DIPLOMA THESIS

Embedded Peer-to-Peer Networking

Submitted at the Faculty of Electrical Engineering, Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Master of Sciences (Diplomaingenieur)

under supervision of

O. Univ. Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
Dipl.-Ing. Dr.techn. Friederich Kupzog

by

Thomas Gamauf
Matr.Nr. 0325401
Schreibersdorf 23
7423 Pinkafeld

April 2009

Kurzfassung

Die Anforderungen an das elektrische Versorgungsnetz sind durch den steigenden Energiebedarf in den letzten Jahren massiv gestiegen. Eine Möglichkeit, die dadurch entstehenden Probleme zu bewältigen, besteht in der Schaffung von intelligenten Stromnetzen, die unter anderem der Steuerung von Lasten auf der Verbraucherseite dienen. Dies kann zum Beispiel durch eingebettete Systeme realisiert werden, die verbraucherseitig elektrische Lasten schalten, wobei es für die effiziente Nutzung dieser Technologie unumgänglich ist, dass sich diese Knoten untereinander verständigen können.
Die Aufgabe dieser Arbeit ist es, ein Peer-to-Peer-Protokoll zu finden, das es den Knoten ermöglicht, sich dezentral zu koordinieren. Peer-to-Peer-Protokolle werden üblicherweise für Heimcomputer entwickelt, welche im Vergleich mit den verwendeten eingebetteten Systemen wesentlich leistungsfähigere Hardware zur Verfügung haben. Aus diesem Grund stellt die eingeschränkte Hardwareaustattung, die diese Steuerknoten aus Kostengründen im Allgemeinen mitbringen, eine große Herausforderung dar. Das gefundene Protokoll wird dann für die vorhandenen Knoten implementiert, um die Verwendbarkeit eines Peer-to-Peer-Protokolls auf solchen leistungsschwachen Geräten zu demonstrieren. Dazu wird die entwickelte Software auf dem existierenden Steuergerät getestet und zusätzlich das Verhalten bei steigender Knotenzahl durch Simulationen ermittelt. Die erhaltenen Resultate weisen darauf hin, dass es möglich ist, diese Technologie und ihre Vorteile auch auf eingebetteten Systemen einzusetzen. Vor allem wird aber auch gezeigt, unter welchen Bedingungen dies möglich ist.

Abstract

The demand for electrical power grows continuously and with it the challenges for the electric power grid, to cope with these changes. Improvement of the situation can be gained by addition of intelligence to the power grid. One option that makes this possible are embedded energy management nodes at the consumer side, which control connected loads depending on the stress of the grid. Coordination of the nodes is an essential condition for efficient execution of this task. In this work a peer-to-peer protocol is to be found that enables these nodes to interact in a decentralised way. Peer-to-peer protocols are usually developed for home computers, which offer far more hardware performance as any embedded system. So the main challenge in this endeavour is to find a protocol that is able to cope with the severely limited hardware resources of these embedded energy management nodes. This protocol is then implemented as a proof of concept, in order to establish that the peer-to-peer technology is applicable even on low-performance embedded systems.
The developed software is tested on the existing embedded energy management node and the behaviour in larger networks of up to thousand nodes analysed in simulations. By these procedures the suitability of the protocol for coordination between these embedded systems is ascertained but not without highlighting existing limits.

To all the people that supported me
in a million different ways
over the last few months and years.

No words could ever transport
the gratitude I feel towards all of you.

Table of Contents

Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CAN	Content-Addressable Network
CRC	Cyclic Redundancy Check
DHT	Distributed Hash Table
DoS	Denial-of-Service
DSM	Demand Side Management
GPRS	General Packet Radio Service
GUI	Graphical User Interface
IP	Internet Protocol
IRON	Integral Resource Optimisation Network
JXTA	Juxtapose
LAN	Local Area Network
P2P	Peer-to-Peer
SOA	Service-Oriented Architecture
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VO	Virtual Organisation
WLAN	Wireless Local Area Network

1 Introduction

Computers and distributed systems have a wide array of applications in our homes and the industry, ranging from low-power sensor networks to high-performance mainframe clusters. They are used in building automation, have their place in nuclear submarines and ensure passenger safety in cars and aeroplanes. Driven by the changes to the structure of the electric power grid, a further area of use for distributed systems has grown in importance over the last few years – energy management. With the rising number of participants in the process of energy production, management of the electric power grid steadily increases in difficulty. Features like self-organisation, which are exhibited by certain types of distributed systems, have the potential to aid in this situation.

Among the multitude of existing forms, peer-to-peer networks are a comparably new type of distributed system. They experienced a real boom in the last decade and have a fixed place on personal computers today. Starting as tools for file sharing, peer-to-peer (P2P) networks have penetrated many different parts of the "Internet life". The idea behind these systems is to make every node equal, with same rights and same duty. This really is a compelling thought that neatly opposes the popular client-server scheme and is considerably betters suited for interaction between common computers in the Internet.
Nonetheless, while peer-to-peer systems are widespread on personal computers, they have not yet penetrated the domain of low-performance embedded systems seriously. One reason for this state definitely is that many peer-to-peer protocols/applications are intended for rather high-performance personal computers and need far more resources as any single embedded system can possibly provide. However, some of the more recent developments show more reasonable demands in this regard and eventually open up the field for embedded systems, too.
As the title suggests, the overall topic in this thesis is establishing a P2P network made of embedded nodes. The aim of this work is to find a suitable peer-to-peer protocol among the multitude of possibilities and subsequently adapt it to an existing distributed system, which consists of low-performance embedded systems that are using LANs or the Internet as communication infrastructure.

1.1 State-of-the-Art

In this work the implementation and deployment of a peer-to-peer protocol is the focus. As P2P systems are a form of distributed system, they are examined in this section in order to provide a background for the rest of the thesis.

Distributed systems became possible by two developments in the last three decades, the increase of performance of computer systems in combination with the massive reduction of cost and the invention of high-speed computer networks [GBKS08, p. 1]. Together the advances in these two areas enabled the design of systems, consisting of multiple linked computers that together are more than the sum of all parts. According to Andrew S. Tanenbaum [TS07, p. 2] a distributed system is

> ”... a collection of independent computers that appears to its users as a single coherent system.”

This is a comparably loose definition but highlights the essential properties that distinguish distributed systems from other related systems. First of all, a distributed system consists of *more than one* computer and each of the parts is *autonomous*. Single machines (nodes) may differ in the hardware they consist of, the operating system each node uses, the programming language of the software and the resources it offers but the system as a whole must not depend on any of these. The second important property of any distributed system and the main distinguishing feature to a simple computer network is that it appears as an *uniform* system to the outside. It builds on an underlying computer network but masks its complexity and provides the means to process a task just like a single computer. Beside computer networks, parallel systems are a second type of system that bears some resemblances to distributed systems. However, while the components in a parallel system are closely coupled via shared memory, the parts of a distributed are connected loosely by a computer network.

It is a self-evident property of any distributed system that the nodes are not necessarily located at the same place. Less obvious is that type of nodes each system consists of can vary strongly. The heterogeneity of nodes can even be a defining property, some forms of distributed system exhibit (for instance grid computing) [GBKS08, p. 26].

In technical terms, the two aspects mentioned above translate into three requirements – transparency, scalability and openness [TS07, p. 5]. First of all, transparency means that the complexity of distribution is hidden by the setup from the user. Several different forms of transparency for distributed systems are distinguishable – Table 1.1 gives a short summary and explanation of them. Which types of transparency are of importance for the specific system and the degree of transparency that is required, depends strongly on the application. It often is necessary to find a balance between the amount of transparency that is employed and the performance of the system.

The second important goal of a distributed system is scalability. Scalability describes the ability of a system to adapt to changing conditions without degradation of the offered service. A scalable system can cope with adding or removing of users and/or resources (scalability concerning size), the service it provides does not change with distance between user and resource (geographical scalability) and it is resilient to fluctuating numbers of organisations that use the system (administrative scalability). The different types of scalability are closely related to certain types of transparency, for instance replication transparency is necessary to enable resizing of a system.

The third and last figurehead is openness. Essentially it implies that the components of a distributed system exhibit well defined interfaces in order to enable interoperation, portability and easy extensibility [TS07, p. 5].

The predominant communication scheme used for distributed systems is the client-server model [GBKS08, p. 105]. Two distinct functions exist in this model – servers and clients. Servers basically provide a service or data to the clients, while these use the service to complete their specific task. An arbitrary number of independent clients and servers can be deployed on any node – it

Table 1.1: Different types of transparency in a distributed system [TS07, p. 5].

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may have moved to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recover of a resource

is even possible to run a client and the server it connects to on the same node. Communication is always initiated by the client and follows a question-answer sequence. The client requests a service (data or a calculation, among others), the server delivers the result and subsequently the client uses the response for its own means.
This principle does have its merits [GBKS08, p. 118]. The partitioning in client and server parts naturally favours concurrent processing on the client side. Furthermore, keeping data consistent on the server side is easy. On the other hand, there are several severe downsides that come hand in hand with the advantages. The server is a bottleneck in many ways. Obviously it has to bear the main load of the system, as it is responsible for a number of clients but like every computer system it has only limited resources. In addition to the available system resources, the connection to the server is a further bottleneck that must be taken into account. Beside the performance issues, a server represents a single point of failure. Malfunction at this point will render the connected clients useless and endanger the availability of the whole system. Closely related is the fact that a server is a single point of attack too. An adversary that gains access to a server poses a thread on several levels. Malicious actions can compromise the system stability and the central presence of data on a server represents a security issue as well, as all data is available at a single point.
It is possible to reduce the impact of some of the issues mentioned above, for instance by replication of the servers. This measure enables deployment of load-balancing techniques and provides the means to apply fault tolerance schemes. While definitely an improvement, replication cannot solve the issue of being a single point of failure [GBKS08, p. 118]. In order to remove the weak spot it is necessary to leave the client-server scheme behind and move towards a more decentralised system. One way to go are service-oriented architectures (SOA), which distribute the services evenly on all computers in the system and circumvent many of the aforementioned problems. Other possibilities are cluster computing, peer-to-peer computing and grid computing [GBKS08, p. 120]. All three are related in many ways but each of them exhibit a number of distinct features and areas of use as well.

- **Cluster Computing** – A computation cluster connects a number of independent computers with a high-speed communication link. It presents itself to the user just like a uniprocessor system, while employing massive parallel processing for handling of a task [GBKS08, p. 415]. The technology was born out of the need for higher computing performance, memory size and data throughput in time consuming problems that arise especially in the natural sciences, physics, medicine and climate research.
  While the idea itself is actually not new – the first cluster computer appeared in the early 1980s – only the decrease in cost and the massive increase in performance of standard computers were ultimately responsible for the success of this technology. Nowadays clusters

consist of a number of standard computers that are connected by a network (usually Ethernet).
The actual performance of a cluster is basically unlimited and only depends on the performance of the used computer systems, the maximum data rate of the employed networking technology and the overhead the system requires for management and maintenance.
Several different types of cluster have been established – the three most important are *Load Balancing Clusters*, *High Performance Computing Cluster* and *High Availability Cluster*. A *Load Balancing Cluster* attempts to distribute the demanded load evenly on all associated computers. *High Performance Computing Clusters* mainly aim at optimal parallel processing of a given task and *High Availability Clusters* improve the availability of the system as a whole, by monitoring the single computers for failures. In case of a malfunction of one of the machines, all corresponding processes are transferred from the original computer to a different, active computer of the cluster.

- **Peer-to-Peer Computing** – This is meant only as a short introduction on peer-to-peer (P2P) systems, a more detailed description can be found in Chapter 2.
P2P systems consist of a large number of nodes that are (ideally) equal in every aspect. The nodes are connected and share a certain set of resources between them [TS07, p. 44]. The concept of connecting computers in a peer-to-peer fashion is about as old as personal computers themselves but the current understanding of the term is only a few years old. It has its origin in the P2P network Napster [7] that was created by Shawn Fanning in 1999 to enable easy sharing of music files. On Napster followed a flood of other P2P protocols, pursuing various persuasion and nowadays peer-to-peer is responsible for a major part of Internet traffic [SW02].
The single nodes in a P2P system are organised in a so-called overlay network, a structure that sits on top of the communication network (usually a local-area network or the Internet). The overlay does not necessarily have anything to do with the underlay network – on the contrary, especially the protocols that were developed early on ignore the used communication network completely (what potentially leads to a huge overhead).
Usually P2P systems are divided into structured and unstructured systems. The difference between the two forms is that nodes in an unstructured P2P network have no global view on the network, while nodes in a structured P2P network build a global structure and use it for routing and lookups.
These days a large number of P2P systems exist and though every single one of them follows its own path, all have a few things in common. The main focus is always on managing a large number (up to several million) of unreliable nodes. As a consequence thereof, only few assumptions are made concerning the reliability of single nodes. Furthermore, even while the configurations might vary, the main parts of the nodes are standard home computers.

- **Grid Computing** – Grid computing emerged out of the need to keep up with the changing scientific community that is moving towards location-independent, international cooperation's [GBKS08, p. 435]. In a certain sense it represents the logical, location independent improvement of cluster computing. Although these two types of systems have a number of things in common, an equal or even greater amount of dividing properties exist.
While clusters are usually placed in one geographical location, one of the defining factors of a computation grid is the geographical distribution. Clusters are constructed mostly homogeneous in respect to hardware and software – grids consist of a largely heterogeneous set of resources, which might include computers or computer systems (even clusters), databases, storage facilities, sensors, highly-specialised scientific instruments, among others [TS07, p.

18]. Each grid now consists of an associated set of resources that is connected by any available communication infrastructure (usually local-area networks and the Internet).
Contrary to the components of a cluster, the individual subsystems of a grid as a whole do not belong to any single organisation but each one is owned and managed by its own institution (universities, companies and similar entities). The grid and its member organisations then are organized in so called virtual organisations (VO), which consist of a defined set of resources and a group of people or institutions. The members of each VO are able to use the assigned set of resources to further their goals.

There exists a close relationship between grid computing and peer-to-peer computing as well [FI03]. Both focus strongly on sharing resources among their members, but while grids originate in the scientific community, P2P is a phenomenon that more or less emerged from the Internet. The respective communities are not the only dividing factor. Another factor is the size of the network each technology maintains. Grid computing manages networks that span a maximum of a few thousand nodes, P2P networks on the other hand can grow up to several million participants. There exists a big divergence concerning reliability – while resources in a grid are considered as more or less always on, the opposite is the case for nodes in P2P systems. Other relevant issues are the type of service each provides, trust and scalability.
Even while these differences exist, some voices see a convergence of the two technologies towards each other [FI03].

1.2 Context: The IRON Project

Current goals of energy and environmental politics [Kup08, p. 13] – like reduction of CO_2, increased independence of EU electricity production from imports of primary energy sources and reduced usage of fossil fuels – advance a change in energy production. The traditional producer-side structuring of the electric power grid, with a small number of large power plants, is more and more replaced by an increasing number of decentralised production facilities. As the current control mechanisms of the system are not designed for the altered structure, this change leads to reduced efficiency of the whole power grid. In order to counter the degrading efficiency of this new topology integration of all participants of the energy cycle is essential.
Integrate Resource Optimisation Network (IRON) is a project realised by the Institute for Computer Technology of the Vienna Technical University. It aims at the aforementioned issues – adaption of the power grid to the changing grid structure and improvement of the overall efficiency of the power grid – by involving the user side of the network in addition to the producing side.
Currently communication between energy user and energy producer is more or less non-existent. In most cases the only interaction of the two sides is limited on reading out the power meter once a year. The actual load of each household is then calculated, using generic profiles that estimate the energy use. This arrangement is very inefficient for both sides. At times of peak-load, the producers have to use expensive forms of energy production in order to fulfil the current demand. On the other hand, the consumers have to pay higher prices for the energy they spend. Better communication could cure this deficiency by giving the user side the means to modify their habitual use of energy. For example, at times of peak-load one could shift unnecessary tasks to times of lower load of the power grid. This process is usually known as demand side management (DSM) and requires highly-distributed communication infrastructure.

The IRON project attempts to use DSM for this purpose [Kup08, p. 13]. It is divided into three parts, IRON study, IRON concept and IRON implementation. The first two parts are already finished and smoothed the ways for a possible field test in IRON implementation.
Demand side management [Kup08, p. 94] does in no way aim at reducing the consumption permanently; it only attempts to delay it until the load on the power grid is lower and as a result distribute consumption more evenly. DSM can be used for several different tasks. One possibility is to provide control energy to the grid and thereby reduce the need to operate power plants at lower efficiency. On the other hand, simply lowering the consumption and with it the stress on the power grid at times of peak-load, reduces the need for expensive peaking power plants.
Management of loads requires knowledge of two factors. The current level of stress on the power grid is the first and a defined point of action is the second one. The current load of the power grid is freely available to every consumer at every point of it, by monitoring the deviation of the grid frequency. It is inversely proportional to the load, so a high stress of the grid results in a reduction, while an excess of available energy is indicated by an increase of the grid frequency. Load shifting in the IRON project is archived by switching virtual electric storages between pre-defined states. The points in time, the virtual storage is activated or deactivated, depend on the currently measured frequency of the grid and the state of the other storages in the DSM network. Virtual electric storages are systems that have the ability to store energy – usually as some other form of energy, like thermal energy – but in contrast to real electric storages they can not feed the energy back to the power grid. Examples for virtual electric storages are rooms with heating or cooling facilities that keep the temperature of the room constant over a certain time, even if the heating/cooling unit is not active. In the IRON project a simple controller is used, that provides three states (the virtual electric storage in combination with the controlling device is called a node from now on). This amounts to a behaviour that is similar to the one displayed in Figure 1.1. In this model, the three states are defined by a certain constant temperature each. Usually the

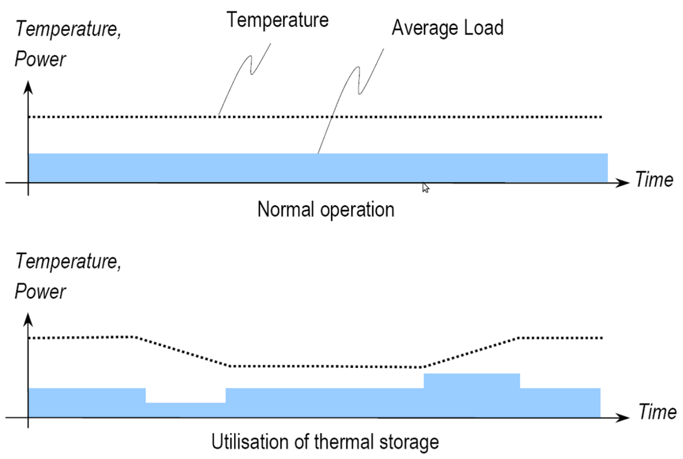


Figure 1.1: Usage of a virtual electric storage [Kup08, p. 96].

virtual storage is in the middle state and the room has the corresponding temperature. At times of high load it is switched to the lower state and reduces the average load of the grid by a certain amount. During this state, the temperature in the room sinks to the pre-defined temperature for this state and remains there. After the unwanted situation is over, the state is changed to the higher state and the room is heated, until the loss in temperature is compensated. As soon as the wished for temperature is reached, the storage is switched back to the middle state [Kup08, p. 94].

Every node measures the deviation of the grid frequency locally and changes its state according to the frequency and a certain second factor that depends on the system as a whole. The reason for this additional influence is that uncoordinated activation of nodes will likely have a negative effect. Assuming that the considered DSM system does manage a relevant storage capacity (if not, the influence of the arrangement will be negligible), uncoordinated activity of the single loads probably will lead to a reduction of load beyond measure and rather increase the stress on the power grid than lower it.

The issue mentioned above is only one of the reasons that demand some form of interaction between the nodes, used for load shifting. Considering the nature of the virtual storages it is conceivable that a single one probably is too small to make any difference – so the activation of several nodes has to be coordinated. Parallel and serial linking comes to mind as possible combinations. Parallel modification of virtual storages provides an increased capacity while serial modification leads to an increased duration. For efficient DSM probably both types are required.

In the IRON project, nodes calculate a system wide storage energy S that represents the sum of all energy parts, provided by the individual storages [Kup08, p. 103]. The value of S is proportional to the grid frequency and used to coordinate the nodes. In order to organise this process, each individual node has a certain pre-assigned activation level, which indicates the value S must reach in order cause an activation/deactivation of the connected virtual storage. In Figure 1.2,

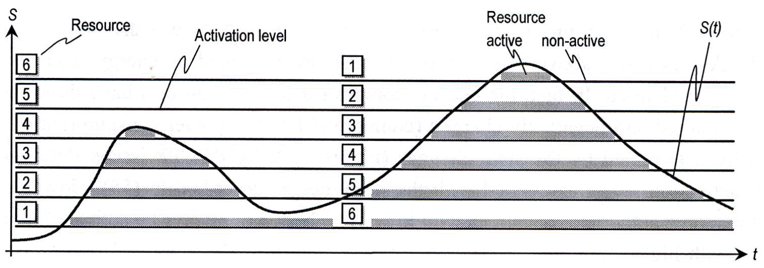


Figure 1.2: S is the sum total of all individual storages. The function is divided into activation levels – each node is activated if S exceeds the corresponding level [Kup08, p. 103].

the system consists of six storages with the levels 1 to 6. Each of the nodes is activated as soon as S exceeds the corresponding level and deactivated the moment it moves below it.

This form of management leads over time to a certain imbalance regarding the number of activations each virtual storage has to carry out. Nodes with a lower level are activated more frequently compared to nodes with a higher level. A good solution for this issue is rearrangement of the activation levels in regular intervals. A simple form of reordering would be to reverse the levels, as illustrated in the right part of Figure 1.2. More complex forms of modification of the activation level require some form of communication, either with a central server or between the nodes. Currently a Java-based IRON server is used for distribution of activation levels and collection of data from the nodes.

Management of the virtual storage requires a controller that offers the functionality mentioned above – either integrated in the storage itself or as an external device. One of the results of the IRON project is the IRON Box [Kup08, p. 106] that implements the means for demand side management in a compact device. The box basically controls the virtual storages depending on the assigned activation level. In addition to this base functionality it has the capability to collect data from external sensors and it is able to connect to the Internet by using a wireless connection. A detailed description of the device can be found in Chapter 3.

1.3 Problem Analysis

As suggested in Section 1.2, the current structure of the IRON sensor network is that of a classical client-server system. The IRON Boxes connect to a server and receive their current activation level from it. This scheme does have its advantage especially in the ease of communication. All information that is needed for the distribution of the level is available on the server. The only information an IRON Box requires for obtaining an activation level, is the server address. Furthermore a central server enables easy control of the connected nodes and simplifies data collection from them. The two downsides of this approach are that a server always represents a central point of failure and has limited resources. In order to improve fault tolerance, the IRON project utilises not a single server but a group of functionally identical servers. At all times only one of the servers is active and handles all incoming requests, while the passive servers only update their data structures. If one of the servers fails an election algorithm determines the next active server. This scheme provides a certain degree of fault tolerance but is tied to additional costs. The issue of scalability is not addressed by this approach.
An alternative to this centralised approach is to connect the nodes in a decentralised manner, as discussed in Section 1.1. While there are a number of other possibilities for decentralised systems introduced in Section 1.1, in this work a peer-to-peer protocol is to be used to connect the IRON Boxes and provide the means to organise the distribution of activation levels. Considering the properties of the IRON network, structuring the network in a peer-to-peer fashion presents itself. This way there is no dependence on any central entity and as such no central point of failure. Additionally, in this context, a decentralised set-up is inherently scalable and does not cause additional costs, as only the existing nodes are used.
In general, several different aspects have to be considered in the selection process of the P2P architecture. Some relate to the used hardware and others originate from the application. At first the comparatively low hardware performance of the device must be taken into account. The IRON Box uses an Atmel ATmega128 microcontroller [Atm08] with 4 kbyte RAM, 4 kbyte EEPROM, 128 kbyte FlashROM and a maximum clock frequency of 16 MHz. It does not use the maximum clock frequency but is configured for 8 MHz. The microcontroller is not equipped with any built-in network connection. As a consequence an external networking module is needed. It is connected to the microcontroller by a serial connection with a data rate of 38400 baud. All three available networking modules offer only wireless access to the Internet. The three different modules are the Sony Ericsson GR47 GSM module, the Avisaro WLAN Modul 1.0 and the Avisaro WLAN Modul 2.0 [10]. The Sony Ericsson GR47 and Avisaro WLAN Modul 1.0 each support one concurrent TCP/UDP connection, while the Avisaro WLAN Modul 2.0 offers up to twelve sockets for simultaneous TCP connections and in addition 6 ports for UDP connections. The latter module was used in this work, due to its higher flexibility and the higher number of possible concurrent TCP connections.
Regarding the network performance, only moderate demands are imposed by the IRON protocol, as its main task is distribution of the activation level. An update assigns a new activation level to every node in the network. During the updates, only a few short messages with a length of a few bytes have to be transferred. As a result, every online node in the network actually has to be reachable too, to enable this process to work properly.
While updates of the activation level only have to be performed in an interval that spans several days, daily communication between the nodes is necessary, in order to ensure synchronisation of the individual clocks. Between these updates an online connection is basically not necessary [FK07].

The presented problem now leads to a number of requirements on the peer-to-peer protocol – these are subsequently explained.
First of all, the chosen solution must be a decentralised system. While this looks like a trivial demand in the context of peer-to-peer systems, it is not the case (more of this in Chapter 2). The next requirements correlate to the weaknesses of a centralised approach, namely fault tolerance and scalability. Of ultimate importance is that the failure of a single entity must not interrupt communication between the remaining nodes. For this reason, the protocol has to sustain the ability to route messages even while facing fluctuations of nodes. While scalability is only of secondary importance, a certain demand on the possible network size is present anyway – the protocol implementation must support a network size of at least one thousand nodes.
The second source of demands on the protocol stems from the limited performance of the IRON Box. It is a magnitude or more below usual peers in a P2P system – P2P protocols are mainly used on x86 computers, with a clock frequency of several hundred megahertz and several hundred megabyte of memory. Especially the small amount of working memory leads to the necessity of few and small data structures. Related to the low clock frequency and the low maximum transfer rate of the serial connection is the number of messages that are necessary for network set-up and network maintenance. No exact number can be given for this item but the issue must be taken into account anyway. The last limitation, imposed on the chosen P2P protocol by the hardware, is the number of network connections that are available in parallel at any point in time. Twelve concurrent TCP and six concurrent UDP connections must not be exceeded.
Finally, there are two demands that originated from the IRON protocol. Number one is that the created system, with a high chance, stays in one piece (low chance of network splitting) in order to enable a continuous distribution of activation levels. For the same reason, every active node in the P2P network must be reachable from every other node at all times. In case that more than one protocol meets the discussed requirements above, the least complex is chosen.
The requirements in decreasing relevance are

1. Decentralised system – no single point of failure
2. Ability to reach every active node in the network
3. Demand for a maximum of twelve concurrent TCP and six concurrent UDP connections
4. Demand of less than 4 kbyte RAM, 4 kbyte EEPROM and 128 kbyte FlashROM
5. Low chance for splitting of the network
6. Possible network size of 1000 nodes

The project is divided into three tasks that build upon each other and are explained in the following.

1. **Analysis and selection of a peer-to-peer protocol** – The landscape of available P2P protocols has to be explored and examined concerning possible candidates. A number of alternatives must be collected and documented as prerequisite to a detailed analysis in respect of the aforementioned requirements. Finally, the candidate that is best suited is chosen for further work.

2. **Implementation and test of the selected protocol** – A software design for the selected protocol must be created and subsequently implemented for the IRON Box. In addition, a software tool must be created that provides the ability to test network set-up, adding/removing of nodes and routing of messages. For this work only one device with Avisaro WLAN Modul 2.0 was available.

3. **Simulation of a large P2P network** – A simulation model of the implemented P2P protocol for Omnet++ must be created and then a network of at least one thousand nodes must be simulated and analysed.

After the general introduction in this chapter, the work proceeds as follows. Chapter 2 is dedicated to a comparably new form of distributed architectures – peer-to-peer systems. At first a short introduction on P2P networks and an outline of their short history is composes in Section 2.1. This is continued by Section 2.2 that holds a presentation of different types for classification of P2P. Building on these information, a number of P2P protocols, which could serve as solution for the present problem, are introduced in Section 2.3 and finally Section 2.4 an analysis of the aforementioned protocols in respect to the requirements is concluded. The following Chapter 3 is focused on the existing distributed system and the design of the chosen P2P protocol. At first the existing system is reviewed in Section 3.1. Following after is Section 3.2 that explains the used implementation design of Symphony, the chosen P2P protocol. Section 3.3 concludes the chapter. It elaborates on how the implementation was evaluated, considering that no resources for the setup of a large P2P network were available. The next chapter deals with the simulation of a large peer-to-peer network. It consists of three parts. Section 4.1 presents Omnet++, the simulation framework that was used. In Section 4.2 the simulation itself is discussed and finally the acquired results are displayed and analysed in Section 4.3. Finally, Chapter 5 contains a summary of the work, followed by suggestions for future improvements.

2 Peer-to-Peer Systems

In Section 1.1 a general introduction on distributed systems was given and cluster computing, peer-to-peer (P2P) computing and grid computing, which represent three important technologies in this field of research, were presented. While all three are possible ways to enable efficient distributed computation, each one aims at a different area of application. The present chapter enlarges upon this topic while considering the application explained in Section 1.2. The energy management system, explained in the aforementioned section, currently uses a client-server approach, suffering from the usual, basic weaknesses this scheme inherits – constitution of a single point of failure and limited scalability acting as the most prominent. In respect to this application that is characterized by the IRON project and the issues it displays related to the use of a client-server structure, a P2P protocol has to be found, in order to employ it on a low-performance embedded computer and as a result benefit from true decentralisation. With this in mind, the current chapter provides the required background by extending Section 1.1 with a more detailed presentation of peer-to-peer systems. Adjoining the theoretical background is the discussion and choice of the P2P protocol used in the remaining chapters. It is to note that P2P system, P2P network and P2P architecture are understood as the same thing, as the difference is of no relevance for this work.
The chapter is divided into four sections that take a direct line towards the peer-to-peer protocol that is implemented in the following Chapter 3. Section 2.1 introduces P2P and its background. Furthermore it outlines the short history of this technology and gives a peek on P2P networks that are widely used today. After that a number of possible classifications for P2P systems are presented and subsequently discussed in Section 2.2. Of particular importance is the differentiation between structured and unstructured architecture, which also is the best known form of classification. In consequence of distinct properties these two forms of peer-to-peer architectures offer, unstructured systems are discarded and only structured protocols are examined further. Section 2.3 continues along this line and presents a number of structured P2P protocols that have the potential to serve as candidate for use in the rest of the thesis. As closing of the chapter serves Section 2.4, which analyses the protocols discussed in the previous section in respect to the requirements in Section 1.3 and elaborates on the choice of the peer-to-peer protocol Symphony.

2.1 Introduction and History of Peer-to-Peer Networks

A peer-to-peer network is a form of network that consists of equal nodes that share resources among them. While P2P systems, along the line of file sharing networks like Napster [7], entered

the spotlight only a few years ago, the idea to connect computers in a peer-like fashion is not that new as public believe suggests. The first computer networks of the pre-Internet age exercised exactly that, they connected computers as equal partners (without separation in clients and servers), in order to share resources like printers throughout the network [GBKS08, p. 1]. This required administration of access rights on every single machine, which severely limited the performance of these early systems. New approaches to P2P and a massively increased performance of modern computer systems though made it possible to revive peer-to-peer systems. Other popular P2P-like systems were the Usenet, a bulletin board systems that is one of the predecessors of modern web forums or the ancestor of today's Internet, the ARPANET [SFS05].
The rise of peer-to-peer systems started in 1999 with Napster, that was developed by Shawn Fanning as a tool to exchange music files (at first over the Northeastern University campus network) [SM06]. The network structure of this first P2P system was only marginally P2P like. It was based around a central server that kept an index of all shared files and managed lookup and routing. Only the transfer of requested files occurred in peer-to-peer fashion between the nodes. Napster was shut down in 2001, after a few very successful years, yielding to the increasing pressure from the music industry, concerning legal issues. While Napster itself had far more in common with a classic client-server than a P2P system, it triggered a wave of development, which led to "pure" P2P systems that interact in a completely decentralised manner.
After Napster followed Gnutella [6] that represents the first true P2P protocol. The reason for development of a P2P network without any central entity was the elimination of any point of vantage on the network by adversaries. It actually was successful with this endeavour but at the cost of efficiency. The protocol creates a network in a random fashion. Each node in a Gnutella network connects to a certain number of randomly chosen other nodes. Now, if a node attempts to find any item, it sends the request to its neighbours. The neighbours again forward the lookup to their neighbours until the item is found or the maximum number of hops is reached. This flooding mechanism, applied for lookups and routing in Gnutella, is very bandwidth expensive especially if it is taken into account that the protocol is completely oblivious to the underlying network structure. Furthermore, often items are not found, even if they exist in the network, as every node is only able to search in a limited area around itself.
Shortly after Gnutella, FastTrack and Gnutella 2 [11] appeared, which both feature a third class of P2P topology that constitutes sort of a hybrid network with two type of nodes – supernodes/hubs and leaves. Supernodes connect in a decentralised fashion among each other and are responsible for a number of connected leaves. They maintain an index of all the files on their leaves and process search/routing requests. The difference to a Napster-like system is that no central server exists and in case of failure of a supernode, the leaves simply migrate to a different, active supernode.
After this point in time, the development of P2P systems split in two parts. On the one hand new protocols were still developed by the P2P community – among them BitTorrent [12], one of the most commonly used protocols for file transfer today. Opposing this are the protocols that have their origin in the scientific community, like CAN (Content Addressable Network) [RFH+01] or Chord [SMK+01]. In the last few years the hype cooled noticeable and relevant new developments of peer-to-peer systems are rare. Nonetheless, P2P earned its place among the "Internet technologies" and is definitely here to stay.

As mentioned above, a P2P network usually consists of a large number (up to several million) of nodes that are connected and share some form of resource. In the optimal case of a pure P2P system, theoretically all connected nodes are equal (making them peers – the terms "peer" and "node" are understood as the same thing in this text). The single peers have the means to

be a server and a client at the same time (a so called "servent" from SERVer + cliENT [TS07, p. 43]). Building distributed systems in peer-to-peer fashion has several positive aspects. First of all, these systems improve on the weak spots of the classic client-server architecture. Server in a client-server architecture always represent a central point of failure – if the server fails, the system goes down with it. Even while there are possible improvements (Section 1.1) it is impossible to eliminate this weakness completely without abandoning the client-server approach [GBKS08, p. 118]. A P2P architecture simply avoids this issue, as per definition no central point exists. Additionally, a basic assumption every peer-to-peer system makes, is that nodes can fail at any time, without prior notice and are in general not very reliable. The second representative weakness of the client-server scheme is scalability in its many aspects. Scalability of a client-server system mainly depends on the performance of the server and on the maximum data rate of the network connection of the server. While naturally there is a limit to the performance of any node in a P2P network too, the load here is spread on many nodes more or less equally. One must not forget though, that this are very general considerations and that the performance of a specific peer-to-peer system in respect to these two issues strongly depends on the used protocol [GBKS08, p. 118].
The notion of nodes being equivalent is intriguing but reality paints different picture. Peer-to-peer is only little more as a label, put on different architectures which actually do not have many properties in common, except not being pure client-server systems. Various definitions range from

> " ... totally distributed systems, in which all nodes are completely equivalent in terms of functionality and tasks they perform ... "

[ATS04], to definitions like

> " ... peer-to-peer is a class of applications that take advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet ... "

[13]. The latter not only encompasses true decentralised systems but stretches as far as the aforementioned Napster or SETI@home [8], which are both more or less client-server systems with only a pinch of P2P communication thrown in. Even barring grid computing, that bears a close relationship to P2P computing, one has to step carefully in talking about P2P.
Every P2P network consists of peers, which are organised in a so-called overlay network [ATS04]. That means the P2P network resides on top of the "real" network (mostly the Internet or local area networks) and uses its infrastructure. The topology of the overlay as well as the placement of nodes in it, are both defining factors for any P2P protocol. They range from completely random, like the structure constructed by Gnutella [6], to well structured topologies alike the ring created from Chord [SMK$^+$01], and encompass everything between. Additionally, the topology and placement of nodes is independent of the position in the underlying network – Figure 2.1 displays a possible network, in its two layers. This eventually leads to a huge overhead in the lower network layers, as for every message between two peers eventually a number of nodes in the underlying network have to be bothered, if it is not taken into account by the protocol [SW02]. Unfortunately every early architecture, like Napster or Gnutella, and most of the newer ones, (Chord, Kademlia [MM02]) create the overlay without any serious consideration of the underlay infrastructure (actually this offers a hint why P2P networks are responsible for a major part of Internet traffic today). Opposing this negative examples are some promising newer protocols, especially Pastry [RD01] and Tapestry [9] that take network locality into account.

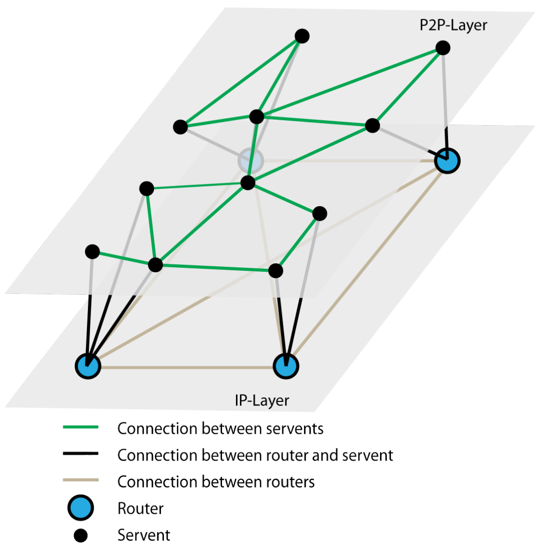

P2P-Layer
IP-Layer
Connection between servents
Connection between router and servent
Connection between routers
Router
Servent

Figure 2.1: Layered Structure of a P2P network. The upper layer represents the overlay network that is created by the P2P protocol. Every peer in the overlay is connected to an entity in the underlying infrastructure [ESZK].

As explained earlier on, file sharing was the main driving factor in the development of P2P networks. Since then many various areas of application for peer-to-peer technology have been found. Androutsellis-Theotokis [ATS04] lists following categories:

- Communication and Collaboration
- Distributed Computation
- Internet Service Support
- Database Systems
- Content Distribution.

Content distribution (file sharing) is still the most widely used application of peer-to-peer systems in the Internet and responsible for a major part of Internet traffic today [SW02]. Despite this imbalance concerning usage, the other forms of application are of no less importance. A well known example for a P2P based communication system is the voice chat Skype, which is used by several millions of users daily.

2.2 Classification of Peer-to-Peer Networks

Peer-to-peer is a very broad term that is applied to a multitude of different systems, which have only a few properties in common. While there are many distinct P2P systems, it is possible to find characteristics that some share. There are actually a number of different ways to categorise P2P networks, each focusing on other aspects of the technology. In this section three interesting ways to classify P2P architectures are presented.
The first possibility is to divide them hierarchically into three levels, depending on the system

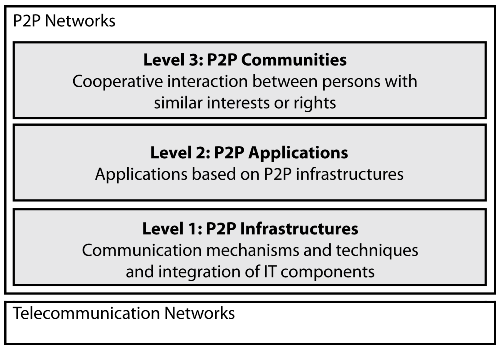

P2P Networks
Level 3: P2P Communities
Cooperative interaction between persons with similar interests or rights
Level 2: P2P Applications
Applications based on P2P infrastructures
Level 1: P2P Infrastructures
Communication mechanisms and techniques and integration of IT components
Telecommunication Networks

Figure 2.2: Levels of P2P networks [SFS05].

level. Schoder et al. refer to the levels as P2P communities, P2P applications and P2P infrastructures, as is displayed in Figure 2.2). On all three layers the term "peer" is used, but with different meaning. A peer in P2P applications and P2P infrastructures represents a technical entity; in a P2P community it is used for a person that is part of the community.
P2P communities are communities of interest, rather than location. They are concerned with social interaction between the users of a common interaction platform. In order to enable communication and collaboration between the members of a P2P community, peer-to-peer applications are employed. P2P infrastructures finally act as common base for usually different P2P applications. They provide services to applications, that include communication, integration and translation between different IT components, management of resources, as well as security processes [SFS05]. Usually the terms P2P networks and P2P infrastructures are used equivalently.
In the early days of P2P, no clear separation of P2P networks along these lines was possible. Systems like Napster [7] or Gnutella [6] do not explicitly define infrastructure, application or community. Meanwhile the situation has changed and many distinct P2P infrastructures can be identified. Well known examples are CAN [RFH$^+$01], Pastry [RD01] and Kademlia [MM02], which all have their origin in the scientific community. Especially Kademlia is used in many current P2P file sharing clients like eMule [1].
P2P communities and applications are of no relevance in this work and all further considerations, including the subsequent forms of categorisation, aim mainly at P2P infrastructures, even if it is often impossible to clearly differentiate.

Division by system level is a rather general form of categorisation. Three other ways are introduced here that relate to certain properties of the inspected P2P systems. The traits that are taken into consideration are degree of centralisation, structure and routing/locating mechanisms [ATS04].
First of all, there is the degree of centralisation. It specifies if the P2P network does rely on one or more central entities and to what degree. Next is the topology structure of the overlay network that describes the amount of order a P2P architecture does display. These two characteristics allow a general classification of all existing P2P networks. Routing of messages and locating of information on the other hand, is very specific and one of the most important peculiarities of each single architecture.
The remaining part of this section presents the two more general forms of classification. At first, division by the degree of centralisation is discussed, followed up by partitioning in respect to structure. These two topics are the premise to Section 2.3, which introduces the most important structured architectures and enlarges further on the routing and locating mechanisms these apply.

2.2.1 Degree of Centralisation

Taking degree of centralisation as the defining factor, leads to three well defined forms of architecture [ATS04]:

- **Hybrid Decentralised Architecture** - This form of P2P architecture is the oldest of the three and is exactly the architecture of the Napster network [7]. It is related closer to a classical client-server system as to any decentralised system. The basic structure is that of a client-server system (see Figure 2.3) with a central server and clients that connect to it. The server keeps an index of all the resources that are found on the connected clients. It coordinates the interaction between them and is responsible for lookups. Only a minor part of the functionality, for instance data transfer, is done in a peer-to-peer way, between the nodes. This approach is much easier to implement as any decentralised way – the server has all information that is relevant for lookup and routing. Decentralised systems in contrast, can only ever see part of the picture and need mechanisms to work around this lack of knowledge. The downside though is the same as for every client-server system. The server is a single point of failure and prone to all the problems that arise from it. The probably best known representative of hybrid decentralised architectures is Napster but many other P2P applications follow the same line.

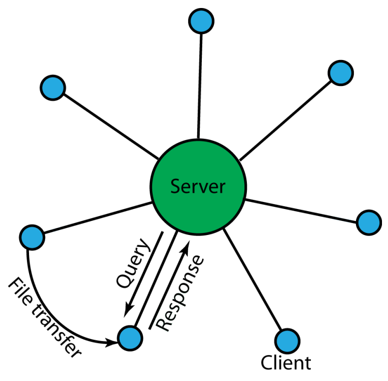


Figure 2.3: Hybrid Decentralised Architecture. A central server maintains an index of all resources on the connected clients and is responsible for coordination of the peers. Only, for instance, data transfer is performed in peer-to-peer fashion [ATS04].

- **Partially Centralised Architectures** - In a partially centralised architecture no central server, and so no central point of failure, exists. Figure 2.4 shows the typical structure of this form of P2P network [ATS04, ESZK]. There are at least two types of nodes – supernodes and leaves (peers) connected to them. Supernodes, the first kind of node in partially centralised architectures, play the same role as the server in a hybrid decentralised architecture. They keep an index of all the resources that are found on the connected leaves, process incoming lookups and are responsible for routing. Unlike a central server, supernodes are easily replaced by other nodes. In case a supernode goes offline its leaves simply establish connections to other supernodes. The issue of a central point of failure does as such not exist in this form of network and scaling of the network is possible too, as the number of supernodes – in limits – is easily increased. This architecture was introduced by FastTrack but many others, among them the well known P2P framework JXTA (Juxtapose) [OTG02], adopted the approach.

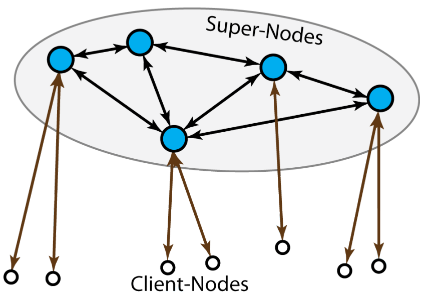


Figure 2.4: Partially Centralised Architecture. Leaves (peers) connect to supernodes, while supernodes establish a, usually decentralised, P2P network among them [SM06].

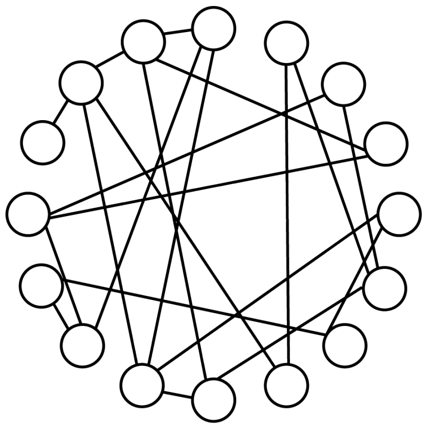


Figure 2.5: Possible structure of a pure decentralised architecture. In this case the peers are connected in a random fashion, alike the Gnutella network. Other possible forms are displayed in Section 2.3 [SM06].

- **Pure Decentralised Architectures** - A pure decentralised architecture is the archetype of a P2P architecture. It takes node as peers and so as equal in every way. The fashion in which the peers are connected depends altogether on the applied P2P protocol and ranges from totally random to very well structured (Figure 2.5) [SM06]. Considering the decentralised overlay structure, a single point of failure can not exist by definition. This does not mean that this form of architecture does not have its issues. Fault tolerance, scalability, stability, the amount of traffic that is caused and many other important characteristics, depend strongly on how the network is organised. For instance, the biggest issue with Gnutella – it was the P2P architectures that used this scheme – is the huge amount of traffic that is caused by lookups and routing (due to flooding of messages to neighbouring nodes).
  The pure decentralised architecture is the most important alternative for this work, as all structured architectures naturally belong to this category.

2.2.2 Topology

Division of P2P networks by structure is another possibility. There are three different types of topology. Two of them are distinct types – unstructured and structured – and the third – loosely structured – is a hybrid of the other two. The basic difference of the two forms is that the connections in unstructured networks, in contrast to the process in structured networks, are created ad hoc without following specific rules [ATS04]. Loosely structured networks stand a little at the side and exhibit properties of both other types. Subsequently, the characteristics of each type are analysed in detail.

- **Unstructured** – Unstructured P2P networks are very common. For all architectures mentioned in Section 2.2.1 exist a number of P2P networks that exhibit an unstructured topology [ESZK]. Every single one of the first P2P protocols belongs to this category. The defining property of an unstructured topology is that they generally have no global view of the network. A node itself only "sees" a certain number of nodes around its position in the overlay and is only able to contact the nodes within its "virtual horizon" directly. In order to locate nodes or resources that are outside of this area, it has to contact the neighbours with the request in a random fashion. The basic mechanism is flooding – the request is sent to all of the neighbours and forwarded again by them to all their neighbours, until the item is found or an exit condition met. Newer protocols often replace flooding by related mechanism, like random walk [ATS04]. One thing all of these mechanisms have in common is that there is no guarantee to find the searched-for item, even if it exists in the network. Concluding, unstructured P2P networks have a relatively low scalability, availability and persistence. On the other hand, they are much better in dealing with high fluctuation of nodes compared to structured networks.

- **Structured** – Contrary to unstructured P2P networks, structured networks are by definition only pure decentralised architectures [ATS04]. The greatest aim of the first peer-to-peer protocols that used this topology was to improve scalability from unstructured networks. However, it is not the only positive aspect of these systems. As the name suggests, networks of this class have a well defined topology – every node and every resource has a fixed, predefined spot on the overlay. This usually is archived by building a DHT (Distributed Hash Table) that maps a certain feature of a node/resource to an id. The identifier then belongs to a specific position on the topology [SM06]. A positive aspect of this – and one of the most important features for this work – is that every single node or resource is found, if it exists in the network. Nonetheless, there is no light without shadow – structured P2P networks of course have a number of weaknesses too. For one thing, highly transient node populations are difficult to handle for most structured protocols. It is hard to maintain the structure and keep up efficient routing while facing constant joining and leaving of nodes. For another thing, maintaining the rigid structure of a structured P2P network is far more expensive than keeping up the more or less random topology of unstructured systems.

- **Loosely Structured Networks** – The last topology type are loosely structured networks that combine properties of both types of networks from above [ATS04]. First of all, they always are pure decentralised architectures, just like structured P2P networks but unlike them, loosely structured P2P networks can only approximate the location of an item in the network. This makes the use of these networks less expensive, as only the relevant parts of the structure must be contacted, but it naturally is not as efficient as real structured networks.

2.3 Selected Peer-to-Peer Protocols

In Section 1.3 a number of requirements are listed that must be taken into account in the selection process of the P2P protocol. A demand the IRON protocol forces onto the system in question, is the ability to reach any online node by every other node in the network. While it doesn't appear like a very strict constrain, the requirement effectively bans all unstructured P2P systems from the pool of choices. The reason for this limitation lies in the none-existence of any order – the item of interest can be anywhere in the network and without doubt only a certain part of it can possibly be searched. As a consequence all P2P protocols explained in this section use a well structured topology and self-evidently are pure decentralised architectures.
There are a number of characteristics most structured P2P architectures share. First of all, the selected protocols assign an id to nodes and keys (that correspond to data items/values). In the text the term key is used for both, the key itself and the id it is mapped to. Nodes either use a randomly chosen id or an id obtained from a hashing function. Items (e.g. files, information and more), on the other hand, are mapped onto the structure by some form of mapping function. Each of them have a fixed place on the topology, which is defined by this id. That implies for items that they do not necessarily belong to any specific node, as it is the case in most P2P applications. With joining and leaving of nodes they eventually move from node to node, depending on which one is responsible for a certain area of the structure at the moment. The mechanism described in the last passage, is called distributed hash table (DHT) and is used in more or less every modern P2P system [SM06, p. 16][GRW]. Figure 2.6 illustrate this arrangement, with the aid of topology

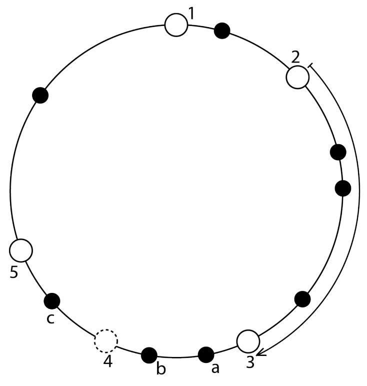


Figure 2.6: A generic ring topology of a structured P2P network.

used by the protocol Chord. The full dots are data items, the circles represent nodes and the dashed circle indicates that node 4 has not yet joined the network. It is a commonly used practice for structured P2P systems that every node manages a certain area of the topology and all the items in it. In this case a ring is used and the nodes and items are mapped onto it. Every node now takes care for the area between itself and its predecessor – for instance, node 2 is responsible for the area spanned by the arrow. Now, node 4 joins the network at the indicated position. Up to this point, the interval between node 5 and 3 is managed by node 5. It now is divided in two parts; the section between 4 and 3 is assigned to node 4 and the remaining piece stays with node 5. The data items a and b that correspond to the new area between 4 and 3, are subsequently moved from node 5 to node 4.
The following subsections present a number of modern, structured P2P protocols, which have the potential to serve as a candidate for implementation. The selection is not exhaustive but was made in order to provide a good overview on what is available.

2.3.1 Content-Addressable Network

CAN (Content-Addressable Network) [RFH+01] was the first structured P2P architecture developed. The idea behind it is to improve the low scalability of decentralised P2P protocols like Gnutella [6] that use flooding (or similar resource consuming procedures) for routing and locating. As a structure, CAN creates an overlay network that spans over a virtual d-dimensional torus (a so called d-torus). A uniform hash function assigns a key to every value (for instance the id/IP address of a node or the name of a file) and the resulting key then is mapped deterministically on the virtual coordinate space of the d-torus (with d the dimension of the torus).
Each node is dynamically assigned a certain area on the d-torus that sum up to the entire virtual space. As explained in the introduction of the section, every key that is created is bound to a

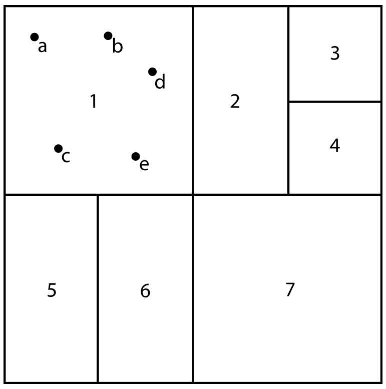


Figure 2.7: 2-d CAN space. The area is divided between the nodes (identified by the numbers in the rectangle) – each node now manages the keys (black dots) in its field of responsibility [RFH+01].

certain area of the structure and as such belongs to the node that manages the area it is mapped to. Figure 2.7 displays an example of how this would be on a 2-d torus. In this example, nodes are represented by a number in the rectangle and each node is responsible for data items (indicated by black dots) in it.
In order to create an overlay network, each node knows the IP addresses of its direct neighbours. Two nodes are neighbours in CAN if they border in one dimension and have overlapping coordinate systems. To illustrate this, consider node 2 in Figure 2.7. The neighbours of each node are the managers of the bordering areas – for node 2 this amounts to the nodes 1, 3, 4 and 7.
The routing information each node in CAN maintains, consists solely of the information about its neighbours. Building on this data, routing works as follows. On a request for a key, the node first checks if it is located in the managed area. If the key does not belong to the node, it is forwarded to the neighbour that is closest in space. This leads to an average path length for routing of $\mathcal{O}\left((d/4)N^{1/d}\right)$ and a growth of the path length of $\mathcal{O}\left(N^{1/d}\right)$. If a node fails it is possible to route around the failed node without any difficulty as there exist more than one route to every node.
As explained before, the information about the neighbours makes up the routing table. Considering this, the size of the table is independent of the number of nodes in the network. The only factor it depends on is the dimension of the d-torus and the number of neighbours. This correlates to a size of about $2d$ for the table.
Besides routing, joining and leaving of nodes are defining properties of a peer-to-peer network. Joining of nodes in CAN is explained, using the example of Figure 2.7. Consider a node that intends to join the network. It needs at least one known CAN node that is online – the protocol provides means to find a node – node 7 in this case. The node in question then sends a join request

for a randomly chosen point in the coordinate space to the known node 7. Consequently, the request is routed by the CAN routing mechanism to the manager of the point in the d-dimensional space – in this case node 1. 1 then splits its managed area in two equally equal parts (by specific rules that enable re-joining of the areas on departure of one of the two managing nodes) and reduces its own area to one of the halves. The other half, including the keys b, d and e that are in it, is assigned to the new node 8 (with all said and done, this amounts to a transfer of the items that correspond to the keys). After node 8 has acquired its part of the d-torus, it receives the necessary neighbourhood information from node 1. Both nodes modify their routing table to the new relationships and inform all neighbours about the change in topology. The result of the process then looks like Figure 2.8. The neighbour state is periodically refreshed in order to learn

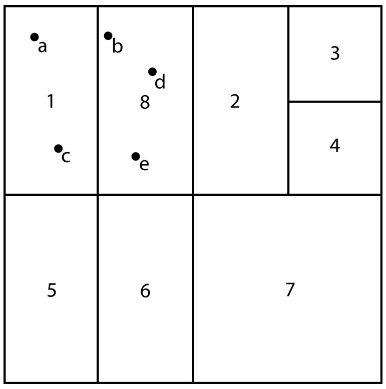


Figure 2.8: Topology of Figure 2.7 after node 8 joined. Node 1 now only manages half of its original area [RFH+01].

about failed nodes quickly. While a node that intends to leave usually informs its neighbours about its departure, it does not really matter as the remaining nodes to learn of it anyway by the periodical refreshes of neighbour states.
In the event of departure or failure of a node, the area it managed is joined with the area it was split of. For instance, if node 6 leaves in Figure 2.8 its part is merged to the area of node 5. After that, node 5 takes care for the merged area. If merging is not possible, the neighbouring node with the smallest area takes on the abandoned area too. It then takes care of both areas until it is reassigned by a special maintenance procedure [RFH+01].

2.3.2 Chord

The P2P protocol Chord [SMK+01] aims at keeping up its operability while facing high node fluctuation. It creates a DHT that maps nodes and data items on the structure, using a m-bit identifier. The identifier space consists of the numbers from 0 to $2^m - 1$ and forms a circle modulo 2^m. In order to create the distributed hashing table, a hash function is employed that uses the IP address of nodes and the a (common) defining feature of data values as input.
Chord uses the scheme illustrated by Figure 2.6 – each node is responsible for the interval between its id and the id of its direct successor on the circle. All keys that map to this area are managed by this node. As so often for this form of peer-to-peer system, keys do not belong to any specific node – quite to the contrary, they stick to the area on the ring, they are mapped to. Joining nodes take over the keys that are in the part of the identifier space that is assigned to it. Likewise, on leaving of a node, all values are signed over to the successor of the departing node.

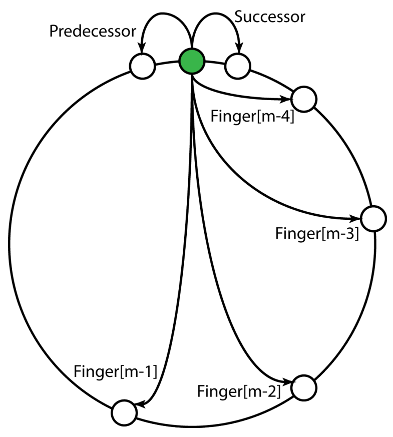


Figure 2.9: Pointer structure of Chord [SM06, p. 28]

Routing in Chord follows a different path as CAN. Figure 2.9 displays the connections each node in maintains – a pointer to its predecessor, a list of its k immediate successors and a table of so called fingers, with a size not exceeding m. The fingers are pointers to remote nodes, on specific positions on the identifier circle. Finger table entries consist of the Chord id, the IP address and the port of the node. The first entry always is the direct successor and the following entries have a distance of at least 2^{i-1} with $1 \leq i \leq m$. This means that the distance between each finger doubles with each step. The finger table is the main source of routing information in Chord.
Every node in the network has only knowledge about a limited number of other nodes and it is informed better about its close vicinity than about places further away. A consequence of this is that no single node is able to locate every id in one step. Routing in Chord is a sequence of unidirectional lookups that hop around the identifier circle. If node A looks for id X, it searches its finger table for the closest id to X and contacts the corresponding node B. Node B again goes through its finger table for the id and returns the closest node C to A, whereupon A contacts C. In every step this process produces nodes at about half the distance between the previous node and the target, until the responsible node is located. That results in a size of the routing (finger) table and an average path length of about $\mathcal{O}(logN)$. It is important to note, that the size of Chords data structures and the number of required connections, are not fixed but grow with the increasing network size.
The procedures required for joining and leaving of nodes are rather straight forward. Imagine, node 14 in Figure 2.9 wants to join the network. Similar to CAN, needs to know at least one active Chord node that represents the gateway to the network (the node has to be obtained by external mechanisms). In case of Figure 2.9, it is node 6. First of all it contacts the node with a search request, with its own id as a target. The request is routed through the network to the manager of id 14 – in this case, node 9. The interval node 9 manages is split into two sections, 9 to 14 and 14 to 18. Subsequently the node modifies its data structures, so that they indicate the reduced area of responsibility. Following after, the nodes 9 and 14 inform their predecessors and fingers about the change. As a last step all affected nodes correct their data structures in order to reflect the change.
In case a node fails (or leaves the network) the predecessor simply replaces the entry in its finger table by the first node in the successor-list that is found to be alive. Furthermore every finger attempts to find the successor of the failed node in order to put fill up its finger table.
Chord is only one of several protocols that used a circular structure, among them Koorde, Symphony and Viceroy that are introduced in the next three subsections [SMK+01].

2.3.3 Koorde

Koorde [KK03] is the first of three presented distributed hash tables that drew inspiration from Chord. The goal in the development of this protocol was to create a DHT that captures the positive simplicity of Chord, while improving characteristics like hop count, degree and maintenance overhead. For this, Koorde combines the basic, circular topology of Chord, with de Bruijn graphs. As a tool for creation of the structure, a hashing function is used that maps nodes and values on a b-bit id. The resulting identifier space is the unity circle and features ids from 0 to $2^b - 1$. In accordance with Chord, a node is responsible for all keys in the area between its id and the id of its direct successor on the ring.
Koorde embeds a de Bruijn graph in the ring, using the edges of the graph as routing links. Figure 2.10 shows the structure of a simple de Bruijn graph with eight nodes – a node for each of

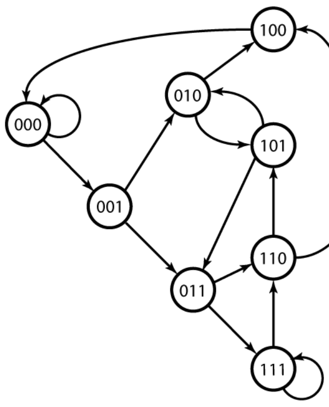


Figure 2.10: A de Bruijn graph with an 8-bit identifier. Each node m has edges to the nodes at $(2m)$ mod 2^b and $(2m+1)$ mod 2^b [KK03].

the b id-bits. Each node in the graph entertains two links to two other nodes at $(2m)$ mod 2^b and $(2m+1)$ mod 2^b, where m is the id of the node. The ids of the two nodes correspond to the id of the node, shifted to the left by one – the most significant bit is dropped – and inserted a new least significant bit. This property of the graph leads to a very convenient routing mechanism. In order to route from the node with id m, to the node with id k, k is simply shifted bitwise into m until all bits of m are replaced by k. Every shifting operation corresponds to one hop, from node to node.
A combination of the Chord circle and a de Bruijn graph produces a structure alike the one displayed in Figure 2.11. As illustrated by the figure, a node basically maintains three connections – one to its successor on the ring and two to the nodes designated by the graph. The de Bruijn routing algorithm now must be adjusted to the new conditions. Koorde's identifier space is, in line with every structured P2P architecture, only sparsely occupied. De Bruijn graphs, on the other hand require a node for every bit of the id. This divergence leads to a large number of virtual de Bruijn nodes. These virtual nodes are handled similar to data items – they belong to the next succeeding node on the ring. A consequence of this arrangement is that both de Bruijn edges a node maintains probably have the same owner, as they point on two succeeding nodes on the circle. Each node now only uses two connections for routing – naturally a link to its direct successor on the ring and in addition, a pointer to the predecessor of the node at $2m$ (which also is a predecessor of the node at $2m + 1$, with high probability).
In principle, routing now is very simple. The first step is to walk along the de Bruijn graph,

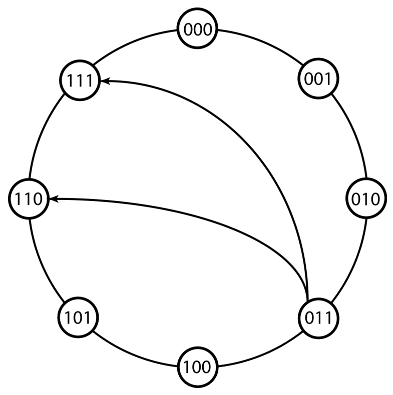


Figure 2.11: Compound structure of Koorde, with the edges introduced by the embedded de Bruijn graph. The edges are only drawn for node 011 in order to preserve the clarity of the picture.

until the successor of the requested node is hit. Then the path goes around the circle – using the successor-pointers – to the target. As most of the nodes are only imaginary, the real predecessors of the virtual nodes are used.
This actually is only a simplified view of Koorde's workings. If this simple procedure is used, the number of hops that are required to route to a node grows up to $3b$, with b the number of id-bits (this amounts to more than 400 hops for a 160-bit id). By choosing the starting point of the used de Bruijn graph carefully, this large number of hops can be reduced to $\mathcal{O}\left(logN\right)$. A further improvement is possible by moving to degree-k de Bruijn graphs. While this leads to a low hop count of $\mathcal{O}\left(log_kN\right)$ (n is the size of the network), the number of outgoing connections that have to be maintained by each node, increases to k. An aspect that must not be left out, is the basically unlimited number of incoming connections, each node has to accept in order to enable construction of the de Bruijn graph. It leads to a logarithmical dependency of Koorde's degree to the network size [SM06, p. 30].
In respect to joining and leaving of nodes, Koorde simply uses Chord's mechanisms. First a random id is obtained and then a join message is routed over a known node to the manager of the section. After that the new node is inserted in the network. A node leaves by simply cutting its connections [KK03].

2.3.4 Viceroy

Viceroy [MNR02] is another P2P protocol, besides Koorde and Symphony, that borrowed from Chord. Though, while all four share the basic circular structure, Viceroy obfuscates this by extending it with a butterfly network [Sie79] and a more complex routing algorithm. The result of this modification is a very intricate topology, which indeed has its perks.
This peer-to-peer protocol constructs a distributed hash table, while using numbers from the interval [0,1) as node id; s a whole, the ids produce the unity circle. Values receive an id as well but in contrast to the ids, it is not chosen randomly. Management of values is just as it is in Chord – each node is responsible for all keys that are in the interval between its predecessor and itself.
Viceroys overlay topology is rather complex, compared to the one applied by its relatives. First of all, it consists of three components – a general ring, several level rings and the butterfly. The general ring is identical to Chords structure – each node has links to the direct successor and

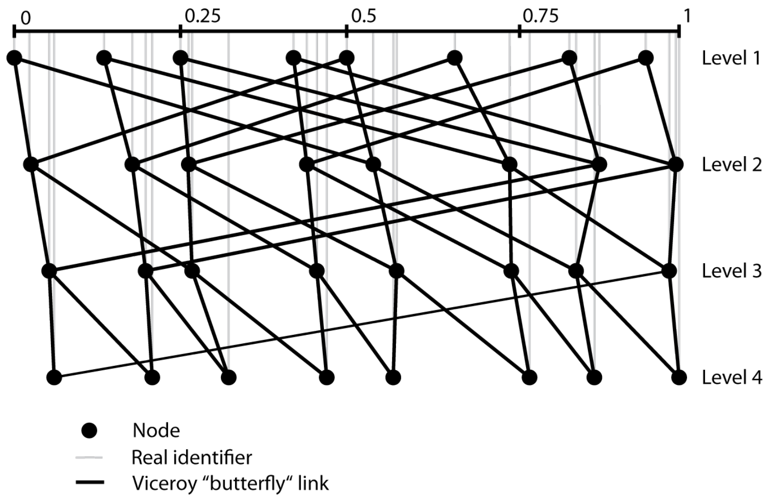


Figure 2.12: Principal structure of a Viceroy network. The dotted lines mark the ids of the nodes on the unity ring. Only the connections up and down the level are drawn – in addition there are connections to predecessor and successor on every level and to predecessor and successor on the unity ring [MNR02].

predecessor, depending on the id. Next, for every level exists a level ring that is created by linking all nodes of the same level. Finally there is the butterfly that requires connections to two nodes on the next higher level and one to the level below. All in all this amounts to seven outgoing connections per node. If seen in the terminology of Chord, two short distance connections exist on the general ring, two fingers/long distance connections on the level and three fingers/long distance connections for the butterfly. Figure 2.12 displays a (simplified) version of an ideal Viceroy network. The combination of the ring and the links to the different levels, approximate the aforementioned butterfly network. One further factor has to be considered. While the out-degree (the number of outgoing connections) is constant and low, the in-degree (the number of incoming connections) is unlimited and in the worst case can go up to $\mathcal{O}\left(logN\right)$. In order to limit the in-degree and reach network size independent degree, Viceroy applies a costly mechanism called "The bucket solution" (for information on this process, see [MNR02]).
Each nodes in Viceroy receives two parameters that identify it on the structure – the first one is the id, which is chosen randomly from [0,1). Secondly, there is the level a node resides on. The choice of the level is basically done randomly too but it depends on an approximation of the number of nodes in the network. While the id is constant throughout the lifetime of the node, the level may change with the network.
Routing in Viceroy follows in line with the architecture – it is comparably complex. Basically it is done by a three step algorithm that is described in a simplified form here. First the message is sent to the top level. In the second step it is routed down again, until a node is encountered that does not have any downlink. Finally the requested key is searched for by moving around the two corresponding rings – the level ring and the general ring – until the item is located. With the low number of seven links, Viceroy archives an average number of routing hops of $\mathcal{O}\left(1/k * log^2 N\right)$ [GRW].
Joining of a node requires, as usual, a known node in the network. The node calculates an id and starts a lookup for its own id through the known node. Using Viceroys routing algorithm, the manager of the corresponding location on the structure is found. After that, the node inserts itself in the network and in the process updates the routing information of the predecessor and

successor on the general ring. Now the node picks a level and repeats the process on the level ring. The last step then, is to connect to the remaining 3 nodes on the butterfly – 2 nodes on the level above and one on the level below.
Node departure is the odd one out of Viceroys mechanisms, in being quite simple. A node leaves by simply cutting its connections. After that the nodes in the network rearrange itself to the changed topology [MNR02].

2.3.5 Symphony

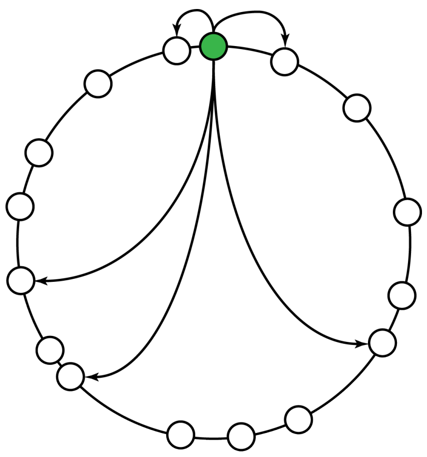
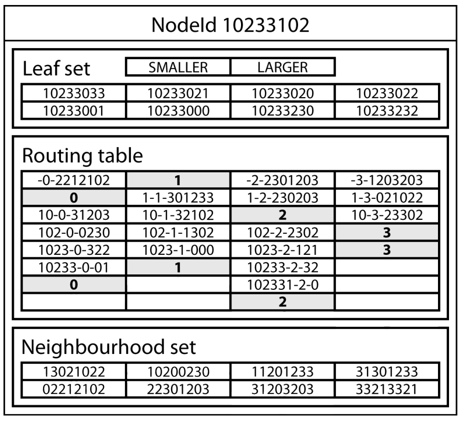


Figure 2.13: Structure of a Symphony network featuring all links that are established [MBR03].

Symphony [MBR03], just like Koorde, is closely related to Chord in many respects. It implements a distributed hash table that takes Kleinberg's small-world phenomenon [Kle00] as inspiration and combines it with Chords basic topology and routing mechanisms. While the protocol is closely related to Chord, it is especially trimmed to work with a very low number of connections per node. Actually the number of connections, and with it the size of the routing table, is fixed and independent of the size of the network.
The nodes and resources in Symphony are evenly mapped by their id on a circle with a perimeter of one (this overlay topology is again similar to the Chord circle Figure 2.6 only with a different type of id). Applying the common scheme, every node is responsible for the resources between its id and the id of its predecessor. While random creation of ids is the way proposed by the authors of Symphony, this is no mandatory property – any other approach is legitimate too (for instance, using a central server for distribution).
The bit size of the real number used as id is fixed by the protocol in no way either – it is open for adaption to the used hardware architecture and preferences. Even if it is not predefined, the bit size of the id relates directly to the possible number of nodes – a lower bit size means a lower precision and so only a smaller number of ids are available for use.
For routing, every node establishes a TCP connection with its neighbours on the ring and, additionally, a fixed number of long distance links, which are similar to the fingers in Chord. Contrary to the Chord fingers, Symphony's long distance links connect to random, evenly distributed nodes on the circle. Figure 2.13 shows the connection structure of Symphony – two connections to its immediate neighbours and an arbitrary number of links to random nodes on the circle. A further difference to Chord is that the number of long distance links is independent of the number of nodes in the network. Basically it is even possible to change it separately on every node during

runtime. In order to distribute the long distance links uniformly over the structure, a simple probability distribution is used (the name Symphony comes from this probability density function – it belongs to the class of harmonic distributions). A precaution, made necessary by the random distribution of the outgoing long distance connections, is the limited number of incoming long distance links each node accepts – if the maximum amount of connections is reached, every further request is denied. The number of incoming/outgoing links is not defined by the protocol but even for large networks the necessary number is as low as four. Two possibilities exist how the incoming connections are treated. Either they are used as additional long distance routing links, or not. Depending on this decision, routing either minimises the clockwise distance or the absolute distance.
The routing algorithm applied by Symphony is fairly simple. First the node checks if the key does belong to the node itself – is this the case, the algorithm is cancelled. If not, it compares the processed key to the intervals the predecessor and successor manage and in the positive case delivers it to them. Else, it is forwarded to the node (either one of the neighbours or one of the long distance connections) whose id is numerically closest to the requested id. All in all, Symphony requires $\mathcal{O}\left(1/k * log^2 n\right)$ routing hops, in average.
Network maintenance in Symphony is fairly straight forward as well. In order to enable a node to join the network it needs to know at least one active Symphony node, which subsequently serves as entry point. The first step is to choose a random id from the id range [0,1). After that, it requests the entry node to route the join message to the manager of the area on the ring, the chosen id belongs to. The two nodes divide the area of the circle between them, in respect to the position of the new neighbour. Both now notify the remaining neighbours of the change and finally the new node establishes its long distance links.
Leaving of a node is no difficulty at all – the node simply cuts its connections, whereupon all affected nodes just establish connections with other nodes [MBR03].

2.3.6 Pastry



NodeId 10233102

Leaf set

	SMALLER	LARGER	
10233033	10233021	10233020	10233022
10233001	10233000	10233230	10233232

Routing table

-0-2212102	**1**	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	**2**	10-3-23302
102-0-0230	102-1-1302	102-2-2302	**3**
1023-0-322	1023-1-000	1023-2-121	**3**
10233-0-01	**1**	10233-2-32	
0		102331-2-0	
		2	

Neighbourhood set

13021022	10200230	11201233	31301233
02212102	22301203	31203203	33213321

Figure 2.14: Data structures of Pastry. The leaf set holds the nodes that are numerically closest in the network. In every line of the routing table are nodes that have a certain part of their id in common. This two structures are used for routing, while the neighbour table is only required for network maintenance and holds nodes that are close in respect to locality in the underlay network [RD01].

Pastry [RD01] and Tapestry [9] are two structured P2P protocols that are closely related and so are not discussed isolated. Only Pastry is discussed here, as it has the greater practical relevancy.

The virtual overlay of Pastry has a circular structure (similar to the Chord identifier circle in Figure 2.6). The nodes and data items are distributed randomly on the ring, dependent on the id. This is usually archived by a hash function that processes the public id/IP address of nodes or any representative feature of values.
The node id is a 128-bit value that is arranged in a sequence of digits with the base 2^b. It is not only responsible for identification but the placement in the overlay and, corresponding to this, the distance between nodes. A shorter identical prefix (number of common leading bits in the id) relates to a longer distance between the nodes and a longer common part of the id indicates a closer distance. It has to be noted that the logical distance on the ring and not the geographical distance of the nodes is considered, even if the protocol is not oblivious of network locality – is taken into account in the process of building its routing structures.
Pastry keeps three different data structures – the leaf set, the routing table and the neighbourhood set. The first two data structures are used for routing, while the neighbourhood set is required for providing network locality. In Figure 2.14 the three tables are displayed for a node with the id 10233102. The leaf set contains a defined number of node ids that are closest to the own id. The routing table holds, in every row, a certain number of nodes that share a prefix (a number of common digits in the id) with the node. This prefix corresponds to the number of the row – so row 0 holds nodes with a prefix of length 0 and row 4 holds nodes with a prefix of length 4). In contrast to the other two lists, the neighbourhood set is not used for routing but for construction and update of the other two structures.
Routing in Pastry is a three-step process; the first step, if a request for a certain id is obtained, is to look up the id in the leaf set. Is it found, the message is directly forwarded to the node. In the other case the routing algorithm tries to find an id with a prefix that is at least one digit longer than the own prefix and, if successful, the message is sent to this node. Should the rare case arise that these two steps do not find any appropriate node, the message is submitted to a node with a same length prefix but which is numerically closer. This amounts to a number of $\mathcal{O}\left(log_2 N\right)$ (the configuration parameter b, is usually 4) routing hops, while maintaining a sum of $\mathcal{O}\left((2^b - 1) * log_2 N\right)$ entries in the routing structures [GRW].
Joining and leaving of nodes in the network is another important issue. In order to gain access to the network, a node needs the id of at least one node already in the network. There are various means to obtain this id, for instance outside channels at setup. The node X now contacts its known node A and sends a join message to it. Node A routes the message to the node that is numerically closest to A (we call this node B). After that every node that received the message on the way, forwards its state tables to A, which inspects the received information and updates its own tables. Afterwards it informs all nodes that have to be aware of its presence.
On departure of a node, the nodes which register it ask their neighbours for their leaf sets, in order to fill the gap in their own leaf set and routing table that was caused by the departed node.

2.3.7 Kademlia

Kademlia [MM02] is a very sophisticated P2P protocol that takes not only routing/lookup performance into account but also delivers a reasonably good protection against malicious attacks, like the denial-of-service (DoS) attack. Kademlia is is used in many different P2P networks, among them eMule [1]. Its routing mechanism is very flexible in respect to the possible routes it can calculate. The main reason for this is that id lookup is not limited to a small number of nodes, as is the case in every other P2P protocol that is discussed in this section. Quiet to the contrary, every known node can be contacted in order to find a certain id. This way it is possible for

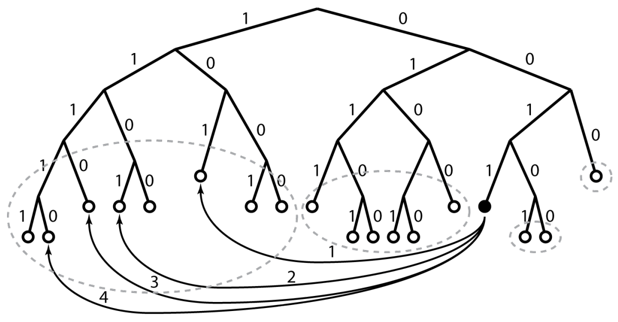


Figure 2.15: Kademlia binary tree with exemplary lookup queries from node 0011 in order to find node 1110 [MM02].

the protocol to use low-latency routes. Furthermore, the mechanisms of Kademlia allow parallel queries in order to reduce waiting time.

In view of node identification, Kademlia follows a common path with the 160-bit keys that are used as node id, as well as id for values. The node ids are randomly chosen 160-bit values, while the keys corresponding to data items are calculated by a hashing function. In contrast to the frequently used approach, where every node is responsible for a certain geometric area or the section of a circle, Kademlia takes a different path – keys that correspond to values, belong to the closest node. The necessary distance for this evaluation is calculated as the bitwise exclusive or (XOR) of the associated ids.

Figure 2.15 illustrates the topology of the overlay network – it implements a binary tree with the nodes as leafs. The position of the node in the tree depends on its id. In Figure 2.15 the id of the node displayed as black dot is 0011. It is found by starting at the root of the tree and walking downwards to the leaf (node).

The routing table in Kademlia is a collection of at most i lists, with $0 \leq i < 160$. Each list (a so called k-bucket, with k a global constant) is filled with routing information for a maximum of k nodes, in the interval between 2^i and 2^{i-1} that have a common part of their id. This set-up ensures that every node knows at least one node in every sub tree, in order to enable routing (in Figure 2.15 the dashed, gray circles are the sub trees from which node 0011 needs to know a node, in order to enable routing). The structure of the routing table (the number of k-buckets) is not fixed. Each node starts with a single k-bucket and increases the number, as the node learns of more and more nodes (until the maximum number of k-buckets is created). New nodes are discovered by receiving a message from them. Kademlia manages a very good routing performance of $\mathcal{O}\left(log_{2^b}N\right)$ hops on average but requires a comparably large routing table of size $\mathcal{O}\left(2^b * log_{2^b}N\right)$ (b is variable but usually is 5).

There is more to the k-buckets than just being an unordered list – ordering of the entries in a k-bucket is not random but by time last seen. If a message from a known node is received, the position of the node in its k-bucket is update. Originates a message from an unknown node, it is entered in the k-bucket if it is either not full or if one of the nodes in it is not reachable. Generally, older nodes get preference over new nodes, in order to contain efforts to infiltrate the network with malicious nodes.

The protocol uses an algorithm that ensures a successful search of any key, if the key exists in the network. In each step it picks α nodes from the k-bucket (α is a global constant), whose id range is closest to the requested key (if there are not enough nodes in the corresponding k-bucket, any close node is taken) and queries all nodes in the list for the key. The queried nodes locate the closest nodes in their table and return it. This process is repeated until the requested node is

found or no closer node is detectable. Figure 2.15 shows a query from node 0011 to node 1110. First the known node 101 is queried and returns 1101. Then 1101 delivers 11111 and finally 11111 passes back the location of 1110.
In order to join the network, a node needs to know at least one active Kademlia node. It then enters the node in the fitting k-bucket and queries it for its own id. By way of the routing algorithm it subsequently learns about nodes that are increasingly closer to its own id and inserts them into the correct k-buckets.
On disconnect or leaving of a node, the entries in the k-buckets time out and are replaced by active nodes over time.

2.4 Comparison of the Peer-to-Peer Protocols

The greater part of this chapter covered peer-to-peer systems in general first and a number of P2P protocols that represent possible candidates for usage in this work, afterwards. This section builds upon the collected information and discusses the presented P2P protocols in respect to a number of performance metrics. While a number of measures capture the different aspects of a peer-to-peer system, only a few correspond to the requirements defined in Section 1.3. The relevant factors are:

1. **Degree** – The amount of state information each node has to store in order to keep up the structure and enable routing. This factor also corresponds to the number of incoming and outgoing connections (in-degree and out-degree) a node has to maintain.

2. **Hop Count** – Characterises the average number of hops a message has to make in order to reach its target. Basically it is a measure of routing efficiency – better routing algorithms imply a lower average number of hops.

Beside these three factors, a number of others exist that are only of low relevancy to this work. Among them are maintenance overhead, fault tolerance (number of nodes that can fail concurrently without rendering the network inoperable), network delay, load balance and security [KK03].

First of all, every discussed protocol belongs to the class of structured peer-to-peer system. There are several reason for this limitation. These systems inherently are only pure decentralised architectures, which is of ultimate importance. Then, in accordance with the last section, only this type of topology possesses the ability to locate every node, integrated in the network. As the second most important requirement demands this feat, it all unstructured protocols are out of the picture. Furthermore, this form of P2P network provides strong network cohesion and is less likely to split as unstructured architectures. Last but not least, improved scalability often was one of the main thoughts in the development of structured peer-to-peer protocols. So this one choice, which was already imposed in Section 2.3, satisfies four of the six listed demands – leaving only the limited number of concurrent connections and the low overall performance of the used system (memory, clock frequency) as key reasons. Both requirements relate to the network degree.
As was already mentioned, the degree of a network is the sum of incoming and outgoing connections to other nodes, necessary for routing. Requirement 3 now defines a hard limit of 12 concurrent TCP and 6 concurrent UDP connections. Such being the case, the degree too must

Table 2.1: Comparison of the introduced P2P systems in respect to degree and hop count (compare to [GRW, SM06]).

Protocol	**Degree**	**Hop Count**
CAN	$2d$	$\mathcal{O}\left((d/4) * N^{1/d}\right)$
Chord	$\mathcal{O}\left(2 * log_2 N\right)$	$\mathcal{O}\left(1/2 * log_2 N\right)$
Koorde	$\mathcal{O}\left(logN\right)$	$\mathcal{O}\left(logN\right)$
Viceroy	$\mathcal{O}\left(2k+2\right)$	$\mathcal{O}\left(c/k * log^2 N\right)$
Symphony	$\mathcal{O}\left(2k+2\right)$	$\mathcal{O}\left(1/k * log^2 N\right)$
Pastry	$\mathcal{O}\left(1/b * (2^b - 1) * log_2 N\right)$	$\mathcal{O}\left(1/b * log_2 N\right)$
Kademlia	$\mathcal{O}\left(2^b * log_{2^b} N\right)$	$\mathcal{O}\left(log_{2^b} N\right)$

not exceed this number. Essential for this is the independence of the degree from the network size, as in the other case a violation of criterion 3 can not be ruled out. Column one in Table 2.1 lists the degrees each protocol exhibits. Protocols Chord, Koorde, Pastry and Kademlia do depend on the number of nodes in the network logarithmically. The direct logarithmical correlation of Chord and Koorde suggests a low enough number of links, even for large networks. But the dependence, in combination with the used $\mathcal{O}()$ notation, make a larger number of connections possible and as such the protocols too must be discarded. The degree of CAN, while independent of the actual network size, does depend on the number of neighbours a node has. The expected number of connections is as low as $2d$ but with a low probability can grow over the maximum of $12/6$ links.

The degree of the remaining two protocols – Viceroy and Symphony – are, independent from network size and bounded to a low number of links. Symphony maintains a maximum number of $2+3k$ of links (with k a configuration parameter) and Viceroy requires at most $7+2c$ connections (with c a configuration parameter).

Next relevant topic is the hop count – the average number of hops a message has to travel from source to target. In Table 2.1 the second column shows a similar result, if the upper bound of two for Viceroy's parameter c is taken into account. Both route a message in about $1/k * log^2 N$ hops in average, denying a selection through this property. Considering the equivalence in degree and hop count, the choice between the two peer-to-peer protocols must be based their complexity. Viceroy and Symphony both build a circular distributed hash table, supplemented with connections to other nodes on the ring. The additional structure Viceroy uses, now is substantially more complex as the simple random connections, Symphony establishes. Furthermore, Viceroy requires an additional mechanism for limiting the in-degree, adding to the already high complexity of the system. Taking all factors, mentioned above, into account, the final decision is on the protocol Symphony.

3 System Design and Implementation

The previous chapter conducted an extensive discussion of peer-to-peer (P2P) systems, followed by selection of the structured P2P protocol Symphony. Building on the given theoretical background, this part of the work goes into detail on the adaption and implementation of the aforementioned protocol in respect to the existing IRON (Integral Resource Optimisation Network) system.
At first, the current structure of the system that represents the base of this work is introduced. Furthermore, details on the used hardware – the IRON Box – are provided, in order to argue for the design decisions made. In the subsequent Section 3.2, a more in-depth discussion of the Symphony implementation is carried out and the operating principles and procedures of the protocol are explained. Finally, the last section of this chapter outlines the process of verification that was used to ensure the operability of the developed software.

3.1 System Overview

As was already mentioned in the first chapter, this work has its origin in the IRON project and is based on a distributed system, already in existence. At this point the existing system, used in the project is introduced.

3.1.1 System Structure and Properties

The idea behind the employed arrangement in the IRON project is reviewed from a different angle that complements the outline given in Section 1.2. Basically it is about controlling certain electric loads, depending on the state of the electric power grid. Any electric load by itself is too small to seriously influence the grid load. As a consequence the individual systems have to cooperate, in order to succeed in this endeavour. Even if cooperation, in certain bounds, is possible without interaction of the single entities, communication is imperative for doing so efficiently [Kup08, p. 106].
Figure 3.1 displays the current structure of the system used by the IRON project [Kup08, p. 106]. A central server is the heart of the operation. It distributes all information that is necessary for cooperation between the connected clients and collects measurement data from the nodes. The clients, represented by the so called IRON Boxes, are placed at the location of the electric loads. Control of the connected loads is the main responsibility but the device is equipped with means

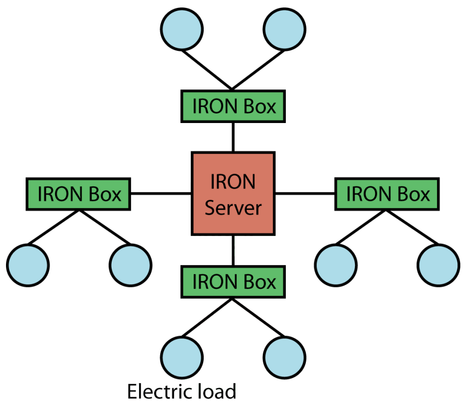


Figure 3.1: Structure of the current system used in the IRON project. The IRON server controls the IRON boxes, while these manage one or more loads.

to pick-up a number of measurement parameter too. These are processed on site and eventually transmitted to the IRON server at certain points of time.
Each node is equipped with a module, serving as network connection – either using WLAN (Wireless Local Area Network) or GPRS (General Packet Radio Service). On activation, a node establishes a TCP (Transmission Control Protocol) connection to the IRON server and, in the current configuration, remains on-line at all times. Despite maintaining an open channel to the server, a node does not send data continuously but collects it and stores it locally. Exchange of information between the two sides is only conducted sporadically. Lacking any other possibility, the Internet serves as communication infrastructure for the system.
An Atmel microcontroller is the heart of the IRON Box. The software that runs on the computer uses the layered design, presented in Figure 3.2 a). At the bottom of the stack is the network module with the corresponding TCP/IP stack – the network stack is executed in the hardware. On top of it, a layer managing the serial communication is located. This layer is topped by the network module handling layer, which has the task to manage the connected network module (either Avisaro WLAN or Sony Ericsson GR64) and to hide it from the higher layers. Next in line is the "IRON communication" layer, responsible for device-independent communication tasks. Finally, the remaining two layers represent the wished-for functionality of the system.
Figure 3.2 b) on the other hand, shows the structure including the Symphony protocol. The position of the peer-to-peer (P2P) protocol is placed in level with the "IRON communication" layer. It is not supposed to replace the current client-server based approach completely but to complement the software with additional functionality [Kup08, p. 106].
It is to note that integration of the Symphony protocol in the current IRON software is no part of this work.

3.1.2 Description of the IRON Box Hardware

The task of the IRON Box was explained in the previous section. In acknowledgement of the importance the hardware structure has concerning the implementation of the chosen P2P protocol, the device is presented here in detail.
Primarily, the IRON Box was developed to enable easy modification and flexible operation [Kup08, p. 106]. Figure 3.4 illustrates the structure of the device in a schematically way. Placed at the core of the system is the Atmel AVR ATmega128 microcontroller [Atm08]. Connected to it are

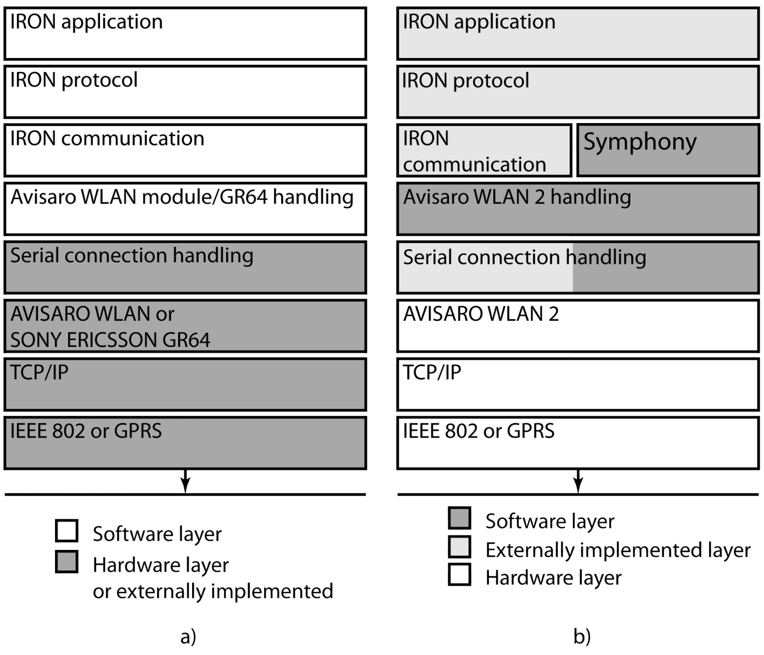


Figure 3.2: a) displays the protocol stack that is used at the moment for the software employed on the IRON Boxes. b) shows the modified protocols stack – the Symphony peer-to-peer protocol is positioned on level with the IRON communication layer.

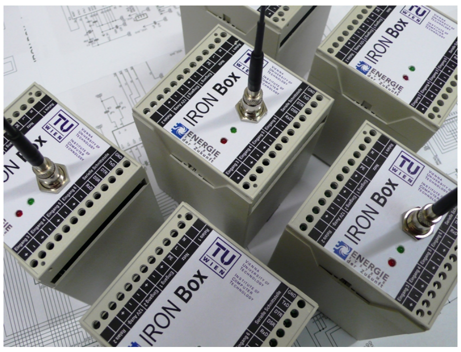


Figure 3.3: Picture of assembled IRON Boxes.

the two control-channels including the measurement equipment (each channel can be used to control and monitor one load), the grid frequency measurement unit, its sensor-inputs (for possible use of various external sensors), the external working memory and the network module.
The IRON Box uses a casing in DIN dimensions that is intended for use in switchgear cabinets – Figure 3.3 shows a picture of six fully assembled devices. Placement on this position opens up some advantages – it enables management of more than one electric load by a single box and furthers easy accessibility. For control of the loads, each of the units is equipped with two control-channels (top, left part of Figure 3.4). Both channels use a relay for switching and monitor the consumption separately in each of the channels, using Hall Effect sensors. Furthermore, six digital inputs for additional sensors are provided. In accordance with the main function – load depending activation/deactivation of loads – every device constantly measures the power grid frequency. All gathered information is then stored locally and transmitted to the IRON server on a regular basis. If for some reason the available memory is depleted, the oldest stored data is

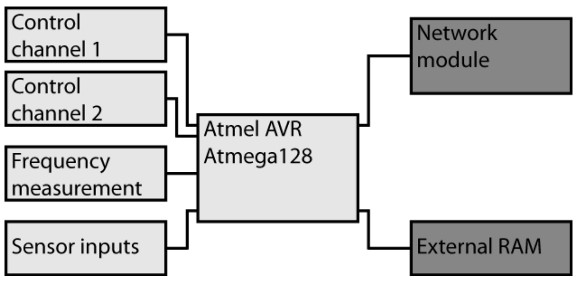


Figure 3.4: Schematic of the IRON Box structure. The components with a light grey background are the same for all versions of the IRON Box. Opposing them, the network module and the external RAM in dark grey, are modular. For the network module three different options are offered, while the external RAM currently is not incorporated.

replaced by a new measurement values, so the most relevant information is available to the server at all times.
One of the prime components of the IRON Box is the network module (top, right part of Figure 3.4). All versions of the device consist of the parts explained earlier on, in addition to the microcontroller. However, there are different choices for the network module. At the moment two different technologies are available – either GPRS or WLAN. All options are represented by independent modules, connected to the microcontroller via serial line. The computer controls the network module by messages (either text or byte code).
Currently three distinct modules are available – the Sony Ericsson GR47 GSM/GPRS Radio Device, the Avisaro WLAN Modul 1.0 and the Avisaro WLAN Modul 2.0. The GR47 and WLAN Modul 1.0 are fairly similar in respect to usage and features (except for the different type of communication technology they use). Both are operated, using simple text commands (AT commands) and support a single concurrent TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) port. This severely limited number of possible network connections rules out these two components for use in this work (the reason was discussed in the problem statement).
The Avisaro WLAN Modul 2.0, on the other hand, is a different matter. It is the latest improvement of the IRON Box and exceeds the other two modules in many ways. This option provides a definite advancement in flexibility, especially through the increased number of possible concurrent TCP/UDP connections. For implementation of any peer-to-peer protocol, a single concurrent TCP/UDP connection is not sufficient – so only this option is a viable solution in this work. As certain design choices depend on peculiarities of this WLAN unit, the module is introduced in detail subsequently.

The IRON Box in its current form is a proof of concept that is used to evaluate various ideas, among them different forms of network access [Kup08, p. 78]. For this reason, the WLAN module is not included in the used microcontroller but represented by an external device. This approach implies that the Avisaro WLAN Modul 2.0 itself is a sophisticated embedded system in its own right that is very adaptable to the needs of the user [10]. First of all, it is modularly designed – the device consists of a base module that runs a real time operating system and offers a serial interface for communication. Other functionality then is added, by extension modules. These are attached and connected to the base module in a piggy-back fashion. Available choices include several different communication protocols – among them CAN, SPI, Ethernet and WLAN – and a module with a mounted SD card slot.
For use in the IRON Box only the WLAN extension module was chosen. This unit supports the IEEE 802.11g standard, providing a transfer rate up to 54 Mbit/s. Furthermore, all parameters

that are necessary to integrate the device in an existing wireless network are available – including channel selection, ad-hoc/infrastructure mode and different types of network encryption. It includes a full UDP/IP and TCP/IP stack, making the most common transfer protocols available to the "user". The device is able to maintain a number of six concurrent UDP connections and twelve concurrent TCP connections – in both, the incoming and the outgoing direction.
However, while the unit is quite powerful, it too has its drawbacks. Featuring highest among them is that branching of TCP connections is not supported. Branching redirects an incoming connection attempt to another port, while keeping the original port open. As this property is not available, a connection is always established to the port, the device listens on and the port will be closed for further connection attempts for the duration of the connection.
Besides the modular structure of the device itself, a broad array of configuration options for every aspect of the unit is offered. For this task, basically every available interface (in this case WLAN and RS232) can be used. The RS232 serial connection, though, is the primary way for configuration. It can be used twofold – as a text based command interface or a packet API. Both variants use a ping-pong scheme for communication – the user sends a request and the module replies (the term "user" is used as loosely as possible). It essentially works as a slave to any controlling instance – communication is always initiated by the user and never by the device. This goes for incoming TCP/UDP packets too – such a packet is buffered in the module, until the user requests its delivery. For this purpose two 1 Mbyte buffer are available, which are used alternately.
The command interface uses text messages in ASCII (American Standard Code for Information Interchange) coding. It is mainly provided as an interface for user-machine interaction and works with every terminal program on a personal computer. Figure 3.5 shows how the WLAN channel is changed, using the text interface. Each command is plain text, followed by its parameters (and eventually by user data). The device responds equally with a text string as acknowledgement.

Host: $\underset{1}{\underline{\text{WLAN}}}$ $\underset{2}{\underline{\text{CHANNEL}}}$ $\underset{3}{\underline{\text{13}}}$ <enter>

Module: $\underset{4}{\underline{\text{OK}}}$

1: Command
2: Parameter 1 (Subcommand)
3: Channel 13 (Data)
4: Response

Figure 3.5: Configuration of the module using the text based interface. In this example the WLAN channel is configured.

Header	Length	Payload			CRC
1 byte	2 byte	Command	Parameters	Data	2 byte

Figure 3.6: Avisaro Packet API frame format [10].

The second interface for configuration of the module, using the serial connection, is the packet API (Application Programming Interface). Contrary to the command interface, it is intended for communication with a microcontroller. For this purpose the simple frame format, displayed in Figure 3.6, is used. The meaning of each part of the frame is as follows:

Header – Type of the packet (command, acknowledgement, ...).

Length – Length of the payload.

Payload – Payload carried by the frame, consisting of the command, the associated parameters and the user data. The command is mandatory, while the existence and amount of parameters and data depends on the specific command.

CRC – Checksum that is calculated (only) over the payload (Cyclic Redundancy Check).

As was already mentioned, the length and CRC field only relate to the payload of the packet. While the header and length fields are mandatory, the checksum need not be calculated. If it is not used, the corresponding field must be filled with zeros. Independent of how the user handles the CRC field, the module always calculates the checksum and attaches it to the return message. Repeating the procedure from Figure 3.5, while using the frame API, results in Figure 3.7. All

Host: $\underbrace{81}_{1}\,\underbrace{00\ 02}_{2}\,\underbrace{74}_{3}\,\underbrace{0D}_{4}\,\underbrace{00\ 00}_{5}$

Module: $\underbrace{84}_{6}\,\underbrace{00\ 00}_{7}\,\underbrace{A9\ 41}_{8}$

1: Header
2: Length - 2 bytes
3: Command id - WLAN Channel
4: Parameter - 13
5: CRC - 0 (not used)
6: Positive Ack
7: Length - 0 bytes
8: CRC

Figure 3.7: Configuration of the module using the packet API. In this example the WLAN channel is configured.

data is sent simply as a string of bytes (the spaces in the picture are only added for clarity). Compared to the command interface, the packet API is far more efficient in combination with a microcontroller. Not only the communication itself is superior – packet commands only require one byte of memory each, while even the shortest text commands that are used with the command interface need 3 bytes. So by using the packet API a substantial amount of memory in the microcontroller is saved. Furthermore, the manufacturer of the module recommends employment of the packet API if several concurrent TCP/UDP connections are used. The Avisaro WLAN Modul 2.0 does not require any configuration of the interface, so both types can be used unconditionally at the same time – a very handy feature for debugging. A list of all available commands for the command interface and packet API are found at [10].
Another way for configuration of the module is using the WLAN interface. The Avisaro WLAN Modul 2.0 offers a website for this process that provides the most important features of the module in a neat and easy-to-use fashion. It is a very convenient way, especially for initial configuration of the device. A downside of using the web interface is that it reduces the number of TCP connections that are available for other tasks to eleven (one connection is reserved for the website, if it is enabled).
Beside the already mentioned features the module offers a special scripting language and an interpreter, enabling execution of custom scripts on the module itself. However, during execution of scripts, all traffic is processed only by the script and, for instance, incoming packets are not forwarded to the serial line.

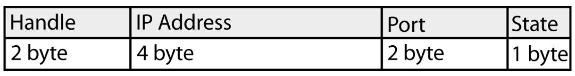
The Atmel AVR ATmega128 microcontroller is at the core of the IRON Box and unites all the different components [Atm08]. It is a cheap, low-power, 8-bit microcontroller that is available in mass production. It provides three built-in memories – 128 kbyte flash program memory, 4 kbyte EEPROM and 4 kbyte SRAM. In addition, it offers an interface for external memory. The flash memory is in-system programmable via SPI (Serial Peripheral Interface), allowing for easy programming of the assembled device. While prepared, no external memory is used in the current configuration of the IRON Box. A further important property of the microcontroller is the clock frequency that ranges from 0 to 16 MHz and is adjusted to 8 MHz for use in the IRON Box. As means of communication, two programmable serial connections are available. These are used to connect the WLAN module and to provide debugging messages to the user. At the moment no flow control mechanism is employed.

3.2 Implementation of the Symphony Protocol

The Symphony protocol was already introduced in Section 2.3.5. This section discusses the protocol in greater detail, with a focus on the embedded implementation.

3.2.1 Implementation Structure

The Symphony protocol implementation is based around TCP connections. Usually these are handled by TCP sockets but due to the structure of the hardware (microcontroller, with a serially attached WLAN module) no TCP socket implementation exists. For this reason handling of TCP



Handle	IP Address	Port	State
2 byte	4 byte	2 byte	1 byte

Figure 3.8: Structure and byte-size of a TCP socket used in the Symphony implementation.

connections is done through the custom data structure displayed in Figure 3.8, which is similar to a TCP socket (in this section TCP socket is used for this constructs). It consists of all information necessary to manage the corresponding connection – the "handle", which is an identifier of each specific connection on the WLAN module, the remote port/IP address and the current state of the socket.
The number of possible concurrent TCP connections is severely limited. This restriction has its origin in the used WLAN module and influences especially the way new TCP connections are established. Every Symphony node maintains a number of connections, where each one has a well defined function:

1. 1 socket for incoming connection requests.
2. 1 socket for outgoing connection requests.
3. 1 socket for the left neighbour.
4. 1 socket for the right neighbour.
5. k sockets for outgoing long distance connections.
6. j sockets for incoming long distance connections (usually $j = 2 * k$).

The task of the neighbour and long distance sockets is to connect to the mentioned nodes, while the sockets for incoming and outgoing requests are used to compensate for the ability to branch incoming TCP connections, which is missing in the WLAN module (Section 3.1.2). If one of the above mentioned sockets is connected, no additional connection to it is possible. Now, if new nodes join the Symphony network, they have to connect to certain other nodes in order to be inserted into the network. Of special importance are the two neighbour connections. On an already integrated node, these connections will be busy and connection attempts to them are futile. To circumvent this problem, these two nodes (and ever other active node too) will listen on the incoming "request port" for connection attempts.
A new node now tries to establish a connection from its outgoing "request port" to the incoming ports of its yet-to-be neighbours and notifies them about its intention to establish a new neighbour connection. Following the notification, the desired counterpart will cut its current connection (if possible) and open the corresponding neighbour port for a connection from the new node.
This approach does work out in case of a slowly changing network, as the one in focus. A highly fluctuating network, though, represents a problem. As soon as two or more nodes attempt to connect to a single node at roughly the same time, one of the rivals has to wait until the connection is free. In case similar situations occur frequently, nodes have to wait a long time for a chance to join the network.

For implementation of the protocol design, the layered structure displayed in Figure 3.9 was chosen, which offers a good balance between clarity and performance. The dashed rectangle at

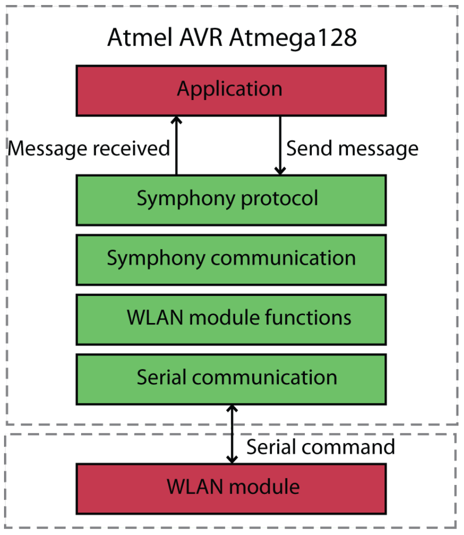


Figure 3.9: Stacked structure of the Symphony protocol implementation.

the bottom represents the network module that is connected to the microcontroller (the upper dashed rectangle) via serial line. While the layers marked in green were created or modified as part of the work, the red layers were already available. On every level, the upper layer use the services provided by lower layer, while these communicate reception of messages and link failures upwards. Each layer is responsible for a distinct part of the functionality:

1. **Application**
   Generic application that uses the Symphony protocol for sending messages to specified Symphony ids. In the other direction, the application receives messages that are intended for the node from the "Symphony Protocol" layer. This layer is no part of this work.

2. **Symphony protocol**
   All procedures that are necessary for the self-organisation of the network, routing of messages and high level error handling are placed in this layer. These tasks mainly consist of network maintenance (joining of nodes, leaving and failure of nodes) and routing.
   The functions in this layer can be divided in the following three classes:

   - *Utility* – To this category belong all functions that mainly have a supporting role. Tasks that are performed here are calculation of ids, estimation of the network size, every action related to routing and handling of lookahead information.
   - *Handling* – These functions are responsible for handling incoming messages and furthermore for processing connection errors. For each of the messages in Table 3.3 exists one distinct function that is called by the "Symphony communication" layer on receiving a message of this type. The same goes for connection errors – if a failed link is noticed by the communication layer, the error handling function in the protocol layer is called.
   - *API* – Only one dedicated API function exist – *symphony_send_request(target, message)*. Its purpose is to enable applications to send messages through the Symphony network.

3. **Symphony communication**
   Responsible for every communication related property and task of the protocol. This includes:

   - *TCP socket handling* – The used TCP sockets are managed by this functions, including initialisation of sockets, connection build-up/tear-down, and monitoring of link states.
   - *Output buffer management* – In order to compensate for the time it takes to establish connections, a ring buffer exists that stores messages that are intended for links that are still in the process of connecting. It holds *BUFFER_SIZE* (default: 5) elements of the size *MAX_MSG_LEN* (default: 150) in bytes (see Table 3.2). All functions concerned with controlling the buffer are collected under this term.
   - *Message management* – Encompasses all functions that relate to creating, sending and receiving of Symphony messages.
   - *Task* – Covers the task-based approach taken for scheduling. Refers to a single function – *symphony_task()*.

4. **WLAN module functions**
   Provides all functions that are required to manage the WLAN module. Basically, everything that is handed down from the upper layers is converted to an API command the module understands. The functionality can be divided into following categories:

   - *Module management* – Every function that is tasked with configuration and control of the module.
   - *Socket management* – Handling of TCP connections. Lower level version of the connection handling functions on the "Symphony communication" layer.
   - *Message handling* – Sending of messages to the WLAN module and receiving of messages/packets from the module.

5. **Serial communication**
   Handles transmission of commands from the "WLAN module functions" layer over the serial connections. This layer is a modified version of the one used in the IRON software – it was extended by a function that enables sending of bytes (instead of strings or numbers) over the serial line.

6. **WLAN module**
   This layer is no software layer but represents the Avisaro WLAN Modul 2.0. All TCP/IP and WLAN related functionality is found here.

To highlight the interaction between the different layers, the process of sending and receiving of a data message is used. In Figure 3.10 the process of sending a data message is displayed. The

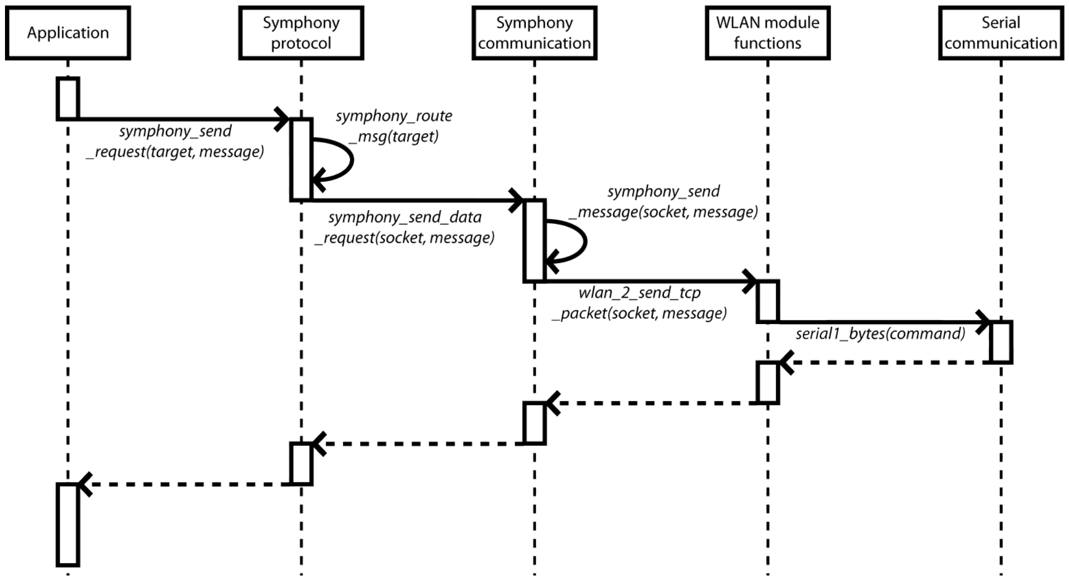

Application
Symphony protocol
Symphony communication
WLAN module functions
Serial communication
symphony_send_request(target, message)
symphony_route_msg(target)
symphony_send_data_request(socket, message)
symphony_send_message(socket, message)
wlan_2_send_tcp_packet(socket, message)
serial1_bytes(command)

Figure 3.10: Message sending procedure used in the Symphony implementation.

first step is that the application uses the API-call *symphony_send_request(target, message)* that is provided by the "Symphony protocol" layer. In this function, three basic steps are executed. First, the user-message and the target id is put into a *DATA* message, followed by a call of *symphony_route_msg(target)*, in order to obtain the TCP socket for the next hop. At this point the message is passed on to the "Symphony communication" layer for sending, through a call of the *symphony_send_data_request(socket, message)* function.
First thing in this layer is to create a generic data frame from the message, which is then forwarded to the *symphony_send_message(socket, message)*. Depending on the type of connection (long distance or neighbour), lookahead information for the next hop is attached to the message. Furthermore, the state of the TCP connection to the next hop is checked and depending on the current status one of two actions is taken. In case that the connection is not yet on-line (still connecting or disconnected), the message is pushed on the buffer and in case of a disconnect, the *symphony_handle_connection_error(socket)* is called to process the error. The message is reclaimed from the buffer and transmitted as soon as the error is handled and a connection available.
On the other hand, if the connection is on-line, the message is passed on to the *wlan_2_send_tcp _packet(socket, message)* in the "WLAN module functions" layer. Here a command for the WLAN

module is created by wrapping the message and socket into an API command frame (Figure 3.6), which is then sent to the network module, using the *serial1_bytes(command)* on the "Serial communication" layer.
In the WLAN module, the message is wrapped in a TCP packet and then transmitted over the TCP connection that corresponds to the chosen route.
Reception of a message works as displayed in Figure 3.11. As was already mentioned, the pro-

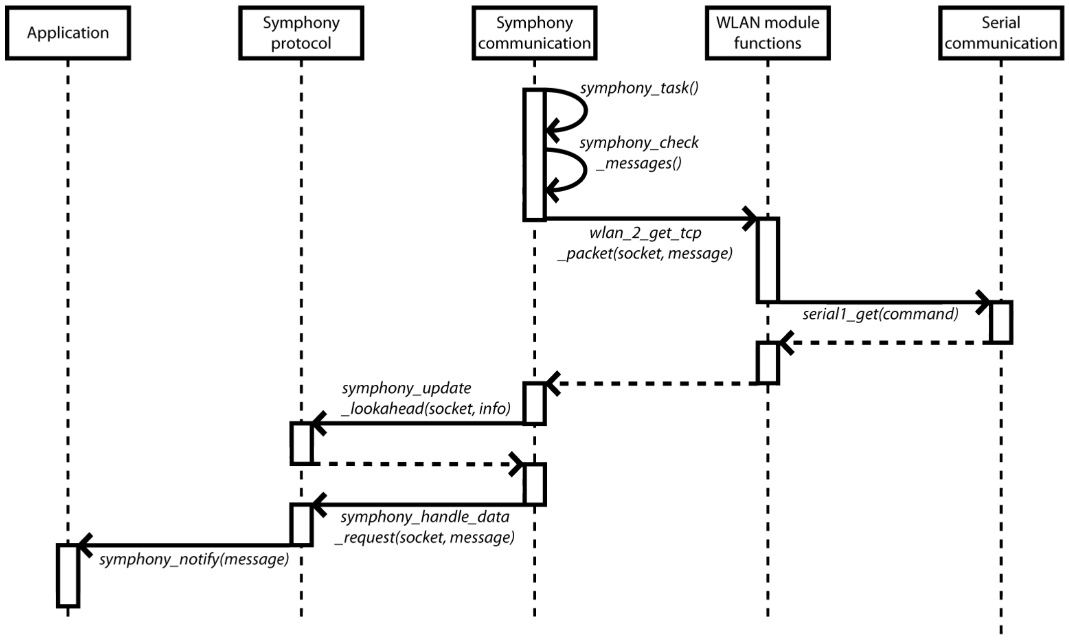

Application
Symphony protocol
Symphony communication
WLAN module functions
Serial communication
symphony_task()
symphony_check _messages()
wlan_2_get_tcp _packet(socket, message)
serial1_get(command)
symphony_update _lookahead(socket, info)
symphony_handle_data _request(socket, message)
symphony_notify(message)

Figure 3.11: Message receiving procedure used in the Symphony implementation.

tocol used by regular calls of the *symphony_task()*. This function checks if new messages are available on the WLAN module, using *symphony_check_messages()* (both functions are placed on the "Symphony communication" layer). It employs the *wlan_2_get_tcp_packet(socket, message)* and *wlan_2_get_packet()* functions to collect messages from the module (the *serial1_get()* is only used for illustration – the actual process of collecting a message from the module is a little bit more complex). If a message was received by this procedure, two things happen. The lookahead information of the entry in the routing table is refreshed, if the socket the message was received on belongs to a long distance connection. Primary though, the message handling function, corresponding to the Symphony message type (Table 3.3) is called and subsequently processes the message (in case of a data message, the application is informed).

3.2.2 Software Configuration

The Symphony protocol does not stand on its own but represents a tool that offers message transmission through the constructed peer-to-peer network. Currently the target application is the software developed in course of the IRON (Integral Resource Optimisation Network) project. While integration of the protocol in the existing IRON application is no part of this work, the means to enable this process are necessary anyway. As a consequence of this, the configuration parameters must correspond to the ones used in the existing software. The important settings are summed up and explained in Table 3.1. Symphony requires a real-time clock for resending

Table 3.1: Symphony configuration settings.

Parameter	Description
Timers	Configured to provide a real-time clock.
Watchdog timer	Reset after 1,8 seconds.
Serial lines	38400 Baud, 8-bit frame size, 1 stop bit, no flow control.

JOIN messages after a certain time – for this reason the timer of the Atmega128 are configured to reflect this need. A further important device is the watchdog timer. It is running at all times and resets the microcontroller after 1,8 seconds, if the timer is not reset.
Most important are the two serial lines. Both use the parameters listed in Table 3.1. Serial line 1 links the Avisaro WLAN Modul 2.0 to the microcontroller while serial line 2 is used to provide debugging messages to the user.

Independent of the application that uses this Symphony implementation, some form of scheduling is required. The currently employed mechanism is very simple – the software is divided into tasks, which are then called alternately.
Execution of the Symphony protocol procedures is done through such a task as well, represented by the *symphony_task()* function. This function is responsible for monitoring socket states and retrieving/sending of (buffered) messages – basically all tasks that have to be processed regularly.

Besides the basically pre-defined options of the software, a number of configuration parameters for the Symphony protocol were introduced, which are defined as pre-processor constants. Table 3.2 lists the options that are of importance for use of the software (see Appendix A for a full reference).

Table 3.2: Software configuration settings.

Parameter	Description
MAX_MSG_LEN	Maximum length in bytes for a message sent over the Symphony network. Default: 150
MAX_USER_MSG _LEN	Maximum length in bytes for an application message. Default: 107
BUFFER_SIZE	Size of the buffer for outgoing messages (number of messages). Default: 5
RESEND_TIME	Amount of time in seconds, the node waits until it resends the JOIN message. Default: 30
SYMPHONY_SEED	Seed for the random-function that is used to calculate Symphony ids. Must be different for each node in order to avoid identical node ids.
LOCAL_IP_X	Local IPv4 address (X is 1,2,3 or 4).
BASE_PORT	Start of the port range, Symphony uses. Default: 2000
INIT_IP_X	IPv4 address of the Symphony node that is used as entry point (X is 1,2,3 or 4). Default: INIT_IP_1 = 192, INIT_IP_2 = 168, INIT_IP_3 = 1, INIT_IP_4 = 74.
INIT_PORT	"Request port" of the node in the Symphony network that is used as entry point. Default: 2000

3.2.3 Protocol Structure and Organisation of Symphony

Symphony organises the network, using the circular structure described in Section 2.3.5. It consists of short links that connect to the neighbours on the ring and the so called long distance connections that point to certain positions on the ring. The target positions are calculated, using the harmonic probability distribution function

$$e^{\log n * (random\_01() - 1.0)}$$

. It is calculated using the exponential and logarithm functions of the maths library for the Atmel AVR microcontroller (*double exp(double x)* and *double log10(double x)*). These functions are programmed in hand-optimised AVR assembly and use values in double (32 bit) precision. The *random_01()* function provides random real numbers between zero and one (again in double precision) and n is the size of the network [MBR03]. Taking into account that the real network size n usually is not known, an estimate is used instead. For this purpose, Symphony uses a simple estimation protocol that only requires already existing information about its neighbours. The estimation formula is s/X_s with s the number of intervals used (three in this work) and X_s the sum of the intervals the node in question itself and its two neighbours manage. This really is only a rough estimate – experiments in this work showed that for a real network size of one thousand, the calculated size ranges from less than five hundred to more than one thousand five hundred nodes. Although this is a rather wide range, it is exact enough for use in the probability distribution function and, according to the authors of the protocol, has only a small impact on routing performance [MBR03]. If a more precise estimation is needed for any purpose outside of the protocol function, a different approach must be taken. Of course, if for instance the exact network size is known, this value can be used unconditionally too.
As a Symphony network usually will grow from a small number of nodes onwards, the distribution of long distance links will become less optimal in time. To counter this process, a re-linking mechanism is proposed by Manku et al. [MBR03] that redistributes the connections in a regular interval. Considering the quality of the used estimation and its low impact on performance, this re-linking mechanism was not used in this work.

For implementation of the Symphony protocol the following node states were defined in this work:

1. ***DISCONNECTED*** – Node has no join message sent.
2. ***WAITING_JOIN*** – A join message was sent but no response was received yet (the message will be resent after a certain time).
3. ***WAITING_NEIGHBOURS*** – A response from the right neighbour was received and the node is busy, establishing connections to both its neighbours.
4. ***WAITING_LONG_DISTANCE*** – Both neighbours are connected, the node has sent k requests for long distance connections and is waiting for a response.
5. ***READY*** – The node is connected to k other nodes and is fully set up.

These states signify the different levels of integration in the network and influence several aspects of the protocol. As long as the node is in a state below WAITING_LONG_DISTANCE it will not process any attempts of nodes to join through it. Furthermore, the routing performance is affected by the different states. Effective routing of messages is possible in the states four and five.

While routing in state four is possible, performance is degraded, compared to the *READY* state, as only zero to $k-1$ long distance connections are available. All states and the possible transition between them are displayed in Figure 3.12. More details on the states and the transitions between them are given in the subsequent sections.

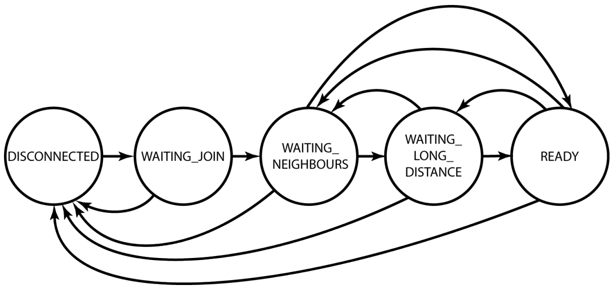
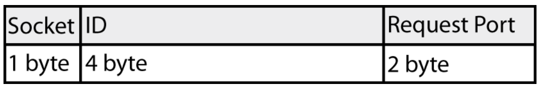


Figure 3.12: State diagram of a Symphony node.

In order to fulfil the needs of the protocol, seven types of messages were defined. A short description of the different types is given in Table 3.3. In case a message arrives that can not be identified, it simply is discarded. All messages with exception of the *DATA* message are part of network maintenance and are discussed further in the context of Section 3.2.5. *DATA* messages contrary to the other messages are used to transmit data over the network and as such represent the intended functionality of the protocol. They include the id of the sending node, the id of the targeted node (this value is used for routing) and a custom data message that was provided by the application.
The seven message types belong to one of two categories that are divided by how they are transmitted. First class of messages are the routed messages. These messages are, as the name implies, routed through the Symphony network – the message types *JOIN*, *CONNECT_LONG_DISTANCE*, *REJECT_LONG_DISTANCE* and *DATA* are in this class. *CONNECT_NEIGHBOUR_REQUEST*, *CONNECT_NEIGHBOUR_RESPONSE* and *ACCEPT_LONG_DISTANCE* on the other hand, are sent directly from the sender to the receiver, without any intermediate nodes.

3.2.4 Routing in Symphony

The routing information, stored by a node in this implementation consists of two main parts. First of all, there is the neighbour table, storing all relevant information about the direct neighbours. Figure 3.13 displays the structure of an entry in the neighbour table and the amount of space it takes up in the working memory. Beside the neighbour table, the routing table is the second,



Socket	ID	Request Port
1 byte	4 byte	2 byte

Figure 3.13: Structure of a neighbour table entry in the implementation.

greater part. It contains information, relevant to routing, about all long distance connections a node maintains (see Figure 3.14). Both tables contain entries that consist of the id, IP address and port of the corresponding nodes. In addition to these values, the routing table also includes lookahead information of every entry in it. This extra data is made up of the ids stored in the routing table of the linked-to node and is exchanged as attachment to messages that are passed

Table 3.3: Symphony message types.

Type	Description
JOIN	Initial message in the network joining process (routed through the network).
CONNECT_NEIGHBOUR _REQUEST	Requests a new neighbour connection from the node at the left. If the message is used for initial inclusion of a new node in the network, information about the current left node of the sending node is included in the message. This message goes always to nodes on the left-hand side of the sender.
CONNECT_NEIGHBOUR _RESPONSE	Answer to the connection request message. It is used to conclude the set-up procedure for neighbour connections and provides the right neighbour with information about the left neighbour of the sending node. This message goes always to the right neighbour.
CONNECT_LONG_DIS-TANCE	Requests a new long distance connection between the sender and receiver of the message (the message is routed through the network).
ACCEPT_LONG_DISTANCE	Is sent as a positive acknowledgement to the *CONNECT_LONG_DISTANCE* message. Concludes long distance connection set-up.
REJECT_LONG_DISTANCE	Is sent as a negative acknowledgement to the *CONNECT_LONG_DISTANCE* message (routed through the network).
DATA	Used for data transmission between two nodes (routed through the network).

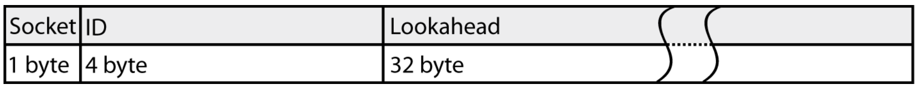


Figure 3.14: Structure of a routing table entry in the implementation.

between the two nodes (no dedicated messages are used for exchange of lookahead information). Figure 3.15 illustrates this construct – the node with id 0.01 maintains a long distance connection to the node 0.41 and stores the id of the node (among other information) in its routing table. To every entry it stores the routing table content of the corresponding node too. This data is responsible for the greater part of the byte-size the routing table exhibits, as is illustrated in Figure 3.14.

The routing process that is used in each Symphony node consists of several steps. These necessary steps are depicted in Figure 3.16. First of all, the node checks if the target id is contained in the interval it manages. If this is not the case, the same action is carried out for the right and then the left neighbour. If this again does not deliver any positive result, the node engages in the ”real” routing process, the so-called ”distance routing”.
Distance routing uses the relationship imposed by the ids, to find the closest known node as a next hop. For this, the numerical distance between the target id and the id of each known node is calculated. Between any two numbers on the unity circle two distances exist – clockwise and anticlockwise – and here always the shorter of the two spans is considered. As new path, the

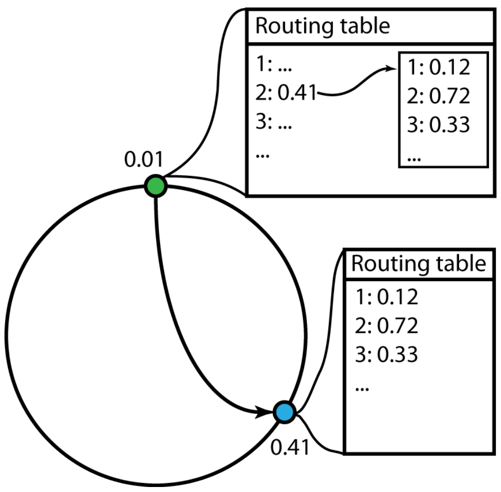


Figure 3.15: Simplified routing table of a Symphony node displaying the additional lookahead information, the node stores.

numerically closest node – the node whose id presents the shortest numerical distance to the targeted id – is chosen. For this process the following ids are considered (in the sequence they are used):

1. Long distance link id.
2. Ids in the lookahead table of each long distance link.
3. Left neighbour id.
4. Right neighbour id.

Only the ids of connected nodes (by neighbour and long distance connections) are needed for finding a route. However, during "distance routing" not only the ids of the nodes in the routing table are used but the ids in the attached lookahead information as well. If one of these ids presents the shortest distance to the target id, the node this lookahead information belongs to is chosen as next hop. Figure 3.15 illustrates this process. Node 0.01 intends to route a message to the id 0.13. It looks up the id in its routing table and determines that node 0.41 is the best route – it is connected to node 0.12 which has the least numerical distance of all known ids in the routing table of node 0.01.
So, basically lookahead information gives a node the ability to "look ahead" one routing step and so offers a better idea which of the nodes it knows is best suited as next hop. While a node uses the ids in the lookahead information to find a better route, it never forwards a message directly to an id it finds in the lookahead information (it doesn't even maintain a connection to them). None the less, nodes determined by this process are not bound to forward the received message to the node calculated by the previous hop – every node is free to choose independently. Disagreement occurs though; the main reason for it is that lookahead information is not always up to date, as it is only updated if any communication happens between two nodes. Of course this feature adds to the complexity of the system but experimental results in [MBR03] indicate that 1-lookahead leads to a performance improvement of about 40% and as such is worth the effort.

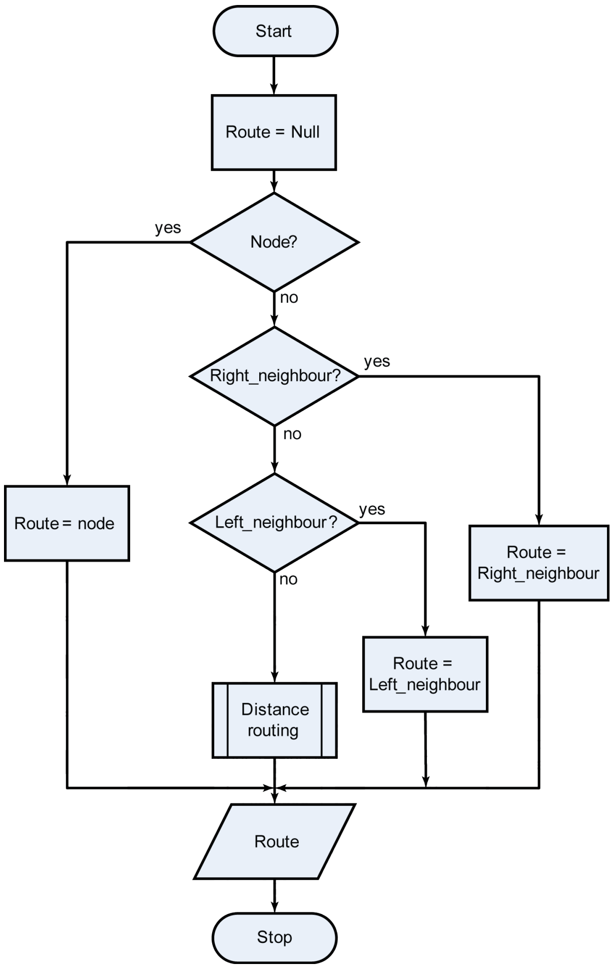


Figure 3.16: Steps a node takes in routing a message.

During the network joining process a symphony node goes through all node states, introduced in the last section (Figure 3.12). While basically the same, routing is slightly different in each state. In the *DISCONNECTED* and *WAITING_JOIN* state, no connection is established and as such no routing can proceed. The *WAITING_NEIGHBOURS* state can not guarantee reliable routing, as in the best case the right neighbour and in the worst case no other node is connected. Next state in line is *WAITING_LONG_DISTANCE*. It declares that the node is connected to its two neighbours but does not yet have established all of its k long distance connections (so the number of long distance connections is 0 to $k-1$. Routing in this state is possible but degraded, compared with the *READY* state. If a node has all its long distance links up, it enters the *READY* state and finally has full routing performance.

3.2.5 Network Maintenance

The last essential area of the Symphony peer-to-peer protocol is maintenance of the network. In this section is discussed how joining and departure of nodes is handled in the Symphony system.

In Symphony no central entity is used as entry point for the network [MBR03]. A node that intends to join, must know at least one node that is already integrated into the network. The protocol itself defines no method for obtaining such a node, so some external mechanism must be used for this task. Clearly the easiest way is to provide one node or a list of nodes as a predefined parameter at start-up (or possibly receiving them from a server). This is an issue, as a certain load imbalance results if only a single node is used as entry point for other nodes. Even if the entry point is in the current implementation design received that way and is not considered any difficulty for the current area of use (due to the slow network growth), this factor has to be considered in a productive environment.

At the beginning, any node that wants to enter a Symphony network is in *DISCONNECTED* state. As a first step it chooses a random id from the range [0,1). Then a connection to the request port of the known entry node is established and *WAITING_JOIN* state is entered. This connection is used to send a *JOIN* message that includes the id, IP address and "request port" of the new node. Starting with this unit, the message is routed through the Symphony network (using the routing mechanism introduced in the last section) towards the managing node of the id selected by the new node.
On receiving a *JOIN* message (that is meant for it), the manager of the target id cuts its connection to the left neighbour and opens the same port in listening mode. This step enables an incoming connection from the new node on this port. Subsequently, a connection to the port of the new node that was defined in the message, is engaged and a *CONNECT_NEIGHBOUR_REQUEST* message is sent through it. *CONNECT_NEIGHBOUR_REQUEST* messages include the id, IP address and "left neighbour" port of the sending node and the id, IP address and "request port" of its left neighbour (the second left neighbour).
After the new node receives this message, it goes over to *WAITING_NEIGHBOUR* state. At this point it has all the information it needs to take its place in the network. So it connects from its local "right neighbour" port to the "left neighbour" port of its new right neighbour and submits a *CONNECT_NEIGHBOUR_RESPONSE* message over this connection. In order to contact the left neighbour it opens a listening connection on its dedicated "left neighbour" port and opens a connection to the "request port" of the same that then is used to send a *CONNECT_NEIGHBOUR_REQUEST* message.
The dedicated new left neighbour receives the *CONNECT_NEIGHBOUR_REQUEST* message, opens a connection to the given port on the new node (this again is found in the message) and responds with a *CONNECT_NEIGHBOUR_RESPONSE* message that includes the id, IP address and "request port" of its own left neighbour.
Receiving this message from its left neighbour is the last step of the basic joining process. Now the node possesses all information about its direct vicinity in the network – its left and right neighbour for routing and its second left neighbour for future reference – and enters *WAITING_LONG_DISTANCE* state. The basic steps of this process are depicted in Figure 3.17.
What happens at this point depends on the estimation of the network size, calculated by the node. If it is below a certain number (currently, the limit is at six) the node remains in this state until it notices that a higher number of nodes exist. In the other case, the node attempts to establish its long distance connections.
As a first step, the node opens k listening ports for long distance links. Then it sends *CONNECT_LONG_DISTANCE* messages (consisting of the id and IP address of the node, the dedicated port and the target id) to positions on the ring that are calculated by the formula given in Section 3.2.3. These messages are then routed through the network, using the target id given in the message.
On arrival of a *CONNECT_LONG_DISTANCE* messages, a node checks two things. First of all, if there still is space for an incoming long distance link and secondly, if the id of the re-

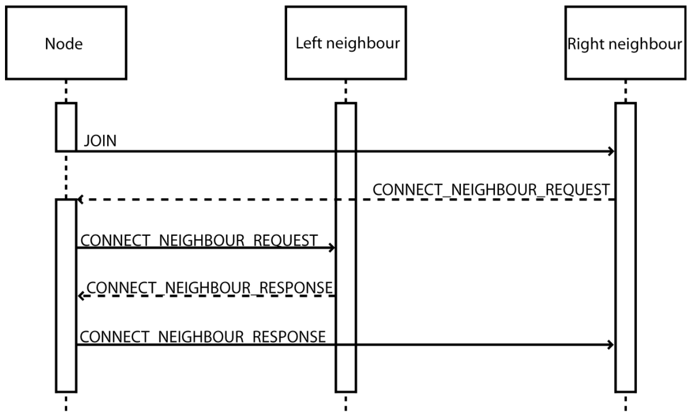

Node
Left neighbour
Right neighbour
JOIN
CONNECT_NEIGHBOUR_REQUEST
CONNECT_NEIGHBOUR_REQUEST
CONNECT_NEIGHBOUR_RESPONSE
CONNECT_NEIGHBOUR_RESPONSE

Figure 3.17: Sequence of events, necessary for adding a node to an existing Symphony network, up to the point the change to *WAITING_LONG_DISTANCE* state occurs.

questing node is already in the neighbour/routing tables. In case that the maximum number of incoming long distance connections is not yet reached and the current candidate is unknown, a connection to the IP address and port given in the message is created and over this link a *ACCEPT_LONG_DISTANCE* message is transmitted, notifying the original node. Otherwise a *REJECT_LONG_DISTANCE* message is routed through the network to the original requesting node.
Depending on the answer to its message, the node either includes the other node in its routing table or resends a *CONNECT_LONG_DISTANCE* message to another random position on the ring. This process goes on, until all k long distance connections are created.
On successful creation of all long distance links, the node moves over to *READY* state and finishes the joining procedure.

Departure of nodes from the network may happen for one of two reasons. The first reason is that the node leaves by its own intentions. It then employs the closing mechanisms provided by TCP, for shut-down of its connections. Otherwise, some failure (in the node or the network) occurred, resulting in disappearance of the node from the system (or parts of it). In the second case, the concerned nodes learn from the failure by the time-out mechanisms defined by TCP. While both origins are quite different, no essential distinction is made by the Symphony protocol.
From the viewpoint of the remaining nodes, the node simply cuts its connections in any case and as such removes itself from the network. Obviously this leads to a hole in the network that must be filled again. The remaining nodes, now take steps to recover the structure, depending on the type of connection that is broken. They are:

1. **Incoming Request Connection** – Listening mode is engaged on the dedicated port, in order to enable further incoming connections.
2. **Outgoing Request Connection** – The connection is invalidated.
3. **Left Neighbour Connection** – First step is to re-enter *WAITING_NEIGHBOUR* state, followed after by marking the entry in the neighbour table as invalid. Then a connection to the second left neighbour is established, using the same process described in the last section. From now on, this node is used as new left neighbour, while the second left neighbour entry is replaced by information gained during the set-up procedure. Finally, the right neighbour is

informed about the change and *WAITING_LONG_DISTANCE* or *READY* state is entered again.

4. **Right Neighbour Connection** – State of the node is changed to *WAITING_NEIGHBOUR*. After that, the only action taken for this connection, is to delete the corresponding entry in the neighbour table (initiative for a new connection always comes from the right neighbour).

5. **Outgoing Long Distance Connection** – *WAITING_LONG_DISTANCE* state is entered and the corresponding entry in the routing table is erased. Afterwards a new *CONNECT_LONG_DISTANCE* message is sent in order to complement the table.

6. **Incoming Long Distance Connection** – The entry is deleted from the routing table.

3.3 Evaluation of the Protocol Implementation

The implementation of the Symphony protocol was tested thoroughly during and after the development. In this section, the test set-up and the tool that was created for this purpose are discussed and eventually the gathered results are presented.

3.3.1 Test System Overview

Testing of software is iterative in any development process and so was one of the essential tasks in the creation of the Symphony protocol implementation as well. In this work the software test was carried out for two purposes. Primarily, the goal was detection of errors and proofing the proper operability of the implementation. Besides this purpose, another aim of the testing procedure was to estimate the delay a node induces through its internal workings. The reason for this is that such a delay possibly influences the efficiency of the protocol. Furthermore, a function test with only a low number of nodes is not likely to show the consequences of a possible high delay. So, further analysis of the behaviour of the node is necessary, especially in view of the considerably slower processing speed of the used hardware and the artificial delays, which had to be introduced for the reasons mentioned in the last section. Because of this need, the delay is measured and subsequently used in the simulation of a large network.

As only a single IRON Box was available in the required hardware configuration, the protocol was not tested in any real Symphony network. On the other hand, set-up of a real network could not even really guarantee operability anyway, considering the poor means of debugging that are provided by the device. For this reason a different approach for testing the Symphony node was taken. First of all, the test network, shown in Figure 3.18 was used, in order to simulate a Symphony network. It consists of three parts – the Symphony node itself, a router and a personal computer. On the computer runs a test software that was developed during this work, which enables a user to manually influence most protocol specific functions and so extensively test the implementation. In addition the wished-for timings are measured and saved for later use. The test tool internally creates a virtual Symphony network, fitted to the real Symphony node. Towards this entity, the software reacts just like a Symphony network would do. It offers TCP ports the node can connect to and responds on the messages appropriately as well.
Test results are now gathered in two ways – first there is the output of the test software and second, the debugging messages the node transmits to a terminal over the serial link dedicated to this task.

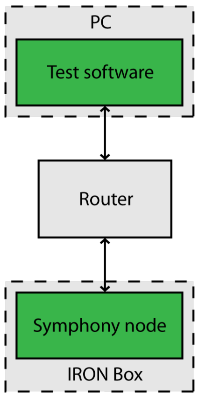


Figure 3.18: Network used for test of the Symphony implementation.

3.3.2 Description of the Protocol Test Software

JAVA was selected as programming language for the test tool. This decision was made for several reasons – JAVA is platform independent and can be used indifferently on any operating system. Furthermore, it is inherently easy to use, while being quit powerful at the same time. The computer language provides an API for TCP sockets and threads as well as a framework for creating user interfaces. It offers as such everything that is needed for the test application, with comparably low performance as the only downside (and high performance is only of low importance).

Base of the software is the Observer pattern [Lar05, p. 463], used in software engineering. Hence, it is divided into the graphical user interface (GUI) and the business logic. Each of the two parts is processed in a separate thread of execution, in order to enable smooth operation. Basically, objects in the business logic are notified of user action and the user interface is notified by the business logic if a change to any GUI object occurred. Message reception and processing of these happens in the business logic layer and, as the name suggests, the GUI is responsible for all user interaction.

The tool now simulates a virtual Symphony network, created around the id of the real Symphony node. In the test software, however, no single nodes are simulated but it simple responds to messages from the real node appropriately. Each "virtual node" is only represented by an id and a corresponding connection to the real node. Every connection that is opened spawns a new thread in order to make the measurement of message arrival as exact as possible (no delay by any loops in the program code can occur this way, as the thread is only listening for messages on the single connection). While despite all efforts, the influence of program execution on the measurements, can not be completely removed, it is expected to be orders of magnitude below the delay induced by the node and the transmission channel.
Creation of the network is not done on initialisation of the tool. Only a single listening TCP connection is opened on the default "request port" (2000), for the real node to connect. Only after it sent a *JOIN* message to the virtual right neighbour, the ids that belong to the two left and the right neighbour, and later on the nodes connected by long distance links, are calculated and subsequently the connections to the real node are established. The ids of the neighbours then are calculated in a predefined numerical distance from the id of the Symphony node and thus relate to a certain network size.

The test application allows evaluation of the following tasks, leaving no protocol function untested:

- Network set-up.
- Adding of nodes (left neighbours, right neighbour and nodes long distance links).
- Removing of single nodes (both left neighbours, right neighbour and nodes long distance links).
- Sending of text messages to any id, using any node as source.

A test run always begins with the initial network set-up, done without any user intervention. After that all other actions are for the user to handle.
During network set-up, the real Symphony node goes through all possible node states, from *DISCONNECTED* to *READY*. At all times, the test tool automatically provides the required responses. After this initial procedure, the remaining protocol functions can be tested by the corresponding options in the software.

In addition to the already mentioned aspects of the test software, a number of configuration options are available to analyse the reaction of the Symphony node under different conditions. The available options are:

- **Seed** – Seed used for the *random()* function.
- **Number of nodes** – The intended network size – translates to a different distance between the id of the real node and the ids of the neighbours.
- **Number of lost *JOIN* messages** – Number of *JOIN* messages that are discarded before one is accepted by the tool (tests the resend timer of the node).

3.3.3 Test and Measurements Results

Three basic results could be gathered from the Symphony protocol implementation itself and the function test, which are discussed subsequently:

1. the code size of the software,
2. proof of operability,
3. delay times for use in the simulation.

Memory, or rather the limited available amount, is one of the major difficulties the IRON Box (and low-performance embedded systems in general) has in respect to any peer-to-peer protocol, the code size and the size of the program in the working memory are essential features of the developed Symphony implementation.
The flash program memory, used for this purpose, is 128 kbyte. In the current state, the code size of the produced software is 21.3 kbyte (the size of the image, used to program the microcontroller), which amounts to about one sixth of the available program memory. However, this includes not only the Symphony functionality but handling of the WLAN module, of the serial connection and a main program. An educated guess of the amount of memory that is needed in addition to the software that already exists and provides this functionality, would be about half of this value – so roughly 10 kbyte. If this is put into perspective to the original IRON software, which requires

about 27 kbyte, the current size of the protocol implementation is easily small enough to fit into the flash memory together with the IRON software.
The size of the code in the flash memory is only one side of the coin, though. Another important factor is the working memory that is required. The demands in working memory consist of a static part, which consists of all variables and constants and a dynamic part, consisting of the stack and the heap (a memory area for dynamical memory allocation). In the current configuration of the Symphony implementation, 2 kbyte of the 4 kbyte that are available are already filled with static data alone – this leaves only 2 kbyte free for memory demands at runtime. If this is taken into account, there is hardly any space for additional software. The IRON application, which is the software the protocol is intended for, requires about 3.4 kbyte of RAM for static data. While the code of the Symphony implementation and the IRON application overlaps for approximately one third overall, this still leads to more than the available 4 kbyte of working memory. From this data only one conclusion can be drawn, currently it is not possible to employ the Symphony protocol implementation as a tool for the IRON application. A number of basic attempts of integration confirm this result – all ended in crashes of the system, usually in combination with garbled debugging messages. However, this issue can be easily solved by use of external working memory.

The archived size of the Symphony implementation is rather small, compared to other available peer-to-peer protocol implementations. Figure 3.19 shows a comparison of the Kademlia implementation used in RevConnect-0.674p [5], JXTA-C 1.0, which is a variety of JXTA for Linux (programmed in C) [4] and Symphony.

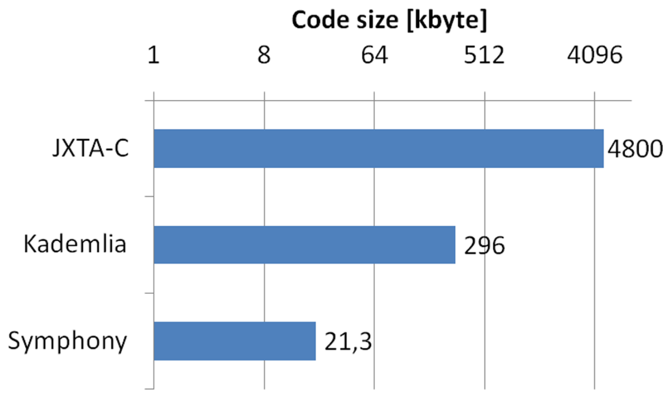


Figure 3.19: Size of different peer-to-peer protocols.

While the size of the code is an ultimate property of the created software, it is useless if the product does not work. To ensure the operability, the function test was carried out 5 times each, using the configurations in Table 3.4, while a different sequences of user actions was used in each run. In addition to the configurations given in the table, a number of runs were executed where the test tool dropped the *JOIN* message two times in a row, in order to check the reaction of the implementation on lost *JOIN* messages.

In respect to operability, no systematic failure of the protocol implementation could be detected, in any of the experiments but two sporadic errors turned up. Now and then, if a *REJECT_LONG_DISTANCE* message was sent from the test application to the Symphony node, it did not reach its destination. It was indeed sent by the test tool but no note of it could be found in the debugging info, collected from the IRON Box. With the help of the WireShark [2] network

Table 3.4: Configurations used for the test of the Symphony protocol implementation.

Seed	Network size
0	100
0	1000
0	10000
140283	100
140283	1000
140283	10000

protocol analyser it was ensured that the packet leaves the computer and due to TCP no packet can simply be lost on the way. So the only conclusion can be that the loss occurs either in the Avisaro WLAN Modul 2.0 or on the serial line towards the microcontroller. As the WLAN module does not allow insights in its internal workings, a detailed analysis was impossible. So no definitive reason for this error could be found. A probably related issue were truncated messages that were received occasionally from the WLAN module. Again no clear reason could be found but the nature of the error suggests usage of flow control mechanism on the serial line.
Beside these problems, two issues of the test software were encountered too. First, occasionally the initial connection between the Symphony node and the test tool was created but then the *JOIN* message disappeared and did not reach the tool. Using WireShark, it was ascertained that the message really reached the computer, the test application was executed on. For this error no reason could be found either but it is of very low relevance anyway, as a restart of the tool solves the problem. The second issue is no error but rather a drawback of the current test software design. As every connection is used with a dedicated thread of execution, the test software slows down the computer dramatically after creation of about 15 connections and must be restarted. This is in principle no issue as by this point enough results are collected for a single run. In this work accuracy of the measurement results is deemed as more important as this topic.

The second results of the function test are the collected delay times. It is to note that the obtained values are the end-to-end delay between the Symphony implementation in the IRON Box and the test software on the computer and not the real latency induced by the Symphony node alone. However, the use in the simulation compensates for this by defining the used TCP connections without any delay and instead introducing the measured values as the only one between two nodes.
Measurement of this delay was conducted with the different configurations from Table 3.4. For illustration, the average delay times that were found for a *CONNECT_NEIGHBOUR_RESPONSE* message are displayed in Figure 3.20. Here no relevant influence of the different configurations is discernible. This outcome is even better highlighted by Figure 3.21 which pictures the measured delay times – again for *CONNECT_NEIGHBOUR_RESPONSE* messages – that were found for all configurations in Table 3.4. In addition to the single timings (the crosses), the average and the standard deviation is shown. Basically all measured values, with exception of a few outliers, are placed around an average of 1860 ms. Yet another angel offers Figure 3.22, which displays the distribution of delay times for the same message type. More than 50% the messages were received in a time frame that ranges from 1500 ms to 2000 ms. Between 1000 ms and 2500 ms, transmission of close to 79% of all sent messages was finished.
For all other message types a similar picture was found – none of the gathered data indicates any relevant influence of either the random seed or the defined number of nodes in the virtual network on the exhibited delay.

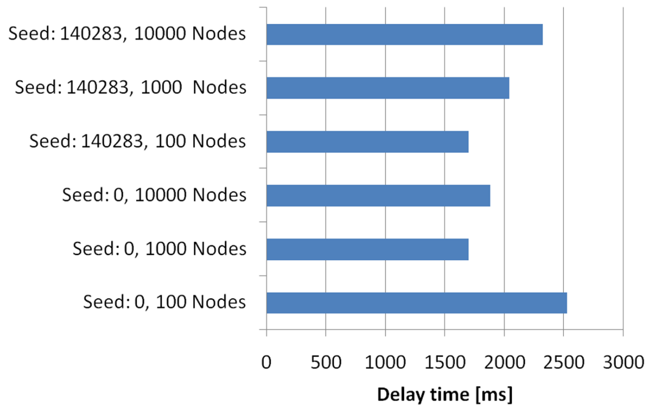


Figure 3.20: Comparison of the average latency a *CONNECT_NEIGHBOUR_RESPONSE* message experiences for the different configurations.

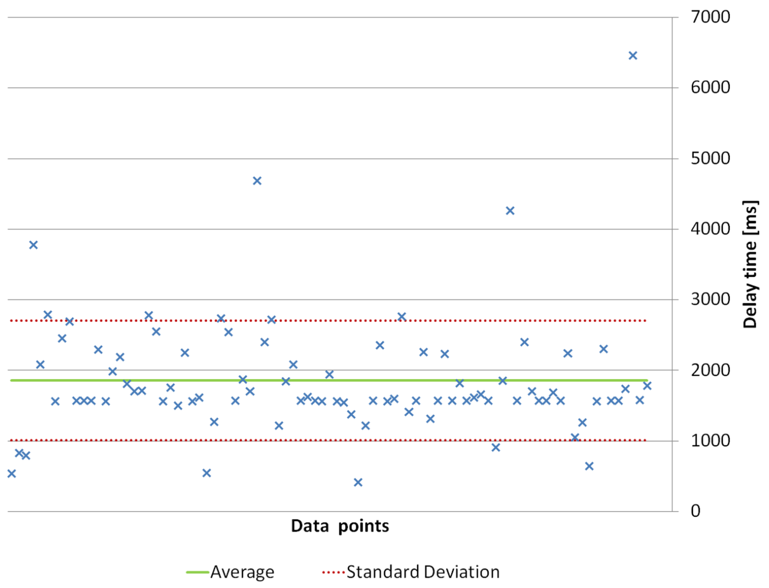


Figure 3.21: All delay times logged for *CONNECT_NEIGHBOUR_RESPONSE* messages, depicted in relation to the overall average and standard deviation.

In order to obtain representative timings for the simulation, the average of the collected values was calculated for the different message types. Table 3.5 shows these calculated average delay times. The values always represent the time by which the corresponding message is delayed before it is sent from a node in the simulation.
Two aspects of these results leap to the eye. Primarily the values are quite high, even for a low-performance embedded system. However, there is a reason that accounts for at least part of it. It is related to a problem that was encountered during development of the software. The Avisaro WLAN Modul 2.0 obviously requires delays between two sequential requests and between a request and the retrieval of the response. These delays are neither documented nor confirmed by the developer but, if not integrated, they lead to missing responses from the module. During the development of the Symphony software (especially the WLAN module control part), delays were inserted at the aforementioned positions and selected other places in the source code, in

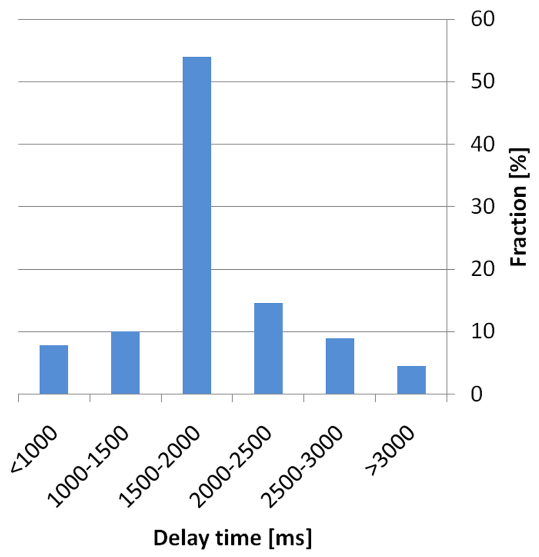


Figure 3.22: Distribution of the delay times for a *CONNECT_NEIGHBOUR_RESPONSE* message.

Table 3.5: Average delay for each message type.

JOIN	0.87 s
CONNECT_NEIGHBOUR_REQUEST_DELAY	2.11 s
CONNECT_NEIGHBOUR_RESPONSE_DELAY	1.86 s
CONNECT_LONG_DISTANCE_DELAY	3.67 s
ACCEPT_LONG_DISTANCE_DELAY	1.82 s
REJECT_LONG_DISTANCE_DELAY	0.88 s
DATA_REQUEST_DELAY	0.87 s

order to ensure operability. These delay times sum up to several hundred milliseconds or even seconds for some tasks and are responsible for the greater part of the node delay, induced by the IRON Box.

Beside the absolute values, the spreading of the measured latency is roughly the range from 1000 ms to 2700 ms, as is shown in Figure 3.21. However, for every message type a small number of values are extremely far of the average – deviations of up to 90 *seconds* were measured. These are probably caused by an extremely high load of the computer that executed this measurements – this might happen if too many connections are spawned during a test run, as was mentioned earlier on.

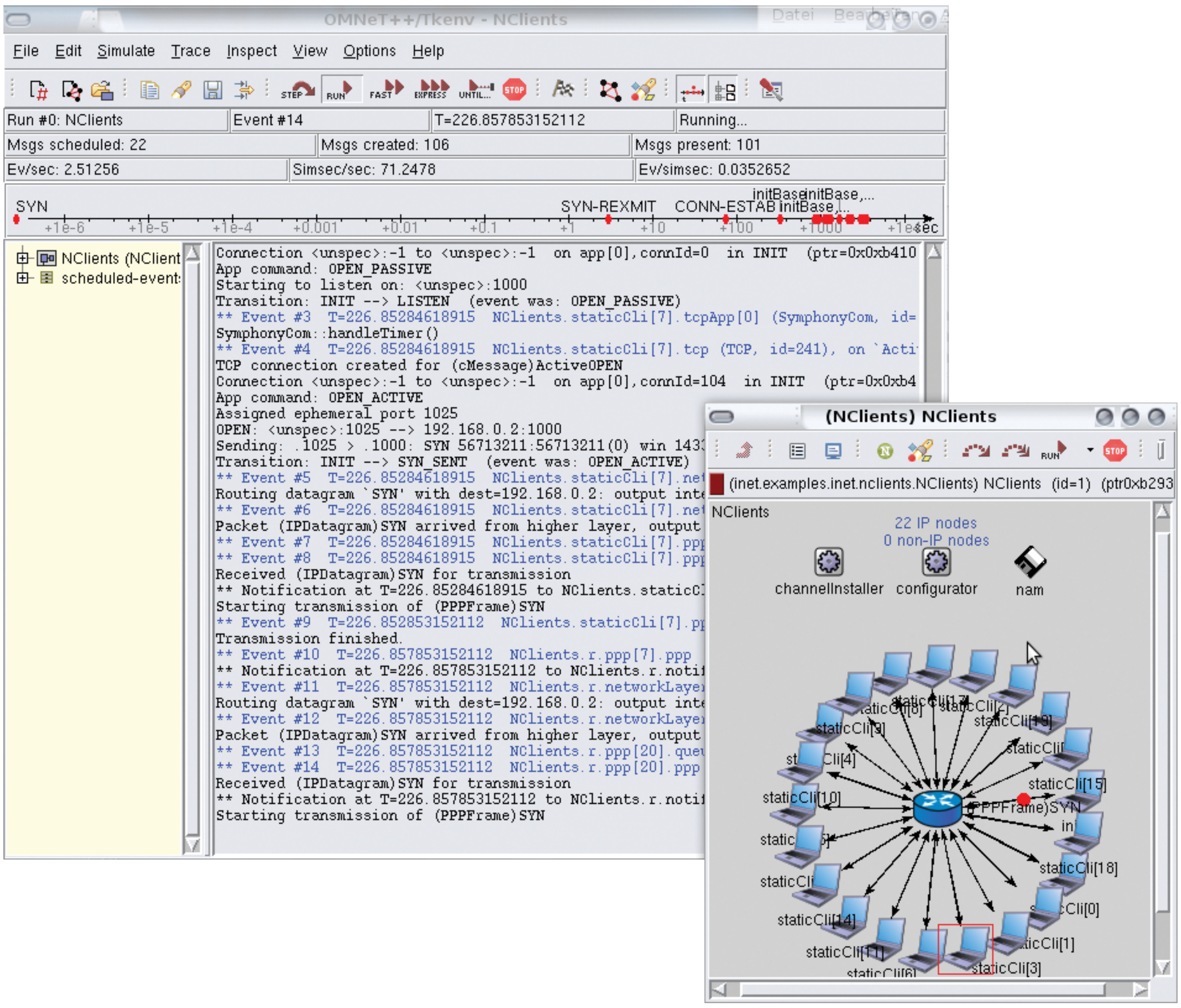
4 Simulation of a large Symphony Network

In the last chapter the designated Symphony protocol design and its implementation were discussed. As a large-scale evaluation is impossible with only a single IRON (Integral Resource Optimisation Network) Box, the normal function test is complemented by simulation of larger networks. These simulations were conducted using the Omnet++ discrete event simulation system [3] and carried out for Symphony networks with up to one thousand nodes. The primary goal of this venture was to test the principal operability of the Symphony design, complemented by comparison of the gathered results with data published by the authors of the protocol.

4.1 The Simulation Environment

As base for the Symphony network simulation, the Omnet++ simulation environment in the version 4.0 was used. It is an extensive and powerful framework that is published under the Academic Public Licence [3], allowing free use in a non-commercial environment. This tool (and its commercial counterpart) sees frequent use in both, the scientific as well as the industrial setting.

One of the essential properties of the simulation framework is its very flexible in every aspect [3]. Besides Omnet++'s availability for different operating systems (among them Linux and Windows), the tool set features a custom integrated development environment and provides both a graphical and a command-line user interface for execution of the created simulations. Additionally, it includes several other tools, among them random number generators and widgets for presentation of scalars/vectors.
The GUI (graphical user interface) allows easy analysis and control of simulations. Events and representations of simulation models are visualised and detailed information about them are available for observation. If defined in the models, even variables of the individual entities can be viewed and in some cases even changed (these variables must be declared accordingly through a macro, though). Beside the data that is available about the individual components, every event is displayed in the GUI screen (as highlighted in Figure 4.1), making it easy to follow the process of the simulation. In most cases, though, control over the system during a simulation run is limited, it mainly consists of starting/stopping of the run and modification of the simulation speed (simulated events per seconds).
Contrary to the GUI, the command-line interface offers neither a graphical representation of the simulated system nor the option to monitor components closely. It is run in a terminal window and prints out certain events, depending on the chosen simulation speed. However, it can be



```
Run #0: NClients          Event #14          T=226.857853152112          Running...
Msgs scheduled: 22          Msgs created: 106          Msgs present: 101
Ev/sec: 2.51256          Simsec/sec: 71.2478          Ev/simsec: 0.0352652
```

```
Connection <unspec>:-1 to <unspec>:-1  on app[0],connId=0  in INIT  (ptr=0x0xb410
App command: OPEN_PASSIVE
Starting to listen on: <unspec>:1000
Transition: INIT --> LISTEN  (event was: OPEN_PASSIVE)
** Event #3  T=226.85284618915  NClients.staticCli[7].tcpApp[0] (SymphonyCom, id=
SymphonyCom::handleTimer()
** Event #4  T=226.85284618915  NClients.staticCli[7].tcp (TCP, id=241), on `Acti
TCP connection created for (cMessage)ActiveOPEN
Connection <unspec>:-1 to <unspec>:-1  on app[0],connId=104  in INIT  (ptr=0x0xb4
App command: OPEN_ACTIVE
Assigned ephemeral port 1025
OPEN: <unspec>:1025 --> 192.168.0.2:1000
Sending: .1025 > .1000: SYN 56713211:56713211(0) win 143
Transition: INIT --> SYN_SENT  (event was: OPEN_ACTIVE)
** Event #5  T=226.85284618915  NClients.staticCli[7].ne
Routing datagram `SYN' with dest=192.168.0.2: output inte
** Event #6  T=226.85284618915  NClients.staticCli[7].ne
Packet (IPDatagram)SYN arrived from higher layer, output
** Event #7  T=226.85284618915  NClients.staticCli[7].pp
** Event #8  T=226.85284618915  NClients.staticCli[7].pp
Received (IPDatagram)SYN for transmission
** Notification at T=226.85284618915 to NClients.staticC
Starting transmission of (PPPFrame)SYN
** Event #9  T=226.852853152112  NClients.staticCli[7].p
Transmission finished.
** Event #10  T=226.857853152112  NClients.r.ppp[7].ppp
** Notification at T=226.857853152112 to NClients.r.noti
** Event #11  T=226.857853152112  NClients.r.networkLaye
Routing datagram `SYN' with dest=192.168.0.2: output inte
** Event #12  T=226.857853152112  NClients.r.networkLaye
Packet (IPDatagram)SYN arrived from higher layer, output
** Event #13  T=226.857853152112  NClients.r.ppp[20].que
** Event #14  T=226.857853152112  NClients.r.ppp[20].ppp
Received (IPDatagram)SYN for transmission
** Notification at T=226.857853152112 to NClients.r.noti
Starting transmission of (PPPFrame)SYN
```

Figure 4.1: The Omnet++ Tkenv graphical user interface. On the left hand side is the main windows with the different controls and the event log and in the right bottom corner is a representation of the currently simulated system.

configured by scripts and enables execution of several simulation runs in sequence, each with run specific parameters. Furthermore, it is far more efficient and allows efficient execution of simulations with large numbers of components.
Independent of the user-interface, Omnet++ has the capability to log diverse information from a simulation. First of all, it is possible to save the event log (created during the simulation) for further analysis. In addition, a number of statistical tools exist that allow collection and calculation of a variety of statistical data about the active simulation.

Omnet++, at heart, is an event based simulator [3]. This has two main consequences – the first one is that a component is only active if an event, related to the entity, takes place. Between two events nothing ever happens in the simulation. However, events can be triggered by many sources, among them messages that are sent from the node to itself at a predefined time in the future.
The second impact of the simulator's nature is that the created simulation models are executed in zero simulation time. So, unlike a real system, execution of code triggered by an event that

corresponds to a certain point in simulation time is finished without any delay. While functions to interrupt or pause execution exist in the C++ toolset, these do not work for simulated models. In order to delay certain events, a method exists that sends a message to the node itself at a certain time in the future. This offers a way to introduce artificial delays in a node, even if no dedicated methods exist for this purpose.
The Omnet++ framework provides a very modular architecture that allows for creation of very complex systems out of components (which can range from very simple to very complex). For this reason each simulation is divided into three parts, alike to the structure shown in Figure 4.2. Any

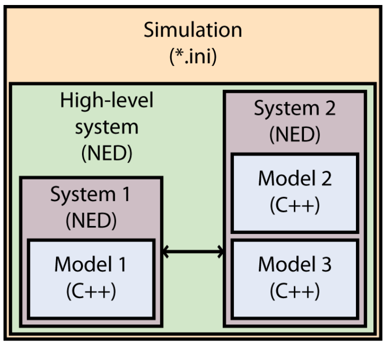


Figure 4.2: Structuring of simulations in Omnet++.

system is created out of custom models, written in the programming language C++. The created models inherit from these basic modules, the properties necessary for integration and interaction in the simulation. All new models base on C++ classes, provided by the framework. While usually custom models are created that reproduce the imagined setup, it is possible to produce simple systems without development of custom C++ models, by simply using the existing basic modules.
Not only models of computing systems are possible, by this approach. Models of communication channels and messages that are passed over them are created the same way. As it is possible to model communication links, a connection between "nodes" is not necessarily a passive line but can have properties, too. This enables, for instance, the creation of lossy and bandwidth-limited channels. There is a similar situation regarding messages in this set-up – they consist not only of the intended content but maintain information about their sender, receiver and furthermore, about the creation, sending and arrival time. They provide interfaces to attach whole objects, which then can be transmitted over the created network. Most importantly, if a message is "sent" the whole object is simply moved, over all intermediate channels and nodes to the target.

From these basic models, larger systems are then assembled using a custom high-level language, the so-called "NED language" [3]. It is required for mapping of the used models and their links to a structure that can be processed by the simulation. This is done in several stages – first of all, every model is described to the simulation by this language, as depicted in Figure 4.2. Then these constructs are further assembled to more complex systems by a layer of "NED language". Then rinse and repeat, until the targeted system is complete.
A simulation basically consists of two parts – the system part that is represented by a file in "NED language" and the simulation part, defined in a $*.ini$ file. In this document all parameters that make up the simulation are listed (for instance, the simulation time or the number of nodes is found in these files). Beside the options relevant for the simulation, parameters that control the system in this specific simulation are defined in it as well. The sum of all these tools enables easy and intuitive creation of very complex systems out of simple building blocks.

While the simulation framework itself provides only a few very basic modules, numerous collections of models for the framework already exist, making it even more powerful as it already is [3]. The different available options range from ”simple” sensor networks to collections, allowing the creation of complex, Internet-like networks. First among them is the INET framework that was created from the same group as Omnet++. This package provides models for basically all technologies that are of importance in local networks and the Internet, including IPv4/v6 (Internet Protocol), TCP/UDP (Transmission Control Protocol, User Datagram Protocol) and even wireless technologies. It is of special importance for this work, as the system in concern uses TCP connections to weave a network.
The INET framework (in the version *INET-20080920*) provides, beside complete network nodes (router for instance) and models of connections, raw models for components with a full TCP/IP protocol stack. From these components and nodes then more or less every desired network can be constructed. Moreover, the links between these nodes are adjustable, allowing for realistic network set-ups. The most important feat for the simulation task in this work, though, is the TCP socket implementation, provided by the framework. It enables use of TCP connections in a similar way as the one used in the Symphony protocol implementation.

4.2 Modelling the Symphony Network

In the last section, the Omnet++ simulation environment and the INET framework were introduced. This section takes the concepts discussed there and describes the simulation of a Symphony network that was created in this work. The discussion is divided into two parts – in Section 4.2.1 the structure and properties of the simulation are the topic. Section 4.2.2, on the other hand, describes and analyses the Symphony model that is used as application on the simulated nodes. However, the focus is in this section is not on the protocol functionality – these was already discussed in Section 3.2 – but especially on the differences between the implementation for the IRON Box and the simulation model. These two versions have much in common but there are a number of differences, none the less.

4.2.1 Overview of the Symphony Network Simulation

The simulated network, used in this work, was not developed from scratch but based on the *Nclients* example application, provided by the INET framework. It simulates a network with a custom number of routers and clients that is similar to a local-area network and consists of following (modified) components, provided by the framework [3]:

- **Router** – Model of an IP (Internet Protocol) router. It uses additional objects for name tracing and configuration of the network.
- **Standard Host** – Generic IP host with the internal structure displayed in Figure 4.3. The figure displays all relevant parts of the model. It basically mirrors the layered structure of the TCP/IP protocol stack. At the bottom is the virtual physical connection to the network (in this case a point-to-point connection – *ppp[0]*). On top of it is the network and finally the transport layer, which either uses TCP or UDP as transport protocol. The whole stack is used by the application (*tcpApp[0]*), in this case of the work, the Symphony ”application”. In addition, a number of utility components exist for the different processes in the network.

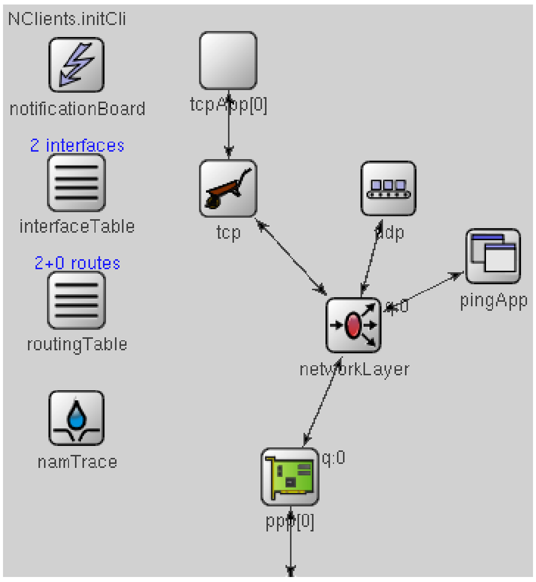


Figure 4.3: Internal structure of an INET framework Standard Host.

This generic simulation was modified to fit the needs of the Symphony network. A simple, star-like topology with only one router in the centre and all other nodes connected to it, was chosen as an underlying network in the simulation (the structure is displayed in the bottom left corner of Figure 4.1). The nodes are represented by *Standard Hosts* and link to the single router by point-to-point protocol. All point-to-point links between the router and the nodes are defined as wired connections with a data rate of 1000 Gbit and no delay. While the framework basically provides wireless and non-ideal channels as well, no more complex setup is needed in the simulation. The reasons for the choice of such a simple network are the employment of TCP as transfer protocol. TCP is a connection-oriented protocol that guarantees successful transmission, even over an extensive and inhomogeneous network, as long as the opposite side is available. It ensures that no data package is lost or garbled on the way, leaving transmission delays as the only influence from the network. However, even the transmission delays on the virtual connections are removed and (effectively) ideal connections are used in the simulated networks. The reason for this step is that the Symphony protocol model introduces artificial delays, which capture the whole delay a message experiences on its way between two nodes (including the desired network delay).

A simulation consists not only of the system that is simulated but is defined by a number of parameters. The configuration options that are important are listed in Table 4.1. Some of the parameters only influence the simulation environment, for instance *user-interface*, which defines the used user-interface. Other options have a direct influence on the nodes, like *startTime* or *sendTime*. However, configuration parameters that influence the behaviour of nodes are not defined for every single node especially but are defined for a whole group of nodes. Currently only two groups of nodes exist – "Initnodes" (initial entry point of the Symphony network – usually only one) and normal nodes.

At start-up of the simulated network, each node is assigned an IP address and connects to the router. While the underlying IP network exists from the beginning, the simulated Symphony network does not start out as a finished network but grows in a predefined time span from a single node to the intended size. For this reason a single dedicated entry node is defined, the

Table 4.1: Important configuration options for the simulation.

Parameter	**Description**
user-interface	Type of user interface used (Cmdenv or Tkenv).
sim-time-limit	Maximum simulation time.
seed-0-mt	Random seed. If none is provided this way, a seed is calculated automatically, depending on the number of the run.
repeat	Number of runs that are conducted sequentially. If no explicit random seed is defined, a new seed is automatically calculated for each run.
n	Number of Symphony nodes in the network, minus one (in every network an additional ”Initnode” is used).
isInitNode	Defines if the node is an ”Initnode” or not.
startTime	Start time of the specified node. This can either be an individual number or a range of values, given by a function (for instance *uniform(1s, 10s)*).
sendTime	Point in simulated time, the node starts to send messages to random nodes. This can either be an individual number or a range of values, given by a function (for instance *uniform(1s, 10s)*).

so-called ”Initnode”. Unlike normal Symphony nodes, an ”Initnode” does not send any *JOIN* message itself but simply waits in the *WAITING_JOIN* state until it receives such a message from any other node. Beside the behaviour at the initial creation of the network, no difference exists between a normal node and an ”Initnode”.
If a node is an ”Initnode” or not is configured by a simulation parameter (*isInitNode* in Table 4.1). After this initial creation of a very small network, basically all active nodes can serve as entry points to the network.

A number of tasks of the Symphony protocol are controlled by random numbers that must be created through the simulation environment. Actually these values are no real random numbers but depend on a seed that is the same for every simulation, if not defined otherwise. This enables comparison of simulations with different numbers of nodes or start-up times, as the sequence of created random numbers will stay the same over all created simulations. Naturally, it is possible to choose another seed as a configuration parameter for a simulation (*seed-0-mt* and *repeat* in Table 4.1 influence the seed). Taking the dependence of the Symphony protocol on random number into account, a different seed causes a different outcome.

In addition to the parameters mentioned above, a number of other important options are available and presented in Table 4.1. Among them are the network size, the type of user interface and parameters that influence the behaviour of the model. All mentioned parameters are of high significance for the created simulation. Beside these are a number of variables that are of lower importance and as such not explained here. All of them are found in the simulation configuration file (omnetpp.ini by default).

4.2.2 The Symphony Simulation Model

After a general description of the simulation in the last section, the simulation model of the Symphony software is introduced here.
A Symphony node consists of several parts, as displayed in Figure 4.4. The *Standard Host* represents the node in the created underlay network. Internally it contains two major parts – the

Network Stack (basically a TCP/IP stack), which handles everything related to the underlay network and the *TCPApp* (*tcpApp[0]* in Figure 4.3) that represents the Symphony model to the *Standard Host* model. *TCPApp* represents a generic interface that declares how the model is

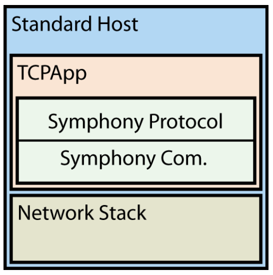


Figure 4.4: Structure of a Symphony node in the simulation.

integrated into the simulation. It specifies the gates that are used to communicate with the lower layers and furthermore, defines the node specific simulation parameters.
The Symphony model is divided into the two layers – *Symphony Protocol* and *Symphony Communication* – again, as was defined by the design in Section 3.2. *Symphony Protocol* is almost the same as in the implementation (Section 3.2.1). Differences are mainly in the code style and the interaction procedures with the *Symphony Communication* layer. However, the *Symphony Communication* layer, is something else entirely, as adaption to the changed environment is done here.

The function of the *Symphony Communication* layer, naturally, has not changed. Again all communication related tasks are handled on this layer but the way it is done is quite different. Primarily this is for the event based nature of the simulation environment.
On every layer, a procedure is only ever started by arrival of a message. As soon as a message is received the *handleMessage()* method of the affected layer is called and subsequently initiates whatever action is required as a consequence. This has one essential implication – collection of messages from lower protocol layers by the Symphony model is not necessary (or possible), as they are directly received through the lower layers of the *Standard Host*. However, these messages must not necessarily come from any other node or layer but can have the origin in the model itself. Omnet++ provides a method that schedules messages for the model itself at a future time. This actually is an important feature, as it enables delayed actions and execution of tasks at regular intervals. In the Symphony model it is used for timer (for instance, resending of a *JOIN* message after the time-out) and for the artificial delays, inherited from the IRON Box.
A further peculiarity of this environment is the handling and properties of TCP sockets. Contrary to the software used on the IRON Box, the INET framework provides a complete TCP socket implementation that eases connection handling and sending/receiving of messages severely. The application must not watch the state of sockets – it is notified by special methods about every change in the state as soon as they are noticed by the underlying layers. This is done by the *socketXXX()* methods – for every change of state a custom method exists. in the name of the method corresponds to the change that occurred (for instance, *socketEstablished()* or *socketDataArrived()*). These methods are found in the *Symphony Communication* layer and each of them triggers further processing.

It was already mentioned earlier on that an artificial delay is introduced by the Symphony model that "pauses" the processing of messages in a node. This is done to enable analysis of the influence the delay has on the protocol operability. The simulation framework does not innately provide any option to interrupt processing of incoming messages in a model. However, it is possible to use the existing scheduling mechanism (sending a message to self at a defined point in the future) for this end. Basically this is archived by pushing outgoing messages on a special buffer (though not the outgoing buffer) and scheduling a self-message for the desired point in the future (the delay for the specific message). If any message is received by the node during this period, it is pushed on a first-in first-out buffer (the incoming buffer) as well and only processed after expiration of the artificial processing time. So, the delay, measured during the implementation test, does not only influence the sending of messages but the processing as well. While this approach does not really pause processing of any code in the node, it results in a behaviour similar to the one exhibited by the software in the IRON Box.

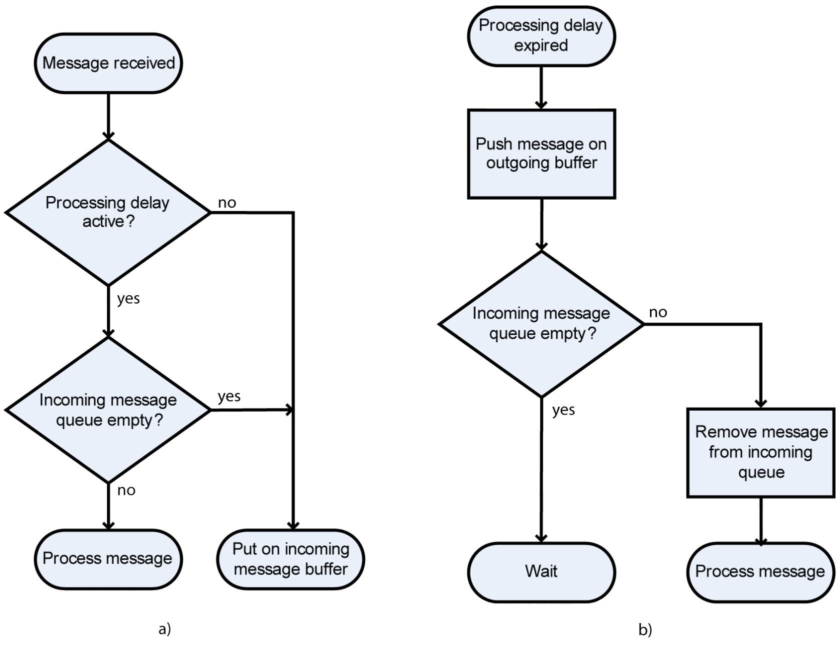

Message received
Processing delay active?
no
yes
Incoming message queue empty?
yes
no
Process message
Put on incoming message buffer
a)
Processing delay expired
Push message on outgoing buffer
Incoming message queue empty?
no
yes
Remove message from incoming queue
Wait
Process message
b)

Figure 4.5: Sequence for processing an incoming message in the Symphony simulation model.

Figure 4.5 illustrates processing of an incoming message and the procedure that is triggered by it. Figure 4.5 a) shows the procedure that is executed on reception of a message and Figure 4.5 b) pictures what happens after a processing delay ends.
The first step taken on reception of a message is to check if currently any processing delay is under way. In case that no delay is scheduled, the in-buffer is examined for any messages that arrived earlier on. Now, if either a message is already in the in-buffer or the node is locked by a processing delay, the fresh message is placed at the end of the incoming buffer as well. Then the node suspends until a further message is received (this can be a scheduled message as well, of course), as no processing must happen during lockdown by an active processing delay.
After a processing delay expires, the message that triggered it is taken from the delay-buffer and put on the buffer for outgoing messages. Now, the in-buffer is checked for any messages that wait for further processing. If no message is found in the buffer, the node suspends until further notice. On the other hand, if one or more exist, the oldest message is removed from the buffer and subsequently processed. Often, this will lead to engagement of a processing delay again and the whole procedure starts over. In case that no outgoing message is produced, for instance if the received message was an *ACCEPT_LONG_DISTANCE* message the next message on the incoming buffer is processed. Up to now, three distinct buffers were mentioned. The first

one is the delay-buffer that is used for messages that trigger a processing delay. The maximum number of messages stored in this buffer is fixed at every point of execution. It only depends on the number of messages that are sent in a single message-handling method (as no further message is processed during the delay). In contrast, the maximum number of messages in the in- and out-buffer is basically not limited in any way. These two constructs are used, as the name suggests, to buffer incoming and outgoing messages.
Use of the inbound buffer was already explained earlier on. The outgoing buffer is used for all outgoing messages. These are placed on it after expiration of the corresponding processing delay and transmitted in a regular interval. So it models the procedure, used in the IRON Box-implementation, despite the event based environment. Furthermore, it is used to compensate for the time that is required to establish TCP connections and messages are stored in it during the time that is required to handle a connection error, if the specific message was supposed to be sent over the failed channel.

The options given in Table 4.1 control the execution of the corresponding simulation. In addition to them exist a number of parameters that have a direct influence on the behaviour of the model. Contrary to the simulation options, which are defined in the simulation control file ($*.ini$), these are represented by compiler constants and required new compilation of the model's source code. While many of the variables are identical to the options used in the implementation, there are some differences as well. Table 4.2 lists the parameters that are of relevance for running the simulation (a full listing is found in Appendix A). The default values, specified in the table, are the values that are used in the Symphony protocol implementation for the IRON Box.

Table 4.2: Important configuration options for the Symphony model.

Parameter	**Description**
LOOKAHEAD	Defines if lookahead information is used for routing (Default: 1).
NUM_LONG_DIS-TANCE	Number of long distance connections that are established per node (Default: 3).
MAX_INC_NUM_LONG _DISTANCE	Maximum number of incoming long distance connections that are accepted (Default: $2 * NUM\_LONG\_DISTANCE - 1$).
MIN_NUM_FOR_LONG _DISTANCE	Minimum estimated network size that is required for use of long distance connections. If the estimated number of nodes in the network is below this value no long distance links are established (Default: 6).
RESEND_TIME	Time that is waited until a *JOIN* message is sent again (Default: 30).
_DELAY	Delay time for the message (Default: Table 3.5).

4.3 Analysis of the Simulation Results

The simulation of a large Symphony network was realised with three goals in mind. Primarily, the aim was to validate the design for networks consisting of a large number of nodes. While the basic functionality was proven during the hardware test (Section 3.3), analysis of the behaviour while facing a large number of nodes can only be done using the simulation. So the task was to find out if it is working in this setting.

Beside this basic analysis, the next goal was to evaluate the performance of the routing algorithm. The reason for this undertaking was not only to gain knowledge about the actual routing performance but to provide some data for a comparison with the original publication of Symphony [MBR03]. Failure of finding any significant difference in the results can be taken as a further proof of the design's operability. On the other hand, if any noticeable deviation shows up, a difference or even a failure in the design must be taken into consideration.
Finally, certainty about the effect of the node delay on the routing process (the message delays, measured during the implementation test) is to be gained. Of special interest is the influence of the node delay on the allocation of the buffer for incoming and outgoing messages.

In order to gain information about the protocol that corresponds with the goals mentioned above, several different simulations were conducted. Each simulation was carried out for network sizes between hundred and one thousand nodes and lasted 80000 simulated seconds. All simulation runs started out with a single node, growing to the defined size in a predefined time (the parameter *startTime* in Table 4.1 influences this time). The specific time of growth for the network, was

Table 4.3: Start time and send time for each network size.

Parameter	**100**	**500**	**1000**
startTime	*uniform(1 ms, 100 s)*	*uniform(1 ms, 500 s)*	*uniform(1 ms, 1000 s)*
sendTime	1000 s	2000 s	4000 s

dependent on the intended network size and is shown in Table 4.3.
In runs, executed with the purpose of measurement of the routing latency, messages were sent in regular intervals, to randomly chosen nodes after set-up of the network. For this purpose the configuration parameter *sendTime* was defined – the values chosen for different node sizes is displayed in Table 4.3 as well. So, the usual sequence of events was

1. Initialisation
2. Network growth
3. Transmission of messages to random nodes.

The first conclusion is that the protocol design does work even for a larger number of nodes. Start-up of the network must happen slowly, though, with only a small number of nodes attempting to connect at any one time. This is mainly for the constrained number of TCP connections that are possible concurrently and the used approach with a signalling (request) connection. If a large number of nodes join the network at the same time, the whole process will stretch severely in time. The reason for this is that at any one time only a single node can connect to the single incoming "request port" of the entry node, in order to transmit its initial *JOIN* message. All other nodes have to wait and will try it again after a certain time. After expiration of this period, all the nodes will attempt to send the message again but, again, only one will succeed. This process goes on until all nodes are connected. While this is no issue for a real network it does lead to errors in the simulation, for the reason that the number of messages increases to a point, the simulation engine can not handle them anymore.
Another limit was found that relates to fault tolerance of the Symphony network. Each node stores only information about a single node that could serve as replacement for a failed left neighbour. In case of failure of more than one immediate neighbour, the network is broken, as no

life node is known anymore which could be used as substitute for the failed nodes. This actually is no error but a design decision, limiting the required amount of data that must be stored to one neighbour entry (10 bytes).
Anyway, if the random distribution of nodes in the underlying network is taken into account, this property usually will not represent any problem. It is very unlikely that several nodes that are neighbours in the Symphony network are in the same physical vicinity too. So this topic does not present any discrepancy to the requirements either.
However, a higher level of fault tolerance can be archived by adding information about more neighbours, while every entry adds 10 bytes. This approach is discussed further in [SMK+01].

4.3.1 Influence of the Number of Long Distance Connections

The number of long distance connections that are used for routing, strongly influences the behaviour of the Symphony network as a whole. This taken into account the average routing latency can be used as an indicator for the correct working of the routing algorithm.
Comparison of the gathered results and the data provided by Symphony's authors only can give a rough estimate. A number of details of the protocol design are not explained in detail in the paper [MBR03]. Furthermore, no exact numbers are available as a source of data but only the published graphs. For these reasons, certain differences are likely to appear. However, while no exact numbers are available for comparison, the basic trend of the results should match to a certain degree (see Figure 4.6 and Figure 4.7).
Beside the usefulness of the average latency for the evaluation of the simulation behaviour, it too is a measure for the overall routing performance. The data collected in the simulation will offer a good clue on the performance that can be expected from the IRON Box-implementation and eventually provide indications for improvements to the currently employed setup.

In order to estimate the influence of the long distance links, a number of simulation runs were executed for 100, 500 and 1000 nodes. All of the runs used the simulation configuration parameters mentioned above. For this part of the simulation, the Symphony protocol is configured to apply lookahead information for routing (see Section 3.2.4) and the number of long distance connections was used as the distinguishing factor between the single runs. Simulation runs were conducted for one to seven long distance connections, maximising compatibility with the existing data. In this configuration, each long distance connection amounts to 106 bytes (see Figure 3.8 and Figure 3.14).
Each run followed the same path – each node joins at a random time, assigned to the node at start-up. After a predefined time (*sendTime*), all nodes in the network start to send messages to randomly calculated ids. The transmitted message collects the number of hops it makes on its way from its origin to the specified id, which is then stored on the targeted node, for later analysis. This was done for several million messages in every run. At the end of each run, the average number of routing hops per message (the average latency) is calculated by

$$average\_latency = \frac{\sum_{i=0}^{n} hops_i}{n}$$

(n is the number of messages and $hops_i$ the number of hops, message i required to reach its target) and stored. The final result is displayed in Figure 4.6.

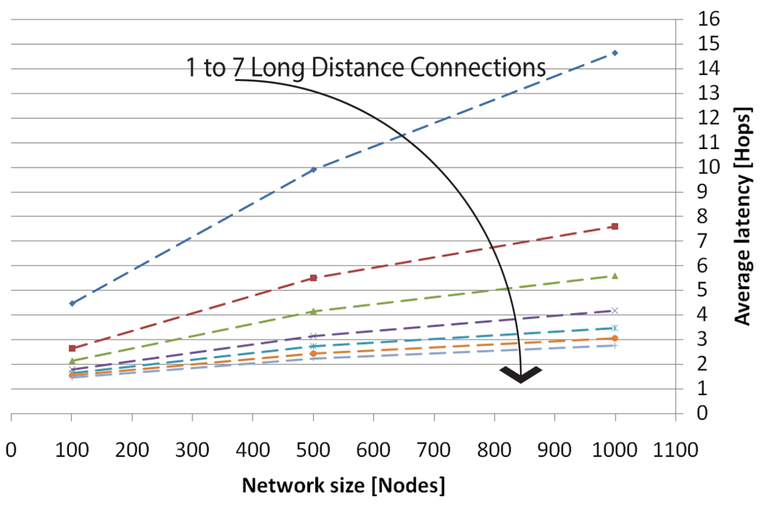


Figure 4.6: Measured effect of different numbers of long distance links on the routing latency.

The figure indicates that even a low number of long distance connections leads to a good average routing performance. For three links in a network of 1000 nodes, 5.61 hops per message are required in average to deliver a message. Increasing the number of links, improves the routing latency but with each additional connection, the improvement shrinks. While the gain between one and two connections is 48% it decreases to 26% if the number is increased to three long distance links. The difference between six and seven connections is even below 10%.
Now if transferred in the time domain, using the timing data in Table 3.5, the following picture appears. Using three long distance connections in a network of 1000 nodes and an average delay of 0.87 seconds per node (the delay induced on *DATA_REQUEST* messages by the IRON Box), a message is exchanged between any two nodes in about 4.8 seconds, on average. One additional long distance link reduces the average transmission time by about one second to 3.7 seconds, while the same process, using only one or two long distance links, requires respective 12.7 and 6.6 seconds.
A higher number of long distance connections are not possible due to hardware limitations, so it is out of the picture. Reduction of the number of links to two, though, is possible and frees up to two TCP connections for other tasks, without impairing the routing performance severely. These two additional connections could, for instance, be used as additional "request ports" or for a client-server connection that is independent of the Symphony network. Further "request ports" possibly enable faster construction of the network but inclusion of these additional ports would require extensive adaption of the protocol implementation. In addition, it comes at the cost of longer delays during use and more complexity (related to the management of several "request ports"). If large scale modification of the Symphony implementation is necessary anyway, a different approach presented in Section 5.2 is more promising than this modification. Anyway, if all this is taken into account, three long distance connections are a good trade-off if no TCP connections are needed outside of the Symphony network.

Figure 4.7 displays the results published by the authors of Symphony (the diagram was reproduced from the original paper [MBR03] and can only provide a rough estimate for the real numbers). The trend of the graphs in both graphics is similar and shows the expected behaviour. In small networks, the routing latency is low and grows with the size of the network.
For four to seven long distance links, the average latency is roughly the same over all network

sizes. However, the graphs are increasingly diverging with node size and decreasing number of connections. While the results in both figures are largely the same for a hundred nodes, the difference is largest, with about 38% at the thousand nodes and one link. A possible reason for this behaviour is the use of lookahead information. The explanation of its use in Manku et al. [MBR03] does leave several degrees of freedom. For instance, it is not specified if lookahead

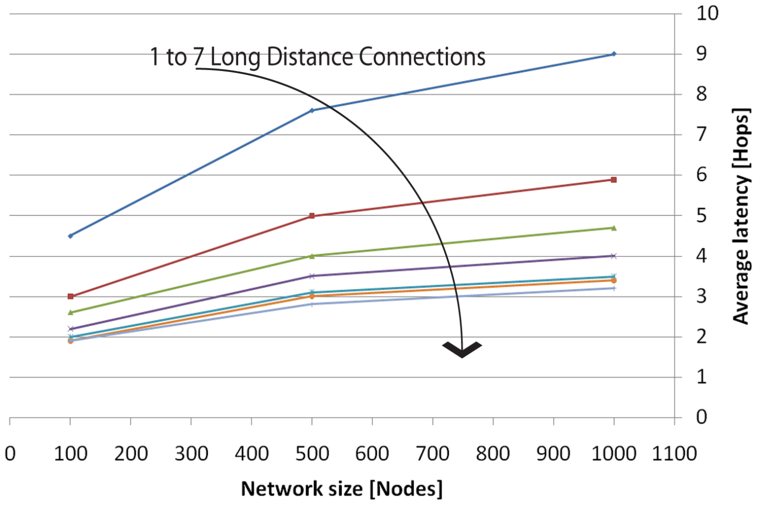


Figure 4.7: Reference result from [MBR03].

information is used only for long distance connections or for the two neighbours as well. In addition, exchange of lookahead information by keep-alive messages is proposed in addition to attaching it to routing messages. Considering this uncertainty, it is possible that these aspects of the design are used different in this work. Lookahead data is only maintained for routing table and no keep-alive messages are used in the simulation (the IRON Box does not provide the means to attach data to keep-alive messages either). As the influence of lookahead data grows with the network size, it is probable that at least part of the deviation is caused by the differences in design. None the less, further analysis is needed in order to resolve this matter satisfactorily.

4.3.2 Influence of Lookahead Information

Symphony offers an improvement for routing that uses additional (lookahead) information, in order to improve the routes. It is described in Section 3.2.4.
According to Manku et al. [MBR03], lookahead information is responsible for about forty percent improvement of the routing latency. The downside of this feature is that for every routing table entry, a quite high amount of data has to be stored additionally. For instance, the currently used size of the routing table is eight – so eight ids have to be stored, in addition to the already existing parts of the routing table entry. An entry consists of the id and the socket information – handle, IP, port and state – all in all 14 bytes. Lookahead data for one entry consists of seven times the id, which makes 32 bytes. This is more than double the data required for the rest of the entry. As this is a huge increase in the memory requirement, a thorough check of the efficiency is in order.

The necessary simulation runs are again executed, using the simulation parameters presented earlier on, for 100, 500 and 1000 nodes. In order to check the influence of lookahead information on the routing performance, the lookahead information was deactivated, for these simulations. Contrary to the simulations in the previous section, three long distance connections were used for all runs, aligning it with the configuration, used for the IRON Box.

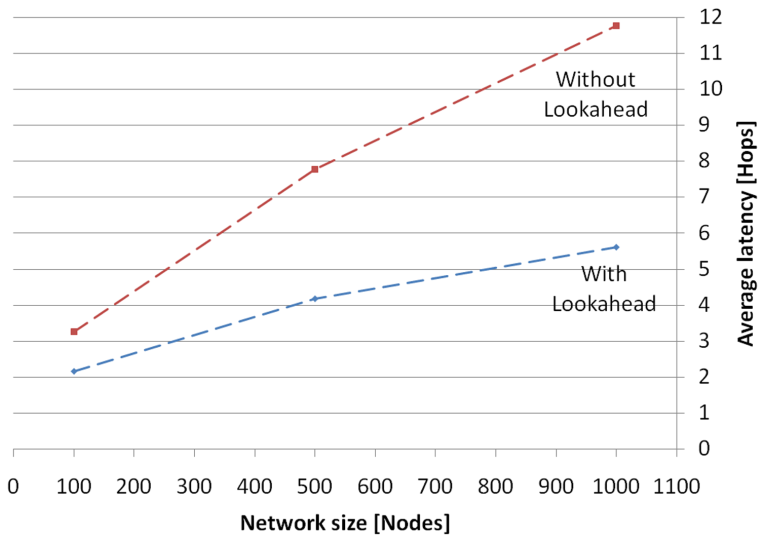


Figure 4.8: Comparison of the routing performance, with and without lookahead information.

Figure 4.8 displays the comparison of the measured average routing performance, with and without 1-lookahead. Over all simulated network sizes, the average latency is significant lower if lookahead information is used for routing. While the improvement in a smaller network with 100 nodes is only 33% and so below the proposed 40%, the situation is quite the opposite for larger networks. At 500 nodes, the improvement is already roughly 46% and routing in a network of 1000 nodes gains the most, with about 56%. On average, this makes an improvement of 44% – about the number given by the authors of the Symphony protocol.

The main conclusion from the gathered data and the considerations concerning the memory requirements is that use of lookahead information strongly depends on the needs of the application that employs the Symphony protocol. If more working memory is required as is available beside the Symphony implementation, removing of is a definite option. On the other hand, if there is sufficient space and fast routing is iterative, then lookahead data is a definite improvement.

4.3.3 Influence of the Node Delay

During the test of the protocol implementation, the delay introduced by the IRON Box was measured for different message types. The gathered timings (Table 3.5) were used in all simulations discussed up to now. After evaluation of Symphony protocol in larger simulated networks, the influence of the node delay is analysed and discussed in this section.

In order to collect data on how the Symphony network is influenced by different timings, a number of simulations were conducted, each with a different message delay. All simulations were carried out for a network of 100 nodes, using the simulation and model configuration parameters proposed above. Furthermore, the number of used long distance connections was fixed at three and lookahead was employed as well, in order to maintain comparability to the implementation.

The selected message delays were taken from the interval [0 s, 25 s] with a step every five seconds, which is used for all message types. This includes delay times below as well as above the default delay and should highlight any influence of the timings on the network. For each delay, seventeen simulation runs were conducted, each with a different random seed. During every run the maximum number of messages in the in- and out-buffer was collected.

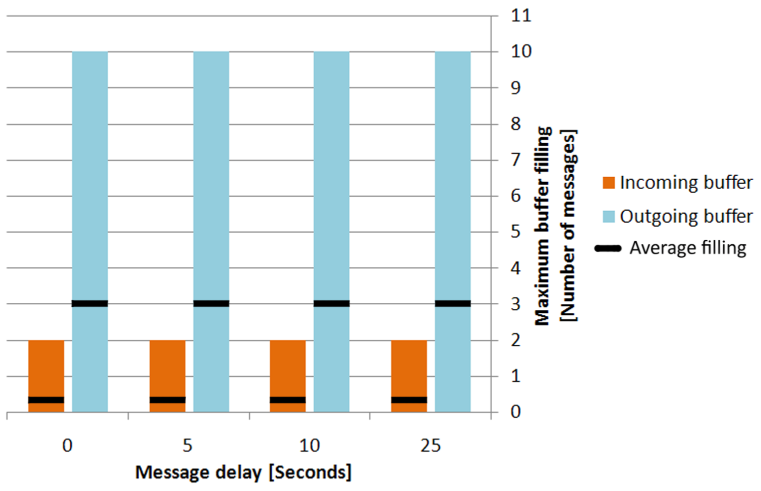


Figure 4.9: Maximum number of messages in the incoming and ougoing message buffer for different message delays.

The maximum number of messages in the input and output buffer for different delay times is displayed in Figure 4.9. Basically there appears to be no influence of different delays on the number of messages that occupy either of the buffers at any time. The number of messages in the output buffer does not depend on the delay of the messages as in the current design no new message can be created during the message delay (no processing happens during this time – see Section 4.2.2). It is mainly affected by the amount of time needed to establish a TCP connection and these timings were not modified in the simulations.
However, the indifference of the input buffer filling towards the changed timings comes as a surprise. From the gathered data, it looks like the only influence of the node delay is a stretching in time – all tasks simply take longer but the overall process of communication stays in proportion.

While no sizeable effect of the node delay on the buffer filling is recognisable, there is one aspect about the output buffer filling that is of importance. The maximum number of messages found in the output buffer (Figure 4.9) is ten at any one time. This is double the currently used size, which is fixed to five in the current implementation for the IRON Box. From this data, it is to be expected that a buffer overflow is possible in the configuration used at the moment. An increase of the output buffer size in the implementation would aggravate the situation in respect to the required working memory, as each additional buffer slot would add about 150 byte to the needed amount (this is the default maximum message size for a Symphony message – see *MAX_MSG_LEN* in Table 3.2).
However, during further investigations in this matter, only a very small number of nodes counted more than three messages in the outgoing buffer at the same time, among several thousand nodes. The most likely reason for this rare occurrence of many messages in the out-buffer, is that certain nodes are not able to connect to the "request port" and so have to attempt resending of the initial *JOIN* message several times, without success. In this case, the buffer is not erased in the simulation, contrary to the same situation in the IRON Box-implementation.

5 Conclusion and Future Work

In this work a peer-to-peer (P2P) protocol, suitable for a low performance embedded system, was selected and subsequently adapted for an existing hardware platform. The resulting software was then adapted for simulation in Omnet++, in order to evaluate the operability in larger networks. In this chapter, the conclusions of this process are summed up, followed by possible improvements.

5.1 Conclusion

The task of this work in a nutshell was to find a P2P protocol that meets the requirements of the IRON (Integral Resource Optimisation Network) project and furthermore, is modest enough to work with the limited resources offered by the IRON Box. Despite the availability of many P2P protocols only a very small part even meets the two basic demands – they either represent no decentralised network or are not able to reach every active node in the network. Among the few remaining protocols only Symphony really manages to cope with the remaining requirements, which are mostly related to the severely limited resource pool. This protocol was then implemented as a proof of concept.
From the implementation and the subsequent simulation a number of results were collected. The first and foremost is that the Symphony implementation for the IRON Box satisfies all given requirements. Symphony as a structured, pure decentralised P2P protocol ensures that the first two demands – no single point of failure exists and it is possible to reach every node in the network – are met. Furthermore, the self-organising properties of the protocol reduce the chance that splitting of the network occurs. Splitting in a Symphony network only can happen if several neighbouring nodes fail within a short period of time, which is extremely unlikely due to the random distribution of nodes on the circular structure.
Further requirements have their origin in the IRON Box. There are 128 kbyte of flash program memory available and only about 21 kbyte are needed by the Symphony implementation (Figure 5.1 a)). This leaves an abundance of program memory for other tasks. Compared to other P2P protocols, the size of 21 kbyte is extremely small, as is displayed in Figure 5.2. It displays a comparison of the developed Symphony implementation with a library that represents the Kademlia P2P protocol, used in RevConnect-0.674p [5] (a P2P file-sharing application). The byte-size of the software created during this works is less than a tenth of the other protocol. In respect to working memory, the developed software requires 2 kbyte, of the 4 kbyte that are offered by the Atmel AVR Atmega128, for static data (Figure 5.1 b)). This leaves about half of the available RAM for the stack and the heap of the microprocessor, which is enough for use of the protocol.

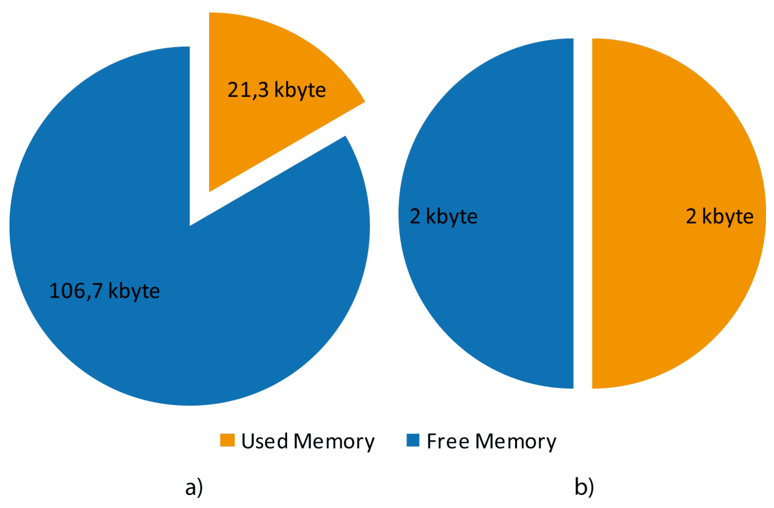


Figure 5.1: Comparison of the amount of memory required by the Symphony implementation to the amount available. a) shows the result for the flash program memory and b) for the working memory.

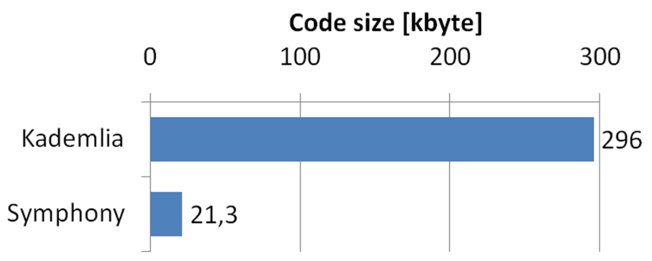


Figure 5.2: Size of different peer-to-peer protocols.

The crucially low number of possible concurrent TCP connections provided by the used WLAN module is just enough for the implementation design, created in this work. While the low number of possible connections reduces the routing performance, it is acceptable for the intended network size. Last but not least is the requirement to allow Symphony networks of up to thousand nodes. Extensive simulations showed that this is basically possible with only minor modifications of the implementation.

The Symphony protocol in the current form works on the IRON Box. However, during the test and simulation certain limits were highlighted as well. While the available working memory is currently not fully allocated by the protocol implementation alone, only a small part remains for any application. The IRON application alone, for instance, uses 3.4 kbyte of the 4 kbyte of available RAM. Even if the intersection of the code is taken into account (the main program, the serial transmission and WLAN control part are the same), the available memory is filled up with statically data alone, leaving no space in the RAM at runtime.
One way to tackle this problem is to optimise the protocol for lower usage of working memory. This can be done by two modifications. First of all, it is possible to reduce the number of outgoing long distance connections to two, from currently three. This lower number of long distance connections would lead to a reduction of the routing table size by two entries and of memory allocation by 106 bytes (see Figure 3.8 and Figure 3.14). The second possibility to reduce the required amount of RAM is to deactivate the use of lookahead information (Section 3.2.4), which reduces the

demand by additional 100 bytes. Beside this direct reduction this modification has an additional advantage. If lookahead information is removed completely from the implementation design, it leads to a further reduction of the static and dynamic memory allocation, as no effort has to be made for management of this information. Altogether, these two adjustments lead to a sizeable reduction of the size of statically data in the working memory, as is displayed in Figure 5.3. This

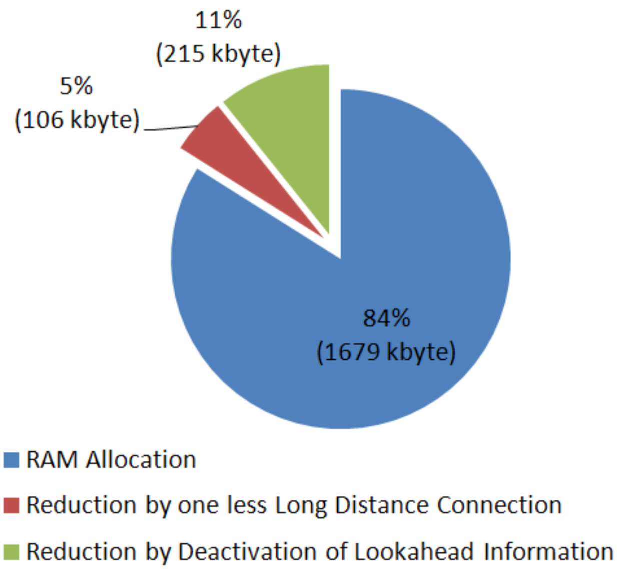


Figure 5.3: Decrease of the size of statically data in the RAM by reduction of the number of long distance in combination with deactivation of lookahead information.

improvement is amplified by an undetermined decrease of the demand for RAM by dynamical data. Although this changes offer a noticeable reduction of working memory allocation, it comes at a high cost. The two changes will reduce the routing performance by estimated 70%, which will correspond roughly to an average transmission time of fifteen seconds for a message that is routed through the Symphony network. Not only the time of a message in the network is quite high, this lengthy delay also increases the time a node requires to join the network.
The memory demand of the implementation is not the only issue. Every decentralised network has the problem that a node that intends to join needs a node which is already active in the network and so is able to serve as an entry point. In the current implementation design only a single node is used for this purpose, whose IP address is hard coded. If a large number of nodes attempts to join at approximately the same time, this poses a problem as each node has to contact this entry node in order to transmit the initial message. In this case, the process of joining will stretch in time by an amount proportional to the number of nodes that attempt to join concurrently. Furthermore, a failure of this specific node implies that no new node can enter the Symphony network anymore. Of course, the IP address of the entry node can be changed. By using a different entry node for different nodes, the load on one node through joining can be distributed over many nodes. While this solves the problem that nodes are delayed seriously in the joining process, it does not help in the case that the indicated node is not available. Eventually, either a list of possible entry nodes or a method of finding such a node dynamically solves this issue.

In this work, Symphony networks of up to thousand nodes were simulated. However, it is of importance as well, to know how the system behaves for larger networks. Although, it is selfevident that only rough estimates are possible from the collected data. Figure 5.4 shows an estimation of how the average routing latency will grow for networks up to several thousand nodes. The upper bound (the red graph) is an approximation of the average latency, which takes the deviation of Figure 4.6 and Figure 4.7 for three long distance connections (the configuration used on the IRON Box) into account. Furthermore, for the lower bound a reproduction of the results gathered by Manku et al. [MBR03] is used. Now, if the worst case is considered, the average routing latency

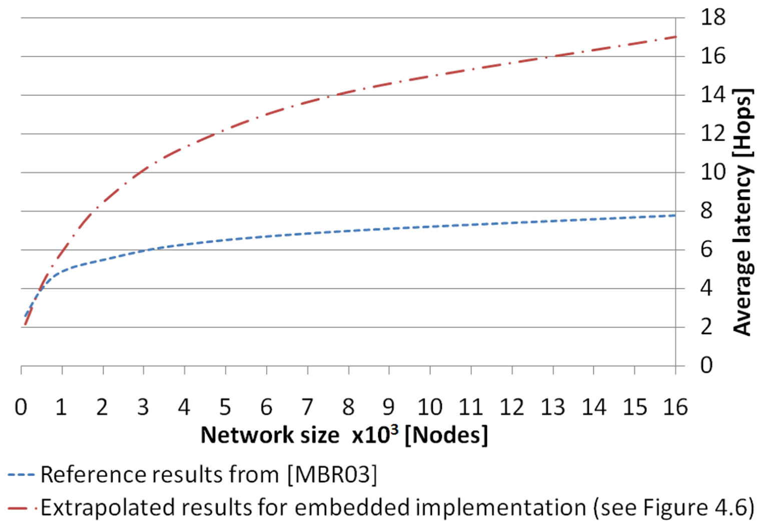


Figure 5.4: Estimation of the routing performance of the Symphony implementation for larger Networks. The extrapolated results from this thesis correspond to Figure 4.6, while the referenced results are reproduced from [MBR03].

in a network of 1000 nodes is about 5.6 hops, for 5000 nodes it increases to 12 hops and in a network of double this size routign of a message will take on average 15 hops.
On their own, these numbers appear quite low. Distribution of activation levels in the IRON network, however, puts a different complexion on them. Even the simplest approach demands that each node must be contacted by at least one other node. Furthermore, the delay time per node, introduced by the IRON Box, is about 0.9 seconds for a data message. On average this means that the (at least) thousand messages that are necessary for distribution of levels take about one and a half hour. For a network of five thousand nodes it will already take fifteen hours and for ten thousand nodes it is more than one and a half day. If these activation level updates must be performed only every couple of days, even networks of several thousand nodes are conceivable. None the less, this estimation assumes that distribution of activation levels only requires one message per node, if more than one is needed, these durations will grow quickly to several hours even for small networks.
The Symphony protocol enables completely decentralised interaction and cooperation of embedded energy management nodes in the IRON network. However, after all it is just a tool and how it is best put to use must still be discovered.

5.2 Future Work

In this work a prototypical implementation of the Symphony protocol for the IRON Box was developed and simulated. However, there still are a few problems to solve, until the protocol implementation can seriously be used in a real-world setting.

The biggest issue is the instability caused by the disappearance of messages and the truncated messages. While no dedicated reason was found for these errors, usage of flow-control mechanisms on the serial line could improve the situation. Currently the IRON Box 2.0 is in development, which will offer support for flow-control mechanisms, among other improvements. A different approach is usage of checksums in combination with resending of truncated messages, which at least would eliminate the influence of bit errors on the serial connection.
A further issue that probably would lead to network instability is the outgoing buffer. According to the simulation, the current buffer size is below the size required for stable operation. A possible solution that does not require any major modification of the design is to conduct a large number of simulations in order to obtain a statistical relevant estimate of the required buffer size and change the size accordingly. If this leads to a buffer size that demands more memory as is available, integration of external working memory is in order.
The buffer is not the only part of the Symphony implementation that suggests use of external RAM. Integration of the Symphony implementation with the IRON software is impossible, using the internal working memory alone, as was discussed in the last section. For this reason, external RAM is essential anyway.

Beside these issues a convenient improvement to the design is possible. Currently a node needs to establish a TCP connection to the designated port, in order to request a new neighbour connection. This approach has several downsides. First of all, two of the available TCP sockets are reserved for this indication process (one outgoing and one incoming socket). Considering that only twelve concurrent connections are possible, this is quite a high sacrificial. Though, that is not the only drawback – as only one socket is available for indication, two nodes that attempt to connect at the same time or try to notify the same node about a change in the network topology are delayed as only ever a single node is able to do so.
If now UDP (User Datagram Protocol) ports instead of a dedicated TCP connection is used for this purpose, these difficulties could be avoided. The Avisaro WLAN Modul 2.0 supports six concurrent UDP connections as well, which could be used to receive packets that notify the node about changes in the network or the intention of a node to join the network. This way, the tedious process that is needed to establish a TCP connection is removed and additionally for this task six UDP ports are available instead of a single TCP connection.

With these adaptions, a well-working P2P network of embedded energy management nodes can be realised. Even if central controlling instances remain in the energy management network, the P2P network can be used to provide autonomous services that remain in working order even if the central entities fail. Altogether, this P2P-based approach is a valuable addition to the currently used system.

Appendix A - Configuration Parameters for the Symphony Implementation and Model

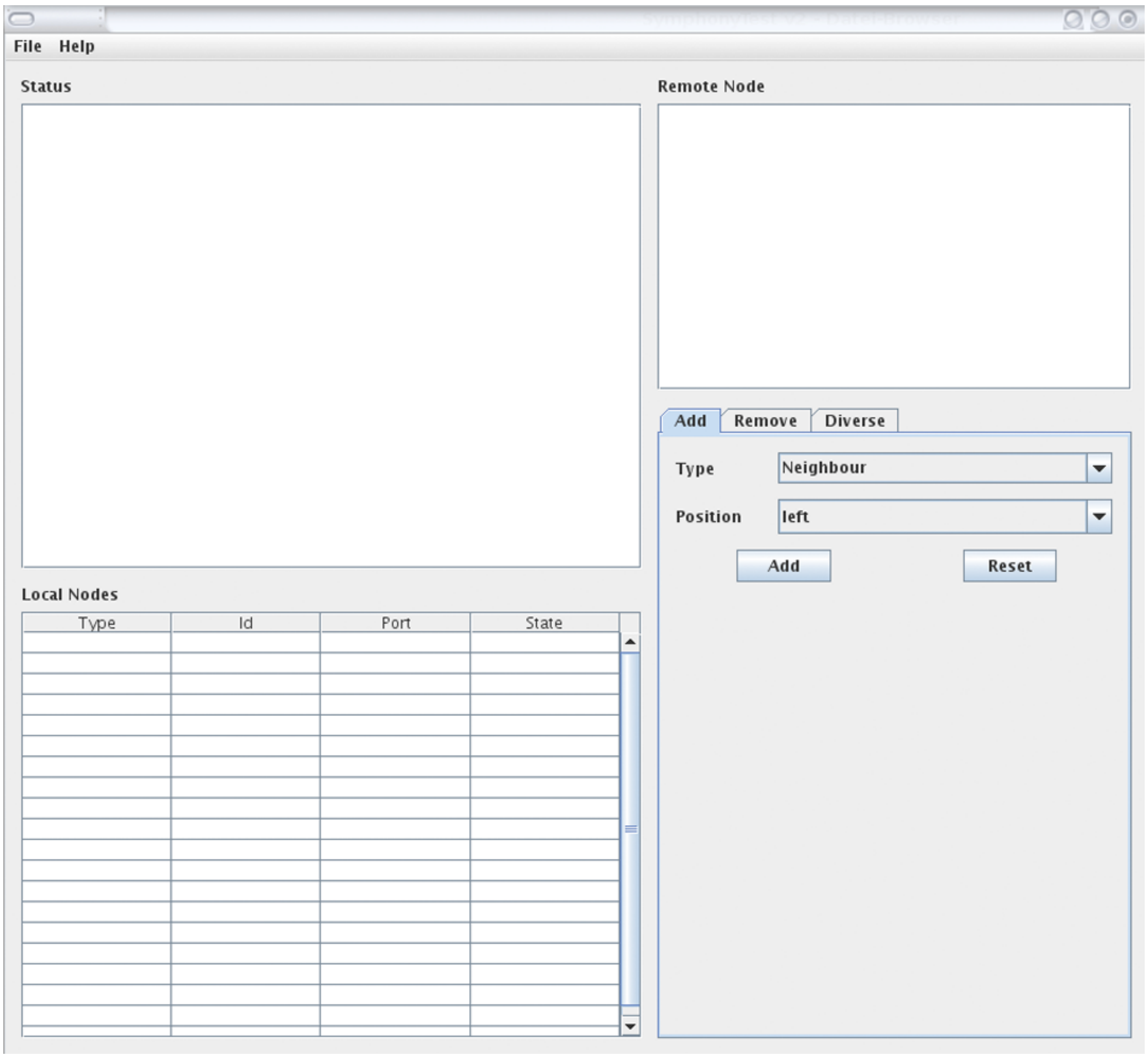
Table 5.1: Listing of the Symphony Implementation Configuration Parameters.

Parameter	**Description**
NUM_LONG_DIS-TANCE	Number of long distance connections that are established per node. Default: 3.
MAX_INC_NUM_LONG _DISTANCE	Maximum number of incoming long distance connections that are accepted. Default: $2 * NUM\_LONG\_DISTANCE - 1$.
MIN_NUM_FOR_LONG _DISTANCE	Minimum number of nodes that are calculated, in order to send long distance connnections. Default: 6.
MAP_SIZE	Number of sockets that are mangaged by the socket map. Default: 4 + NUM_LONG_DISTANCE + MAX_INC_NUM _LONG_DISTANCE.
MAX_MSG_LEN	Maximum length in bytes for a message sent over the Symphony network. Default: 150.
MAX_SYM_MSG_LEN	Maximum length for a Symphony message. Default: MAX_MSG_LEN - 4 * (NUM_LONG_DISTANCE + MAX_INC_NUM _LONG_DISTANCE).
MAX_USER_MSG_LEN	Maximum length in bytes for an application message. Default: MAX_SYM_MSG_LEN - 11.
BUFFER_SIZE	Size of the buffer for outgoing messages (number of messages). Default: 5.
FIRST_HANDLE	Start of the number range, used for handles. Must be between 100 and 199. Default: 150.
BASE_PORT	Start of the port range, Symphony uses. Default: 2000.
MIN_ID	Minimum value for a Symphony id. Default: 0.00001.
SYMPHONY_SEED	Seed for the random-function that is used to calculate Symphony ids. Must be different for each node in order to avoid identical node ids.
LOCAL_IP_X	Local IPv4 address (X is 1,2,3 or 4).
INIT_IP_X	IPv4 address of the Symphony node that is used as entry point (X is 1,2,3 or 4). Default: INIT_IP_1 = 192, INIT_IP_2 = 168, INIT_IP_3 = 1, INIT_IP_4 = 74.
INIT_PORT	"Request port" of the node in the Symphony network that is used as entry point. Default: 2000.
RESEND_TIME	Ammount of time in seconds, the node waits until it resends the JOIN message. Default: 30.

Table 5.2: Listing of the Symphony Simulation Model Configuration Parameters.

Parameter	**Description**
DEBUG	Activates/Deactivates debugging messages.
OUTPUT	Activates/Deactivates output of status information.
LOOKAHEAD	Defines if lookahead information is used for routing. Default: 1.
BASE_PORT	Start of the port range, Symphony uses. Default: 2000.
LONG_DISTANCE _PORT	Starting port for long distance connections. Default: BASE_PORT + 5.
NUM_LONG_DIS-TANCE	Number of long distance connections that are established per node. Default: 3.
MAX_INC_NUM_LONG _DISTANCE	Maximum number of incoming long distance connections that are accepted. Default: $2 * NUM\_LONG\_DISTANCE - 1$.
MIN_NUM_FOR_LONG _DISTANCE	Minimum number of nodes that are calculated, in order to send long distance connnections. Default: 6.
RESEND_TIME	Time that is waited until a *JOIN* message is resent. Default: 30.
MAP_SIZE	Number of sockets that are mangaged by the socket map. Default: 4 + NUM_LONG_DISTANCE + MAX_INC_NUM _LONG_DISTANCE.
XXX_DELAY	Delay time for the XXX message. Default: Table 3.5.

Appendix B - Symphony Test Software



Type: Neighbour

Position: left

Type	Id	Port	State

Figure 5.5: Main window of the test client.

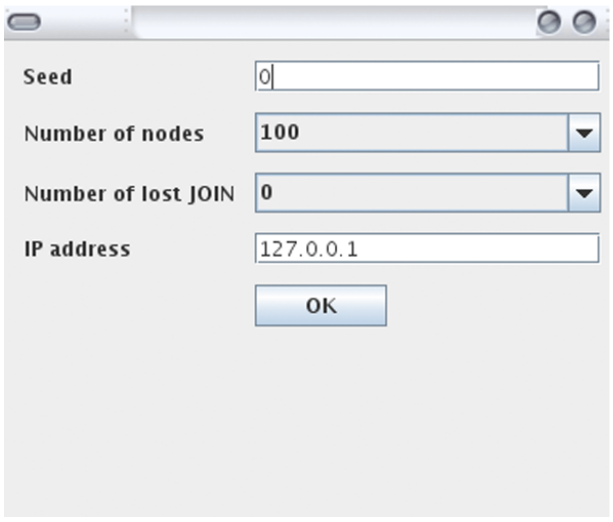

Seed: 0
Number of nodes: 100
Number of lost JOIN: 0
IP address: 127.0.0.1

Figure 5.6: Settings window of the test client.

Literature

[Atm08] Atmel: *Atmel AVR ATmega128/ATmega128L*. 2467Q. 05 08

[ATS04] Androutsellis-Theotokis, S. ; Spinellis, D.: A survey of peer-to-peer content distribution technologies. In: *ACM Comput. Surv.* 36 (2004), Nr. 4, S. 335–371. – ISSN 0360–0300

[ESZK] Eberspcher, J. ; Schollmeier, R. ; Zls, S. ; Kunzmann, G.: Structured P2P Networks in Mobile and fixed Environments. In: *Proc. of the International Working Conference on Performance Modeling and Evaluation of Heterogeneous Networks*

[FI03] Foster, I. ; Iamnitchi, A.: On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In: *Peer-to-Peer Systems II*. Berlin/Heidelberg, Germany : Springer Berlin / Heidelberg, 2003. – ISBN 978–3–540–40724–9, S. 118–128

[FK07] F. Kupzog, P. P.: Wide-Area Control System for Balance-Energy Provision by Energy Consumers. In: *Proceedings of 7th IFAC International Conference on Fieldbuses & Networks in Industrial & Embedded Systems (FeT 2007)*, 2007, S. 337–345

[GBKS08] G.Bengel ; Baun, Ch. ; Kunze, M. ; Stucky, K.-U.: *Masterkurs Parallele und Verteilte Systeme*. 1.0. Wiesbaden, Deutschland : Vieweg+Teubner, 2008. – ISBN 978–3–8348–0394–8

[GRW] Goetz, S. ; Rieche, S. ; Wehrle, K.: Selected DHT Algorithms. In: *Peer-to-Peer Systems and Applications*

[KK03] Kaashoek1, M. F. ; Karger, D. R.: Koorde: A Simple Degree-Optimal Distributed Hash Table. In: *Peer-to-Peer Systems II*. Berlin/Heidelberg, Germany : Springer Berlin / Heidelberg, 2003. – ISBN 978–3–540–40724–9, S. 98–107

[Kle00] Kleinberg, Jon: The small-world phenomenon: an algorithm perspective. In: *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 2000. – ISBN 1–58113–184–4, S. 163–170

[Kup08] Kupzog, F.: Endbericht: Integral Resource Optimization Network Concept. (2008)

[Lar05] Larman, C.: *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. 3. Upper Saddle River, New Jersey, USA : Pearson Education, Inc., 2005. – ISBN 0–13–148906–2

[MBR03] Manku, G. S. ; Bawa, M. ; Raghavan, P.: Symphony: Distributed Hashing In A Small World. In: *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA : USENIX, 2003

[MM02] Maymounkov, P. ; Mazires, D.: Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In: *Peer-to-Peer Systems*, Springer Berlin / Heidelberg, 2002. – ISBN 978–3–540–44179–3, S. 53–65

[MNR02] Malkhi, Dahlia ; Naor, Moni ; Ratajczak, David: Viceroy: a scalable and dynamic emulation of the butterfly. In: *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing.* New York, NY, USA : ACM, 2002. – ISBN 1–58113–485–1, S. 183–192

[OTG02] Oaks, S. ; Traversat, B. ; Gong, L.: *JXTA in a Nutshell.* 1.0. O'Reilly Press, 2002. – ISBN 0–596–00236–X

[RD01] Rowstron, A. ; Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: *Middleware 2001*, Springer Berlin / Heidelberg, 2001. – ISBN 978–3–540–42800–8, S. 329–350

[RFH$^+$01] Ratnasamy, S. ; Francis, P. ; Handley, M. ; Karp, R. ; Schenker, S.: A scalable content-addressable network. In: *SIGCOMM Comput. Commun. Rev.* 31 (2001), Nr. 4, S. 161–172. – ISSN 0146–4833

[SFS05] Schoder, D. ; Fischbach, K. ; Schmitt, Ch.: Core concepts in Peer-to-Peer Networking. (2005)

[Sie79] Siegel, H. J.: Interconnection networks for SIMD machines. In: *Computer* 12 (1979), Nr. 6, S. 57–65

[SM06] Schindelhauer, C. ; Mahlmann, P.: *P2P Netzwerke.* 1.0. Springer Heidelberg / Berlin, 2006

[SMK$^+$01] Stoica, I. ; Morris, R. ; Karger, D. ; Kaashoek, M. F. ; Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications.* New York, NY, USA : ACM, 2001. – ISBN 1–58113–411–8, S. 149–160

[SW02] Sen, S. ; Wang, J.: Analyzing peer-to-peer traffic across large networks. In: *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment.* New York, NY, USA : ACM, 2002. – ISBN 1–58113–603–X, S. 137–150

[TS07] Tanenbaum, A. S. ; Steen, M. V.: *Distributed Systems: Principles and Paradigms.* 2.0. Upper Saddle River, NJ : Pearson Prentice Hall, 2007. – ISBN 0–13–239227–5

Internet References

[1] *Emule*, July 2008. `http://www.emule-project.net`.

[2] *WireShark*, February 2009. `http://www.wireshark.com`.

[3] *Omnet++ Discret Event Simulation System*, February 2009. `http://www.omnetpp.org`.

[4] *JXTA-C*, March 2009. `https://jxta-c.dev.java.net`.

[5] *RevConnect*, March 2009. `http://www.revconnect.com/`.

[6] *Gnutella 0.4*, July 2008. `http://www.stanford.edu/class/cs244b/gnutella_protocol_0.4.pdf`.

[7] *Napster*, July 2008. `http://opennap.sourceforge.net/napster.txt`.

[8] *SETI@home*, September 2008. `http://setiathome.berkeley.edu`.

[9] *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*, July 2008. `http://www.cs.huji.ac.il/labs/danss/p2p/resources/an-infrastructure-for-fault-tolerant-wide-area-location-and-routing.pdf`.

[10] *Avisaro WLAN Modul 2.0*, December 2008. `http://www.avisaro.com/tl/index.php/avisaro-wlan-modul-20.html`.

[11] *Gnutella2 Developer Network*, January 2009. `http://g2.trillinux.org/index.php?title=Main_Page`.

[12] *The BitTorrent Protocol Specification*, January 2009. `http://www.bittorrent.org/beps/bep_0003.html`.

[13] O'Reilly. *What is p2p and what isn't*, July 2008. `http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html`.