS network, and the distribution of network load.

The early days of Internet show that the network would not have reached its importance without the appropriation of P2P communication technologies. Today's P2P developers may, and must learn from the experience that was gained during this development process - problems that are fronted today are similar to those that have been solved 15 years ago.

2.4 Peer-to-Peer in the Future

Peer-to-Peer has established as a serious technology for distributed systems. Recent concepts and ideas will improve all aspects of P2P computing, including functionality, performance, stability, and security. Especially the aspect of storing, managing, and retrieving large amounts of data does and will challenge researchers, as the list of Peer-to-Peer conferences[53] shows. Danezis and Anderson[51] present an approach where the storage location of data is assigned depending on the "interests" of participating nodes. Usenet, a "classic" P2P application, is still of interest: an implementation of this idea recreating the Usenet is to be created.

Sit et al.[55] re-implement the Usenet infrastructure using a Distributed Hash Table.

For practical applications, software designers must decide if to include P2P concepts into their systems. Roussopoulos et al.[52] present a "decision tree for analyzing the suitability of a P2P solution to a problem", where they formulate five design criteria and seven classes of P2P solutions.

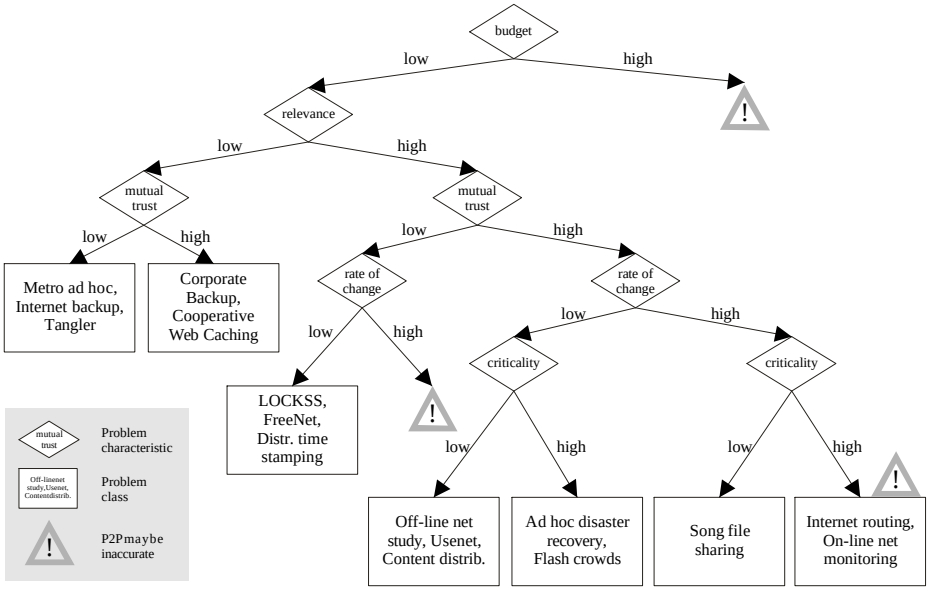

Figure 2.1: Decision tree for P2P suitability (from [52])

To formulate how P2P networking can lower expenses, [54] presents a model to describe the costs of peers participating a network, and compares different topologies regarding economical aspects.

Security aspects play a key role in P2P evolving, as no serious distributed system can exist without such mechanisms. CorSSO[56] is a distributed single-sign-on service, enabling the authentication service to tolerate attacks and failures, and removing users' objections to sign-on services managed by a single administrative entity.

Chapter 3

The Hypercube Topology

This chapter discusses the basic principles of hypercubes in general, as well as a discussion about the usability and the advantages of the application of a hypercube-based topology. It presents the algorithms for the construction and maintenance of a hypercube topology.

This section is mainly based on [10]. It is recapitulated here to give a short and concise view on the concepts and ideas of the hypercube topology applied for a peer-to-peer environment.

3.1 Definition of Terms

Hypercube A *hypercube* is a cube of more than three dimensions. A single ($2^0 = 1$) point (or *node*) can be considered as a zero-dimensional cube, two (2^1) nodes joined by a line (or "edge") are a one-dimensional cube, four (2^2) nodes arranged in a square are a two dimensional cube and eight (2^3) nodes are an ordinary three dimensional cube. Continuing this geometric progression, the first hypercube has $2^4 = 16$ nodes and is a four-dimensional shape ("four-cube") and an d-dimensional cube has 2^d nodes ("$d-$cube").

To make an $d + 1$-dimensional cube, take two d-dimensional cubes and connect each node on one cube to the corresponding node on the other. A four-dimensional cube can be visualized as a three-cube with a smaller three-cube centered inside it, with edges radiating diagonally out (in the fourth dimension) from each node on the inner cube to the corresponding node on the outer cube, or with two three-dimensional cubes connected by parallel edges.

A (complete) d-dimensional cube has 2^d nodes, and $2^{d-1}d$ edges. Each node in a d- dimensional cube is directly connected to d other nodes.

Figure 3.1 shows the construction steps from a 0-dimensional to a 4-dimensional hypercube. In every step, the number of nodes is doubled.

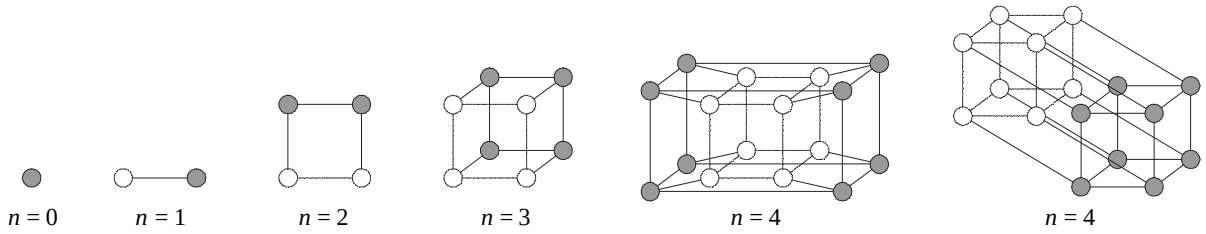

Figure 3.1: Construction of a hypercube

Coordinates and Dimensions We can identify each node by a set of d Cartesian coordinates where each coordinate is either zero or one. Two nodes will be directly connected if they differ in only one coordinate, in other words, they have a Hamming distance (see below) of 1. In this report, we will identify the dimensions of a hypercube starting with 0, and denote the number of dimensions with d_{max}, so a 3-dimensional cube ($d_{max} = 3$) will have dimensions 0, 1 and 2, and will have $2^{d_{max}} = 8$ nodes. The coordinates are identified from left to right, so the first position identifies the position along dimension 0, the second position identifies the position along dimension 1, and so on. The numbering schema for a 3-dimensional hypercube, as well as the arrangement of dimensions, is shown in figure 3.2.

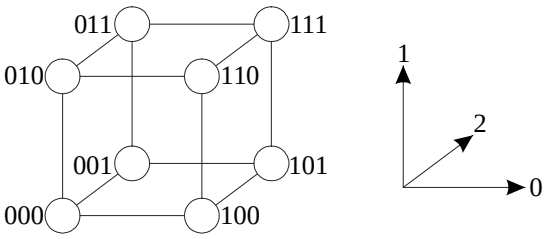

Figure 3.2: Identifying the nodes and the dimensions of a hypercube

We will denote the coordinate vector of node i as $\vec{p}_i$.

Hamming Distance The *Hamming distance* between two Cartesian coordinates is the minimum number of bits that must be changed in order to convert one coordinate string into the other one. Note that adjacent nodes, or *direct neighbours* in the hypercube, have always a Hamming distance of 1. The Hamming distance can be formulated as

$$H(\vec{p}_x, \vec{p}_y) = \|\vec{p}_x \oplus \vec{p}_y\|$$

where $x \oplus y$ denotes the *bitwise XOR* of binary fields x and y, and $\|\vec{z}\|$ denotes the *weight* of the binary vector $\vec{z}$, i.e. the number of positions that are set to 1. One could also interpret the Hamming distance between two nodes as the number of hops that traveling from one node to the other one requires. Note that in an d_{max}-dimensional hypercube, the maximum Hamming distance between two nodes is d_{max}.

Link Dimensionality The *link dimensionality* $L(\vec{x}, \vec{y})$ between two nodes, identified by their coordinates $\vec{x}$ and $\vec{y}$ is expressed by

$$L(\vec{p}_x, \vec{p}_y) = f(\vec{p}_x \oplus \vec{p}_y)$$

where $f(\vec{b})$ returns the position of the most significant bit set to 1 in the binary vector $\vec{b}$. As stated above, and depicted in figure 3.2, the most significant bit is the leftmost bit, and the enumeration of bits starts with index (or dimension) 0. For example, the link dimensionality between the node on position 000 and the node on position 010 would be $L(000, 010) = f(000 \oplus 010) = f(010) = 1$.

One important aspect concerning link dimensionality is the *dimensionality ordering*: Lower dimensions correspond to larger distances. Two nodes that are connected by a link of dimensionality 2 are considered closer than two nodes connected by link of dimensionality 1. This definition of distance is used later in the description of node integration and departure.

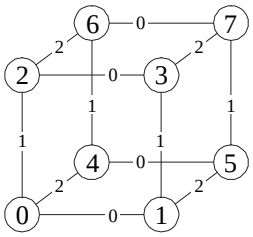

Figure 3.3: A hypercube graph

3.2 Hypercube Applications for Network Topology - Motivations

The simple, regular geometrical structure and the close relationship between the cube's coordinate system and binary numbers make the hypercube an appropriate topology for a parallel computer interconnection network. The fact that the number of directly connected, "nearest neighbour", nodes increases with the total size of the network is also highly desirable for a parallel computer.

What makes the hypercube highly interesting for the application of data distribution, are the following facts[10]:

- *The maximum Hamming distance between any two nodes (the* network diameter*) in the hypercube is* d_{max}. This implies that every message does, presumed an appropriate routing algorithm, in no case require more than d_{max} hops to reach its final destination node.

- *The network structure is symmetric.* In terms of the network's topology, no node incorporates a more prominent position than others, which is crucial for load balancing in the network: Every node can become the source of a broadcast (i.e. the root of a spanning tree of the network), yet the load will always be shared equally.
- *The topology allows to create an optimal spanning tree in terms of messages sent.* Crucial for an efficient broadcasting algorithm, it is always possible to create a spanning tree, starting with any node of the hypercube, so that the maximum number of message transmissions (the edges in the spanning tree) is $2^{d_{max}} - 1$, and every node receives the message exactly once (see figure 3.4). The drawback of this spanning tree is that the workload distribution is imbalanced: the number of messages sent during a broadcast varies between 1 and d_{max} from node to node, depending to its distance to the broadcast originator. However, there exist algorithms that can construct a balanced spanning tree as described in [26].

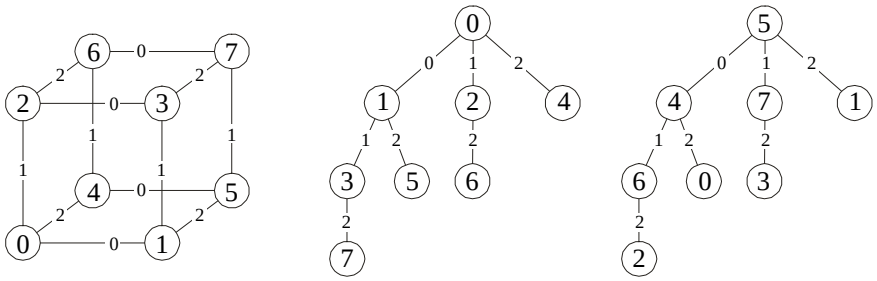

Figure 3.4: 3-cube and two spanning trees

- *The topology provides redundancy.* The *connectivity* (the minimum number of nodes to be removed in order to partition the graph) is optimal, i.e. equal to d_{max}.
- *The network is scalable.* The number of dimensions, d_{max}, can be chosen to adjust the network diameter according to the requirements of the application. As we will see later, it is also possible to create *incomplete hypercubes*, where the number of nodes is not exactly $2^{d_{max}}$. This allows the creation of networks with any number of nodes ranging from 1 to $2^{d_{max}}$. Additionally, d_{max} can be increased at any time to further enlarge the network.

3.3 Algorithms

3.3.1 Broadcast

As mentioned above, an optimal broadcast can be formulated. The algorithm presented below guarantees that in an hypercube with d_{max} dimensions and $n = 2^{d_{max}}$ nodes:

- Any node receives the broadcast message exactly once.

- Exactly $n - 1$ message transmissions are required to reach all nodes in the hypercube.
- The last nodes are reached after $\log n$ steps.

As the ordering of dimensions in the hypercube is unambiguous, it can be used to formulate a broadcast algorithm. The originator of a broadcast (call it O) sends the message to all its neighbours. As stated above, the link dimensionality between two peers can be calculated from their position vectors. Every peer receiving a message to be broadcast forwards it to neighbours along higher dimensions than it received the message on. As an example, refer to figure 3.4, which shows two *forwarding trees* for broadcasts originated by two different peers.

3.3.2 Search

A search in the hypercube may be understood as a broadcast and a subsequent response to the originator. To limit the search, so that not the whole hypercube is affected, two possibilities exist:

- The search broadcast may be endorsed by a *time-to-live* field to restrict the searching to nodes whose distance to the searching node is smaller than a given value.
- The search broadcast may initially be sent not to all neighbours of the originator, but only to neighbours along dimensions $d \geq d_{search} > 0$. In this case, not all nodes get aware of the search, because forwarding of broadcast messages is always bound to higher dimensions. This method has the disadvantage that some of the searching peer's neighbours are excluded from searching.

The reply to the search (the search result) can be sent to the origin node either by contacting it directly (out of the scope of the hypercube) using its physical network address, or with a simple message addressed to the node's hypercube position, which can be forwarded through the hypercube by the routing algorithm.

3.3.3 Routing

A message from a node I at position $\vec{p}_i$, addressed to node J at position $\vec{p}_j$, can be routed through the network as follows: I sends the message to its neighbour on any dimension which is marked as 1 in $\vec{p}_i \oplus \vec{p}_j$. Any following node K compares the destination address with its own address and forwards the message along a dimension marked as 1 in $\vec{p}_k \oplus \vec{p}_j$, gradually matching the destination address. For instance, if node I at position 0111 is to send a message to node J at position 1001, the routing path may be $0111 \rightarrow 1111 \rightarrow 1011 \rightarrow 1001$. The maximum length of a routing path through the hypercube is equal to the maximum Hamming distance between two nodes, d_{max}.

3.4 Incomplete Hypercubes

For the algorithms described in section 3.3, the hypercube must be complete, i.e. all positions must be vacated. Any "hole" in the cube would cause the process to stop or not to finalize successfully. Even under the assumption that the hypercube may grow indefinitely in terms of dimensionality, this implies that the hypercube can only contain exactly $2^{d_{max}}$ nodes, where d_{max} is the number of dimensions.

It is obvious that this is a restriction which is not acceptable for practical applications of the hypercube topology. A network should be able to incorporate any number of nodes, possibly bounded by a maximum number. To accomplish this, the following construction and maintenance algorithm is based on the notion that nodes in the evolving hypercube graph take over responsibility for more than one position.

To keep the cube consistent and symmetric up to a certain degree, nodes are not allowed to cover any combination of positions, but only *sub-cubes* of the whole hypercube. The positions that a node covers in addition to its "own" position can be expressed by a *cover map*: a set of d_{max} bits, each bit representing whether the node "expands" to this dimension (1) or not (0). So a node covering the whole hypercube has a cover map containing only 1s, a node covering no additional positions has a cover map containing only 0s. We will denote the cover map vector of node I with $\vec{c}_i$.

The idea of nodes covering multiple positions leads us to a new definition of the term *immediate neighbour*: Immediate neighbours are nodes which would also be neighbours if the hypercube topology was complete, i.e. all nodes were present. A node V is an immediate neighbour of node W if and only if

$$H(\vec{p}_v, \vec{p}_w) = 1$$

This definition implies that each node may have exactly one immediate neighbour per dimension.

3.5 Randomization

To ensure an even distribution of nodes over the hypercube, three steps of randomization are executed when a node enters the hypercube.

Contact Node Selection A node which wants to join the hypercube may contact any peer which is already integrated. If a sufficient number of nodes is publicly known, the workload of the "first contact" is distributed equally amongst them.

Integration Position Selection A node contacted by a newcomer node does not immediately integrate the node by itself, but randomly selects a position, $\vec{p}_{dest}$, on which the new node is to be integrated. Thus, every position of the hypercube may be the destination position with equal probability. Then, the integration request is routed through the network until the first *integration champion*[1] is found.

This routing procedure adds a maximum delay of d_{max} (worst case) messages to be sent to the integration process. As, in incomplete hypercubes, certain nodes may cover more than one position, these nodes have a higher probability to be selected as integration champion. On the other hand, this approach helps to distribute workload among nodes: the more positions a node covers, the earlier it will be selected as integration champion, thus reducing its number of covered positions.

Routing Path Shortening To shorten the procedure of routing the integration request to the first integration champion, any node which receives the request on its way may pick it and immediately become the integration champion instead of continuing to forward the request to its primary destination. The probability that a node V receiving an integration message to forward becomes the integration champion is set to

$$p = 1 - \frac{H(\vec{p}_v, \vec{p}_{dest})}{H(\vec{p}_{src}, \vec{p}_{dest})} = 1 - \frac{\|\vec{p}_v \oplus \vec{p}_{dest}\|}{\|\vec{p}_{src} \oplus \vec{p}_{dest}\|}$$

where $\vec{p}_{src}$ is the position of the node that initially issued the integration message (the "first contact" node) and $\vec{p}_{dest}$ is the integration position for the new node. So the probability for picking the message depends on the length of the path that the integration request has travelled so far: the longer the travel did last, the higher the probability that it will be truncated.

Simulation results for this randomization scheme can be found in [10].

3.6 Distributed Construction and Maintenance of a Hypercube

In a distributed topology, integration and departure of nodes must be executed so that the hypercube topology is always *consistent*. So, nodes are required to perform well-defined steps in order to keep the consistency of the network.

After the first integration champion has been selected according to the rules described in section 3.5, the first integration champion carries out the following steps:

1. *Integration dimension selection.* This is the dimension along which the new peer will be integrated by the integration champion.

[1] We denote a node as *integration champion* if its set of covered positions is changed during the integration of a newly arriving node.

2. *Integration champion node appointment.* If necessary, the integration responsibility is passed over to the appropriate node(s).

3. *Node integration* The positions which are covered by the node are assigned, the new node is notified, and connections to its neighbours are established.

3.6.1 Integration Dimension Selection

The node which currently covers the integration position - selected by the randomization process described above (call it V) - checks if it covers any additional positions. If this is the case, the *integration dimension*, d_{int}, is set to the position of the most significant 1 in its cover map.

If the dimension can not be defined because V does not cover any additional positions, it looks for non-immediate neighbours in its set of known peers: if a node W has a Hamming distance > 1 to node V, it may be a candidate to be selected as actual integration champion, since this node must cover additional positions near to $\vec{p}_{int}$. Let $\mathcal{N}_v$ denote the set of non-immediate neighbours of V: the integration dimension is set as

$$d_{int} = min_{w \in \mathcal{N}_v}(L(\vec{p}_v, \vec{p}_w))$$

where $L(\cdot)$ denotes the link dimensionality between two nodes. This rule implies that lower dimensions are filled up prior to higher dimensions, keeping the number of "partially fractured" dimensions low. As an example, if the network consists of four nodes, there is no need to establish a 3-dimensional cube: a 2-dimensional cube is able to accommodate all nodes. However, the goal of building the most "dense" hypercube (for n nodes, the dimensionality should never exceed $\log n$), cannot always be satisfied, as shown in [10].

Now, the *integration position* for the new node is updated by flipping the bit on position d_{int} of the integration champions own position $\vec{p}_v$:

$$\vec{p}_{int} = (p_v^0, p_v^1, \ldots, p_v^{d_{int}-1}, \overline{p_v^{d_{int}}}, p_v^{d_{int}+1}, \ldots, p_v^{d_{max}-1})$$

3.6.2 Integration Champion Node Appointment

If V is not the actual integration champion, it forwards the request to the selected node. Otherwise, it executes the following algorithm by itself.

3.6.3 Node Integration

The new node will be integrated in the network, and existing nodes which are affected by the integration may be classified by two groups:

Integration champions are nodes which covered additional positions that will be covered by the new node after integration.

Prospective neighbours are nodes which do not transfer positions to the new node, but will be neighbours of the new node.

The first step that the integration champion performs is to identify other integration champions: if V at position $\vec{p}_v$ is integrating a new node at position $\vec{p}_{int}$, any node W is integration champion if it is closer to V than V is to $\vec{p}_{int}$, formally if

$$H(\vec{p}_w, \vec{p}_v) < H(\vec{p}_{int}, \vec{p}_v)$$

It is clear that node V is closest to the integration position $\vec{p}_{int}$ since it is currently covering this position. Neighbours W of V may cover positions that are closes to $\vec{p}_{int}$ than to $\vec{p}_w$ and thus will be transferred to the new node. All these nodes will become integration champions and have to be informed that they are to abandon these positions.

The initial cover map for the new node will be a modified copy of V's cover map $\vec{c}_v$: for dimensions lower than d_{int}, it is the same as $\vec{c}_v$. The bit at d_{int} will be a 0 since V will be the new node's neighbour along dimension d_{int}. For dimensions higher than d_{int}, the cover map is filled with 1s:

$$\vec{c}_{int} = (c_v^0, c_v^1, \ldots, c_v^{d_{int}-1}, 0, 1, 1, \ldots)$$

As V did now compute the integration data for the new peer, it confirms the request by sending the peer its position, its cover map, and the list of integration champions. The new peer now registers itself at every integration champion.

An integration champion receiving the integration request from the new peer firstly calculates its "local" d_{int} as the dimensionality of the link between the new peer and itself, then it flips its cover map on bit d_{int}, as the new peer is its neighbour along this dimension from now on. As positions have to be transferred to the new peer, links to neighbours have also to be transferred: the integration champion computes the list $\mathcal{L}_{champions}$ of positions that remain covered by the integration champion node, and the list $\mathcal{L}_{new}$ of positions that will be covered by the new node after the integration has been carried out[2].

Firstly, the integration champion node V calculates its list of positions: each 1 in V's cover map $\vec{c}_v$ means that the node covers additional positions along this dimension. Thus, the length of the list $\mathcal{L}_v$ is determined by the weight $\|\vec{c}_v\|$ of its cover map: If a node's position is $\vec{p}_v = 011$

[2]The computation of the lists may cause problems due to the fact that a full list of covered positions may have a length of $2^{d_{max}}$, and the algorithm that creates a full enumeration of covered positions presented in [10] also has a complexity of $O(2^{d_{max}})$. We will discuss this problem in section 5.1.6 and present a possible solution in section 5.1.7.

and its cover map is $\vec{c}_v = 101$, its list of covered positions is $\mathcal{L}_v = (011, 111, 010, 110)$. We spell this operation as

$$\mathcal{L}_v = \vec{p}_v \times \vec{c}_v$$

The lists $\mathcal{L}_{champion}$ and $\mathcal{L}_{new}$ are calculated using V's position vector and its cover map with the flipped bit d_{int}, respectively the position vector of the new node and the new node's cover map.

Then, for each position $\vec{p}_i \in \mathcal{L}_{new}$, the integration champion must identify neighbour positions along dimensions $0 \leq d < d_{int}$ where the cover map is 0, i.e. where an immediate neighbour node exists:

$$\vec{p}_{neighbour(p)} = \vec{p} \oplus \vec{d}$$

(where $\vec{d}$ is a vector consisting of 0 and a single 1 at position d. For each of these positions the node which currently covers this position is identified out of V's set of neighbours; this node (call it W) will be a neighbour of the new node.

In the next step, the integration champion V must determine if W remains to be V's neighbour, or if the new node will be located "between" them and thus V can abandon its link to W. This is accomplished by carrying out the same algorithm with positions $\vec{p}_i \in \mathcal{L}_{champion}$: any node which is no closest node to any of V's remaining positions may be abandoned; any node which has the smallest Hamming distance to a position covered by V must remain V's neighbour.

In both cases, W becomes a new neighbour of the new node, so V notifies the new node which then registers itself at W; W then executes the integration algorithm on its part.

Using this algorithm, the new node gradually is informed about integration champions and prospective neighbours, which it collects in temporary lists. For each node it is notified about, it registers itself with its hypercube position coordinates and its network address. After the integration champion, or respectively, the prospective neighbour, has carried out the integration algorithm, it commits this execution to the new node. After all participating nodes have committed their work, the new node, in return, commits the finalization of the integration to all its partner nodes. This message finalizes the integration algorithm.

3.7 Node Departure

A node wanting to leave the network must execute a *departure protocol* since the topology must always be kept in a consistent state. All nodes must be able to rely on this consistency to carry out the algorithms independently of each other to minimize communications overhead.

The positions that the departing node (call it V) has covered must be handed over to other peers which are neighbours of the departing peer. To select the nodes to transfer the positions, V selects the *buffering dimension* d_{buf}, which is set to the highest dimension where V has neighbours on:

$$d_{buf} = \max_{W \in \mathcal{N}_v} (L(\vec{p}_v, \vec{p}_w))$$

In the binary hypercube, one or two nodes will thus become the buffering nodes of V, since a node may have two neighbours along one dimension. The reason that V selects node(s) along the highest dimension is related to the definition of link dimensionality: neighbours along higher dimensions are considered to be *closer* than neighbours along lower dimensions. The buffering nodes are assembled in the list $\mathcal{L}_{buf}$:

$$W \in \mathcal{L}_{buf} \Leftrightarrow W \in \mathcal{N}_v \wedge L(\vec{p}_v, \vec{p}_w) = d_{buf}$$

where $\mathcal{N}_v$ denotes the set of neighbours of node V. The positions of nodes $W \in \mathcal{L}_{buf}$ are known, but it is also necessary to estimate the cover maps $\vec{c}_w$ because the buffering nodes are not necessarily aware of each other: for dimensions $0 \leq d < d_{buf}$, the buffering nodes' cover maps are identical to that of V, $\vec{c}_v$, because nodes see the same neighbour situation along dimensions lower than the link dimensionality between each other. For dimensions $d > d_{buf}$, the cover map bits are set to 1, and are then modified by pairwise comparing all node positions in $\mathcal{L}_{buf}$:

$$\forall \vec{p}_i, \vec{p}_j \in \mathcal{L}_{buf} : c_i^x, c_j^x = 0 \Leftrightarrow L(\vec{p}_i, \vec{p}_j) = x$$

which reads: the cover maps of nodes I and J are set to 0 on position x, iff the link dimensionality between these two nodes is x.

Using the position vector and the estimated cover map of nodes in $\mathcal{L}_{buf}$, a list of positions $\mathcal{L}_w$ which will be transferred to this node can be calculated: the root position for the list calculation is the position that W will take over:

$$\vec{p}_w^{cover} = (p_w^0, p_w^1, \ldots, p_w^{d_{buf}-1}, \overline{p_w^{d_{buf}}}, p_w^{d_{buf}+1}, \ldots, p_w^{d_{max}-1})$$

Using this root position and the cover map as described above, the list can be calculated: $\mathcal{L}_w = \vec{p}_w^{cover} \times \vec{c}_w$.[3]

As the last step, for each position $\vec{p}_i \in \mathcal{L}_w$ which is transferred to node W, the neighbour positions along dimensions $d < L(\vec{p}_w, \vec{p}_i)$ on which the cover map is 0 are iterated, in order to find the neighbour which is closest to this position, and thus is to be handed over to W.

[3] A computation which again is of complexity $O(2^{d_{max}})$.

After the information for all buffering nodes has been assembled that way, the departing node informs all its buffering nodes of the departure. The buffering nodes update their cover map as required and connect to their new neighbours, then send a confirmation to the departing peer. After the confirmation from all buffering nodes have been collected, the departing peer sends its finalization message to all participants and leaves the network.

3.8 Construction Example

We will demonstrate the ideas stated above on an example. Consider a 3-dimensional hypercube, the maximum number of dimensions of the hypercube, d_{max}, is set to 3. All position vectors and cover maps thus consist of three digits.

Note that we use the terms *node* and *peer* mixed in this paper. The reason is that they are synonymous when talking about one certain hypercube. The difference between the two terms in the context of OPAX is that a peer may act as several nodes in different hypercubes. This does not affect the fact that a peer may act as exactly one node in one hypercube.

In the beginning, only peer 0 populates the network. The first node is always set to be at position 000, and has, as it vacates all positions, a cover map consisting only of 1s (see figure 3.5a).

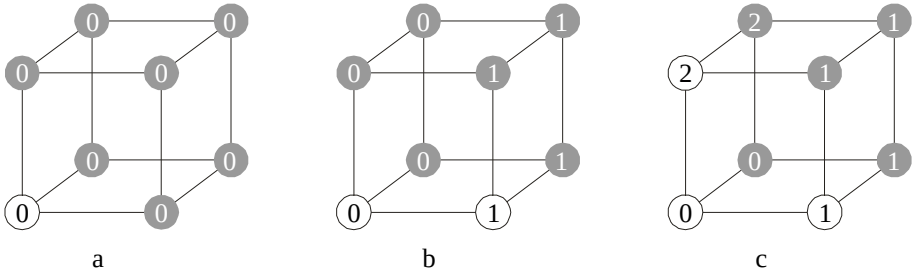

Figure 3.5: Construction example (1)

$$\vec{p_0} = 000$$
$$\vec{c_0} = 111$$

A white bullet indicates the actual position of a peer, gray bullets indicate positions that are covered by a peer in addition to its root position.

Now, peer 1 joins the network. After having contacted peer 0, it will be assigned position 100, as dimension 0 is the lowest dimension where peer 0 has no neighbour. As peer 0 now transfers positions to peer 1, the situation is as shown in figure 3.5b, with the following coordinates and cover maps:

$$\vec{p}_0 = 000 \quad \vec{p}_1 = 100$$
$$\vec{c}_0 = 011 \quad \vec{c}_1 = 011$$

Peer 2 contacts peer 0, which again selects the lowest dimension whereon it has no neighbour, dimension 1, and integrates peer 2 as its new 1-neighbour. It also selects peer 1 as integration champion, because peer 1 will be the 0-neighbour of peer 2. Now, the 2-dimensional cube that was occupied by peer 0 is split to be covered by peer 0 and peer 2 from now on (figure 3.5c).

$$\vec{p}_0 = 000 \quad \vec{p}_1 = 100 \quad \vec{p}_2 = 010$$
$$\vec{c}_0 = 001 \quad \vec{c}_1 = 011 \quad \vec{c}_2 = 001$$

Two more peers, 3 and 4, join the network. For peer 3, peer 0 transfers its last additionally covered position, 001, to be peer 3's position, so both peers 0 and 3 have now a cover map of 000, covering no additional positions. Peer 4 is integrated on position 110, causing peer 1 to give up its position covering along dimension 1, and becoming peer 2's 0-neighbour (figure 3.6a).

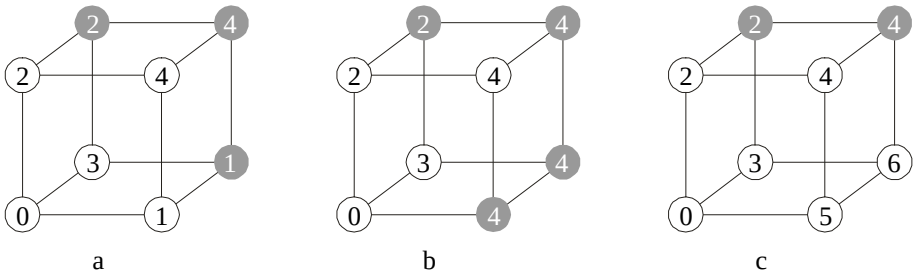

Figure 3.6: Construction example (2)

$$\vec{p}_0 = 000 \quad \vec{p}_1 = 100 \quad \vec{p}_2 = 010 \quad \vec{p}_3 = 001 \quad \vec{p}_4 = 110$$
$$\vec{c}_0 = 000 \quad \vec{c}_1 = 001 \quad \vec{c}_2 = 001 \quad \vec{c}_3 = 000 \quad \vec{c}_4 = 001$$

Imagine peer 1 leaves the network right now: its positions will be taken by peer 4, as it was the nearest neighbour of peer 1; dimension 1 is the buffering dimension for this departure. This situation is depicted in figure 3.6b.

$$\vec{p}_0 = 000 \quad \vec{p}_2 = 010 \quad \vec{p}_3 = 001 \quad \vec{p}_4 = 110$$
$$\vec{c}_0 = 000 \quad \vec{c}_2 = 001 \quad \vec{c}_3 = 000 \quad \vec{c}_4 = 011$$

If now peer 5 would join the network, it will possibly be assigned peer 1's former position, 111. Peer 6 will be integrated on position 110, and both of them are 1-neighbours of peer 4, since this peer covers one position along dimension 2 (figure 3.6c).

$$\begin{array}{llllll} \vec{p}_0 = 000 & \vec{p}_2 = 010 & \vec{p}_3 = 001 & \vec{p}_4 = 110 & \vec{p}_5 = 100 & \vec{p}_6 = 101 \\ \vec{c}_0 = 000 & \vec{c}_2 = 001 & \vec{c}_3 = 000 & \vec{c}_4 = 001 & \vec{c}_5 = 000 & \vec{c}_6 = 000 \end{array}$$

Finally, peers 7 and 8 join the network. Peer 7, which gets peer 4 as integration champion, will be positioned at 111; peer 8, being integrated by peer 2, is set to position 011. The complete hypercube, shown in figure 3.7, accommodates now 8 peers, each of them covering exactly one position. In order to integrate more peers, an additional dimension would have to be opened: all peers would then have to add one digit to their cover map set and set this new digit to 1. Then, 8 more peers could be integrated into the network.

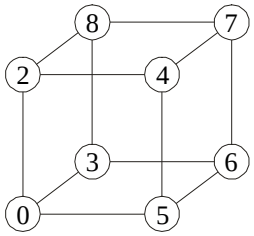

Figure 3.7: Construction example (3)

$$\begin{array}{llllllll} \vec{p}_0 = 000 & \vec{p}_2 = 010 & \vec{p}_3 = 001 & \vec{p}_4 = 110 & \vec{p}_5 = 100 & \vec{p}_6 = 101 & \vec{p}_7 = 111 & \vec{p}_8 = 011 \\ \vec{c}_0 = 000 & \vec{c}_2 = 000 & \vec{c}_3 = 000 & \vec{c}_4 = 000 & \vec{c}_5 = 000 & \vec{c}_6 = 000 & \vec{p}_7 = 000 & \vec{p}_8 = 000 \end{array}$$

Chapter 4

The OPAX Framework

This section describes OPAX, an Open Peer-to-Peer Architecture for XML Message Exchange. OPAX is an open, modular framework which allows the creation of P2P messaging networks. One key feature of OPAX is its ability to use any topology for the construction of the network, which allows the application to optimize the network structure according to its requirements. The requirements, basic architecture, and used message types are depicted.

4.1 Terms

In this section, we describe the main terms that are used in the OPAX framework.

Peer An OPAX peer is an entity which is able to participate an OPAX network. It must have the capability to send and receive OPAX messages. A peer may become member of multiple networks at the same time.

Space In OPAX, networks are called *spaces*. A space is an agglomeration of peers, which are arranged in a well-defined topology. A space can be uniquely identified by its *space URI*. A space is created (*opened*) by one peer, other peers participate the space by *joining*, and later *leaving* (or *departing* from) it.

Message An OPAX message is an XML document which is valid against the OPAX message schema (see appendix A). There exist several types of messages, which are defined in section 4.5.

4.2 Requirements

4.2.1 Openness

To allow the integration of and communication between heterogeneous systems and networks, the framework should be open in a way that allows the implementation of the specification on various platforms.

4.2.2 Extensibility

As requirements to a P2P network may change over time, it is important to allow the framework to grow with its requirements, without the need to reject previous efforts. Implementations of OPAX should be designed taking into consideration this requirement.

4.2.3 Modularity

To satisfy different applications' requirements, the main components of an OPAX instance should be easy to remove, ideally, they should be replaceable or switchable by configuration without the need to restart a running instance. Additionally, a modular concept allows the easy implementation, integration and evaluation of different concepts without the need to rewrite non-affected parts of the system. Parts of the framework that should be replaceable include:

- *Topology management* - To allow the creation, maintenance and management of different network topologies, the appropriate component should be switchable. In OPAX, this component is called *topology manager*. As probing and evolution of peer-to-peer topologies evolves, it should be possible to create and integrate new topology managers.
- *Network communication* - The physical communication layer of the OPAX framework may be exchanged to select the XML message transfer mechanism. Currently, messages are exchanged using a straightforward, TCP-socket-based XML document transferring protocol. As requirements to the framework may increase, HTTP[12, 13]-based protocols or other transfer protocols may be introduced.

It is desired that switching between different implementations of a component should be accomplished using widely accepted standards. One such standard is the usage of *Uniform Resource Identifiers*[11], or *URIs*, to identify implementations in a platform-independent way.

4.2.4 Convenience

As OPAX is a framework for P2P-based applications, it should provide an easy-to-use interface to the application. A software developer creating an application using OPAX should not be

overloaded with internal details of the OPAX implementation. The implementation must provide simple API calls for the following operations:

- *Peer Creation* - The first step is the creation of a peer instance. On creation, the peer starts to listen for incoming OPAX messages, and is ready to perform the operations stated below.
- *Opening / Joining* - To enable the exchange of XML messages with other peers, a peer must open or join an arbitrary number of OPAX spaces.
- *Message Broadcast* - Applications may broadcast messages into a space. The framework must guarantee that all participating peers will receive the broadcast message.
- *Message Receipt* - If other peers broadcast messages, the framework must inform the application about the received messages.
- *Leave* - After a peer has left a space, no communication with other peers within this space is possible.
- *Peer Destruction* - Quitting the application may require some finalization actions for the implementation.

4.2.5 Further Functional Requirements

Synchronization In peer-to-peer networks, it is common behavior that peers are temporarily offline. OPAX provides a mechanism to ensure that messages that have been broadcast during a peer's offline period will be transferred to the peer as soon as it is back online. In OPAX, this mechanism is called *synchronization*. The specification for the synchronization protocol is described in appendix C.

Space Directory Service A *space directory* is a database of known spaces, as well as a dendriform taxonomy of spaces. Using this database, peers may lookup spaces according to criteria like keywords. The database resides on well-known peers whose network address is made public to enable peers that join an OPAX network for the first time to establish contact. The functionalities of a space directory include:

- *space lookup* - allows peers to find spaces using mechanisms like keyword search, or to retrieve a list of spaces that match a given path within a space categorization tree.
- *peers lookup* - allows peers to retrieve addresses of peers that are potential members of a space in order to contact them for sending a join request.
- *topology manager lookup* - allows a peer to retrieve the URI of the topology manager that is used in a space. This is required because different topologies may require different types of join requests.

4.3 Architecture

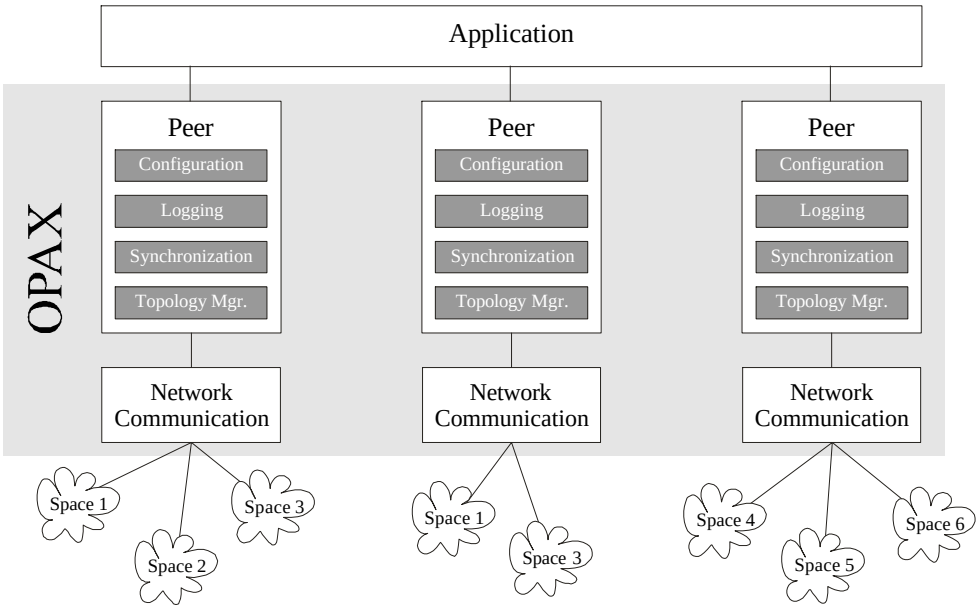

Figure 4.1: The basic architecture of an OPAX instance

Figure 4.1 shows the architecture of an OPAX system. An application may create multiple peers, called *local peers*, running in parallel. Using a local peer, an application may open or join spaces. One local peer may join one space exactly once, but one application is enabled to join a space repeatedly using different local peers.

Each local peer administers several components, each of them operating independently of the corresponding components of other local peers. These components are:

- *Configuration* - Each local peer manages its own set of configuration information. The configuration component includes the local peer configuration (network address, logging) as well as lists of well-known other peers, spaces, and topology managers.
- *Logging* - A local peer owns a logging component, whereby all peer components are enabled to log their activities to enable debugging and the retracing of workflows.
- *Synchronization* - After a peer temporarily left a space, it is the task of this component to synchronize it with remote peers in order to make good for delivery of missed messages.
- *Topology Manager* - As stated above, the topology manager is responsible for carrying out peer integration and departure protocols and forwarding of broadcast messages.

For each local peer, there exists a *network communication component* which provides the functionality to send messages to and receive messages from other peers. Each peer component

may communicate with other peers directly through the network communication component. Incoming messages are buffered, parsed and validated against the message schema. Subsequently, they are delivered to the local peer which then sets adequate actions and dispatches the messages to the appropriate components.

4.4 Protocol

OPAX uses a simple, straightforward, asynchronous message transfer protocol. Each message is transferred with no response. A peer sends a message to another peer in the form of the *header*, followed by a newline and the actual message (which is an XML document). The data is transferred using UTF-8 encoding.

```
OPAX 0.2 541
<Message from= ... </Message>
```

Listing 4.1: Message transmission

The header line contains three fields, separated by one white space:

- *Identifier* - identifies the transmission as standard OPAX transmission. Must be `OPAX`.
- *Protocol version* - OPAX uses a <*major*>.<*minor*> numbering scheme to indicate versions of the protocol. The current (and only) version number which is allowed is `0.2`.
- *Message length* - the length of the following XML document in characters.

The OPAX protocol envisages no handshake or direct reply for message transmission. Transmission control functionality must be provided by the underlying transport layer. Therefore, it is not recommended to use unreliable protocols like UDP[18]. Instead, a protocol that guarantees transmission (like TCP[19] or any protocol on top of it) should be used.

4.5 Message Types

In general, an OPAX message is an XML document that complies with the OPAX message schema, which is in its full text reproduced in appendix A. A simple OPAX message could look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Message from="62.178.0.208:9875"
         space="opax://www.somebody.net/testspace2"
         timestamp="1084781639703"
         uuid="267042b0-a7da-11d8-8fb3-ecfcaeed9c12"
```

```
        xmlns="http://www.mminf.univie.ac.at/opax/message/namespace"
            >
<Ping/>
</Message>
```

Listing 4.2: Exemplary `Ping` message

As you can see, `<Message>` is the root element for all messages, whereas the actual message element, `<Ping>` in this case, is a direct sub-element of `<Message>`. Currently, OPAX defines the following message types:

- Ping
- Application
- ApplicationConfiguration
- Topology
- Synchronize
- Directory
- Unknown

In the following, we will discuss these types in more detail. For each message, the syntax is described, together with a enumeration of the message type's fields and their meanings. As OPAX defines messages in terms of XML documents, there exist fields in the form of *elements* as well as *attributes*. We will use these terms here in the sense they have in XML context.

4.5.1 Common Fields

Table 4.1 shows the fields that are common for all message types. Thus, they are defined to be attributes of the `Message` element.

Name		**Type**	**Notes**
`from`	A	network address	
`space`	A	URI	may be the internal space, see below
`timestamp`	A	long	in milliseconds from Jan. 1, 1970 UTC
`uuid`	A	UUID	time-based
`xmlns`	A	URI	fixed, see below

Table 4.1: Common attributes for all message types

from - contains the network address of the peer that sent the message. This address must be in a format that is recognized by the peers. Currently, only a network address of the type `host:port` is supported.

space - identifies the space that the message is bound to, using its URI[11]. Certain types of messages may require to be sent using the *internal space*, which has the URI `opax://www.mminf.univie.ac.at/opax/space/internal`. OPAX uses the schema prefix `opax` to identify spaces.

timestamp - marks the time that this message was created, expressed in milliseconds, since midnight of January 1, 1970 UTC.

uuid - the 128-bit Universally Unique Identifier (UUID) that is used to identify the message. Each message must be assigned an time-based UUID as specified in section 3.2 in [14].

xmlns - specifies the XML namespace[17] to be used for this message. OPAX defines `http://www.mminf.univie.ac.at/opax/message/namespace` as the XML namespace for its messages.

4.5.2 Ping

The `Ping` message is used to test connectivity between peers. If a peer accepts a `Ping` message, it can be excepted to accept other message types as well. The `space` attribute must be set to the *internal space* (see above). No further attributes are required for the `Ping` message type.

4.5.3 Application

The `Application` message is used to transport application data. The parameters specified in table 4.2 are attributes of the `Application` element (A), or a sub-element of this element (E).

Name		**Type**	**Notes**
`originator`	A	network address	
`routing-control`	A	string	interpretation depends on topology-manager
`valid-from`	A	timestamp	in milliseconds from Jan. 1, 1970 UTC
`valid-to`	A	timestamp	
`Data`	E	N/A	the root element for the application data

Table 4.2: Attributes and elements for `Application` message type

originator - identifies the application message's originator. Note that this must not necessarily be equal to the `from` field of the `Message` element, because the sender of the message may already be a relay peer for the broadcast. Unlike the `from` field, the `originator` field does not change through the forwarding of the message.

routing-control - allows the topology manager to transport routing information. There is no fixed syntax for this field, as the requirements for the routing control data is dependent on the network topology and the used topology manager. Nevertheless, there are some predefined values of `routing-control` that have special meaning.

- `http://www.mminf.univie.ac.at/message/application/routing-control/RC-NOT-ROUTED` identifies that the message should not be routed through the network
- `http://www.mm ... trol/RC-SYNCH-LOCAL` identifies a message that was received through a *local synchronization* and should therefore not be forwarded
- `http://www.mm ... trol/RC-SYNCH-REMOTE` identifies a message that was received through a *remote synchronization* and should therefore not be forwarded

The latter two codes identify messages that are received via the synchronization subsystem. From the peer's point of view, an application message received by virtue of synchronization is not distinguishable from a "regular" broadcast message. The difference is that "synchronized" application messages are not to be forward to other peers, which is indicated by setting the `routing-control` field to the appropriate value.

OPAX implementations must ensure that during the forwarding, the `Message`'s `uuid` and `timestamp` fields may not change in order to preserve the message's original information.

valid-from and *valid-to* - Application messages in OPAX have a validity period which is defined by the values of the `valid-from` and `valid-to` fields. Messages that have expired may be discarded and do not have to be processed furtherly. Also, expired messages should not be delivered to the application.

Data - As application data in OPAX is always valid XML, the `Application` element contains a `Data` sub-element which acts as the root element for the application-dependent elements. Thus, an `Application` message may look as follows[1]:

```
<Application originator="192.168.1.2:9871"
             routing-control="000000"
             valid-from="1089042614437"
             valid-to="1089042754547">
  <Data>
    <aTestMessage attribute="attr-value">
      <OneSubNode/>
      <TwoSubNode/>
    </aTestMessage>
```

[1] **Note:** In the following message listings, the encapsulating `Message` element is omitted to shorten the listings. Nevertheless, in a real message transmission, this element may never be omitted as this would cause the message to be invalid against the OPAX message schema.

```
    </Data>
</Application>
```

Listing 4.3: Exemplary `Application` message

4.5.4 ApplicationConfiguration

Similar to `Application` messages' `Data` field, the application's configuration set is also dependent to the application. OPAX defines `ApplicationConfiguration` messages, which are used to inform an application about the current configuration. These messages consist of a set of properties (pairs of name / value), encapsulated in an element called `Properties`. Usually, a peer receives an `ApplicationConfiguration` message when it joins a space, or when the application's configuration set is updated by any authority. `ApplicationConfiguration` messages are not to be forwarded to other peers. The OPAX specification does not define policies about which peers are allowed to send `ApplicationConfiguration` messages and how to solve contradictions that may occur when multiple `ApplicationConfiguration` messages are received.

OPAX does not define a convention for property `name` fields, nor a syntax for the `value` fields.

```
<ApplicationConfiguration>
    <Properties>
        <Property name="applRunAsService" value="true" />
        <Property name="applRunOnStartup" value="false" />
        <Property name="applMinimize" value="false" />
        <Property name="applServicePort" value="8200" />
    </Properties>
</ApplicationConfiguration>
```

Listing 4.4: Exemplary `ApplicationConfiguration` message

4.5.5 Topology

`Topology` messages are used by the space's topology manager to transfer "administrative" messages. `Topology` messages are required every time a peer joins or leaves the network, or when the topology of the network is changed because of any other reason. `Topology` messages are never directly forwarded to other peers, although they may cause further `Topology` messages to be sent.

```
<Topology type="http://www.mminf.univie.ac.at/opax/message/topology/
    type/hypercube2/ExecuteIntegration">
    <Properties>
        <Property name="DestinationPeer" value="183.112.14.212:9800" />
        <Property name="DestinationPeerCoordinates" value="00010011" />
```

```
    <Property name="DestinationPeerCoverMap" value="00000100" />
    <Property name="MinControlForwardingDim" value="0" />
  </Properties>
</Topology>
```

Listing 4.5: Exemplary `Topology` message

The structure of the `Topology` message is very similar to the one of `ApplicationConfiguration`: mainly, it is a set of properties (pairs of name / value), together with a `type` attribute, identifying the topology manager-dependent function of the message.

Name		**Type**	**Notes**
`type`	A	URI	
`Properties`	E	set of `<Property>` elements	interpretation depends on topology manager

Table 4.3: Attributes and elements for `Topology` message type

type - the `type` fields uniquely identifies the type of a message. As the used message types differ dependent on the topology manager used, an URI is used to prevent naming conflicts.

Note that the `Topology` message type serves as a generic for all messages that the space's topology manager uses. It is left to the specification of a certain topology manager to define the valid sub-types and the properties that are valid for each sub-type. As different topologies require different types of messages, these definitions can not be generalized for OPAX.

4.5.6 Synchronize

OPAX is designed to allow peers that were temporarily out of a space to synchronize themselves with other peers. Synchronization in this context means that a peer belatedly receives `Application` messages that it missed during its off-line period. Usually, synchronization is initiated after the re-joining of a peer is finalized.

Name		**Type**	**Notes**
`type`	A	URI	message sub-type
`UUID`	E	UUID	multiple entries allowed

Table 4.4: Attributes and elements for `Synchronize` message type

For a detailed description of the OPAX synchronization protocol, refer to appendix C.

4.5.7 Directory

The communication between a peer and a *space directory* is processed using a special message type, `Directory`. The meaning of a `Directory` message is defined using the `type` attribute. See appendix D for more details on the directory lookup protocol.

Name		**Type**	**Notes**
`type`	A	URI	message sub-type
`reference-message`	A	UUID	identifies the message to which this message is a reply
`Item`	E	N/A	list of items, meaning depends on `type`

Table 4.5: Attributes and elements for `Directory` message type

4.5.8 Unknown

The `Unknown` message may only be sent in reply to a received message, identifying that the message could not be processed. Note that there is a distinction between *not delivering* and *not processing* a message: The sender of a message can assume that the message was successfully delivered when there was no network error during the transmission. This does not imply that the message could successfully processed. Imagine a peer which is ready to accept message receives an `Application` message for a space that it is not member of. At first, it will accept the message, but it will respond with an `Unknown` message to inform the sender of the message about the problem.

Name		**Type**	**Notes**
`reason`	A	string	the reason of the problem
`text`	A	string	an optional detailed description
`reference-message`	A	UUID	the message that could not be processed

Table 4.6: Attributes and elements for `Unknown` message type

Note that although OPAX does define the data type of the `text` attribute as string, the usage of an URI is recommended to avoid misinterpretation.

Chapter 5

Implementation of Hypercube Topology Types in OPAX

5.1 A Distributed Hypercube

5.1.1 Overview

One of the major goals of this thesis is to evaluate the concept of the hypercube topology for a P2P network. The characteristics of a hypercube, described in section 3, makes this idea interesting for the field of P2P networking. In section 4, we described the basic structure of OPAX as well as the different message types. Now, we describe the *hypercube2* topology manager, a fully distributed hypercube topology upon that a P2P network can be built. The name *hypercube2* is a reverence to the HyperCuP system, an JXTA[24]-based implementation of a hypercube as part of the *edutella* project, which can be found at [25]. The *hypercube2* topology manager basically reproduces the algorithms of the JXTA-based implementation using the mechanisms of OPAX.

The workflow for the main operations is described in section 5.1.5. Similar to the JXTA-based implementation, this implementation firstly does not deal with the case of peer or link failures. Section 5.1.6 describes why such failures may cause the `hypercube2` topology to collapse under certain circumstances. Consequently, section 5.1.7, as a key section of this paper, offers extensions to the `hypercube2` topology that may solve some of the mentioned problems.

The `hypercube2` topology manager is identified within OPAX as `http://www.mminf.univie.ac.at/opax/topology/hypercube2`.

5.1.2 Data Structures

The data structure that is required for each instance of the *hypercube2* topology manager must accommodate the information described in section 3. As stated there, a node V stores its *position vector* $\vec{p_v}$ and its cover map vector $\vec{c_v}$. Usually, this data structure will be implemented using two binary arrays. Additionally, for each of its neighbour nodes W, the node stores a tuple $W = (\vec{p_w}, addr_w)$, where $\vec{p_w}$ denotes the neighbour node's position vector, and $addr_w$ the neighbour node's network address to which it can be sent messages. This data will be represented by a collection of binary arrays (for the node positions) and a collection of a data type representing the network address (depending on the underlying network system).

In addition to the data structures that represent the network topology, each peer needs to hold its current *state*, a value indicating the current situation the peer is in. Depending on this state, a peer may only process certain message types. The states and their valid transitions are depicted in figure 5.1.

5.1.3 Message Types

The *hypercube2* topology manager defines the following sub-types of the `Topology` message.

Message	**Description**
`Join`	Sent by a peer which wants to join the space to a randomly selected peer which potentially is member of the space
`RouteIntegration`	Used to route the integration request through the hypercube until the first integration champion is found
`ConnectNeighbours`	Sent by an integration champion or a prospective neighbour containing topology data for the new peer
`ExecuteIntegration`	Sent by a new peer to its integration partners to initiate the actual integration
`FinalizeIntegration`	Sent by the new peer to its integration or departure partners (integration champions, prospective neighbours) to indicate the completion or abruption of the topology modification
`StartBuffer`	Sent by a peer which wants to depart to its neighbours
`ConfirmBuffer`	Sent by a peer which is a buffering node (i.e. taking over positions from a departing peer) to confirm its buffering

Table 5.1: *hypercube2* message types

More message types are defined by the *peer watch protocol*, described in appendix E.

5.1.4 Peer States

Before we can describe the workflow of the topology manipulation operations, we have to define the states that a peer can be in. In addition to the self-explanatory states of `INTEGRATED` and `OUT`, the distributed trait of the *hypercube2* topology requires the introduction of "transaction" states, i.e. states that are entered temporarily during the execution of topology manipulation operations. *hypercube2* defines the states that are given in table 5.2, the valid state transitions and the events that cause the transitions are depicted in figure 5.1.

Name	Description
`OUT`	the peer is not member of the space and has no connections with any other peer, thus it is not ready to broadcast or receive `Application` messages or to perform topology manipulation operations
`CONTACTING`	the peer is not member of the space, but is trying to establish a connection to a member in order to kick off the integration process
`JOINING`	the peer did successfully contact a peer and is waiting for the integration confirmation
`INTEGRATED`	the peer is member of the space and ready to receive messages and to perform topology manipulation operations
`INTEGRATING`	the peer is member of the space and is currently busy with integrating a new peer within its neighbourhood
`BUFFERING`	the peer is member of the space and is currently busy with removing a departing peer out of its neighbourhood
`DEPARTING`	the peer is member of the space but is departing and thus waiting for the departure confirmation

Table 5.2: *hypercube2* peer states

Note that a peer's state is always related to a certain space. One and the same peer may be `OUT` of space A, `JOINING` space B and `INTEGRATED` into space C simultaneously. We denominate the `INTEGRATING` and `BUFFERING` states as *transitional states* because they are slight modifications of the `IN` state and can only be entered from and left to the `INTEGRATED` state.

5.1.5 Workflow

OPAX provides the `Topology` message type for administrative messages between instances of a topology manager on different peers. The `Topology` message is consciously designed to be flexible for the needs of different topology managers. The *hypercube2* topology manager defines several sub-types of the `Topology` message. This section describes the flow of messages for every transaction that affects the topology of the network.

The following operations are supported by the *hypercube2* topology manager:

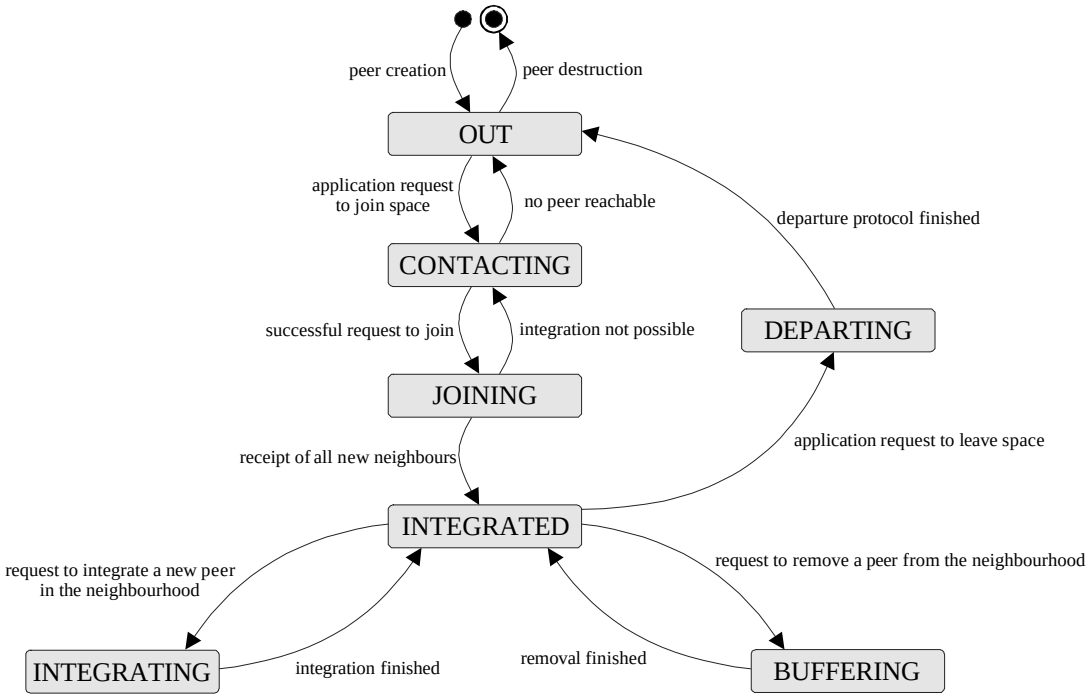

Figure 5.1: Valid state transitions of *hypercube2* peers

- *Peer Integration* - Integration is the key operation for constructing a hypercube-based topology. This section describes the process of contacting already integrated peers, methods to forward the integration request, different approaches of randomization to provide a balanced distribution of load, and finalization messages that commit the integration to the new member peer.
- *Peer Departure* - When a peer wants to leave the network, it must carry out a departure protocol in order to guarantee that the topology is in a consistent state after the departure. This includes the selection of buffering nodes, the assignment of positions to the nodes, and the commitment of the transaction.
- *Message Broadcast* - As the key feature of OPAX is broadcast of messages into the network, this section describes the implementation of the broadcast algorithm described in section 3.3.

Obviously, the implementation of topology operations in a distributed environment is non-trivial. Every operation requires a number of steps, a sequence of messages to be sent, and the collection of acknowledgement messages from neighbour peers. It must be stated again in this context that the this protocol was designed without consideration of peer or link failures. A lost message causes the participating peers to remain in an inconsistent state. As the OPAX protocol in general is asynchronous, this may result in peers no more accepting messages or replying to requests.

In this section, the terms *network* and *space* are used synonymously, as every space has its own topology which is totally independent of other spaces. Also, joining a space may in no case affect the peer's membership to other spaces.

Peer Integration

The integration of a new peer into the space is the fundamental operation of the hypercube construction process. The integration algorithm is performed once for every peer that joins the network. Naturally, the first peer in the network (the peer that *opens* the space) does not have to perform any joining algorithm. It sets its state to `INTEGRATED` and is from now on member of the space.

Figure 5.2 shows the sequence of messages that are sent during a "normal", i.e. error-free integration process.

The integration process can be divided into three phases: *approach*, *integration*, and *finalization*. During these phases, the contacting peer and all participating peers repeatedly change their states and adjust their data sets to the new situation.

Phase 1 - Approach The joining process is initiated by a peer that is not part of the network (i.e. the space). It sets its state to `CONTACTING`, indicating that it is demanding incorporation into a space. It sends a `Join` message to its first contact peer[1] and waits for reply.

As described above, the first contact peer randomly selects an *integration position* and initiates the routing of the integration request. Actually, a `RouteIntegration` message is forwarded, holding all necessary data to carry out the integration process.

Note that if a peer cannot accept a `Join` request, it must reply using an `FinalizeIntegration` message with the `success` field set to `false` in order to indicate that the request was not handled properly. Thus, the joining peer must restart the joining process by searching a different first contact peer. This exception is not depicted in figure 5.2.

Phase 2 - Integration After the joining request reaches its final recipient (the peer that will become the integration champion), this peer carries out the steps according to the algorithm described in section 3.6. During the execution of this algorithm, the integration champion identifies other integration champion and/or prospective neighbours for the new peer. Thus, it will send a `ConnectNeighbours` message to the joining peer, handing it the positions network addresses of these peers, and causing the new peer to enter the `JOINING` state, indicating that the integration is in process.

It is up to the new peer to register itself at the new integration champions and prospective neighbours, in order for them to carry out the integration algorithm on their own and possible

[1] It is out of the scope of this section to define how the peer became aware of its first contact peer. This is the task of the *space directory service*, described in appendix D.

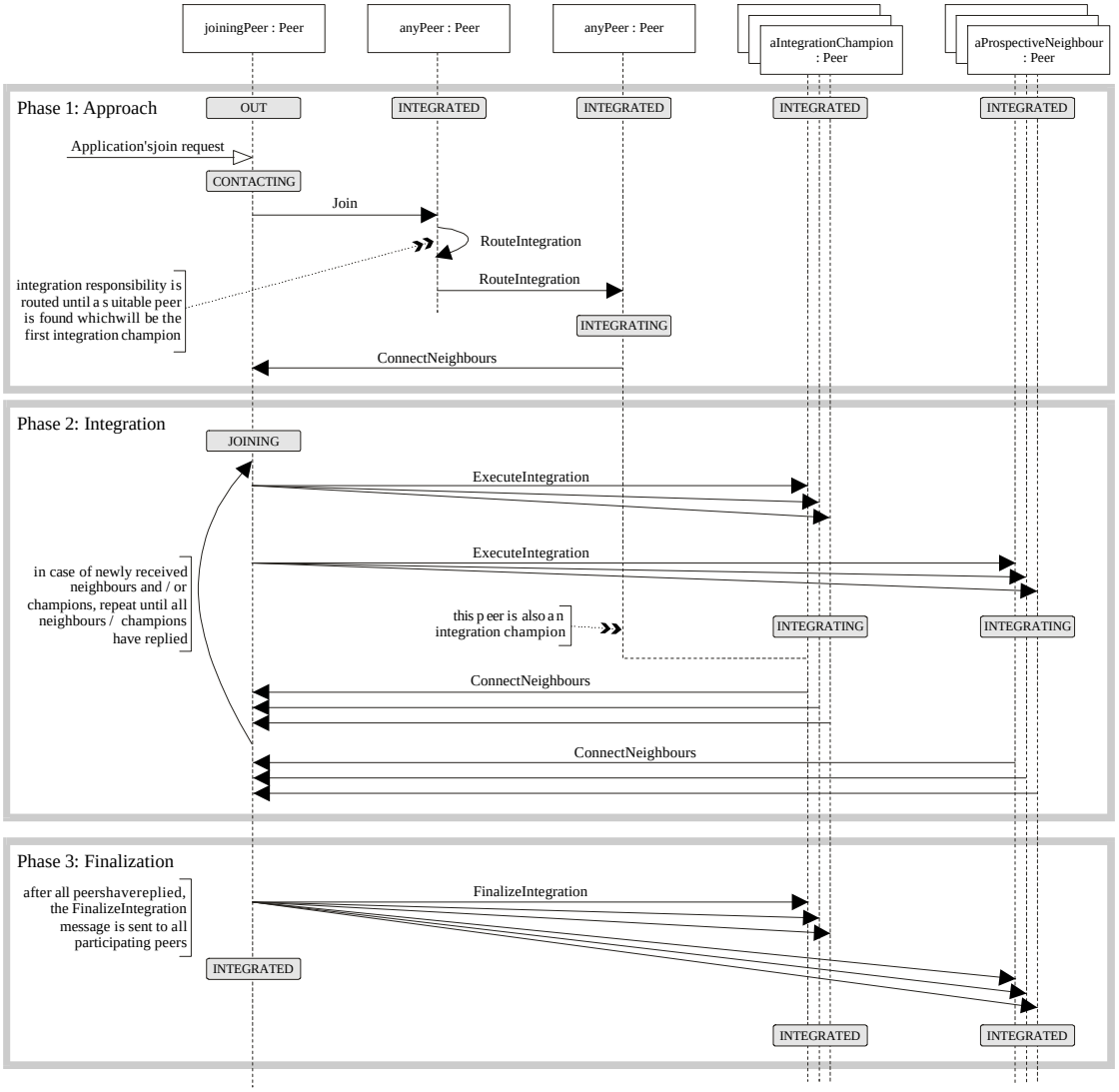

Figure 5.2: Peer joining a *hypercube2* space

sending `ConnectNeighbours` messages to the new peer, too. All participating peers temporarily enter the `INTEGRATING` state to indicate that they currently can accept no other topology manipulation request. This loop is executed until no new integration champions or prospective neighbours are identified, and this marks the end of the integration phase.

Phase 3 - Finalization Now, the new peer has registered itself at all its new neighbours. After it collected the `ConnectNeighbours` replies from all its integration partners, it finalizes the integration by sending a `FinalizeIntegration` message with the `success` field set to `true` to all its partners. This message transmission marks the end of the integration process, and all peers now can return to the "normal" `JOINED` state.

Peer Departure

As stated in section 3.6, peers which want to leave the hypercube must carry out a departure protocol in order to keep the topology in a consistent state. This includes the selection of peers that take over positions that become vacant and the assignment of these positions to their new occupier. Similar to peer integration, if a peer has no neighbours, it can expect to be the last peer in the network and can therefore "leave" it without performing any protocol.

Although the departure protocol is slightly less complex than the integration protocol, we can divide the departure into three phases: *initialization*, *departure*, and *finalization*. The phases, the messages that are exchanged, and the state transitions that peers perform during these phases, are depicted in figure 5.3.

Phase 1 - Initialization The peer that wants to leave the network (it must be in `INTEGRATED` state) selects buffering nodes that will take over positions from the departing peer and selects positions from its set of covered positions that this nodes will cover after the departure. Then the peer sends `StartBuffer` messages to all buffering nodes and sets its state to `DEPARTING`, indicating that is to leave the network.

Phase 2 - Departure A peer that receives a `StartBuffer` message becomes a buffering node, and sets its state to `BUFFERING` in order to indicate this. It checks whether to connect to new neighbours; if not, it directly commits to the departing peer by sending the `ConfirmBuffer` message to this peer. If there are new neighbours for the buffering node, it registers at them by sending a `ConfirmBuffer` message to every one of its new neighbours. It is then up to the new neighbours (which, on their part, change their state to `BUFFERING`) to update their neighbour set and to confirm the departure to the departing peer by sending another `ConfirmBuffer` message.

Phase 3 - Finalization Similar to the peer integration protocol, the departing peer waits for all participating peers to confirm the departure. Upon reception of the appropriate `ConfirmBuffer` messages, the departing peer sends a `FinalizeIntegration` message to all

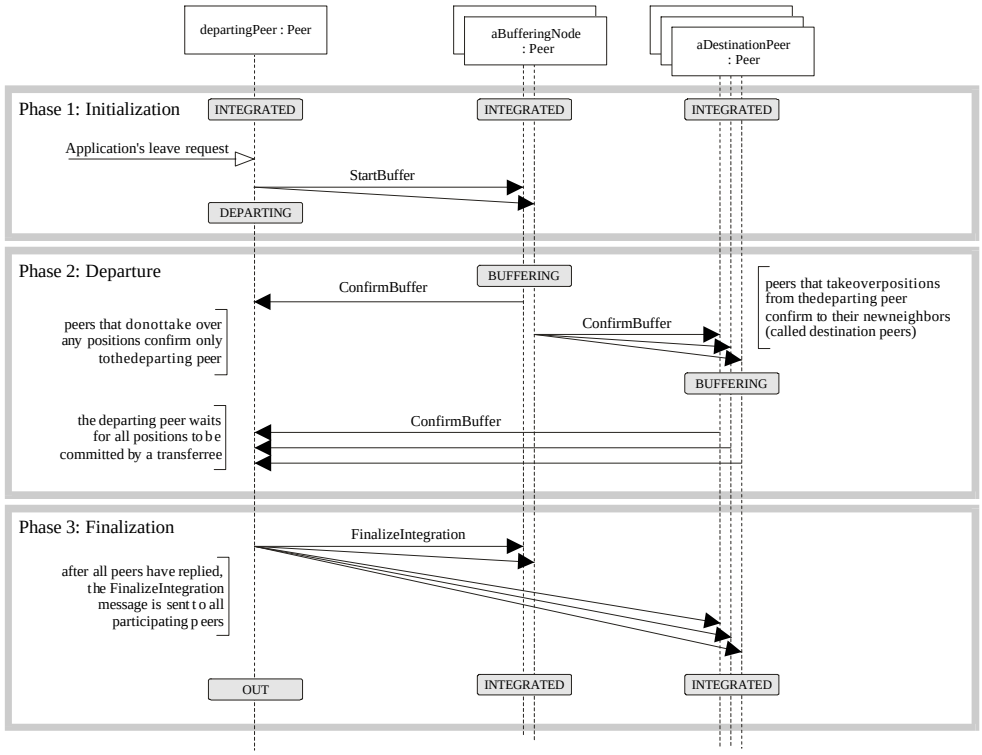

Figure 5.3: Peer leaving a *hypercube2* space

its partner peers, thus finalizing the departure protocol. Then, it sets its state to `OUT`, indicating that it is no more member of the network. All participating peers which are currently in `BUFFERING` state fix their neighbour sets and return to the `INTEGRATED` state.

Message Broadcast

If peers carry out the integration and departure protocol as described above, they can always expect the topology in a state that allows the application of the message broadcast algorithm described in section 3.3. OPAX provides the `routing-control` field of the `Application` message type to allow topology managers to control the forwarding of application messages. The *hypercube2* topology manager populates this field with a binary vector $\vec{r}$ indicating on which positions the messages has already been forwarded.

The first peer, call it V, which is called the *originator* of an application message sends the message to every peer W which is a direct neighbour of V, with a `routing-control` field set to 1 for all dimensions $d \leq L(\vec{p}_v, \vec{p}_w)$ and 0 for all dimensions $d > L(\vec{p}_v, \vec{p}_w)$. Note if there are two nodes that have the same link dimensionality relative to V, the message is sent only to the direct neighbour, ensuring that the message is forwarded only once per dimension.

A peer X that receives an application with a routing control vector $\vec{r}_m$ message must check all of its neighbours: for every neighbour W, it computes the link dimension d_w to that neighbour: $d_w = L(\vec{p}_x, \vec{p}_w)$. If $r_m^d = 0$ and no message has yet been sent along dimension d_w, the peer creates a new routing control vector $\vec{r}_t$ with all field set to 1 for dimensions $d \leq d_w$ and 0 for all dimensions $d > d_w$ and forwards the message to its neighbour W with the `routing-control` field set to $\vec{r}_t$.

Thus, every peer

- forwards a message only to dimensions higher than the dimension on which the message was received
- forwards a message only once per dimension, and
- forwards a message at most once per neighbour peer

gaining a broadcast procedure which is optimal in terms of messages sent - it is always $n-1$, the number of peers in the hypercube minus one.

It must be stated again that this broadcast algorithm is *not* optimal in terms of symmetry. As every node forwards messages only to dimensions the message has not yet been forward, the number of messages a node sends in the course of broadcasting is inversely proportional to its distance to the message's originator node.

5.1.6 Disadvantages and Weaknessess

Although the hypercube topology is a good solution for the problem of broadcasting, several problems arise when an actual implementation has to be made. In this section, we will describe some problems.

Algorithm Complexity

As described in [10], the integration and departure algorithms have a complexity of $O(d_{max})$ in terms of messages sent during the algorithm. However, in terms of local execution, algorithms may have a complexity which make them infeasible for the construction of hypercubes of higher dimensions. Consider the following code:

```
1  procedure computeCoveredPositions(IN BitField rootPos, IN Integer
       currentDigit, IN List numericalCoverMap, OUT List posList )
2  begin
3    if numericalCoverMap.length = 0 then
4      posList.add(rootPos)
5    else
6    begin
7      BitField zeroRootPos = rootPos.copy()
8      zeroRootPos.set( numericalCoverMap.get(currentDigit), 0)
9      BitField oneRootPos = rootPos.copy()
10     oneRootPos.set( numericalCoverMap.get(currentDigit), 1)
11
12     if(currentDigit < numericalCoverMap.length - 1) then
13     begin
14       computeCoveredPositions(zeroRootPos, currentDigit + 1,
           numericalCoverMap, posList)
15       computeCoveredPositions(oneRootPos, currentDigit + 1,
           numericalCoverMap, posList)
16     end
17     else
18     begin
19       posList.add(zeroRootPos)
20       posList.add(oneRootPos)
21     end
22   end
23 end
```

Listing 5.1: Computation of covered positions

The code shown in the listing above depicts the calculation of a node's covered positions, as described in section 3.6.3. Here, the node's cover map is not represented by a binary vector, but by its numerical representation, called *numerical cover map*: a list of integer values, one for

each position in the cover map being set to 1. So, a cover map $\vec{c}_v$ of 011010 would result in a numerical cover map of $num(\vec{c}_v) = \{1, 2, 4\}$.

For every element in the numerical cover map, the algorithm creates two new root positions, one with a 0 and one with a 1 at the cover map position in question. Then, the algorithm recursively iterates through the elements of the numerical cover map. If the recursion reaches its end (by reaching the end of the numerical cover map list, decided by the *if* query in line 12), the two resulting positions are added to the list.

This results in a list of the length $2^{\|\vec{c}_v\|}$, which is a problem especially in the "beginning phase" of the hypercube. Consider the first node that opens the hypercube: as it covers all positions, its cover map consists only of 1s, and thus this list would have a length of $2^{d_{max}}$.

d_{max}, on the other hand, must be selected high enough so that the hypercube can accommodate the desired number of nodes, as this capacity is also delimited by $2^{d_{max}}$.

Peer and Link Failures

In a straightforward implementation, a sudden failure of a peer or a link between two peers may lead to the loss of messages. An abortive delivery of messages may cause peers to remain in a state where it is no more able to continue its algorithms.

Topology Manipulation Consider the case of peer integration, depicted in figure 5.2 on page 41: after having sent `ExecuteIntegration` messages to all its integration champions and prospective neighbours, a joining peer waits for `ConnectNeighbours` messages from all those peers in order to get a confirmation to its request. If one of the integration champion or prospective neighbour peers crashes during or after receipt of the message, it is no more able to send the reply. Thus, the joining peer waits indefinitely, and so do the other participating peers, which stay in their `INTEGRATING` state until they would receive the `FinalizeIntegration` message.

A similar effect may occur during the execution of a peer departure, as shown in figure 5.3 on page 43. Peers may stay in `BUFFERING` state if communication is broken during the second phase of a peer departure.

So, one crashed peer or one faulting link may cause a "deadlock" on many peers in the network. As peers being in a transitional state (`INTEGRATING`, `BUFFERING`) should not accept and/or forward `Application` or `ApplicationConfiguration` messages, this may result in a "black hole" where no application messages can be transferred over. Also, while being in a transitional state, peers should not accept join or departure requests from other peers, as those requests could not be executed on a hypercube being in an inconsistent state.

Message Broadcast In addition to topology inconsistencies, a peer failure may cause `Application` messages not to be reliably forwarded over the whole hypercube. Consider the

situations depicted in figure 5.4: the figure shows the forwarding of a message originated by node 0.

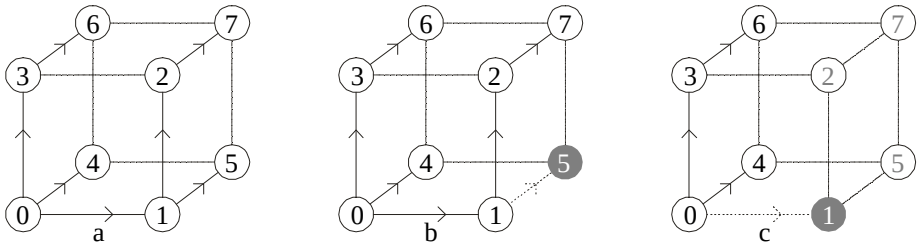

Figure 5.4: Peer or link failure effects to a message broadcast

In figure 5.4a, all nodes and links work, so the broadcast can be executed according to the broadcast algorithm specified above. In figure 5.4b, node 5 fails prior to receiving the message. In this case, this has no direct impact on other peers, as node 5 does not act as relay in this constellation, as it has no neighbours in higher dimensions than the one over which it received the message. In figure 5.4c, the failure of node 1 prior or during the receipt of the message has fatal effects on the message transmission: as nodes 2, 5 and 7 are "behind" node 1 in the message forwarding tree, they are not able to receive the message originated by node 0.

Even worse, nodes 3, 4, and 6, which could stand in for node 1 in order to deliver the message to nodes 2, 5, and 7, have no chance to identify node 1 as faulty, because they do not even know about node 1, as every node in the hypercube only knows about its direct neighbours. Only node 0, as originator of the message, could get aware of node 1's failure and perform eventual operations to fix the situation. Additionally, node 7 has no chance to detect the failure of node 1 as it does not know its physical address since it is no direct neighbour.

Multiple Peer or Link Failures

As we see in the following section, there exist methods to solve the problem of a single peer or link failing up to a certain degree of contentment. However, the problem of multiple peers or link failing simultaneously may lead to an irrecoverable loss of the hypercube topology and, in the worst case, cause the hypercube to break apart, as every peer has only local knowledge about the hypercube topology (i.e. only its direct neighbours). Most solution approaches are only able to cover the failure of one peer and fail by themselves when multiple peers fail simultaneously.

5.1.7 Methods of Resolution

In the previous section, some problems that arise in a distributed hypercube topology were stated. With the following approaches, some of these problems can be solved. Some of these ideas have been implemented and tested in the reference implementation, partly they have only been considered and are to be evaluated in further work.

Dimension Increase

The problem of calculating a peer's covered position list has complexity $O(2^{d_{max}})$. When d_{max}, the number of dimensions the hypercube can have at most, is set to higher numbers in order to accommodate high numbers of peers, the calculation of the list is infeasible.

To address this problem, a *maximum used dimension*, d_{use} is introduced. d_{use} denotes the highest dimension that a peer must consider when calculating its covered positions in order to integrate a new peer. d_{use} is defined to be the maximum of (1) the index of the last 0 in a peer's cover map $\vec{c}_v$, and (2) the index of the last 1 in the peer's position vector $\vec{p}_v$.

The first peer in the network sets d_{use} to -1, as its cover map does not contain any 0, and its coordinate vector does not contain any 1. Every time the peer needs to integrate a new peer, it checks if it has positions along dimensions $d \leq d_{use}$ to select as integration dimension. If this is the case, it integrates the peer as described. If not, it increases the maximum used dimension by 1, thus "opening" a new dimension in the hypercube, as long as $d_{use} < d_{max}$. The dimension d_{use} must be passed along to all the integration champions and prospective neighbours, in order for them to adapt their local d_{use}.

After departure of a peer, each participating peer may calculate its local d_{use} independently by the rule stated above, to possibly "close" unneeded dimensions.

The `computeCoveredPositions` procedure stated above does not have to be changed. Instead, the definition of the numerical cover map $num(\vec{c}_v)$ has to be reformulated to $num(\vec{c}_v, d_{use})$: instead of adding all cover map positions that are set to 1, only those positions $p \leq d_{use}$ are added to the numerical cover map. So a cover map $\vec{c}_v$ of 01001111 and $d_{use} = 3$ would lead to $num(\vec{c}_v) = \{1, 4, 5, 6, 7\}$, but $num(\vec{c}_v, d_{use}) = 1$.

The dimension increase algorithm has been integrated into the OPAX reference implementation in the `hypercube2` topology manager. A field `maxDimension` holds the currently highest dimension d_{use}. This field is added to each message during a topology manipulation, and peers which are affected of a topology change update their `maxDimension` field after every such operation.

Topology Modification Rollback

To prevent the emergence of deadlocks, caused by peers or links failing during an topology modification, peers being in a transitional state (`INTEGRATING` or `BUFFERING`) may fall back to the `INTEGRATED` state - and thus ignore all modifications caused by the integration or departure of a peer - after a certain period of not receiving a commitment message (timeout). This potential problem resolution has the drawback of not being able to cope with peers that reply to a message belatedly, because of overload or other (local) reasons. The length of the timeout has to be selected with care, and has to be equal for all peers. However, we discourage the use of this mechanism because peers may fall back independently or delayed, and thus causing situations where peer states are mixed (transitional and non-transitional).

The rollback mechanism is improved when one peer acts as *transaction principal* which then coordinates an eventual rollback. The selection of one peer as principal is straightforward: when integrating, the new peer is the only one that gets aware of all participating peers and thus can act as principal. When departing, it is up to the departing peer to inform all its neighbour nodes and to commit (or rollback) the departure procedure.

This implicates problems when the principal peer fails during the transaction or when a link failure causes a finalization message not to be transmitted. So it is crucial for the system to use a protocol which guarantees the delivery of messages, like TCP[19].

Currently, the rollback mechanism is not implemented in the OPAX reference implementation.

Stale Peer Ignoring

Another approach, albeit not actually being a solution of the problem of peer failures, is to ignore peers that are not reachable. "Ignoring" in this context means that

1. the peer is not expected to forward `Application` messages,
2. the peer is ignored in topology modification operations (i.e. it is not expected to reply to `Topology` messages, and
3. after a certain period of absence, the peer is dropped from the topology.

Message Broadcast (1.) implies that a modified broadcasting algorithm is required. As the originator of a broadcast may not know about the failure, the peer that attempts to forward the message to the faulty peer must carry out modified broadcast rules: if a peer V forwards a message along dimension d_{fail} to peer W, and this transmission fails, V must ask its neighbour along dimension $d_{fail}+1$ (if it has such), call it X, to forward the message to dimension d_{fail} on behalf of V. X, if it has a neighbour along dimension d_{fail}, starts a *limited broadcast* which it only sends to this neighbour, which, on his part, handles this message as "ordinary" broadcast message and handles it according to the rules of the (extended) broadcast algorithm. This algorithm is to be performed for every broadcast, including the "limited" ones.

We will demonstrate this modified broadcast with an example. Consider figure 5.5a, where peer 0 is the originator of a broadcast, and peer 1 (which is neighbour of peer 0 along dimension 0) fails to accept the message. As peer 0 detects the failure of peer 1, it requests the peer on the next highest dimension (peer 3) to forward the message to dimension 0 on behalf of peer 0 (this request is indicated by a double arrow along the corresponding link). So peer 3 starts a limited broadcast which it sends only to peer 2 along its dimension 0 (figure 5.5b). Peer 2 receives the message along dimension 0 and thus forwards it to dimensions 1 (peer 1) and 2 (peer 7).

Now peer 2 detects that peer 1 fails, and again asks its neighbour along the next highest dimension (2), peer 7, to carry out the broadcast to dimension 1 on behalf of peer 2 (figure

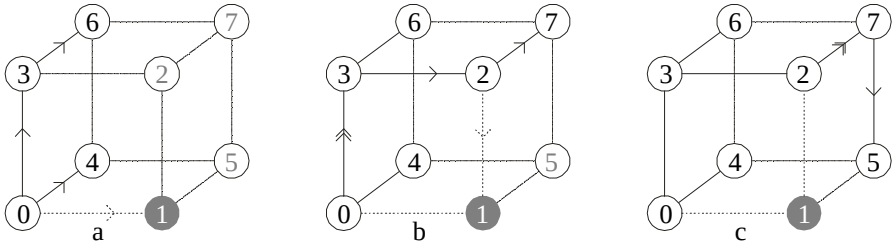

Figure 5.5: Modified broadcast algorithm for peer failures

5.5c). Peer 7, originally receiving the broadcast message along dimension 2 (which is d_{max} in this case) and thus not forwarding it, starts the broadcast on all dimensions higher than 1 and so, finally, sends the message to peer 5. Peer 5, receiving the message along dimension 1 and thus having it to forward to dimension 2, detects that its 2-neighbour, peer 1, is faulty. As d_{max} in this example is 3 (the cube's dimensions are 0, 1, and 2), there exists no neighbour along dimensions higher than 2, and so peer 5 can ignore the fact that peer 1 fails because peer 1 would not act as an relay for the broadcast originated by peer 5.

Figure 5.6, as a further example, shows the modified broadcast algorithm in a 4-dimensional hypercube. O marks the originator of the broadcast, F marks the faulty peer.

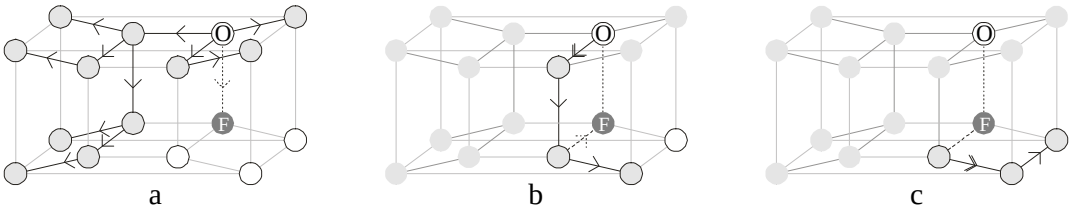

Figure 5.6: Modified broadcast algorithm in a 4-dimensional hypercube

By introducing another slight modification, a broadcast algorithm may eventually cover situations where multiple peers fail. Consider figure 5.7a, which shows a broadcast similar to the one above, with the exception that now two peers are faulty, marked with F_1 and F_2. Similar to before, the originator, after recognizing peer F_1 along $d_{fail} = 1$ as faulty, requests its 2-neighbour (marked A) to limitedly forward the message to dimension 1 (figure 5.7b). Peer A attempts to do so, but its 1-neighbour, peer F_2, is also faulty, thus it sets $d_{fail} = 1$. If it would execute the modified broadcast algorithm stated before, it would now ask its neighbour along dimension $d_{fail} + 1 = 2$ to forward the message. But this would be peer O, the one that A received the forwarding request from. Selecting peer O would result in an endless loop.

Thus, peer A now has to increase dimensions from d_{fail} up to $d_{max} - 1$ until it finds a neighbour to send the forwarding request to. In this case, peer B is the one to be selected, although it did receive the broadcast message yet. Nevertheless, B is requested to forward the message. After B's 1-neighbour, peer C, receives the message, it performs the broadcast forwarding algorithm - forwarding messages to all neighbours along dimensions heigher than the

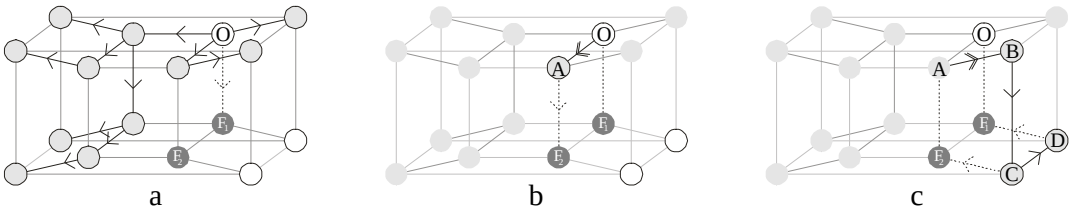

Figure 5.7: Broadcast algorithm with multiple nodes failing

dimension along the message was received. The message was received along dimension 1, so C forwards it to dimension 2 (peer D), and dimension 3 - again detecting that its 3-neighbour, peer F_2, is faulty. As this results in a $d_{fail} = 3 = d_{max} - 1$, there is no need to execute the modified broadcast algorithm: peer F_2 would not act as relay for a message broadcast. The same for peer D, which does not need to take action upon detection of the faultiness of its 3-neighbour, peer F_1.

It is still to be proved if, using this algorithm, any peer failure may be covered, presumed that every non-faulty peer V is still "reachable" in the sense that there is a path from the broadcast originator to V. It is still to be determined how many peers simultaneously failing the algorithm may cope and how faulty peers may distributed over the hypercube.

Topology Manipulation For topology manipulation operations, it can currently not be clearly stated if (temporarily) ignoring a stale peer may affect the overall state of the topology. It seems that ignoring a faulty peer during a topology manipulation operation is possible at first glance: positions that the stale peer would cover are no more assigned to newly arriving peers, and covered positions which would have to be transferred to the stale peer by a departing peer expire. But, with increasing number of topology operations, more and more positions get lost to the stale peer, in the worst case causing the loss of connectivity.

The idea of ignoring stale peers has been tentatively integrated in the reference implementation (`hypercube4` topology manager), but no satisfying results have been achieved up to now.

Peer Watch

The idea of the *peer watch* concept is to designate one substitute, the *monitor peer*, for each peer in the network. It is up to each peer to find a substitute in its neighbourhood, usually the closest peer, after its joining, and every time its local topology changes. This substitute initially gets a copy of the watched peer's local topology information. Every time the local view of the topology changed (because a peer joined or left the network), every peer must again select a neighbour to be its monitor.

The information that each monitor must receive from its monitored peer V consists of

1. its cover map,
2. its list of neighbour positions, and
3. their network addresses.

It is the task of the monitor peer to regularly check if its monitored peer is still reachable, and to initiate a *departure on behalf* if the monitored peer is lost. The protocol of a "departure on behalf" is similar to the one of the regular departure, as depicted in figure 5.3 on page 43, with the difference that all messages from and to the departing peer is sent from and to the monitor peer. So any peer may be removed of the topology if it is not reachable for a certain period of time.

Naturally, this workaround fails if both the monitored and the monitor peer fail simultaneously, or if a monitored peer and one of its neighbours fail, as then it is no longer possible to process the departure protocol as a whole.

In the reference implementation, the monitor concept has successfully been integrated, introducing four new `Topology` message types that control the selection and notification of monitor peers. These messages are described in appendix E. Additionally, respective *hypercube2* message types have been extended a `boolean` field `OnBehalf`, which indicates that the departure process is carried out by a monitor peer instead of its charge. Each peer receiving a message with the `OnBehalf` field set to `true` must reply not to the departing peer as indicated in the message, but to the sender of the message, which is the monitor peer.

Global Shutdown

The simultaneous loss of several peers may cause the hypercube to "break apart" and because of the limited topology knowledge of each peer, it may be impossible to locally reconstruct the adjustment of peers. One possible approach addressing this would be a *global shutdown*: initiated by a *shutdown authority*, and transparent to the application, peers leave the network suddenly (without executing a clean departure protocol) and re-join after a short period. The re-joining is performed according to the integration protocol, resulting in a different but consistent topology.

Problems that arise with this proceeding include:

- *Shutdown Authority Question* - One basic principle of a distributed hypercube is its symmetry: all peers are completely equal. The global shutdown command must be issued by some authority, to avoid overlapping of multiple shutdown commands. The only peer that is prominent is the network's *opener*. So one could implement a global shutdown as follows: the opener of a space is known to all peers, and a peer recognizing a multiple peer failure in its neighbourhood must report this to the opener, which then issues the shutdown command via a broadcast and, after this, re-opens the hypercube immediately.
- *Opener Failure* - Obviously, if the network's opener fails, there exists no authority to issue the global shutdown command. By combining the global shutdown with the peer watch

concept described above, the concept could be saved as long as not both, the opener and its monitor, fail.

- *Shutdown Broadcast Loss* - It is the purpose of the global shutdown command to abandon a broken hypercube topology and create a new one instead. Actually, the shutdown command must be broadcast through the network. As the topology may already be broken or divided into parts, it is not guaranteed that the broadcast receives all participants of the network, resulting in some peers leaving the cube and establishing a new one, and some not.
- *Permanent Shutdown* - Together with the number of peers in the hypercube, the number of multiple peer failures increases. A big network with a certain probability of failing causes peers to issue a global shutdown too often, retarding any productive work.

Implicit Coordinates

Another approach is to use a value which is out of the scope of OPAX as foundation for the peer's coordinate vector, thus making the topology of the hypercube inherent to the physical conditions of the network. Verdy considers the calculation of a hash value of a peer's IP address and port number[42]. Another approach would be to use IPv6[43] as the basis for hypercube addresses.

Fallback Topology

It would be an option for a hypercube network to maintain a fallback topology to which the system could switch in case of a breakdown of the "main" hypercube topology. The problems stated for a global shutdown apply to the fallback topology concept as well. Additional overhead is produced because the fallback topology must be maintained and updated every time a node joins or leaves the network. Consequently, a fallback broadcast mechanism is required because the hypercube broadcast algorithm can not be applied to any topology.

The *Global Shutdown*, *Implicit Coordinates*, and *Fallback Topology* mechanisms have not been integrated into the reference implementation.

Centralized Approach

In a centralized approach, the topology is not maintained by distributed peers, but by a central instance which controls join and departure operations. A centralized hypercube is described in more details in the following section.

5.2 A Centralized Hypercube

5.2.1 Overview

Opposite to the decentralized hypercube, where the topology was managed in a distributed way by the participating peers, in the centralized approach there exists a server which manages the topology. Each peer joining or leaving the network does this by sending the appropriate request message to the server. The server then informs the peer about its position and its neighbours. The transfer of broadcast messages continues to be executed completely distributed, without any need for the server to intervent.

The problems that a symmetrically managed hypercube causes, led to the idea of centralizing the topology administration and a simplification of topology recovery in the case of faulty peers. Although this step means averting from the *decentralized but structured* concept to a *centralized* system, and thus introducing the big disadvantage of a single point of failure, it may be an alternative to a distributed management.

The centralized management of a hypercube implies significant changes in the peers' data structures and procedures: instead of implicitly storing the topology on distributed peers, whereof every one has only limited knowledge about the overall topology (i.e. only its position, cover map, and direct neighbours), the hypercube must explicitly be stored at the manager, and all topology manipulation operations have to be performed on this data structures.

5.2.2 Peer States

As the idea of a centralized hypercube introduces asymmetry into the P2P network, a peer may be started as managing server or as "stupid" peer. In the first case, no peer states are required, as the managing server is not part of an OPAX space. In the latter case, the peer states and transitions are depicted in figure 5.8 and listed in table 5.3:

Name	Description
`OUT`	the peer is not member of the space and has no connections with any other peer, thus it is not ready to broadcast or receive `Application` messages or to perform topology manipulation operations
`JOINING`	the peer did successfully contact the managing server and is receiving neighbours from the server
`INTEGRATED`	the peer is member of the space and ready to receive and forward messages
`DEPARTING`	the peer has sent a departure request to the server and waiting for the departure confirmation

Table 5.3: *hypercube4* peer states

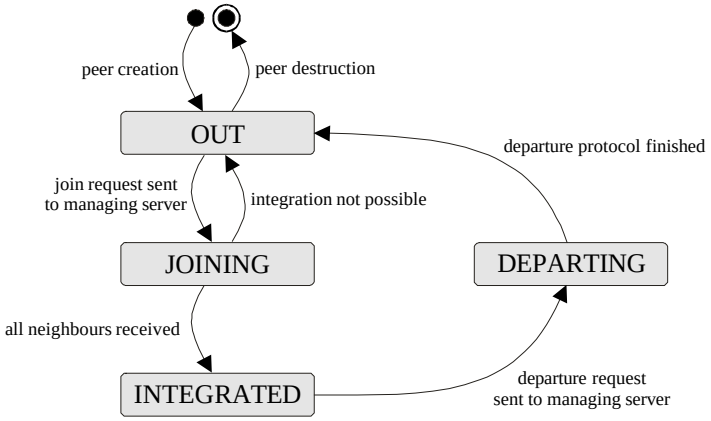

Figure 5.8: State transitions for a centralized hypercube

As you can see, the transitional states `BUFFERING` and `INTEGRATING` were discarded. As "normal" peers are not involved in topology manipulation operations, there is no need to keep these states.

5.2.3 A Hypercube Tree Structure

A data structure to accommodate the topology of a hypercube is required. The following criteria may be formulated for such a data structure:

- The data structure must formally be equal to a hypercube: the mapping function from the topology to the data structure must be one-to-one.
- All aspects of the topology must be represented in the data structure.
- The data structure must adapt to an arbitrary number of dimensions.
- No aspect should be represented multiply.
- Topology manipulation operations should execute with good performance, ideally $O(\log n)$ in a hypercube with at most n peers.
- The data structure should easily be divisible to enable the distribution of the centralized topology to enable load-balancing.

A possible representation for a hypercube is a *tree structure*. Recall the construction of the hypercube, as depicted in figure 3.1 on page 12. For the tree structure, every dimension of the hypercube is represented as one level of the tree, enabling the tree to grow indefinitely. In each node, neighbourship relations between peers are stored for one sub-cube of the hypercube. The construction of such a tree, confronted with the construction of the hypercube, is depicted in

figure 5.9: initially, the hypercube consists of 8 0-dimensional sub-cubes (a). The tree nodes on level 0 represent the four 1-dimensional subcubes of the hypercube (b), the nodes on level 1 represent the two 2-dimensional subcubes (c), and the node on level 3 represents the 3-dimensional cube as a whole (d).

In this data structure, every node consists of a list of mappings, whereof each one (called a "slot") represents one edge of the cube. By traversing the tree from the bottom to the top, one can determine the *coordinates* for each peer by simply concatenating the 0 or 1 along all traversed edges.

This tree structure is also suitable for accommodating hypercubes that are not fully populated: similar to the decentralized approach, peers may occupy multiple positions along dimensions that they have no neighbour on. Figure 5.10 shows a partially populated hypercube and its tree representation.

As we will see below, the idea of centrally constructing the hypercube from its sub-cubes leads to a slightly different construction algorithm and a different alignment of peers if the hypercube is not fully populated. The reason for this is that - contrary to the decentralized approach where neighbours along higher dimensions are considered as closer as neighbours along lower dimensions, and dimensions along which a peer has to cover additional positions are determined at the beginning of the integration - in the centralized approach additional positions to be covered are designated *as late as possible, as soon as necessary.* We will further describe this idea in the workflow section below. The hypercube from figure 5.10 would, if constructed centrally, look like the one in figure 5.11. Note that the positions of the peer remain equal, only the additionally covered positions change.

In figure 5.11, slots marked with an $\mathcal{X}$ are non-populated positions of the hypercube. For every mapping entry, peers that have an $\mathcal{X}$ as opposite are covering the appropriate position. On higher dimensions, fields that are gray indicate that the position is not occupied by a peer, but additionally covered. By taking this into account, one can calculate a peer's *cover map* again by traversing the tree: for every dimension where a peer has an opposite, the cover map position is set to 0, otherwise - if the peer has no opposite - it is set to 1.

The *neighbour set*, i.e. the list of neighbours for a peer V can be determined by traversing all nodes from the top (root) node, and adding each opposite of V in each mapping to the list. In figure 5.11, the neighbours of peer 1 would be peers 4 and 5 along dimension 2, and peers 2 and 3 along dimension 1. Along dimension 0, peer 1 has no neighbour, as it covers the position by itself.

In the following description of the integration and departure workflow, we will denote the ith mapping list on dimension d as $\mathcal{M}_d^i$, and its jth entry with $\mathcal{M}_d^i[j]$.

Peer Integration

A peer which wants to join the network contacts the topology manager server by sending a `Topology` / `Join` message. The server executes the following steps:

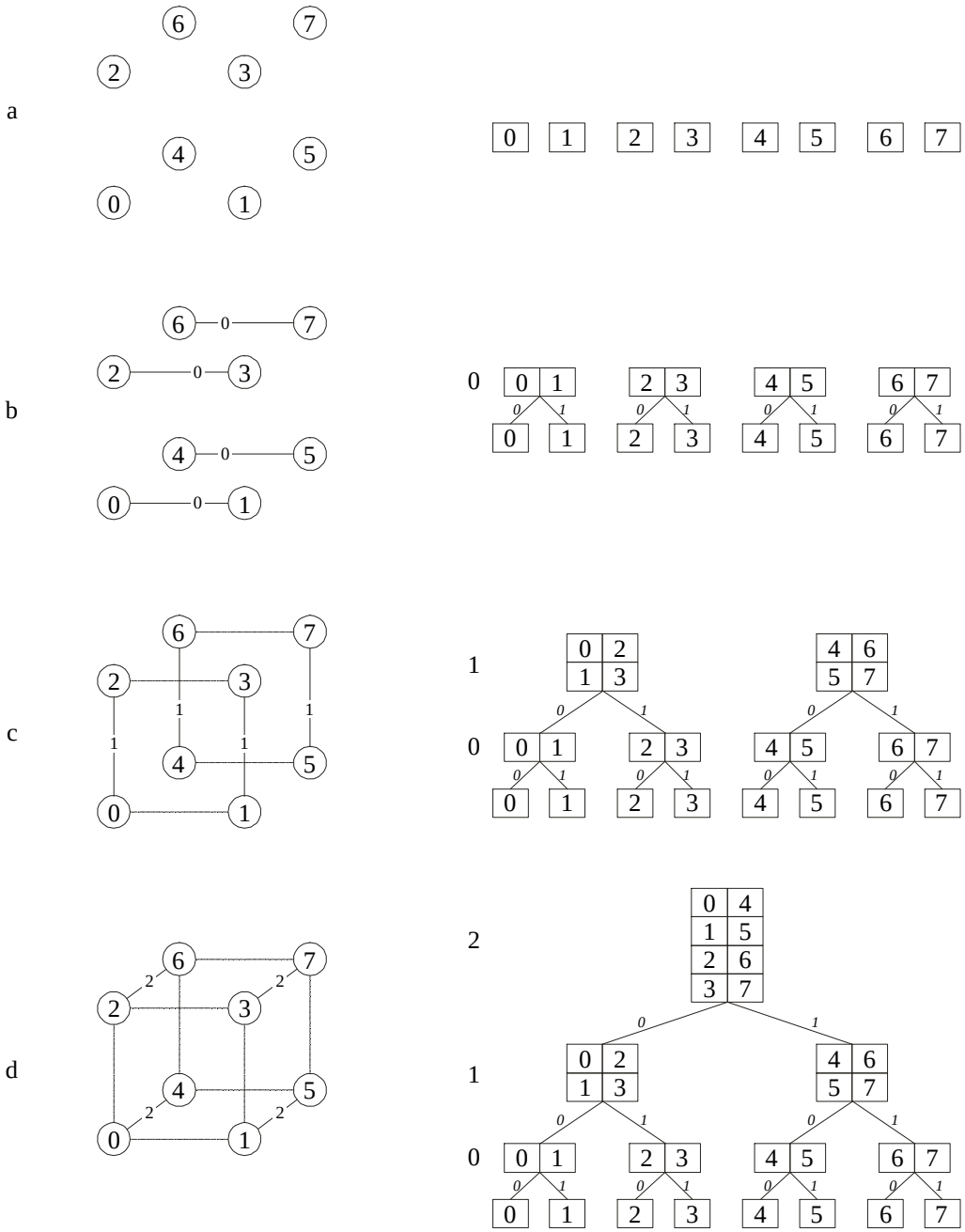

Figure 5.9: Construction of a hypercube's tree representation

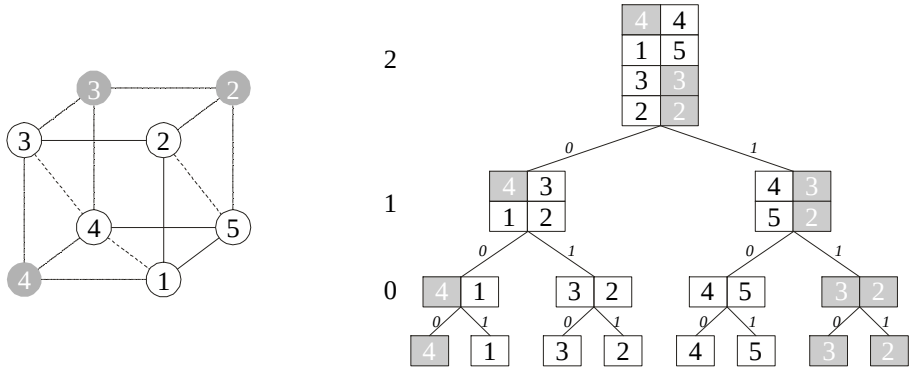

Figure 5.10: A partially populated hypercube and its (hypothetical) tree representation

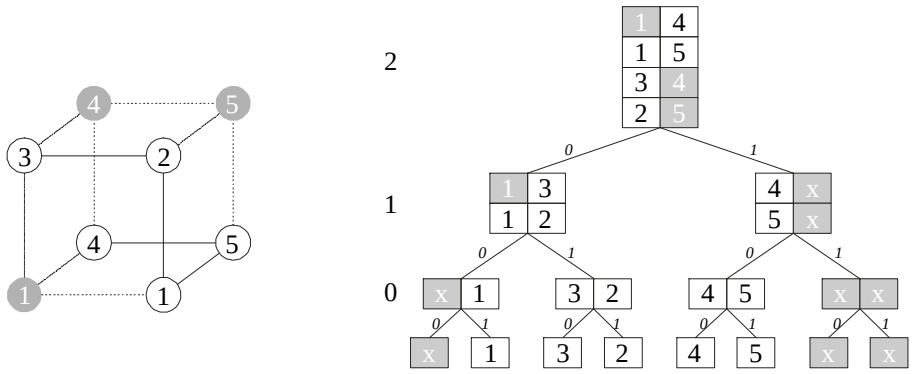

Figure 5.11: A centrally constructed, partially populated hypercube and its tree representation

1. *Position Selection* - The server, aware of a list of vacant positions, selects one position randomly and designates it to the new peer.

2. *Neighbour Collection* - The server now traverses the topology tree from the position that the new peer was assigned (which is a leaf at the bottom of the tree) up. For every dimension, the peer is charted into the mapping table on the appropriate positions. On every dimension, a number of opposites (neighbours) for the peer is identified and sent to the new peer. This step is repeated for each dimension, up to $d_{max} - 1$.

   Formally: To integrate peer V, for every mapping list $\mathcal{M}_d^i$ along the traversing path from the peer's leaf to the root of the tree, the following rules apply:

   - If $\mathcal{M}_d^i$ is empty, all slots on the integration side are occuppied by V, and for all slots, the opposite side is set to $\mathcal{X}$.
   - Otherwise, V is put on positions according to its positions in $\mathcal{M}_{d-1}^{2i-1}$ resp. $\mathcal{M}_{d-1}^{2i}$: for all positions $\mathcal{M}_{d-1}^{2i-1}[j]$ resp. $\mathcal{M}_{d-1}^{2i}[j]$ that are occupied by V, V also occupies the left resp. the right side of $\mathcal{M}_d^i[j]$ if V covers the left side of $\mathcal{M}_{d-1}^{2i-1}[j]$ resp. $\mathcal{M}_{d-1}^{2i}$, or $\mathcal{M}_d^i[2^{d-1} + j]$ if V covers the right side of $\mathcal{M}_{d-1}^{2i-1}[j]$ resp. $\mathcal{M}_{d-1}^{2i}$.

3. *Finalization* - After having traversed all dimensions, the new peer is fully integrated into the hypercube and may start to send and forward `Application` messages.

We will demonstrate the integration process with an example. Consider a centralized hypercube with $d_{max} = 3$. Initially, the cube is empty, there are no peers, thus all slots in the tree are empty. This situation is depicted in figure 5.12a.

Now, peer 0 contacts the server, wishing to be integrated into the cube. The manager randomly selects position 011 and assigns peer 0 to it. Then, the tree is traversed upwards. As there are no other peers in the tree, peer 0 occupies all positions without having any opposite. For instance, on dimension 0, the opposite field for peer 0 is empty. This means that peer 0 must additionally cover a position along dimension 0, now marked with an $\mathcal{X}$. This situation is depicted in figure 5.12b.

Next, peer 1 is integrated. The server selects 010 as the position and again traverses the tree, similar as with peer 0, up to dimension 2, where it is the first time that a peer has an opposite. Both peers 0 and 1 are notified about their new opposites, and the integration is finished (figure 5.12c).

As now peer 2 joins the space, it is designated position 001. Peer 2 will have two neighbours: on dimension 1, it will become neighbour of peer 0, and on dimension 2, it will become neighbour of peer 1 (although they are not *immediate* neighbours, i.e. their Hamming distance is $\neq 1$). After integrating peer 2, the hypercube looks as in figure 5.13. Note that along dimension 1, peer 0 did not have a neighbour (and thus covering additional positions). Now, as peer 2 is peer 0's 1-neighbour, peer 0 did hand over the positions along dimension 1 to peer 2.

As one can see, the cube topology can exactly be matched with the tree representation: for every immediate connection (i.e. Hamming distance = 1) between peers (indicated by a

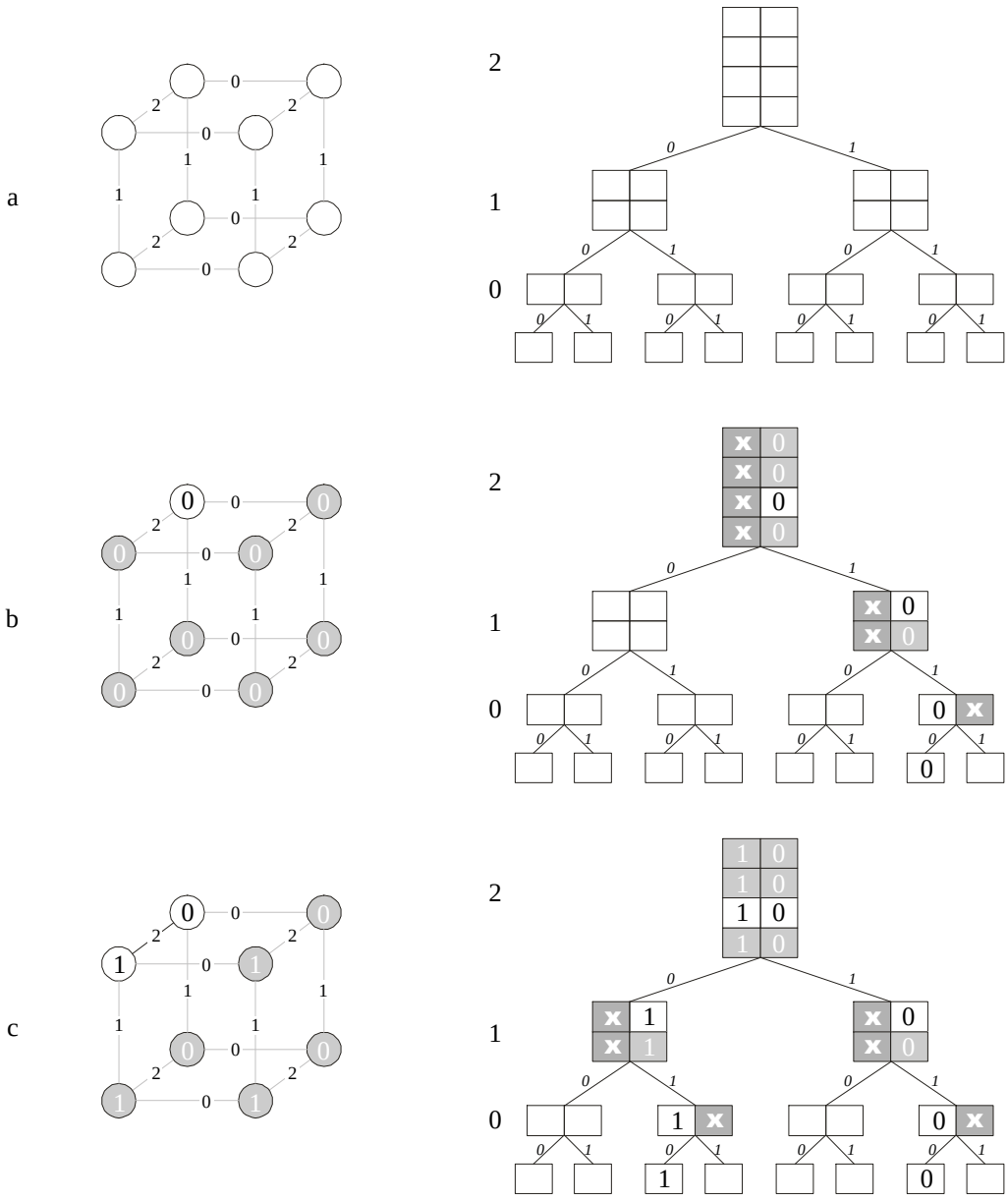

Figure 5.12: Centralized hypercube construction example

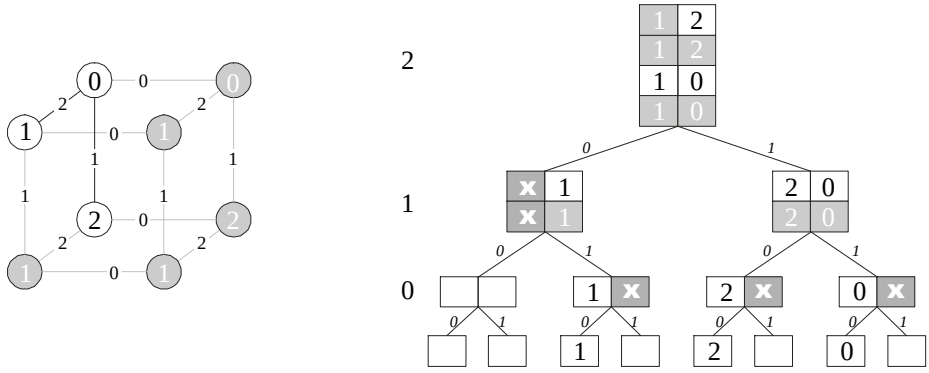

Figure 5.13: Centralized hypercube construction example continued

black edge in the cube), there exists one "white-white" mapping entry between two peers. For every dimension where a peer covers positions, there is a mapping between this peer and an $\mathcal{X}$. For every non-immediate connection (i.e. Hamming distance > 1), there exists a "white-gray" mapping entry between the corresponding peers.

Peer Departure

A peer wishing to leave the network must also send a `Leave` message to the topology manager server. The server removes the peer from its occupied position and traverses the tree upwards. For each dimension, another peer must adopt the positions that the departing peer was covering. For every matching table entry, two cases must be distinguished:

1. There exists one or more peer(s) on the same side of the table: those peers must take over the positions of the departing peer and thus may get new neighbours.
2. There exists no peers on the same side: the position(s) that the departing peer was covering are marked with an $\mathcal{X}$ and thus overtaken by its former opposite peer.

Formally: To remove peer V, for every mapping list $\mathcal{M}_d^i$ along the traversing path from the peer's leaf to the root of the tree, and for every mapping entry $\mathcal{M}_d^i[j]$ which is occupied by V on the left resp. right side, the following rules apply:

1. If there are entries $\mathcal{M}_d^i[k]$, $k \neq j$ occupied by a peer W, $W \neq V$ on the same side as V in $\mathcal{M}_d^i[j]$, W occupies the left resp. right side of $\mathcal{M}_d^i[j]$.
2. If there are no such entries, the left resp. right side of $\mathcal{M}_d^i[j]$ is set to $\mathcal{X}$.

Consider the hypercube depicted in figure 5.14a. When peer 4 wants to leave, it is first removed from its position at the leaf of the tree. On dimension 0, as peer 4 had no opposite, the

corresponding mapping list is cleared in its entirety. On dimension 1, as peer 4 covered positions along dimension 0, peer 4 had two opposites: peers 1 and 3. As there is no other peer on the left side of the mapping list, the positions that were covered by peer 4 are marked with an $\mathcal{X}$, peers 1 and 3 loose their neighbour and, from now on, cover a position along dimension 1. On dimension 2, peer 4 is substituted by peers 1 and 3, as they now cover its positions, thus peer 1 gets one new neighbour (peer 2) and peer 3 gets one new neighbour (peer 5). Now the root of the tree is reached, and the departure is completed. The data structures after the departure is depicted in figure 5.14b.

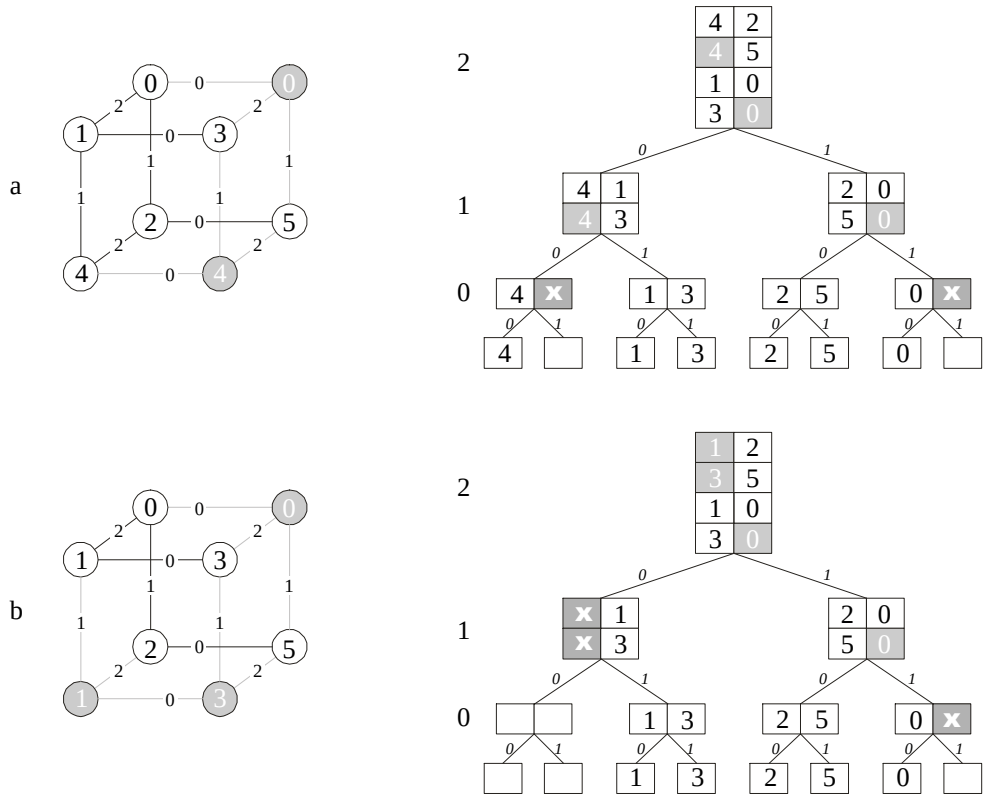

Figure 5.14: Centralized hypercube - peer departure

Broadcast

A message broadcast can be performed exactly as in a distributed hypercube, as the centralized hypercube is also complete and compliant to all demands that the broadcast algorithm makes.

5.2.4 Reduced Hypercube Tree Structure

The tree presented in the previous section has one big disadvantage: its memory requirements. Consider a hypercube with d_{max} dimensions: to represent dimension d, $2^{d_{max}-d}$ tree nodes, each

of them with capacity 2^d, are required: there are $2^{d_{max}-d-1} \cdot 2^d = 2^{d_{max}-d-1+d} = 2^{d_{max}-1}$ links per dimension.

To store the whole hypercube topology, we can derive the number of slots (i.e. the number of edges in the cube) by considering the procedure to construct a hypercube from a point (as depicted in figure 3.1 on page 12). For a 0-dimensional cube, the number of edges is 0. For a 1-dimensional cube, it is 1. For higher dimensions, the number of edges can be calculated as follows: in every step, the hypercube (and all its edges) is doubled, and every node of the "old" hypercube is connected to its counterpart in the "new" hypercube. For the number of edges of a cube of dimension d, $e(d)$, we get (with $d > 1$)

$$\begin{aligned} e(d) &= 2e(d-1) + 2^{d-1} \\ e(d+1) &= 2e(d) + 2^d \end{aligned}$$

The solution of this difference equation leads to

$$\begin{aligned} e(d) &= \sum_{k=0}^{d-1} 2^{d-1-k} \cdot 2^k \\ &= \sum_{k=0}^{d-1} 2^{d-1} \\ &= 2^{d-1} \cdot d \end{aligned}$$

which is also the number of slots required to fully represent the hypercube topology. So, the number of edges grows with $O(2^d)$.

To address this problem, we can construct a *reduced tree structure*: for all dimensions $d \geq d_{red}$, $0 < d_{red} < d_{max}$, the number of slots per tree node is set to 2^d_{red}. Consider figure 5.15, which shows a tree structure for a hypercube with $d_{max} = 4$ and $d_{red} = 1$.

Using such a data structure, we can now calculate the memory requirements as follows: disregarding the lowest dimensions $d < d_{red}$, and setting $c = 2^{d_{red}}$, the number of edges for dimension d with $d > d_{red}$ in a reduced hypercube, $e_{red}(d)$, is

$$\begin{aligned} e_{red}(d) &= 2e_{red}(d-1) + c \\ e_{red}(d+1) &= 2e_{red}(d) + c \end{aligned}$$

which leads to

$$e_{red}(d) = (2^d - 1)c$$

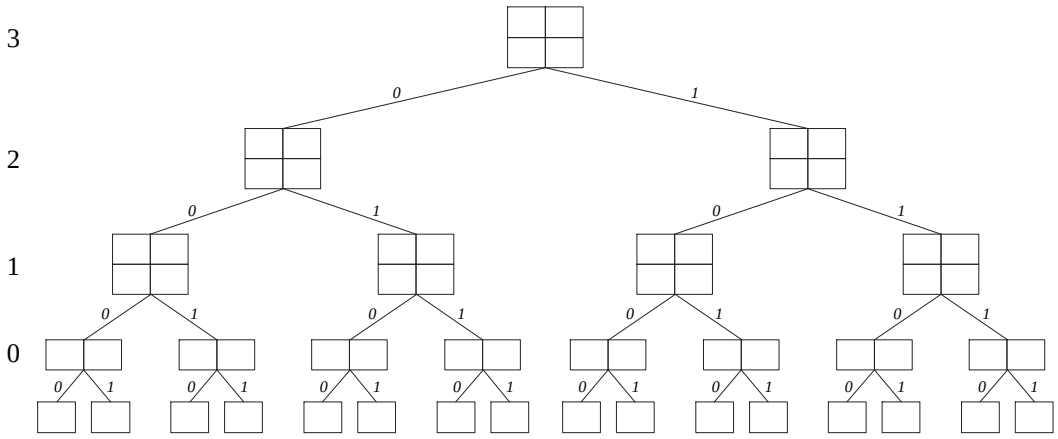

Figure 5.15: Data structure for a reduced tree representation

or, expressed as function of the number of peers, $n = 2^d$,

$$e_{red}(n) = (n-1)c$$

This is a much better consumption than the full hypercube: adding one dimension (i.e. doubling the hypercube's capacity) causes a doubling plus a constant increase of memory requirements, viz. the memory consumption for a cube for n nodes is $O(n)$.

Naturally, this data structure is not suitable to hold a full representation of the hypercube. The reduced capacity of higher-dimensional levels only allows the representation of a *degenerated hypercube*, i.e. a hypercube wherein not all edges are present. Two possible degenerated hypercubes are depicted in figure 5.16.

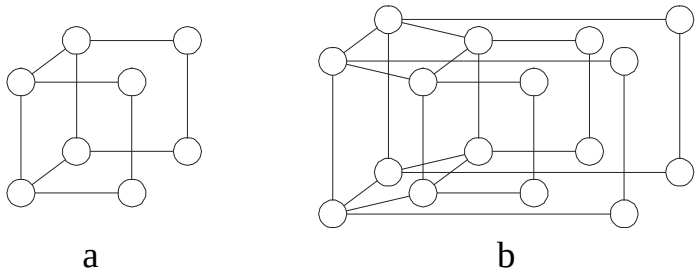

Figure 5.16: Two degenerated hypercubes

Peer Integration

As the reduced tree can not accommodate all edges for dimensions higher than d_{red}, it is required to cancel the up-traversing of a new peer on a dimension between d_{red} and $d_{max} - 1$. Actually,

the dimension on which the propagation stops will depend on the filling degree of a tree node - when it is full, no new neighbours can be assigned to the newly arriving peer. Note that, to ensure a minimal connectivity of the cube, a new peer *must* be up-propagated as long as possible.

A new peer becomes opposite of all peers in a mapping list if the appropriate side of the list is empty. It takes over entries from a peer on "its" side if this peer has multiple opposites.

We can again express this rule formally: to integrate node V, for every mapping list $\mathcal{M}_d^i$ along the path from V's leaf to the tree's root leaf, V is added to $\mathcal{M}_d^i$ if there is capacity in this list. Let the index of the new entry be j. V is added to the left side of $\mathcal{M}_d^i[j]$ if V was added to $\mathcal{M}_{d-1}^{2i-1}$ before, or to the right side if it was added to $\mathcal{M}_{d-1}^{2i}$ before.

The opposite of V in $\mathcal{M}_d^i[j]$ can be determined as follows: if the new entry, $\mathcal{M}_d^i[j]$, is now the only entry in $\mathcal{M}_d^i$, the opposite is set to $\mathcal{X}$. Otherwise, one peer from the side on that V was *not* added is selected to be the opposite of V in entry $\mathcal{M}_d^i[j]$.

Peer Departure

The departure of a peer is critical because the hypercube is no more complete in terms of edges, and thus removing a peer without properly "replacing" the subsequently vanishing edges may cause the hypercube to be split. To prevent this, a departing peer must be substituted by another peer when the departure is propagated through the tree nodes. As soon as there is one candidate for substituting (i.e. as soon as the departing peer has an opposite in one mapping list), the departing peer must be processed in combination with its substitute.

Formally: to remove node V, for every mapping list $\mathcal{M}_d^i$ along the path from V's leaf to the tree's root leaf, and for every mapping entry $\mathcal{M}_d^i[j]$ which is occupied by V on the left resp. right side, the following rules apply:

1. If V's opposite in entry j was $\mathcal{X}$, the entry is removed, and V is propagated up to $\mathcal{M}_{d+1}^{\frac{i}{2}}$ with no substitute.
2. If V was juxtaposed by peer W, two cases must be distinguished:
   - If there are entries $\mathcal{M}_d^i[k]$, $k \neq j$ with a peer U, $U \neq V$ on the left resp. right side, V is replaced by U in entry $\mathcal{M}_d^i[j]$. From all the peers U, one is randomly selected to be propagated up as substitute for V.
   - If there are no such entries, V is removed, leaving W with no opposite, and V's former opposite, W, is propagated up as substitute for V.
3. If, after removing all occurrences of V in $\mathcal{M}_d^i$, any entry which occurs repeatedly is removed.

Example

We will construct a hypercube using the integration and departure rules from above. Figure 5.17a shows the situation when only one peer, peer 0, is in the hypercube. Its randomly selected position is 100, and as there exists no other peer, and all positions of all mapping tables are empty, it is propagated up to the highest dimension, 2.

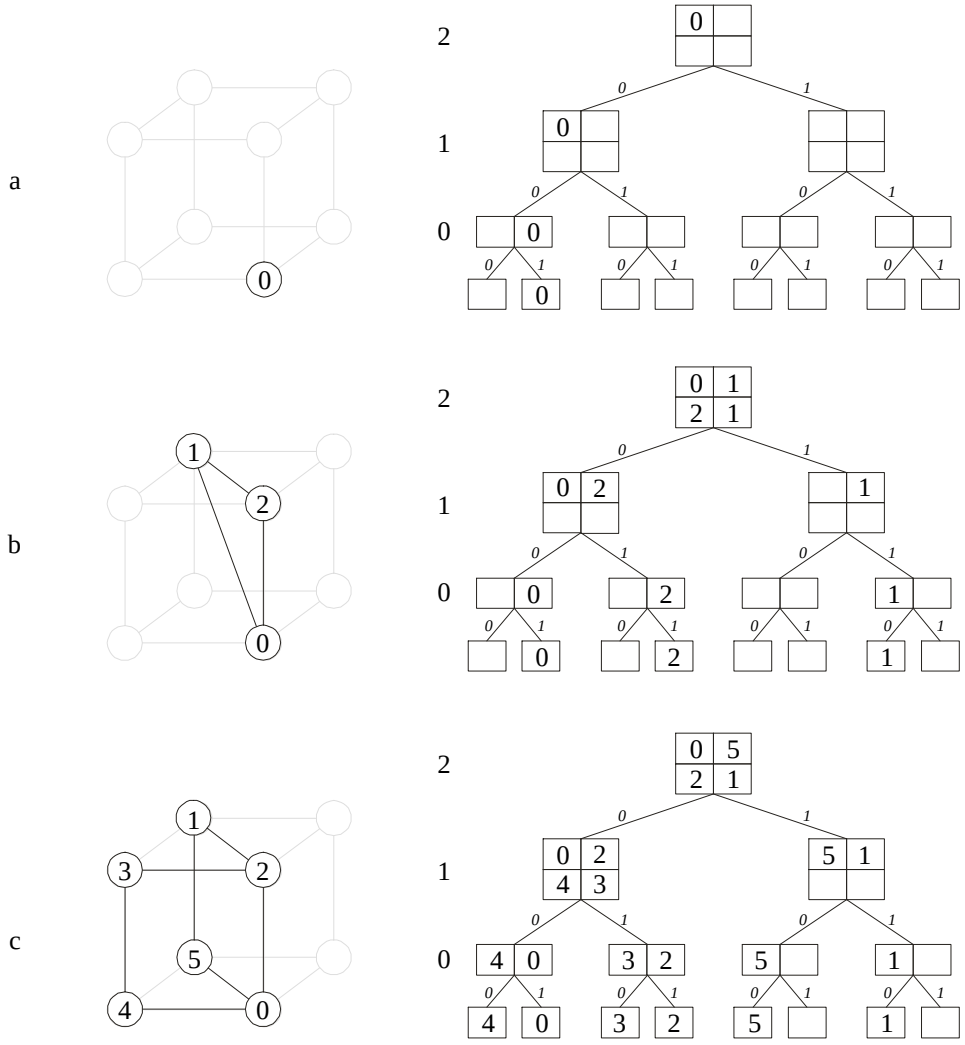

Figure 5.17: Reduced hypercube construction example

When peer 1 arrives, its position is randomly set to 011. It is propagated up and becomes 2-neighbour of peer 0. This is the first abnormality in the hypercube since the edge between coordinates 100 and 011 is a diagonal, as $(100 \oplus 011) = 0$.

The same procedure takes place with peer 2, which is put on position 110 and becomes 1-neighbour of peer 0 and 2-neighbour of peer 1, whose entry in the mapping table of dimension

2 is doubled in order to become opposite of both peers 0 and 2. This situation is depicted in figure 5.17b.

Now three more peers join the space, peer 3 on 010 (causing peer 0 to cover two entries on the left table on dimension 1), peer 4 on 000 (peer 4 takes over one position of the two that were covered by peer 0 on dimension 1), and peer 5 on 001 (taking over one position on the mapping list on dimension 2 from peer 1). After the integration of the three peers, the degenerated hypercube looks as in figure 5.17c.

Broadcast

As a set of edges is missing in the hypercube, and there may exist non-orthogonal edges, the broadcast mechanism must be reformulated as follows:

- The originator of a message sends the message to all its neighbours.
- Any peer V receiving an `Application` message from peer W performs the following steps on it:
  1. If it has received and processed this message already, it does no further processing and discards it.
  2. Otherwise:
     - If the message was received through an orthogonal link (this can be calculated by calculating the Hamming distance between the coordinate vectors of V and W, $\vec{p}_v$ and $\vec{p}_w$ - a link is orthogonal if $H(\vec{p}_v, \vec{p}_w) = 1$), the message is forwarded "as usual" to all dimensions higher than the link dimensionality $L(\vec{p}_v, \vec{p}_w)$.
     - If the message was received through an non-orthogonal link, the peer initiates a *limited broadcast* by forwarding the message to all its neighbours except W.

This broadcast is no more efficient in terms of messages sent, as the last case (the limited broadcast) causes more messages than required to be sent. Nevertheless, this increase of messages sent makes up for the decrease in edges of the cube.

Search

Similar to a distributed hypercube, the search can be expressed as a broadcast and a subsequent sending of results to the originator. The non-optimal broadcast presented above may be modified, if it is not required that every node in the network receives the broadcast, hence introducing a "limited" search, by ignoring the fact that there are non-orthogonal links and forward the search broadcast according to the rules of a complete hypercube.

5.2.5 Load Sharing

The centralized management of the hypercube's topology causes considerable workload on the server, especially for cubes with higher dimensions and increasing peer numbers. The tree representation of the hypercube makes it easy to distribute this workload onto several servers without reducing the functionality of the system.

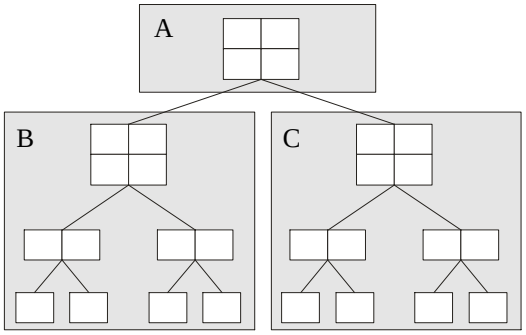

Figure 5.18: Distributing a centralized hypercube topology

Figure 5.18 shows how a reduced hypercube tree can be distributed amongst three servers, A, B, and C, to achieve equal disposition of workload. For this reduced tree, d_{red} is set to 1: all tree nodes on dimensions $d \geq 1$ have a capacity of $2^{d_{red}} = 2^1 = 2$. Assumed an equal distribution of peers to the position range, if the network has a capacity of n, both peers B and C have to process $\frac{n}{2}$ integrations, and so does A, as after $\frac{n}{2}$ its mapping table is full: it may process only $\frac{n}{2}$ requests from B and $\frac{n}{2}$ requests from C.

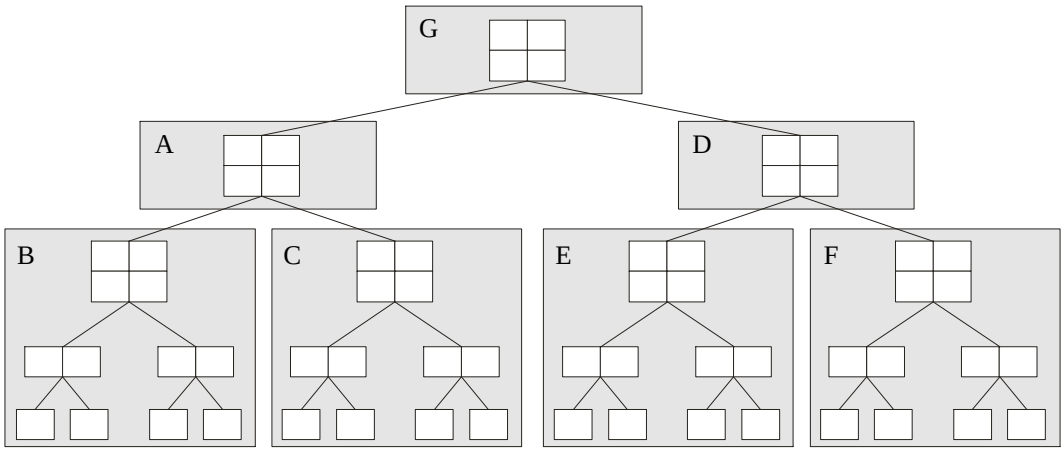

Figure 5.19: The doubled hypercube tree

In figure 5.19, the network is doubled to accommodate $2n$ peers. Servers B, C, E, and F have the responsibility to integrate $\frac{2n}{4} = \frac{n}{2}$ peers each, and so do the servers A, D, and G: from every one of their two "children servers", they receive $\frac{n}{4}$ requests, which totals $\frac{n}{2}$.

Communication between the servers is of low effort: each subsystem only must propagate joining or departing peer to its "parent" server. No communication between servers on the same layer is required; nor communication from a server on a higher level to a server on a lower level.

This concept is scalable if the hypercube has to be extended to accommodate higher peer numbers: the system can be doubled, and a new level can be set to be the parent of the two old systems.

5.2.6 Fault Tolerance

Centralized Manager

As the centralized topology manager is a "normal" server, well-known mechanisms can be applied to provide redundancy, recovery, and backup strategies.

Peers

A failing peer will be detected by its neighbours when they try to forward a broadcast message to it. In this case, neighbours must hold the message to be forwarded and notify the centralized topology manager of the failed peer, which then will remove it from the topology. After the topology has been transferred back to a consistent state, the broadcast may continue.

Chapter 6

JIO - Java-based Implementation of OPAX

6.1 Introduction

This section presents an implementation of OPAX in the Java[44] programming language. This implementation has been made with the goals of OPAX stated in section 1.2. Its purpose is to verify and further examine the concepts and ideas that arose in the context of applying using a hypercube as the topology for a P2P network. Although the hypercube topology is the main field of interest for this paper, some aspects of the implementation which are specific to Java are illustrated. In the last section, we present some classes which may be of interest for usage in other context, as they are not directly related to the fields of OPAX or the hypercube.

This implementation has been developed using the Eclipse IDE[45], version 2.1, on a Windows XP machine, and was compiled and tested using the Sun Java SDK, Version 1.4.2_03.

6.2 Required Libraries

JIO, in the available version, uses several open-source libraries. Those libraries have to be installed according to their installation guide prior to using JIO.

- *Java UUID Generator*[47] - by Tatu Saloranta, enables the creation and manipulation of UUID[14] objects
- *Xerces2 XML Parser*[48] - a XML[15] parser with support for XML Schema[16]
- *JGoodies Forms*[49] and *JGoodies Looks*[50] - by Karsten Lentzsch, libraries to for GUI layout

6.3 Application Programming Interface

6.3.1 API Overview

This section gives a short introduction into the JIO API. An application wishing to use OPAX must do the following:

1. *Create a configurator.* The configurator is a class that provides configuration data to the OPAX system. Any configurator must implement the `at.ac.univie.mminf.opax.config.Configurator` interface. JIO already provides `XMLConfigurator`, which uses XML configuration files to configure the OPAX framework.

   For more information on how to use the `Configurator` interface, see section 6.3.4.

2. *Import JIO classes.* The following classes are required for the application to be imported:

   ```
   import at.ac.univie.mminf.opax.Application;
   import at.ac.univie.mminf.opax.NetworkAddress;
   import at.ac.univie.mminf.opax.Peer;
   import at.ac.univie.mminf.opax.message.ApplicationMessage;
   ```

   If the `XMLConfigurator` is to be used, the following line has to be added, too:

   ```
   import at.ac.univie.mminf.opax.config.xml.XMLConfigurator;
   ```

3. *Create an* `Application` *instance.* JIO uses callback methods to notify the application of events that are caused by the OPAX network. The callback methods that an application must provide are defined in the `at.ac.univie.mminf.opax.Application` interface. The application must provide one implementation of this interface to JIO. For more information on the `Application` interface, see section 6.3.3.

4. *Create a* `Peer` *instance.* The application communicates to the OPAX network using an instance of `at.ac.univie.mminf.opax.Peer`. To create a peer instance, use the following code:

   ```
   try {
       Peer aPeer = new Peer ( configurator, NetworkAddress.
           getNetworkAddress("62.178.0.208:9870"), null );
   }
   catch ( Exception e ) { /* do exception handling here */ }
   ```

   The parameters for the constructor of `Peer` are:

   - `Configurator configurator` - the configurator to be used for this peer
   - `NetworkAddress address` - the network address that this peer should use. To obtain an instance of `NetworkAddress` from its textual representation, use the static method `NetworkAddress.getNetworkAddress(String text)`. Note that this address must be bound to a local network interface, otherwise the peer can not be created.

- `KeyPair keys` - the public/private key pair to be used for this peer. Currently, no security features are implemented, so this parameter should be set to `null`.

From now on, the peer is alive, i.e. it is listening for incoming messages. As it is not yet member of any space, all messages (except `Ping` messages) will be ignored.

As an application may create several peers, it is possible to repeat this step as often as desired. Note that one instance of `Configurator` may serve many `Peer` instances. Note also that there may not be two or more peers with the same network address.

5. *Open a space.* To open a space, you must prepare two objects prior to calling the `open()` method:

   - Create an instance of `java.net.URI` containing the URI of the space to open:

   ```
   URI spaceUri = new URI( "opax://www.mminf.univie.ac.at/space/
       aTestSpace" );
   ```

   - Create and populate an instance of `java.util.Properties` containing the application configuration set for the space:

   ```
   Properties prop = new Properties ();
   prop.setProperty("space-owner", "University␣of␣Vienna,␣
       Department␣of␣Computer␣Science␣and␣Business␣Informatics,␣
       Multimedia␣Information␣Systems␣Group");
   prop.setProperty("space-owner-address", "http://www.mminf.
       univie.ac.at");
   prop.setProperty("space-creator", "Bernhard␣Schandl␣<bes@aon.
       at>");
   ```

   Now, the space can be opened:

   ```
   aPeer.open(spaceUri, this, prop );
   ```

   assuming that this call is made within a class that implements the `Application` interface. If not, an instance of such a class must be specified as second parameter.

6. *Join a space.* Alternatively, an existing space can be joined:

   ```
   aPeer.join(spaceUri, this, true );
   ```

   The third parameter, `boolean lookup`, indicates whether the peer should lookup potential members of the space to contact on a space directory or not. If yes, the configurator must supply the network address of at least one space directory server. If no, the configurator must supply at least one peer which may be member of the space to send the join request to.

   As a peer may open and/or join multiple spaces, steps 5 and 6 may be executed as often as desired. Note that one peer may open or join one space (identified by its `URI`) only once.

7. *Broadcast messages into the space.* After a spaces has been opened or joined, the application may broadcast XML documents into this space. OPAX uses the *Document Object Model*[46] to represent XML documents, so the document to be sent must be stored in an instance of the `org.w3c.dom.Document` interface. Two instances of `java.util.Date`, which mark the beginning and the end of the message's validity period, have to be supplied.

```
org.w3c.dom.Document message = ... // create the document
java.util.Date fromDate = new Date (); // now
java.util.Date toDate   = fromDate + ( 20 * 60 * 1000 ); // 20
    minutes
aPeer.broadcast(spaceUri, fromDate, toDate, message);
```

8. *Receive messages from the space.* The receipt of a message is passed to the application by the `messageReceived()` method of the `Application` interface. See section 6.3.3 for more details about this interface.

9. *Shutdown the peer.* A clean shutdown procedure is required mainly because of two reasons: (1) As spaces may be organized in distributed topologies, other peers have to be notified that a peer shutdown (and thus leaves the space) in order to keep the topology data in a consistent state. (2) As JIO works with multiple threads running during a peer's lifetime, those threads have to be quit orderly.

   To cleanly finish working in the OPAX network, shutdown all instances of `Peer` as follows:

```
aPeer.shutdown();
```

   Any spaces that the peer is still member of are left during the peer shutdown. This method blocks until all spaces have been left.

To get an overview of the processing sequence in as a whole, refer to the source code of the demo application, described in the following section.

6.3.2 OPAX Demo Application

The class `at.ac.univie.mminf.opax.demo.Demo` is a demonstration of how to use OPAX within an application. Using the appropriate GUI elements, the user may create and destroy peers, open, join, or leave spaces, and broadcast a test message into a space. Furthermore, the application displays internal information on the peer instances and their sub-components.

The Demo application uses the XML configurator described below. Figure 6.1 shows a screen shot of the demo application.

The OPAX Demo Application GUI is divided into four groups:

- *Display* - Use the "Refresh display" button to update all display elements.

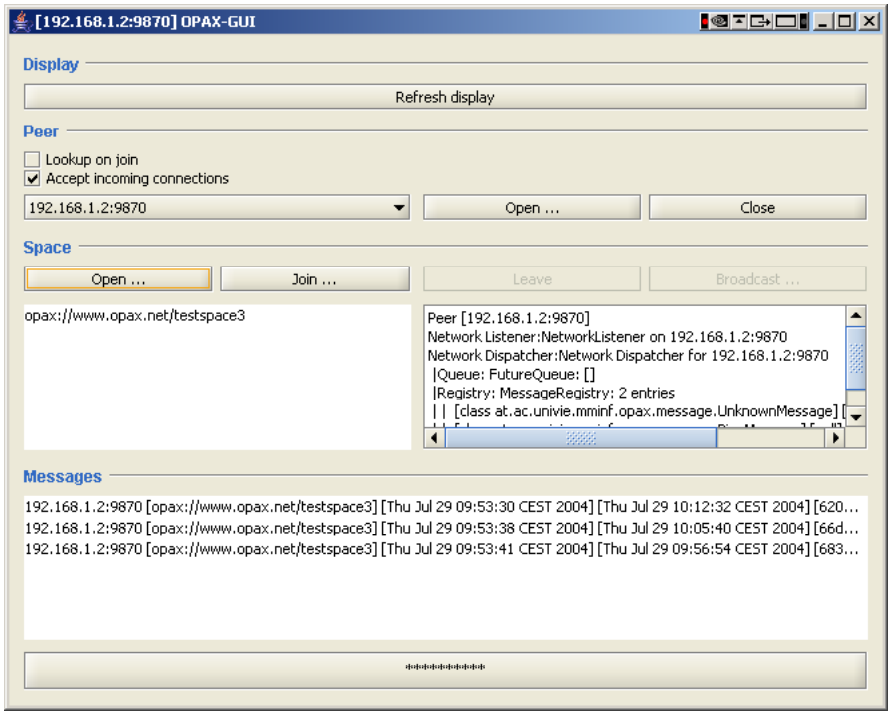

Figure 6.1: OPAX Demo Application

- *Peer* - Use the "Lookup on join" checkbox to enable/disable the lookup of space information on a space directory server. Use the "Accept incoming connections" to temporarily disconnect the peer from the network (only for debugging purposes). Use the "Open" and "Close" buttons to create and destroy peer instances, which then can be selected from the drop-down list.
- *Space* - After selecting a peer from the drop-down list, use the "Open" or "Join" buttons to make the peer member of a space. All spaces that the peer is member of are displayed in the list below these buttons. Selecting one space enables the "Leave" and "Broadcast" buttons, and information about the space, like the peer's address and its neighbours, is displayed in the field to the right of the list.
- *Messages* - In the list under "Messages", all application messages that have been received from the space are displayed, together with their validity period, and (not visible) their UUID.

6.3.3 Callback Interface

JIO uses the `at.ac.univie.mminf.opax.Application` interface to forward events to the application. An application using OPAX must provide an instance of `Application` to the OPAX system. The following methods are defined in this interface:

- `public void applicationMessageReceived ( Peer peer, ApplicationMessage msg );`

  Called every time an broadcast message is received from a space. `peer` identifies the peer instance that received the message. More information about the message, like the space wherein it was broadcast, and its validity period, can be obtained from the `msg` object.

- `public void applicationConfigurationUpdate ( Peer peer, URI spaceUri, Properties configuration );`

  Called when a peer `peer` receives an update of the application configuration set for the space identified by `spaceUri`. The set of properties is passed to the application by the `configuration` parameter.

- `public void spaceJoined ( Peer peer, URI spaceUri );`
  `public void spaceLeft ( Peer peer, URI spaceUri );`

  These methods notify the application that peer `peer` successfully joined resp. left the space identified by `spaceUri`. Note that these methods are called not only due activities initiated by the application (e.g. by calling the `Peer.leave()` method), but also because of "external" events, e.g. the space is shut down by its authority and thus forcing the peer to leave the space.

6.3.4 Configurator

It is the purpose of the `Configurator` interface to provide a pluggable configuration infrastructure for an application using OPAX. As OPAX requires extensive configuration data, the configurator concept provides the feasibility to fetch configuration data from any source, like files, databases, or remote servers. The `Configurator` interface contains methods to retrieve configuration properties, and to retrieve a set of classes which are responsible for managing groups of configuration information, called *registries*, each of them defined in a separate interface. All configuration interfaces can be found in the package `at.ac.univie.mminf.opax.config`. The following registries must be provided:

- `PeerRegistry` - provides methods to verify messages using public/private key encryption and signature methods, and to register new peers with their public key. (Note: as security features are not yet implemented, this registry should behave as if security requirements are fulfilled.)

- `SpaceRegistry` - stores information about spaces, like the list of a space's potential member peers, and the topology manager which is used for a space.

- `SpaceDirectoryRegistry` - maintains a list of space directory servers to be queried for retrieving up-to-date information about OPAX networks.

- `TopologyManagerRegistry` - maps topology managers, identified by their URI, to Java classes which implement these topology managers.

6.4 Internals

6.4.1 Message Processing

The most important task of the JIO implementation is to handle incoming and outgoing messages. As outgoing messages may be created by various sources, and incoming messages are received and distributed to various destination objects, all messaging is centralized within the `Peer` class. `Peer` utilizes several helper classes to manage message sending and receiving. Classes which are used to send or receive messages are grouped in the `at.ac.univie.mminf.opax.message.io` package. In the JIO implementation, these classes are designed to work with TCP connections for incoming and outgoing messages; currently, there exists no support for any other transportation layer like UDP.

Outgoing

Any component which wants to send a message into the OPAX network does this by calling the `Peer.send(NetworkAddress recipient, Message msg)` method. Each `Peer` instance holds one instance of `NetworkSender`, with which it is connected through the `messageOutboxQueue`, an instance of the queue class described below. `NetworkSender` is a class derived from `Thread`; it permanently reads messages from the queue and sends them to their recipients. As all components (`Peer`, the `Queue`, and `NetworkSender`) are fully synchronized, it is guaranteed that the ordering of sent messages is kept until the message is transmitted by the physical layer.

Figure 6.2 schematically shows the workflow when a message is to be sent.

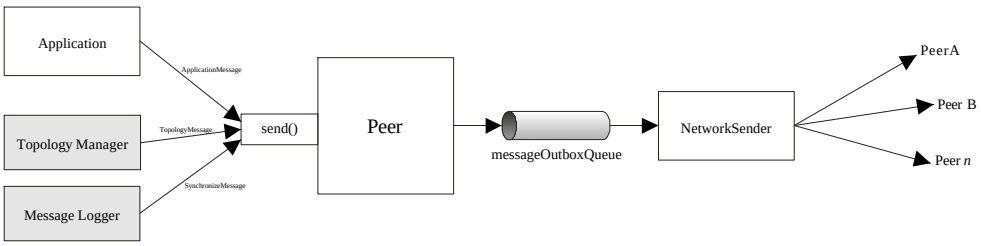

Figure 6.2: Components involved in sending messages

`NetworkSender` writes messages to a socket using a `MessageOutputStream`, which buffers the XML source of any message (obtained by the `Message.toXML()` method. After the header (see section 4.4 on page 29) and the message source is written to the socket, the output stream is closed.

Incoming

Receipt of messages is more difficult to accomplish in a distributed system where message transmission is asynchronous and stateless. To ensure the ordering of incoming messages, and, at the same time, avoiding blocking of the "server port" by voluminous message transmissions, a buffering mechanism must be established. The usage of future objects and a future object queue (see below) in combination with a multi-threaded "server" component ensures that messages that were sent from peer V to peer B in a certain order are processed by peer B in exactly the same order.

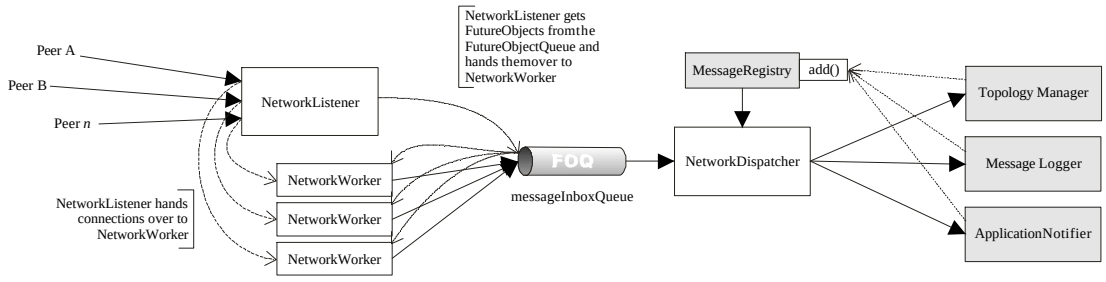

Figure 6.3: Components involved in receiving messages

`NetworkListener` is a thread which is listening for incoming connections on a `java.net.ServerSocket`. `NetworkListener` owns the "in-opening" of a `FutureObjectQueue` where all incoming messages are put into. For every accepted connection, a `FutureObject` is created by calling `FutureObjectQueue.add()`. Then, a `NetworkWorker` thread is started, and the input stream of the socket as well as the future object returned by the future object queue are passed to the network worker.

While the network listener has done its job and is waiting for the next incoming connection, it is now the task of the network worker to read the message from the socket and create an instance of the appropriate sub-class of `Message`, depending on the XML source of the message transmission (this work is performed by an instance of `MessageInputStream`, which, on its part, engages one instance of `MessageFactory` to create the message object.

After the message object has been created, the corresponding future object is cleared by assigning the message object to it, using the `FutureObject.set()` method. Now, the object may leave the future object queue, ready to be processed by the `MessageDispatcher`. This class uses a `MessageRegistry`, where components may register themselves (if they implement the `MessageHandler` interface) as destination for incoming messages, based on the space URI and the type of the message. The message dispatcher gets all message handlers for the incoming messages and subsequently calls the appropriate method. Now it is up to each message handler to process the incoming message.

6.4.2 Utility Classes

This section describes classes that were developed in context of OPAX and JIO but are not directly related to these applications and thus may be of interest for other applications and use cases. These classes have been grouped in the package `at.ac.univie.mminf.opax.util`. For more information on the detailed usage and syntax, refer to the OPAX javadoc pages.

BitField

`BitField` implements a binary vector, which can also be interpreted as an array of `boolean` values. Instead of actually using an `boolean[]` or `Boolean[]` array, `BitField` internally uses an array of `byte` values, so the memory consumption for a `BitField` of size n is $\lceil \frac{n}{b} \rceil$ if b is the size of the `byte` data type, which is usually 8 bit.

`BitField` objects can be constructed "from scratch", by copying an existing object, by parsing the content of a `String` consisting only of "0" and "1", and by converting a `long` value into its binary representation using little endian or big endian representation.

`BitField` provides methods to set and get bits identified by their index, to set blocks of bits, to compare two `BitField` instances, to calculate the Hamming distance and the XOR combination of two binary arrays, and many more. Most of these operations have a runtime of $O(n)$ for arrays of length n.

Queue

Java provides no FIFO Queue, so the class `Queue` implements such one. `Queue` is synchronized, in order to be used in multithreaded environments, and provides the standard methods `add()`, which adds an object to the end of the queue, and `get()`, which returns the first element in the queue.

FutureObject / FutureObjectQueue

`FutureObject` is a concept to create a reference on an object that does not yet exist. In other words, it is a synchronization mechanism which can be used when one thread is waiting for an object that is yet to be created by another thread. In this case, the common Java synchronization fails because using the `synchronize` keyword always requires an instance of an object as monitor.

Using a `FutureObject`, threads may create such an object and pass a reference to it instead of passing a reference to the actual object. After a future object has been created, the receiver of the object may query the object by calling the `FutureObject.get()` method. This method blocks until the creator "fills in" the actual object into the wrapper by calling `FutureObject.set()`. Immediately, the `get()` method returns with a reference to the actual object. The data flow is

depicted in figure 6.4.

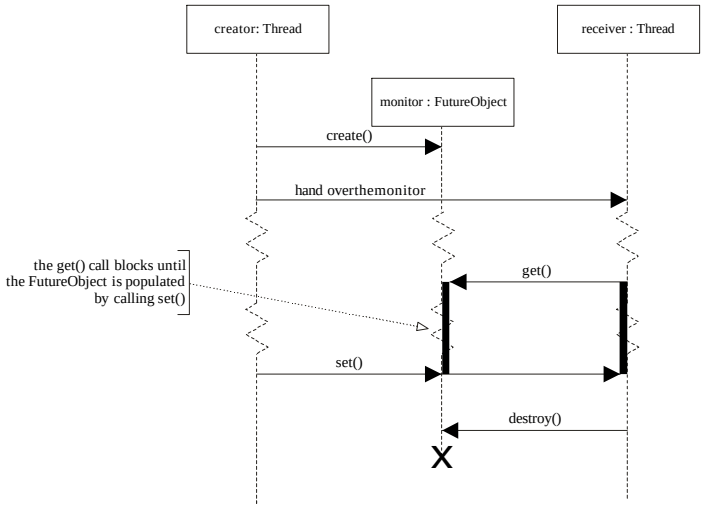

Figure 6.4: Sequence diagram for usage of `FutureObject`

If multiple future objects have to be passed between two threads, and the ordering of the objects is important, they may establish a `FutureObjectQueue`, which ensures that a set of future objects is passed according to the FIFO method.

Randomizer

Often, applications need to process the elements of a collection in random order. The `Randomizer` class wraps a Java `Collection`, and by calling `get()`, objects from the collection are returned in random order. Doing so, `Randomizer` guarantees that each object from the collection is returned exactly once; after all objects have been processed, `null` is returned. The randomizer may be reset using the `reset()` method.

XMLProperties

The `java.util.Properties` class is widely used to store configuration data. To provide a means of easily storing and retrieving properties in XML format, the `XMLProperties` class wraps a `Properties` instance and enables reading and writing of XML property files using the Xerces XML parser. A property file written and read by `XMLProperties` looks as follows:

```
<Properties>
  <Property name="name1" value="abc" />
  <Property name="name2" value="def" />
</Properties>
```

Listing 6.1: A sample XMLProperties file

SimpleDialog

As Java does not provide a "standard" message box, the `SimpleDialog` has been introduced. It provides a window which can be populated with any `java.awt.Component`, and adds a prompt message as well as "OK" and "Cancel" buttons to the window. The user may close the window by pressing either "OK" or "Cancel", and the selection can be queried by the application using the `isOKPressed()` and `isCancelPressed()` methods.

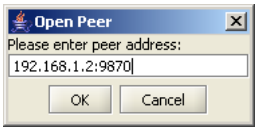

Figure 6.5: A `SimpleDialog`

Chapter 7

Future Work

7.1 Topology

Topology Stabilization As described in sections 5.1, the distributed topology has the disadvantage of being fault-prone because of the lack of knowledge about the overall peer situation. To enable the network to cope with multiple peer failures, further work has to be done in the field of topology stabilization.

Fallback Topology A different approach to make the network more resistant against errors is the introduction of a fallback topology, as described in section 5.1.7 on page 53. This feature, maintained parallel to the hypercube topology, may lead to higher robustness; together with alternative broadcast modes, it may allow efficient broadcast event when the network hypercube is degenerated.

Switchable Broadcast Modes If a fallback topology is being introduced, alternative algorithms for broadcasting messages have to be implemented, and the network must switch over to the alternative broadcast if it switches over to the fallback topology. To switch the network in a consistent way, and to prevent the intersection of different topologies or broadcast algorithms, a protocol must be introduced which ensures synchronization over all peers in the network. As the network may grow up to an arbitrary number of peers, this may lead to a highly complex algorithm.

7.2 Security

OPAX currently does not specify mechanisms to provide security aspects like message signing or authentication. To prevent attacks on the network and to make message transmission confident, such mechanisms have to be introduced.

7.3 Network Layer and Addressing Support

Currently, OPAX requires a network layer which guarantees the transportation of messages, and the only supported addressing system is the IP hostname:port system. To make OPAX suitable for heterogeneous environments, support for different protocols and addressing systems has to be implemented.

7.4 Directory Integration

The space directory lookup mechanism (described in appendix D) is planned to be rewritten to implement the `Configurator` interface, thus making it easier for peers to retrieve all their configuration information from remote sources.

Chapter 8

Conclusion

Peer-to-peer networks are one means to establish a messaging infrastructure for applications. OPAX serves as a network infrastructure which hides the details of network organization from the application and allows it to communicate with remote peers through XML message broadcasts.

In OPAX, the topology, which defines the arrangement and connections of network member peers, may be freely selected. By this, OPAX allows to adopt the topology to the application's requirements and allows designers to develop and integrate their own topology managers.

A hypercube, as one possible network topology, has been implemented. The approach presented in this work demonstrated that the applicability of a distributed hypercube topology for a structured P2P network is questionable. In the distributed variant, the networks lacks of stability. High efforts have to be invested to protect the network against peer or link failures, especially if they incidence in larger numbers.

The strategies presented in this work are still workarounds, as they do not solve the underlying problem: the distribution of topology knowledge amongst peers, and the inelastic structure of the hypercube which does not forgive any topological irregularity. Single local failures (failures where one peer fails, while all of its neighbours are still fully functional) may be caught and the topology may be ascribed to a consistent state. In cases where multiple failures occur, it may be impossible to return to a stable state: peers have mutual knowledge of their topology data, and if a set of peers fail, peers outside this set have no possibility to reconstruct the inner structure of this set.

A second approach, whereby the topology is managed by a central instance implicates the disadvantages of a single point of failure. Together with an adequate load balancing mechanism, this idea may serve well as topology infrastructure for networks with smaller amounts of nodes.

Appendix A

Message Schema

This section describes the OPAX message schema, an XML Schema that all OPAX messages must be valid to. Each implementation of OPAX should perform a validation of incoming messages against this schema.

```
<?xml version="1.0"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:opax="http://www.mminf.univie.ac.at/opax/message/namespace"
            targetNamespace="http://www.mminf.univie.ac.at/opax/message/namespace"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified"
>

  <xsd:element name="Message" type="opax:tMessage" />

    <xsd:complexType name="tMessage">

      <xsd:sequence>

         <xsd:choice>

           <xsd:element name="Unknown">
             <xsd:complexType>
               <xsd:attribute name="text" type="xsd:string" />
             </xsd:complexType>
           </xsd:element>

           <xsd:element name="Ping" />

           <xsd:element name="Application">
             <xsd:complexType>

               <xsd:sequence>
                 <xsd:element name="Data">
                   <xsd:complexType>
                     <xsd:sequence>
                       <xsd:any />
                     </xsd:sequence>
                   </xsd:complexType>
                 </xsd:element>
               </xsd:sequence>

               <xsd:attribute name="originator" type="xsd:string" />
```

```
40                <xsd:attribute name="valid-from" type="xsd:string" />
41                <xsd:attribute name="valid-to" type="xsd:string" />
42                <xsd:attribute name="routing-control" type="xsd:string" />
43
44              </xsd:complexType>
45            </xsd:element>
46
47            <xsd:element name="Topology">
48              <xsd:complexType>
49
50                <xsd:sequence>
51                  <xsd:element name="Properties">
52                    <xsd:complexType>
53
54                      <xsd:sequence>
55                        <xsd:element name="Property" maxOccurs="unbounded" >
56                          <xsd:complexType>
57                            <xsd:attribute name="name" type="xsd:string" />
58                            <xsd:attribute name="value" type="xsd:string" />
59                          </xsd:complexType>
60                        </xsd:element>
61                      </xsd:sequence>
62
63                    </xsd:complexType>
64                  </xsd:element>
65
66                </xsd:sequence>
67
68                <xsd:attribute name="type" type="xsd:string" />
69
70              </xsd:complexType>
71            </xsd:element>
72
73            <xsd:element name="Directory">
74              <xsd:complexType>
75
76                <xsd:sequence>
77                  <xsd:element name="Item" minOccurs="0" maxOccurs="unbounded">
78                    <xsd:complexType>
79                      <xsd:attribute name="data" type="xsd:string" />
80                    </xsd:complexType>
81                  </xsd:element>
82                </xsd:sequence>
83
84                <xsd:attribute name="type" type="xsd:string" />
85                <xsd:attribute name="reference-message" type="xsd:string" use="optional" /
                      >
86
87              </xsd:complexType>
88            </xsd:element>
89
90            <xsd:element name="ApplicationConfiguration">
91              <xsd:complexType>
92
93                <xsd:sequence>
94                  <xsd:element name="Properties">
95                    <xsd:complexType>
96
97                      <xsd:sequence>
98
99                        <xsd:element name="Property" minOccurs="0" maxOccurs="unbounded">
100                         <xsd:complexType>
101                           <xsd:attribute name="name" type="xsd:string" />
102                           <xsd:attribute name="value" type="xsd:string" />
103                         </xsd:complexType>
```

```
                        </xsd:element>

                      </xsd:sequence>

                    </xsd:complexType>
                  </xsd:element>
                </xsd:sequence>

              </xsd:complexType>
            </xsd:element>

            <xsd:element name="Synchronize">
              <xsd:complexType>

                <xsd:sequence>
                  <xsd:element name="UUID" minOccurs="0" maxOccurs="unbounded" />
                </xsd:sequence>

                <xsd:attribute name="type" type="xsd:string" />

              </xsd:complexType>
            </xsd:element>

          </xsd:choice>

          <xsd:element name="Signature" type="xsd:string" minOccurs="0" />

        </xsd:sequence>

        <xsd:attribute name="uuid" type="xsd:string" />
        <xsd:attribute name="timestamp" type="xsd:string" />
        <xsd:attribute name="from" type="xsd:string" />
        <xsd:attribute name="space" type="xsd:string" />

      </xsd:complexType>


</xsd:schema>
```

Listing A.1: XML Schema for OPAX messages

Appendix B

Used URIs

Spaces

Spaces are identified by an URI with `opax` as prefix. The identifier for the *internal space* is `opax://www.mminf.univie.ac.at/space/internal`.

Messages

The root URI for all identifiers specified in this section is `http://www.mminf.univie.ac.at/opax/message` and is abbreviated as `[ROOT]` in the following. The following fields are used to identify message related information.

Namespace

The XML namespace for OPAX message documents is `[ROOT]/namespace`.

Message types

ApplicationConfiguration

Used to identify the routing control information for `ApplicationConfiguration` messages.

```
[ROOT]/application/routing-control/RC-NOT-ROUTED
[ROOT]/application/routing-control/RC-SYNCH-LOCAL
[ROOT]/application/routing-control/RC-SYNCH-REMOTE
```

Directory

Used to identify the sub-type of `Directory` messages.

```
[ROOT]/directory/type/unknown
[ROOT]/directory/type/get-spaces
[ROOT]/directory/type/search-spaces
[ROOT]/directory/type/get-peers
[ROOT]/directory/type/get-topmgr
```

Synchronize

Used to identify the sub-type of `Synchronize` messages.

```
[ROOT]/synchronize/type/get-uuid-list
[ROOT]/synchronize/type/send-uuid-list
[ROOT]/synchronize/type/get-message
[ROOT]/synchronize/type/msg-unavailable
```

Topology

Used to identify the sub-type of `Topology` messages.

```
[ROOT]/topology/type/UNKNOWN
```

```
[ROOT]/topology/type/hypercube2/0.1.3/ConfirmBuffer
[ROOT]/topology/type/hypercube2/0.1.3/ConnectNeighbours
[ROOT]/topology/type/hypercube2/0.1.3/ExecuteIntegration
[ROOT]/topology/type/hypercube2/0.1.3/FinalizeIntegration
[ROOT]/topology/type/hypercube2/0.1.3/Join
[ROOT]/topology/type/hypercube2/0.1.3/RouteIntegration
[ROOT]/topology/type/hypercube2/0.1.3/StartBuffer
[ROOT]/topology/type/hypercube2/0.1.3/MonitorMe
[ROOT]/topology/type/hypercube2/0.1.3/NoMoreMonitorMe
[ROOT]/topology/type/hypercube2/0.1.3/MonitorYou
[ROOT]/topology/type/hypercube2/0.1.3/NoMoreMonitorYou
```

```
[ROOT]/topology/type/hypercube3/0.1.1/ConfirmBuffer
[ROOT]/topology/type/hypercube3/0.1.1/ConnectNeighbours
[ROOT]/topology/type/hypercube3/0.1.1/ExecuteIntegration
[ROOT]/topology/type/hypercube3/0.1.1/FinalizeIntegration
[ROOT]/topology/type/hypercube3/0.1.1/Join
[ROOT]/topology/type/hypercube3/0.1.1/RouteIntegration
[ROOT]/topology/type/hypercube3/0.1.1/StartBuffer
```

[ROOT]/topology/type/hypercube3/0.1.1/TransferOwnership

[ROOT]/topology/type/hypercube4/0.1.1/NewNeighbour
[ROOT]/topology/type/hypercube4/0.1.1/ReplaceNeighbour
[ROOT]/topology/type/hypercube4/0.1.1/ForgetNeighbour
[ROOT]/topology/type/hypercube4/0.1.1/Join
[ROOT]/topology/type/hypercube4/0.1.1/JoinAccept
[ROOT]/topology/type/hypercube4/0.1.1/Leave
[ROOT]/topology/type/hypercube4/0.1.1/LeaveAccept
[ROOT]/topology/type/hypercube4/0.1.1/PeerUnreachable

Appendix C

Synchronization Protocol

Introduction

The OPAX synchronization protocol enables peers that join a space to make up the receipt of `Application` messages that were broadcast into the space during their absence, and to notify the application of messages that have been received in previous sessions but are still valid. Thus, two types of synchronization are distinguished: *local* and *remote* synchronization.

Local Synchronization

It is the goal of the local synchronization to keep applications up-to-date as soon as they join a space. As each application message has a validity period, OPAX releases applications from managing this, instead it provides the feature that messages which have been received in previous sessions are stored and automatically re-sent to the application. This happens transparently to the application, as "cached" application messages are transmitted to the application in the same way as messages which are received from the network.

Remote Synchronization

A peer joining a network must become aware of `Application` messages that have been broadcast during its absence. The remote synchronization protocol allows a peer to query its neighbours for messages that it did not yet receive. It must be initiated by the joining peer after the successful completion of the joining procedure.

The synchronization is performed in four steps:

1. *Synchronize Request Initiation* - At first, the peer wishing to synchronize sends a `Synchronize` / `get-uuid-list` message to its neighbours, together with the UUID of the last message that the peer did receive before its departure of the space.

2. *UUID Announcement* - A peer receiving a `get-uuid-list` message checks its message cache and constructs a list containing UUIDs of messages that (1) have been broadcast after the one that was sent by the peer to be synchronized, and (2) are still valid. Then, it sends a `Synchronize` / `send-uuid-list` message, together with the list of UUIDs, back to the requesting peer.

3. *Peer Selection* - The peer wishing to synchronize, after having received a set of `send-uuid-list` messages, selects one peer to synchronize for each message that it has not yet received. A `Synchronize` / `get-message` message, together with the UUID of the required message, is sent for each message that must be synchronized.

4. *Message Transmission* - Any peer receiving a `get-message` message reads the `Application` message with the given UUID from its cache and sends it, using a "normal" `Application` message, to the peer. The `routing-control` field of this message must be set to `SYNCH-REMOTE` in order to indicate that this message is sent for synchronization purposes and thus is not to be forwarded to other peers.

   If the peer can not satisfy the message request (because maybe the message expired in the meantime), it sends back a `Synchronize` / `msg-unavailable` message.

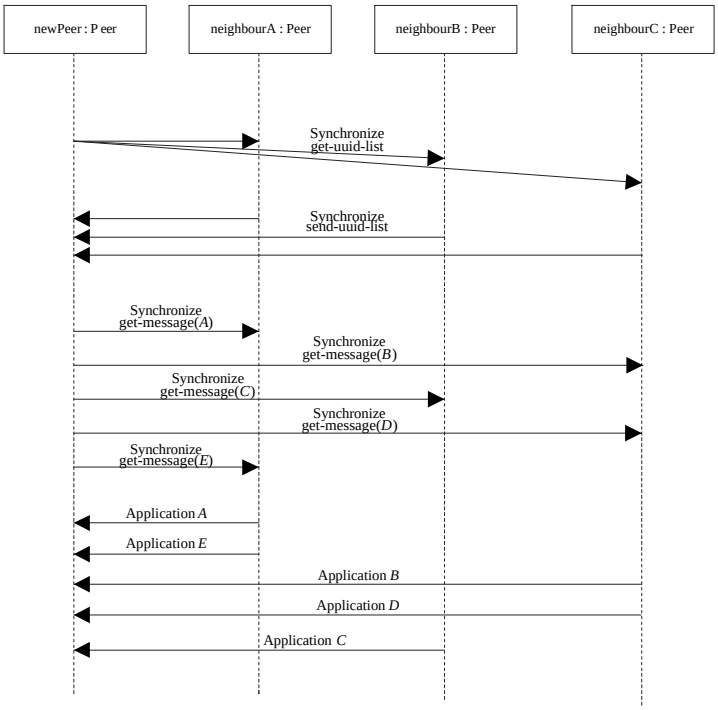

Figure C.1: Sequence diagram for the synchronization protocol

Synchronize Message Type Synchronization requests are transmitted by `Synchronize` messages. This message has one attribute, `type`, which indicates the type of the request. The exact URIs for this field are to be found in appendix B.

Furthermore, `Synchronize` may have an arbitrary (0 to many) number of `UUID` elements. `UUID` has no attributes and no sub-elements, instead may contain one UUID as text content. The set of UUID elements is referred to as *UUID list*. The exact meaning of the UUID list depends on the message's `type` attribute:

Type	# UUID elements	Meaning
`get-uuid-list`	0	to be ignored
`send-uuid-list`	$0 \ldots n$	messages that the sending peer holds in its local cache
`get-message`	1	the message to be sent to the sending peer
`msg-unavailable`	1	the message is not available at the sending peer

Table C.1: Meaning of a Synchronize message's UUID list

```
<Message from="192.168.1.2:9870" space="opax://www.opax.net/
    testspace3" timestamp="1091173710328" uuid="d96da80d-e1fc-11d8-
    b241-e6c8d709d640" xmlns="http://www.mminf.univie.ac.at/opax/
    message/namespace">
  <Synchronize type="http://www.mminf.univie.ac.at/opax/message/
      synchronize/type/get-message">
    <UUID>db860055-e1fc-11d8-9973-c2748d1f5d88</UUID
  </Synchronize>
</Message>
```

Listing C.1: A sample `Synchronize` message

JIO Implementation

In JIO, synchronization is accomplished by the `at.ac.univie.mminf.opax.logging.MessageLogger` class. Initialized for each space, it uses the local file system to store messages (one file per message) and the index file.

`MessageLogger` receives incoming `Application` and `Synchronize` messages from its `Peer` instance. `Application` messages are being written on disk, while `Synchronize` messages are evaluated and the appropriate actions are taken. `MessageLogger` sends outgoing `Synchronize` messages through its `Peer` instance.

For local synchronization, `Peer` calls `MessageLogger.synchronizeLocal()` after a space has been joined. `MessageLogger` then scans its message cache for messages that are still valid. Those messages are sent with the `routing-control` field set to `rc-sync-local` through the `Peer.messageReceived()` method, which then forwards the message to the application.

For remote synchronization, `Peer` calls `MessageLogger.synchronizeRemote()` with the UUID of the last received message and a list of peers that the `MessageLogger` should synchronize with. `MessageLogger` then initiates the synchronization protocol as described in the previous section.

Appendix D

Space Directory Lookup Protocol

Introduction

A *space directory* is a public server which keeps information about existing OPAX networks in order for peers to find their way in larger environments, and for users and applications to learn about which OPAX spaces exist, which configuration properties they have, and which peers to contact in order to join a certain space.

To accomplish this, a space directory server holds a tree structure of spaces that it knows. This tree is organized to reflect the semantic properties of the spaces; the tree is comparable to a web directory service, where web pages are grouped based on their topic. In the space directory tree, the nodes which form the "folders" of the directory are called *space groups*.

Although OPAX messaging is used for the interaction between a space directory server and "normal" OPAX peers, a space directory server is not part of a particular OPAX space; the *internal space* is used for all `Directory` messages.

A space directory server can be queried to obtain the following information; the corresponding sub-type of the `Directory` message is also listed:

- a list of spaces that are registered in a given space group (`get-spaces`)
- a list of spaces that are registered in a space group whose name matches a given search string (`search-spaces`)
- a list of peers which are potential members of a given space; those peers may be contacted in order to integrate a new peer (`get-peers`)
- the URI of a topology manager which is used in a given space; as different topology managers exist, and the topology manager must be known to the peer prior to integration, this information is required for a peer that wants to be integrated into a space (`get-topmgr`)

Messaging

To accommodate the requirements of the different sub-types (as enumerated before), and to ease later extensions, the `Directory` message type is kept very generic. It is similar to the `Topology` message type: it has one attribute `type`, indicating the sub-type, one attribute `reference-message` which holds the UUID of the message that this message is a reply to (if such one exists), and an arbitrary number of data item elements `Item` whereof each one has an attribute `data` holding the actual information.

Each talk of a peer to a space directory server consists of two messages: First, the peer sends the desired request message with the `type` field according to the request, an empty `reference-message` field and no `Item` elements. Then, the directory server processes the request using its data base and replies with a `Directory` message of the same `type`, the `reference-message` field filled with the UUID of the original request message, and `Item` elements containing the data from the data base.

Implementation

In JIO, classes for directory services are grouped in the `at.ac.univie.mminf.opax.directory` package. JIO provides a class `SpaceDirectoryPeer` which extends the `Peer` class and provides functionality to serve directory requests from other peers. On creation, it initializes its database (a tree constructed of the classes `ManagedSpaceGroup` and `ManagedSpace`) with data from the supplied configuration file. The `SpaceDirectoryPeer` registers itself as listener for `Directory` messages and processes the requests using its database. `SpaceDirectoryPeer` may be instantiated by any application similar to the "normal" `Peer` class.

On client side, space directory lookup is currently implemented in the `Peer.join()` method. This method has a boolean parameter `lookup`, which is set to `true` if the peer should lookup space data. The actual lookup is performed by the `DirectoryClient` class, which encapsulates the asynchronous communication to a space directory server in its methods. The appropriate methods are called from the owning `Peer` instance during the join process.

In future, it is planned to integrate the space directory lookup in one implementation of the `Configurator` interface, thus making it possible to retrieve every aspect of configuration from a directory server, including public key lists and actual topology manager implementations.

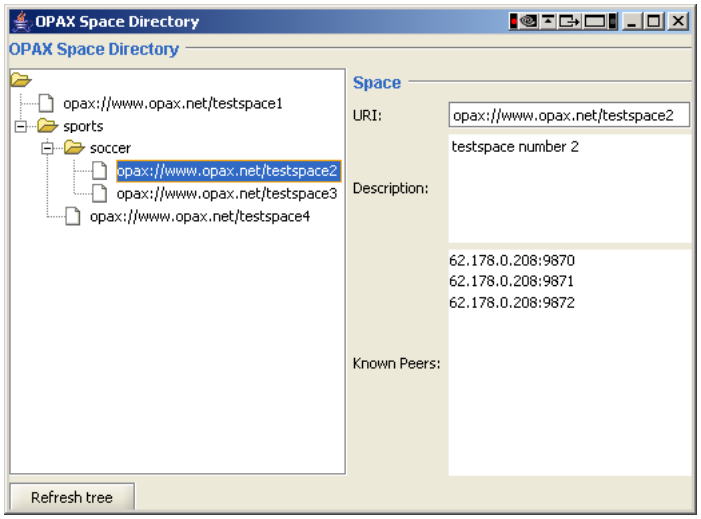

Figure D.1: Space directory demo application

Appendix E

Peer Watch Protocol

Introduction

The peer watch concept has been introduced to enable the consistent, over-directed departure of peers in case of a failure. Without peer watch, in case of a failure, there is no means for the neighbours of the failing node to reconstruct the distributed topology, because they lack knowledge about the failing peer's neighbourhood. The idea of the peer watch concept is to designate one deputy (or *monitor*) per peer which supervises this peer and carries out the departure protocol on its behalf if the peer fails. The peer watch protocol is able to recover the loss of one peer, but it does not succeed if multiple peers - e.g. one peer *and* its monitor - fail simultaneously.

In the hypercube, to remove one peer *on behalf of itself*, it is neccessary for its deputy to know all its local topology data, i.e. its position vector, its cover map vector, and the list of its neighbours and their position vectors. Four sub-types of the `Topology` message type have been introduced for the peer watch protocol (for a detailled description see below) which topology manager instances can use to designate or release a peer monitoring responsibility, to inform the monitor of a monitor's local topology knowledge, and to notify a monitored peer in case of its monitor is no more able to keep the responsibility, e.g. because it is to leave the space.

A peer's monitor is required to regularily check the liveliness of its protege, and to initiate the *departure on behalf* if it detects that the peer failed. The liveliness check is carried out using standard OPAX `Ping` messages.

The peer watch protocol has successfully been implemented within the *hypercube2* topology manager.

Message Types

- *MonitorMe* - is sent by a peer which is to be monitored to its designated monitor as a request to take over the monitoring responsibility.
- *NoMoreMonitorMe* - is sent by a protege to its monitor notifying it that it is to release the responsibility. This message may be sent because of a change of the protege's neighbourhood which causes it to select a new monitor.
- *MonitorYou* - is the reply of the monitor which received a `MonitorMe` message; using this, the peer confirms its responsibility and is from now on monitoring its protege.
- *NoMoreMonitorYou* - is sent by a monitor to its protege to indicate that it does no longer own the monitoring responsibility. This may be because of a protege's `NoMoreMonitorMe` message sent, or because the monitor is to leave the space and therefore is no longer able to be the peer's monitor.

The messages of the peer watch protocol use the `Properties` list of the `Topology` message type to pass information which further describes the monitor requests: the `MonitorMe` and `NoMoreMonitorMe` messages have a field `MonitoredPeer` which contains the network address of the monitored peer; the `MonitorYou` and `NoMoreMonitorYou` messages have a field `MonitorPeer`, containing the network address of the monitor peer.

Additionally, `MonitorMe` has the following fields:

- `MonitoredPeerCoordinates` - contains the position vector of the monitored peer
- `MonitoredPeerCoverMap` - contains the (binary) cover map of the monitored peer
- `MonitoredPeerNeighbours` - contains a space-separated list of the monitored peer's neighbour peers in no particular order
- `MonitoredPeerNeighboursCoordinates` - contains a space-separated list of the monitored peer's neighbours' position vectors, in the same order as the `MonitoredPeerNeighbours` list

The monitor peer must keep all this data as long as it keeps the monitoring responsibility. A change in the protege's local view of the topology causes the deselection of the monitor and the election of a new one, so there is no need for any "update" message.

Heartbeat

A peer having the responsibility to monitor another peer has to regularily check the availability of its protege by sending `Ping` messages. In case of a failure it has to initiate the *departure on behalf*, as described below. It is up to the implementation if the departure is initiated immediately after a failed `Ping`, or even after a certain number of failed heartbeats.

Departure on Behalf

The protocol for the departure on behalf is exactly the same as the common departure protocol as described in section 5.1.5 on page 42. To indicate the fact that the departure is carried out on behalf of the departing peer, a boolean field called `OnBehalf` is added to all concerned messages (`StartBuffer`, `ConfirmBuffer`, `FinalizeIntegration`. This field set to `true` indicates that any communication should not be directed to the departing peer (whose network address is transmitted in the `DepartingPeer` field), but to the sender of the message, which is the monitoring peer.

List of Figures

List of Tables

Listings

Bibliography

[1] Collins, M. et al. *P2P Search Engines*. Networks and Telecommunications Research Group. http://ntrg.cs.tcd.ie/undergrad/4ba2.02-03/p8.html

[2] Eastlake, D. et al. *RFC 3174 - US Secure Hash Algorithm 1 (SHA1)*. RFC3174. 2001. http://www.faqs.org/rfcs/rfc3174.html http://www.ietf.org/rfc/rfc3174.txt

[3] *Napster homepage* http://www.napster.com

[4] *Gnutella homepage* http://www.gnutella.com

[5] *Freenet homepage* http://freenet.sourceforge.net

[6] Boll, S. and Westermann, G. *MediÆther - an Event Space for Context-Aware Multimedia Experiences*. In: Proceedings of International ACM SIGMM Workshop on Experiential Telepresence (ETP'03), Berkeley, USA, 2003.

[7] *Merriam-Webster Online Dictionary* http://www.m-w.com

[8] *Wikipedia Online Encyclopedia* http://www.wikipedia.org

[9] Tsaparas, P. *P2P Search* http://www.cs.unibo.it/biss2004/slides/tsaparas-myP2P.pdf

[10] Schlosser, M. *Semantic Web Services*. Technical Report, University of Hannover and Stanford University, 2002.

[11] Berners-Lee, T. *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396. 1998. http://www.faqs.org/rfcs/rfc2396.html http://www.ietf.org/rfc/rfc2396.txt

[12] W3C. *HTTP - Hypertext Transfer Protocol*. http://www.w3.org/Protocols/

[13] Fielding, R. et al. *Hypertext Transfer Protocol - HTTP/1.1*. RFC 2068. 1997.
http://www.faqs.org/rfcs/rfc2068.html
http://www.ietf.org/rfc/rfc2068.txt

[14] Leach, P. J. and Salz, R. *UUIDs and GUIDs*. Internet Draft, 1998.
http://www1.ics.uci.edu/~ejw/authoring/uuid-guid/draft-leach-uuids-guids-01.txt

[15] W3C. *Extensible Markup Language (XML)*.
http://www.w3.org/XML/

[16] W3C. *XML Schema*.
http://www.w3.org/XML/Schema

[17] W3C. *Namespaces in XML*. Recommendation, 1999.
http://www.w3.org/TR/REC-xml-names/

[18] Postel, J. *User Datagram Protocol*. RFC 768. 1980.
http://www.faqs.org/rfcs/rfc768.html
http://www.ietf.org/rfc/rfc768.txt

[19] Postel, J. (editor). *Transmission Control Protocol*. RFC 793. 1981.
http://www.faqs.org/rfcs/rfc793.html
http://www.ietf.org/rfc/rfc793.txt

[20] Hauben, M. *History of ARPANET*.
http://www.dei.isep.ipp.pt/docs/arpa.html

[21] Postel, J. (editor). *Telnet Protocol Specification*. RFC 854. 1983.
http://www.faqs.org/rfcs/rfc854.html
http://www.ietf.org/rfc/rfc854.txt

[22] Postel, J. (editor). *File Transfer Protocol (FTP)*. RFC 959. 1985.
http://www.faqs.org/rfcs/rfc959.html
http://www.ietf.org/rfc/rfc959.txt

[23] Kalin, R. *Simplified NCP Protocol*. RFC 60. 1970.
http://www.faqs.org/rfcs/rfc60.html
http://www.ietf.org/rfc/rfc60.txt

[24] *Project JXTA homepage*.
http://www.jxta.org

[25] Nejdl, W. et al. *Edutella - RDF-based Metadata Infrastructure for P2P Applications*.
http://edutella.jxta.org/

[26] Johnsson, S. L. and Ho, C.-T. *Optimum Broadcasting and Personalized Communication in Hypercubes*. In: IEEE Transactions on Computers, 38. 1989.

[27] Balakrishnan, H. et al. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. In: Proceedings of ACM SIGCOMM. 2001.

[28] Maymounkov, P. and Mazires, D. *Kademlia: A Peer-to-peer Information System Based on the XOR Metric*. In: Proceedings of IPTPS02, Cambridge, USA. 2002

[29] Melville, L., Sommerville, I., and Walkerdine, J. *P2P Reference Architectures*. Research Report, Lancaster University. 2003.
http://www.atc.gr/p2p_architect/brochure/0309F05_ReferenceArch.pdf

[30] Breslau, L. et al. *Making Gnutella-like P2P Systems Scalable*. Presented at SIGCOMM '03, Karlsruhe. 2003.
http://www.acm.org/sigs/sigcomm/sigcomm2003/papers/p407-chawathe.pdf

[31] Backx, P. et al. *A comparison of peer-to-peer architectures*.
http://allserv.rug.ac.be/~pbackx/cgi-bin/countdown.cgi?A%20comparison%20of%20peer-to-peer%20architectures.pdf

[32] Tanenbaum, A. and van Steen, M. *Distributed Systems - Principles and Paradigms*. Prentice-Hall. 2002.

[33] Neuman, B. *Scale in Distributed Systems*. In: Readings in Distributed Computing Systems, IEEE Computer Society. 1994.

[34] Lv, Q. et al. *Search and Replication in Unstructured Peer-to-Peer Networks*. In: Proceedings of the 16th international conference on Supercomputing. 2002.
http://www.cs.princeton.edu/~qlv/download/searchp2p_full.pdf

[35] Hildrum, K. and Kubiatowicz, J. *Asymptotically Efficient Approaches to Fault-Tolerance in Peer-to-Peer Networks*. In: Proceedings of the 17th International Symposium on Distributed Computing. 2003.
http://oceanstore.cs.berkeley.edu/publications/papers/pdf/disc.pdf

[36] Rowstron, A. and Druschel, P. *Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-Peer Systems*. In: Lecture Notes in Computer Science, vol. 2218, 2001.
http://www.cs.rice.edu/CS/Systems/PAST/pastry.pdf

[37] *Pastry - A Substrate for Peer-to-Peer Applications*.
http://research.microsoft.com/~antr/Pastry/

[38] Zhao, B. Y. et al. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*. UC Berkeley, 2001.

[39] Terpstra, W. W. et al. *A Peer-to-Peer Approach to Content-Based Publish/Subscribe*. Second International Workshop on Distributed Event-Based Systems (DEBS'03), San Diego, USA.
http://www.eecg.toronto.edu/debs03/papers/terpstra_etal_debs03.pdf

[40] Tam, D. et al. *Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables*.
http://www.eecg.toronto.edu/ tamda/dht/dht.pdf

[41] Castro, M. et al. *Scribe: A large-scale and Decentralized Application-level Multicast Infrastructure*. In: IEEE Journal on Selected Areas in Communications, Vol. XX, 2002.
http://www.cs.rice.edu/~druschel/publications/Scribe-jsac.pdf

[42] Verdy, P. *Approximating the Hypercube Network Topology for Better Scalability*.
http://www.limewire.org/pipermail/gui-dev/2002-February/000262.html

[43] *IPv6 Information Page*.
http://www.ipv6.org

[44] *Java Technology Homepage*.
http://java.sun.com

[45] *eclipse.org Homepage*.
http://www.eclipse.org

[46] W3C. *Document Object Model (DOM)*.
http://www.w3.org/DOM/

[47] Saloranta, T. *Java Uuid Generator (JUG) Homepage*.
http://www.doomdark.org/doomdark/proj/jug/

[48] *Xerces2 Java Parser Homepage*.
http://xml.apache.org/xerces2-j/index.html

[49] Lentzsch, K. *JGoodies Forms*.
http://www.jgoodies.com/freeware/forms/index.html

[50] Lentzsch, K. *JGoodies Looks*.
http://www.jgoodies.com/freeware/looks/index.html

[51] Danezis, G. and Anderson, R. *The Economics of Censorship Resistance*. Technical Report, University of Cambridge.
http://www.cl.cam.ac.uk/users/gd216/redblue.pdf

[52] Roussopoulos, M. et al. *2 P2P or not 2 P2P?*. In: Proceedings of 3rd International Workshop on Peer-to-Peer Systems, 2004.
http://iptps04.cs.ucsd.edu/papers/roussopoulos-to-or-not.pdf

[53] List of Peer-to-Peer Conferences.
http://www.neurogrid.net/twiki/bin/view/Main/PeerToPeerConferences

[54] Christin, N. and Chuang, J. *On the Cost of Participating in a Peer-to-Peer Network*. In: Proceedings of 3rd International Workshop on Peer-to-Peer Systems, 2004.
http://iptps04.cs.ucsd.edu/papers/christin-cost.pdf

[55] Sit, E. et al. *UsetnetDHT: A Low Overhead Usenet Server*. In: Proceedings of 3rd International Workshop on Peer-to-Peer Systems, 2004.
http://iptps04.cs.ucsd.edu/papers/sit-usenetdht.pdf

[56] Josephson, W. et al. *Peer-to-Peer Authentication with a Distributed Single Sign-On Service.* In: Proceedings of 3rd International Workshop on Peer-to-Peer Systems, 2004. http://iptps04.cs.ucsd.edu/papers/josephson-sign-on.pdf