Subject **MURRAY**

Creating a P2P chat application

Authors **Eva Katharina Tischner, Sebastian Fontius, Florian Haas, Timm Hofmann**

`kathrin.tischner@googlemail.com, smc@fsfe.org, mail@florianhaas.net, timm@thofmann.info`

Course **TCS04AIM**

Matriculation Numbers **127405, 129045, 186076, 139684**

Supervisor **Jürgen Schultheis**

Project Duration **October 2nd 2006 – June 15th 2007**

This document was created using LaTeX from `murray.tex`.

Copyright © Eva Katharina Tischner, Sebastian Fontius, Florian Haas, Timm Hofmann

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License" (see appendix C on page 99).

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald E. Knuth

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

- dass ich die vorliegende Studienarbeit ohne fremde Hilfe angefertigt habe,
- dass ich die aus fremden Quellen direkt oder indirekt übernommenen Gedanken an den entsprechenden Stellen innerhalb der Arbeit als solche kenntlich gemacht habe,
- dass ich meine Studienarbeit bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht habe.

Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Mannheim, den 15. Juni 2007 ______________________

Eva Katharina Tischner

Mannheim, den 15. Juni 2007 ______________________

Sebastian Fontius

Mannheim, den 15. Juni 2007 ______________________

Florian Haas

Mannheim, den 15. Juni 2007 ______________________

Timm Hofmann

Contents

1. Preface **8**

2. Introduction **9**
2.1. Project description . . . 9
2.2. Motivation . . . 9
2.3. Requirements . . . 10
2.4. Aspects covered . . . 11
2.5. Challenges . . . 13

3. Protocol basics **14**
3.1. Existing Peer-to-Peer approaches . . . 14
3.2. Basic functionality . . . 18
3.3. Basic protocol specifications . . . 19

4. Detailed specification **23**
4.1. Peer-to-Peer layer . . . 23
4.2. XML schema . . . 32
4.3. Crypto layer . . . 34
4.4. Application data . . . 43

5. Implementation **47**
5.1. Development process . . . 47
5.2. Programming language and tools . . . 48
5.3. Network logic . . . 50
5.4. Cryptography . . . 53
5.5. XML schemas . . . 54
5.6. XML Conversion . . . 58
5.7. The XMPP Server . . . 65

6. Testing **71**
6.1. Unit tests . . . 71

7. Additional Aspects **74**
- 7.1. Security . . . 74
- 7.2. Features left out . . . 75

8. Conclusion **77**
- 8.1. The project . . . 77
- 8.2. The programming language . . . 77
- 8.3. The result . . . 77

9. Bibliography **79**

10. Abbreviations **81**

A. XML schemas **83**
- A.1. Client . . . 83
- A.2. Cache request . . . 84
- A.3. Cache response . . . 84
- A.4. Echo request . . . 85
- A.5. Echo response . . . 85
- A.6. Hello . . . 86
- A.7. Public key request . . . 87
- A.8. Public key response . . . 87
- A.9. Search request . . . 88
- A.10. Search response . . . 89
- A.11. User data . . . 90

B. XML examples **91**
- B.1. Cache request . . . 91
- B.2. Cache response . . . 92
- B.3. Echo request . . . 93
- B.4. Echo response . . . 93
- B.5. Hello . . . 94
- B.6. Public key request . . . 94
- B.7. Public key response . . . 95
- B.8. Search request . . . 96
- B.9. Search response . . . 97
- B.10. User data . . . 98

C. GNU Free Documentation License **99**

List of Figures

3.1. MURRAY Layers 20
3.2. Basic Architecture 21

4.1. MURRAY's Communication Model 25
4.2. Hybrid encryption scheme 37
4.3. A man in the middle attack 42

5.1. Architecture of the Jabber/MURRAY translation 66
5.2. The internal architecture of the MURRAY XMPP Server 67

1. Preface

Information and communication are the major resources of the 21st century. The internet has created totally new possibilities for sharing information and communication with people all around the world. Furthermore, the ability to use this technology to interact with others will be of central importance in years to come.

While there are numerous ways of online communication - eMail, internet relay chat, instant messaging services, newsgroups to name a few - all existing solutions (known to the authors) share one basic principle: They are based on a client-server architecture.

While this is generally an efficient and mature approach, it also incorporates some problems considering reliability and privacy. The topic of this student research project is to address the shortcomings of the client/server approach and develop an alternative system based on the "Peer-to-Peer"(P2P) concept.

The aim of the project called MURRAY is to define a protocol and develop a client implementation for a distributed communication system offering the same basic functionality as existing chat- and Instant Messaging (IM) systems. It should be usable both in Local Area Networks(LANs) and the Internet and incorporate the advantages of the serverless approach, like the capability of ad-hoc communication, higher robustness and the distribution of required resources among all users.

2. Introduction

2.1. Project description

Aim of the project is to define and implement a method for text-message communication over the internet and local networks. In contrast to the infinite number of existing solutions, this one should work completely on a Peer-to-Peer basis – without dedicated servers of any kind.

The resulting protocol and implementation shall be open, extendable and offer the necessary functions to enable users to communicate with others.

2.2. Motivation

The existing methods of communication over the internet all suffer from their drawbacks. The most common protocols for instant messaging (ICQ, AIM, MSN) are commercial, centralised and proprietary. As a result they are always endangered of becoming unavailable to the non-paying users or being turned off completely.
Email over the Simple Mail Transfer Protocol (SMTP) on the other hand is open and pretty decentralised, but does not offer the functionality to manage known contacts, find out if they are online or chat with them in real time.

While open instant-messaging protocols like Jabber alleviate these problems to some extent, they still rely on a server infrastructure with according resources.

MURRAY is designed to completely overcome these problems by replacing the client-server-approach. Peer-to-Peer networks are generally more reliable, their performance usually scales very well with increasing number of users and they cannot be controlled, censored or just turned off.
So, it is a viable goal to adapt the Peer-to-Peer model for instant messaging.

2.3. Requirements

The requirements for the new protocol and implementation are to offer the same basic functionality as existing instant-messaging systems, but without any server infrastructure managing the network. Additionally, some extra features have been defined specifically for this protocol.

2.3.1. Requirements for the protocol

- Ability to transmit text messages in near-realtime
- Creation of “chat rooms” with two or more users
- Entering and leaving existing chat rooms
- Definition of access rules for chat rooms (open, private, invitation)
- Network-wide search for users, chat rooms and user details
- Creation of a unique user ID and user details
- Mechanism for transmitting the user’s online status
- Encryption of all user data
- Complete serverless, pure P2P network
- Low network load
- High performance

2.3.2. Requirements for the implementation

- Support of main protocol features
- Graphic user interface

2.4. Aspects covered

2.4.1. Peer-to-Peer

The classic approach for internet communication is the client-server model. One central server with impressive system resources (computing power and connection bandwidth) keeps all data and offers all functions needed for a specific task. This functionality is used by a great number of - compared to the server - “weak” clients. But as developments in the last 15 years have shown, this approach does not always fit the requirements of modern applications. Especially when it comes to situations where much data has to be distributed to many clients (for example in file distribution) or response times are important (real-time voice communication, VoIP) the client-server-model reaches its limits. In these cases the resources required on the server-side are extreme. On the other end of the line the capabilities of the clients have grown rapidly. Taking these developments into account, a number of technologies have emerged since the mid-1990s which reduce the role of the server or eliminate it completely - these communication models are commonly summarised under the term “Peer-to-Peer”.

Taking the server out of the equation of internet communication has become a more widespread approach in recent years. In many areas, the classic client-server design has been replaced by more flexible concepts using direct communication between the clients. But still, in most cases servers are responsible for keeping track of clients and data. Here the servers are freed from the bulk of network traffic, but they still need to be fast and reliable, as the whole network depends on them.
Good examples for this approach are Voice over IP using the Session Initiation Protocol (SIP) and the bittorrent protocol used for file-distribution.

There are also examples for “pure” Peer-to-Peer systems which can exist completely without servers. Although these networks are mainly designed for file distribution, their basic concepts can also be used for text-based communication.
The aim of this project is to develop a system based on this principle without a central management instance while still offering the functionality known from existing instant messaging systems.

2.4.2. Cryptography

The use of cryptographic methods was one of the central requirements for the system from the beginning. Every bit of user data sent using the network is - by default -

encrypted. Not only does cryptography increase privacy and data protection, it also offers possibilities for user identification and transmission control.

Asymmetric encryption is a basic component of the protocol implementation. It is integrated into the architecture.

2.4.3. XML

The Extensible Markup Language (XML) offers a concise, straightforward way to structure and formalise data. Many tools are available for creating and processing XML in various platforms, making it an ideal data exchange format.

2.4.4. Network architecture

A Peer-to-Peer network has to fulfil many functions which are usually of no concern to applications using a client-server system. Things like address resolution, synchronisation of data on each peer, information on changes etc. have to be done by all peers in a way that all necessary information is available.

On the other hand, it would be impractical to store all information at every node, meaning the network has to work in a way that each node has to know a bit about the network.

2.4.5. Protocol definition

The data packages exchanged between the peers, their reactions and handling of these data packages define the behaviour of the system. Accordingly, their specification has to reflect the requirements specified for the protocol.
It has to be defined which data format should be used, which packages types are part of the system, and which type of data is sent when and how.

2.4.6. Implementation

A prototype for a reference implementation of the protocol is also part of this thesis. Since there are no servers managing the network, all the logic needed for this task has to be done by the "client" programs. The implementation should be platform-independent, demonstrate all central functions of the protocol and be well documented as it may serve as a platform for others to create their own implementations.

2.4.7. Testing

The protocol as well as its implementation have to be tested. The protocol has to work in all kinds of TCP/IP-based networks, but especially in the internet. The prototype should be free of critical bugs and work reliably.

2.5. Challenges

The Peer-to-Peer approach to communication offers many advantages, but also has some disadvantages making it difficult to implement IM using this method.
Compared to a client-server system, considerably more network overhead is created because of the management information which has to be distributed among all peers.
Additionally, information on changes like a client changing its IP-Address take time to reach all peers which need it.

These two inherent disadvantages of the Peer-to-Peer approach create the most severe challenges in defining a Peer-to-Peer protocol. On the one hand, changes should be communicated very fast so communication can resume as good as possible. On the other hand this can only be achieved by sending more management information - creating more overhead.
Finding a practical compromise for this problem is essential for the usability of the protocol.
The system has to be designed in a way that requested information can be found by a node as quickly as possible, but also the effort of synchronisation between the peers has to be as small as possible so the overhead created by management information won't get too big.

An additional problem which specifically arises in the chat-environment is identification and secure authentication without a central controlling instance. The central question is how to create a unique identification for each peer without an instance keeping track of which peers already exist.

Another difficulty is created by the mandatory usage of encryption. Cryptography is a very useful, necessary feature, but it creates considerable strain on system resources - especially asymmetric encryption. So again, usable methods for achieving privacy and security have to be found while keeping the system load reasonable.

3. Protocol basics

In this chapter, the definition of the MURRAY-protocol is outlined, as well as the process and decisions leading to it.

Generally, the specification of the protocol followed the premise of keeping it as simple as possible in order to make the protocol itself as well as later implementation lightweight and avoid the introduction of unnecessary features and possible error sources.

The process of defining the protocol was divided into four steps:

1. Analysis and comparison of existing P2P protocols
2. Definition of necessary functionality
3. Specification of technologies and basic principles to be used
4. Detailed specification

Analysing other P2P approaches before the actual definition of requirements was done made it possible to define the actual features of MURRAY according to what existing protocols do in order to achieve similar goals.

Steps one through three are laid out in this chapter, while the detailed protocol specifications will be described in the following sections.

3.1. Existing Peer-to-Peer approaches

The first step in the process of defining the network architecture of the protocol was an analysis of existing P2P approaches and how.

Generally, all P2P systems suffer from the same advantages and disadvantages.

3.1.1. Advantages

- No expensive server infrastructure required
- No “single point of failure”
- It is nearly impossible to monitor or manipulate the network from the outside
- The user has full control over the data and communication to and from his computer
- The network load is (theoretically) distributed equally

3.1.2. Disadvantages

- Overhead is created because management information has to be exchanged additionally to the data
- Information about changes take time to reach every peer

3.1.3. Analysis

Peer-to-Peer protocols can generally be divided into three basic groups:

- With central managing instance
- Completely without central instance
- Anywhere between

3.1.3.1. Peer-to-Peer protocols with central managing instance

During the last years many applications have emerged in the internet utilising the approach of “centrally controlled Peer-to-Peer” data transmission. Many providers of communication services which are usually used one-to-one have begun to reduce the “server” function to keep track of all the users while the real data is transmitted directly from one “client” to another. The most prominent example for this approach is the Session Initiation Protocol (SIP) used in many Voice over IP solutions.
With SIP, the “VoIP-Provider” only holds a central register of all subscribers which is responsible for call initiation, while the real Voice Communication is transmitted directly

between both clients. Other examples for this communication model is the file-transfer feature in commercial instant messaging services or the early file sharing programs (e.g. Napster).

This approach is usually utilised where a central provider wants to keep control of "its" service but does not have the resources to route all the traffic over one (or several) servers. Additionally, technical implementation of this approach is relatively simple.

3.1.3.2. Peer-to-Peer protocols without central instance

"Pure" Peer-to-Peer communication without any servers managing the network usually work by distributing the necessary management information among all peers, using different methods. In most cases, a "distributed hash table" is used to locate peers and stored files in the network [Maymounkov and Mazières, 2001].
Although representatives of this network model are (until now) relatively seldom, the approach is - nowadays - technically mature and enjoys growing popularity.

Systems using this approach (e.g. Kademlia, Fasttrack) are designed and almost entirely used for file sharing. Accordingly, the protocols and methods used in the network are designed in a way that files distributed through the network can be easily found and re-assembled from various sources.

3.1.3.3. Combinations of both approaches

Most of the more broadly used P2P systems work with a model somewhere in between, trying to combine the advantages of both models by either decentralising the servers (e.g. eDonkey, bittorrent) or "promoting" certain, reliable peers to fulfil management functions (e.g. gnutella).

3.1.4. Comparison

A Peer-to-Peer protocol for the primary purpose of instant text messaging between peers does - according to the author's best knowledge - not exist until now.

When comparing existing P2P systems, the advantages and shortcomings of each of the three approaches mentioned above became evident.
While the approach of using a centralised server infrastructure has its place with organisations who can afford it, this model is clearly not viable for a network aimed at

users with - typically - standard PCs and dial-up internet connections. Although the resources required for the servers are a lot lower than with a pure client/server approach, it still has to be reliable and capable of handling all the management information in the network. Another major issue is also not resolved - there is still a single point of failure. Without the central server(s), the whole network is crippled.

On the other hand, distributing the management information among all the peers considerably increases the complexity of the network and the resources each client has to invest. The data which would otherwise be handled by a server infrastructure has to be distributed among the peers, dramatically increasing the amount of logic required. Additionally, the peers in such a network are usually unreliable as the "normal" user only has a dial-up connection at his disposal, thus making the management of the network more complex as the "normal" peer will only be online for a few hours a day with frequently changing IP-addresses. Since this protocol will have the same problems, the approaches used to overcome them were of great interest during the specification of the protocol.

3.1.5. Implications

Generally, contemporary systems are very elaborate and fulfil their respective tasks very well, proving that P2P communication has become a viable alternative to the client/server approach.

As explained in 3.1.3 on page 15, several definitions of "Peer-to-Peer" exist. From the beginning, it was the goal of the project to utilise the "pure" Peer-to-Peer approach - completely without servers of any kind. Accordingly, other protocols using the same model were of great interest.

Unfortunately, most "real" P2P systems are designed primarily for file sharing, which of course has totally different requirements than P2P instant messaging, so not all of the methods used in existing systems could be used in the design. Some of the problems encountered in P2P file distribution were of no concern during the specification of the P2P chat protocol (e.g. how to retrieve a file from several peers and securely reassemble it) while some of the challenges of P2P chat - such as secure user identification - are of very little interest in a file-distribution environment.

But many of the lessons learnt from existing systems were vital in getting a better understanding of how such a system has to work. Methods and ideas used in other systems form a considerable basis for the specification of the protocol. Since the basic issues are the same in all P2P networks, important ideas on how to handle these could

be found by analysing existing solutions.
They provided crucial information and ideas on several important issues:

- How to distribute information on the network among the peers
- How to perform a search
- The “bootstrapping problem” (how can a new peer enter the network?)
- The general makeup of such a system

3.2. Basic functionality

The basic layout of the network architecture depends on the requirements specified in 2.3.1 on page 10. According to these specifications and the information gathered by analysing other P2P systems, the central functionality was predetermined to a great extend.

Other basic decisions about the structure of the protocol where later made based on general considerations (such as portability) and revision of existing and commonly used technologies.

Based on the analysis of existing P2P systems and the ideas about which features the platform should offer, the core functionality was specified.

3.2.1. TCP/IP

As the system will exclusively be used in the internet and other IP-based networks, the protocol will be based on TCP/IP. Usage of UDP instead of TCP will be possible with the protocol design, but a UDP-model is not part of the reference implementation.

3.2.2. Distributed database

The P2P system will have to take over the functions of a central user register, keeping track of which users are online, which user can be reached under which IP-address etc.

Since the main point of the system is text-based communication between peers, only information on the peers has to be available (in contrast to e.g. file-sharing systems, where the main focus is on files and not users).

3.2.3. “Pure” Peer-To-Peer

Since the system should not be attackable through shutting down “some central servers”, and also offer the possibility to work ad-hoc in any IP-based network, a “pure” (meaning completely serverless) P2P approach is to be utilised.

3.2.4. User identification

Secure identification of users is a central part of the services the network needs to offer. As there is no server infrastructure taking care of user authentication, the generation of unique user identifiers as well as authentication of the users among each other has to be tackled.

3.2.5. Privacy and security

The communication should be conducted securely. Although Peer-to-Peer communication is generally harder to track or record than communication with central servers at the key points in the network, it is still possible.
Additionally, secure communication also means that messages cannot be tampered with (adding or leaving out information). Privacy and security of the communication are central goals of the protocol, which is why cryptography has to play an important role.

3.3. Basic protocol specifications

Because the protocol has to provide several very different functions, it became evident that separating these functions into relatively independent components was necessary, resulting in a layered architecture.

1. The *P2P-Layer* is responsible for keeping track of the peers and acts as the mentioned “distributed database”. For this part of the protocol, the actual packages sent back and forth as well as the commodities of transmission are defined.
2. The *Crypto layer* takes care of encryption and user identification.
3. *Application Data* is the actual data created by the application. The protocol is not to be limited to instant messaging, but the architecture is optimised for this type of data.

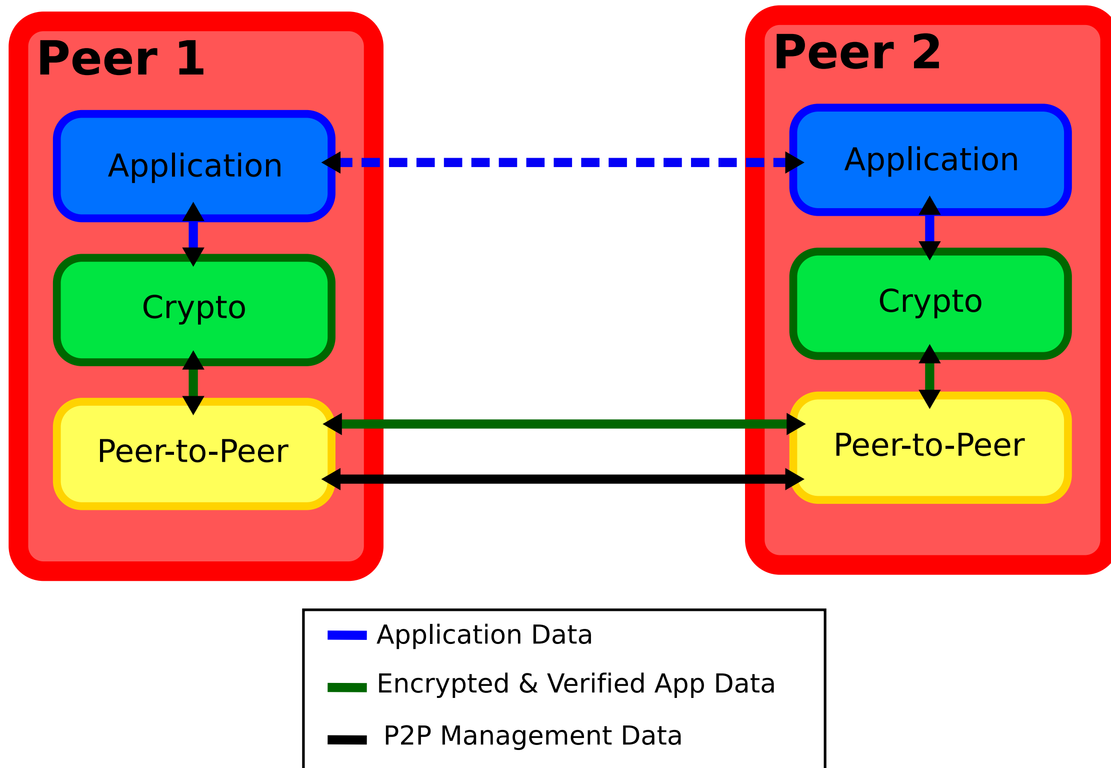

Figure 3.1.: MURRAY Layers

Having defined the three “layers” and their respective functionality, the technologies to be used in each of them were defined. This was done by comparing several available technologies and checking them against the requirements.

3.3.1. XML

For the usage of the MURRAY protocol considerable amounts of data have to be transmitted between the different clients. XML is used to define the format in which the data is transmitted. XML is a subset of SGML, the Standard Generalized Markup Language. XML “describes a class of data objects called XML documents”[1]. “An XML document may consist of one or many storage units. These are called entities; they all have content and are all [...] identified by entity name”[2]. The content of these entities are the data that needs to be transmitted according to the MURRAY protocol.

The advantages of XML are that is formal, concise, easily usable over the internet and supporting a wide variety of applications. Other advantages are that XML documents

[1] [Bray et al., 2006a]
[2] [Bray et al., 2006c]

are easy to create, human-readable and reasonably clear. It is also easy to write programs which process XML documents.

3.3.2. Public key cryptography

In order to ensure privacy, all messages exchanged between chat-partners have to be encrypted. As no secure means of communication exists between the partners, this means that Public-Key cryptography has to be used.

Additionally, the Public-Key acts as a decentralised identifying mechanism: as only the original creator of the key can successfully sign messages, a signed message is proof that the sender is the original owner of the key.

3.3.3. Jabber/XMPP

Jabber or XMPP is a decentralised, open, XML based and extensible instant messaging protocol. It has widespread client support and multiple server implementations exist. See section 4.4.1 on page 43 for details and section 4.4.2 on page 46 why Jabber/XMPP is used.

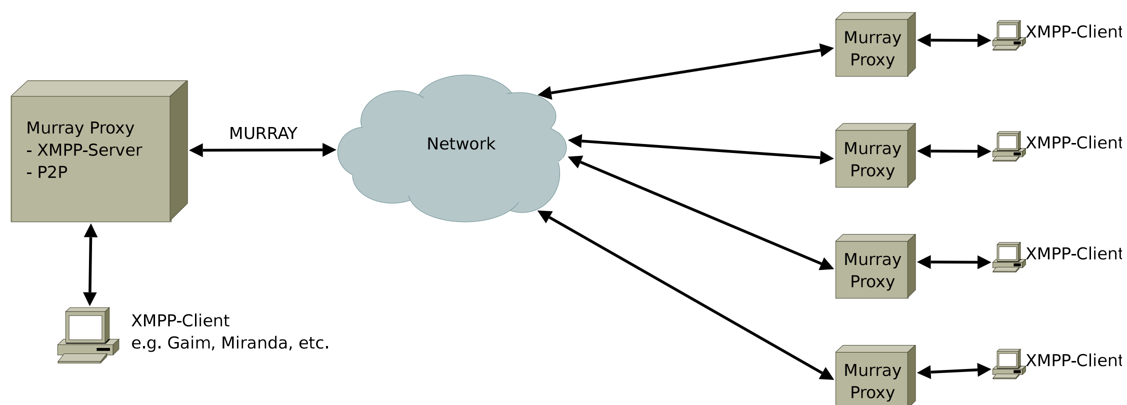

Figure 3.2.: Basic Architecture

3.3.4. Alternatives

Possible alternatives to these technologies were considered during the early stages of specification, but later abandoned in favour of the aforementioned methods.

XML Binary Faster than XML and creating less Overhead[3]. Turned out to be unusable for the protocol because there is no finished specification yet.

Symmetric Encryption *Much* faster than PK-encryption, but with severe practical problems since keys would have to be exchanged securely

Implement an own client Time-consuming and inflexible

Usage of a distributed server infrastructure and/or supernodes Very complex, requires many peers to work properly

Usage of UDP instead of TCP UDP creates less network load and is slightly faster than TCP, but creates problems considering flow control and reliability (lost packages etc.).

[3] http://www.w3.org/TR/xbc-characterization

4. Detailed specification

As explained in 3.3 on page 19, the protocol consists of three components: the P2P layer, the Crypto layer and the User Data layer.

4.1. Peer-to-Peer layer

The P2P layer acts as a distributed database and is responsible for the communication between the peers. In this part of the protocol specification the network architecture, the packages transmitted between the peers and the treatment of these packages are defined.

4.1.1. General considerations

Since the designated usage of the P2P-component is to transmit IM data between peers, the purpose of this part of the protocol is to offer all necessary information required so peers can exchange messages directly.
Because the system will be exclusively used in IP-based networks, there are three attributes required to send a message directly to another peer:

- The other peer's IP-address
- The TCP-port on which his instance of the MURRAY-software listens for incoming connections
- The IP-protocol version used

Out of these three, IP-address and TCP-port are essential. The version of the used IP-protocol is included primarily in respect to possible future developments and can – at the current point in time – safely be assumed to be IP version 4.

With these three properties available, communication between peers can take place. Making this information available to all the peers is the purpose of the P2P protocol.

4.1.2. Distribution of peer information

The real challenge in IM communication over a P2P network is figuring out under which IP-address and Port the communication partner can be reached.
In a classic client/server infrastructure or a centrally managed P2P as described in section 3.1.3.1 on page 15, a server would keep a central register of all connected computers.

But in the serverless P2P-world, this information needs to be distributed among all peers. So the system has to be designed in a way that each peer has to keep information on the addresses of the other peers. And since static IP-addresses are a rare commodity for “normal” internet users in a world of dial-up and xDSL connections, the protocol has to be designed in a fashion reflecting that the “contact information” for peers are changing in relatively short intervals.

In this situation, it is unrealistic for a peer to know *all* the other peers actual address. Accordingly, the system has to work in way that it only requires each single peer to have current information on *some* other peers.
This cache of known participants in the network is called *known peers cache* and is the central database for managing the MURRAY network.

Each peer has one known peers cache and keeps it up-to-date by monitoring communication with other peers and also by exchanging information out of the cache with other known peers.

4.1.3. Identification

Within the decentralised P2P network, a mean for safely identifying the peers is necessary. In the MURRAY network, this is done by each peer assigning himself a “unique identifier” (UID). How the UID is defined and created is described in section 4.3.3.2 on page 40.
Together with the properties described in section 4.1.1 on the preceding page, a peer in the MURRAY network has four attributes:

- UID
- Current IP-address
- TCP-Port
- IP-Version

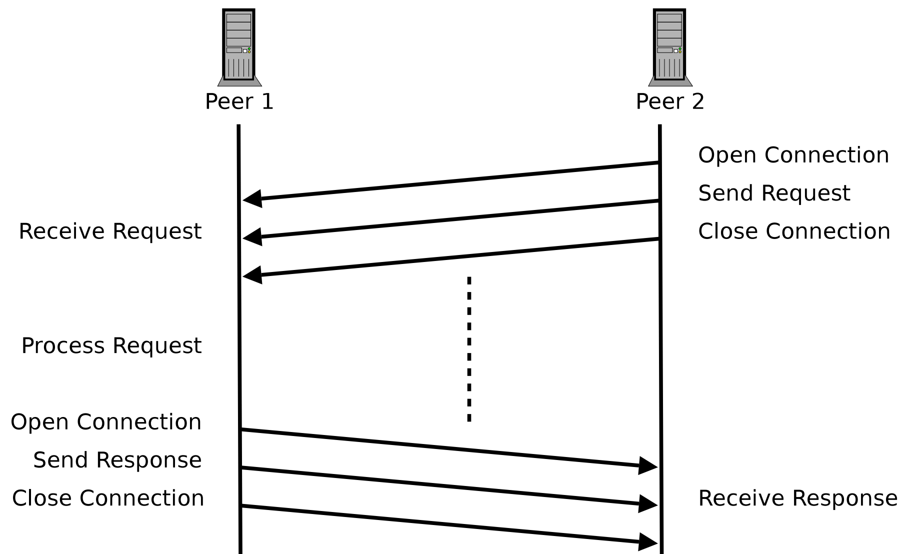

Figure 4.1.: MURRAY's Communication Model

In order to keep the data distributed among the peers up-to-date, this information (both for sender and receiver) is sent with all messages traversing the network.

4.1.4. Communication scheme

The communication in the MURRAY network follows an asynchronous approach and always takes place on a one to one basis. It is based on a simple request → response scheme, with the notable exception of application data which is sent without any response being necessary.

The exchange of management information in the network is conducted asynchronously, meaning that in contrast to other protocols, e.g. HTTP [Fielding et al., 1999] or SMTP [Klensin, 2001], requests and responses are sent over different channels without the need for prompt response.

A normal exchange of information between two peers would work like this:

The peer requesting information (peer 1) opens a direct connection to the receiver (peer 2) and sends its request. After completing the transmission, the connection is closed and normal operation continues. The receiver receives and processed the request. Once the required information is gathered or computed, peer 2 formulates its response and opens a new connection to peer 1. The response is sent over this new channel, which is closed afterwards. Now peer 1 can process the response.

This model is used in order to reflect the P2P approach (all kinds of message may arrive at any time - requested or not) as well as possible problems concerning bandwidth or system resources. A synchronous approach would have required that requests would be processed in relatively short time which cannot be guaranteed in a decentralised and highly heterogeneous network.

4.1.5. P2P network functions

The network has to provide several functions to enable the peers to communicate with each other. The definition of which functions the P2P-layer has to offer is the basis for the individual data packages defined later on. The difficulty in defining these functions is to cover all the necessary functions without "over-engineering" the network, i.e. keeping the protocol as simple as possible, but not simpler.

4.1.5.1. Exchange of known peers data

In order to keep the known peers cache of all the peers up-to-date, it is necessary to exchange known peers data from time to time. This is a very basic function, found in all P2P networks.

4.1.5.2. Search for peers

Although in case of an IM environment, the known peers data will most likely be up-to-date for the people the user is currently communicating with, there is also an expectable high number of searches within the network. These searches could both be system-generated searches for unknown UIDs and user-generated searches for other users or chatrooms.

4.1.5.3. Bootstrapping

The network has to offer a possibility for a "new" peer to enter the network, a process called "bootstrapping". Once a peer enters the network, there has to be a function for it to both "register" with other peers and obtain initial information on the network.
The process of bootstrapping also has to take into account that upon first connecting to the net, a peer might not know his public IP address because of widely-used techniques like Network Address Translation (NAT) or Port Address Translation (PAT). In these cases, a method for communicating the peer his own IP-address has to be integrated.

4.1.5.4. Echo

While not strictly necessary for the network to function, a method for simply checking if a peer is still reachable under the available information is generally a good idea.

4.1.5.5. Public key exchange

Since Public Key cryptography is used for encryption and authentication of application data (see 4.3.2.2 on page 35), it is necessary to have the public key of the communication partner in order to encrypt the packages and verify the signature. So a way for exchanging the public keys has to be specified as a service function for the crypto-layer.

4.1.5.6. Application data

Because of the layered architecture, it is possible to transmit virtually all kinds of data over the MURRAY network, so there has to be a package format defined for the encrypted data generated by the application and crypto layers.

4.1.6. Definition of data packages

The functions defined in 4.1.5 on the preceding page have to be represented in the network by defined packages, which also have to represent the asynchronous communication model described in 4.1.4 on page 25, meaning for each function there has to be a "request" and a "response" package with the exception of the application data. These packages were defined with all the necessary and optional attributes that can be sent within them.

4.1.6.1. Attributes common to all packages

Certain information needs to be sent with all data packages defined in the MURRAY network. This is basically the information on the peers involved, as described in 4.1.3 on page 24. These are included for both the sender (`source`) and the receiver (`destination`) in order to check both if the package reached the right peer and the information stored in the own known peers cache is still up-to-date.

Additionally, for each request/response pair some kind of identification has to be sent, in order for the peers to keep track of requests and responses. This data field contains

an – ideally unique – number assigning a response to a certain request and is called `request-ID`. It is included in all packages, again with the exception of the packages containing user data.

Obviously, also the type of the package has to be included in the transmission.

Summed up, all packages share the following attributes:

- Package type
- Destination
  - UID
  - IP-version
  - IP-address
  - TCP Port
- Source
  - UID
  - IP-version
  - IP-address
  - TCP Port
- Request-IDs (not in “User Data” package)

4.1.6.2. Known peer data exchange

A pair of packages is used to synchronise known peer caches between peers. In certain intervals, a peer should contact the peers in his cache and request them to transmit an excerpt from their caches.

Cache request In order to request an update from another peers cache, he sends a `cache request`. Additionally to the common data fields, the cache-request has only one attribute: The maximum number of cache entries he wants the other peer to send in the response. This field is called `count`.

Cache response Upon receiving a cache request, a peer accesses his cache and answers with a cache response, containing entries from his cache in a list of `nodes` with entries for UID, IP version, IP address and TCP port for each peer. The list may have any length, but it has to have at least one entry and at maximal the number the requester specified in the `count` value.

4.1.6.3. Search

In case a peer searches for another peer, he does so by sending a search request to other peers. In case the receiver of a search request cannot respond to it, it has to be forwarded to other peers. In case the receiver does know results for the search, he still might forward the request additional to sending back a response.

Time to live A search request has to be forwarded to other peers by the receiver in case he cannot answer the request himself. Since a search request is – depending on the settings in the implementation – sent to several peers, they should not be allowed to swirl around the network indefinitely. Therefore, a hop count called `TTL (Time To Live)` is defined.
Each peer forwarding a search request must reduce the TTL by one and must not forward any search requests with a TTL of 0.
Also, it is a good idea to check if a search request was already processed and was just relayed back by another peer. In this case, the package should be discarded.

The TTL can be set freely by the user or software used, but it is suggested to keep it a low number by default. As a search request is send to a certain number of peers which again forward it to several peers, the number of packages sent can accumulate real fast. For example, if each peer sends a search request to 100 others with a TTL of 4, this would already result in $100^4 = 100,000,000$ packages. In order to – at least at some point – restrict the number of packages going around, a peer has to discard searches with a TTL of more than 10.

Search request The search request contains the common data fields, the TTL, an arbitrary long list of search fields in `fieldname : searched value` format and optionally an operation (`AND` or `OR`) to connect the search fields logically.
There is also an optional list `relays`, to which a relaying peer may add himself if he wants to receive the response as well.

Search response If a receiver has information on peers fitting to the search, he answers in a search response package which is sent to the original sender and optionally to the relays. The search response contains the original search request and a list of `found nodes`.

4.1.6.4. Bootstrapping

When a peer enters the network, he "registers" with a number of active peers using a `Hello` package. This is the only package where the sender's IP address and version may be omitted (in case the sender does not know his public IP). In other respects the `Hello`-package is built up and treated as a cache request.

4.1.6.5. Echo

Echo request and response are used in order to check if a known peer is still reachable with the data available. These are the most simple packages, only containing the destination, source and request ID.

4.1.6.6. Public key exchange

Public keys of peers are exchanged via an own pair of packages: `pk-request` and `pk-response`.

PK request The public key request contains only the common data fields.

PK response The public key response package contains the common data fields and the signed public key of the sender.

4.1.6.7. Application data

Application data sent over the MURRAY network is encapsulated in a `user data` package. In contrast to all the other packages, there is no response sent for user data. Accordingly, a request ID is not included in the user data package. So the user data package contains only information on destination and source together with the encrypted application data.

4.1.7. Preconditions & obstacles

Although MURRAY is designed to be as universally usable as possible, certain preconditions have to be met for people in order to use the protocol and some obstacles which need to be circumvented either by means of the protocol or the users themselves.

4.1.7.1. TCP/IP

MURRAY is designed for the internet and other IP-based networks and will not work on other network protocol stacks without great changes to the protocol. On the other hand, MURRAY does not use any internet resources aside from the IP addressing scheme, so it is also fully usable in IP-based Local Area Networks (LANs).

4.1.7.2. Firewalls

As each peer has to listen for incoming connections on an open TCP port, eventually existing Firewalls have to be configured accordingly. This of course creates some security problems, described in more detail in 7.1 on page 74. But there is no other technical way, as a P2P network requires each peer to listen for unmotivated connections from outside. Unfortunately, most Firewalls are designed for client/server traffic and to make P2P systems like MURRAY work is only possible by using settings which can also open up security issues.

4.1.7.3. NAT/PAT

NAT and PAT can seriously impact the network as it effectively disturbs the peer's ability to find out about their own public IP address. This problem is circumvented by sending a `Hello` package with an empty ip address and version, which can then be extracted by the receiving side and will be integrated in the response.
Additionally, NAT and PAT usually are operated in combination with a Firewall, which again creates problems with port forwarding etc.

4.1.7.4. Getting into the network

A severe problem in serverless P2P is to get into the network in the first place. First entering the network with only resources of MURRAY itself is, like in other P2P systems,

not possible. In order to first obtain information on the network one has to contact at least a peer who is already taking part.
Unfortunately, retrieving the information to do this cannot be achieved without "help" from the outside world. As all peers are capable to export and share their known peers cache also through other channels, a reasonable possibility would be to publish lists of especially reliable peers over the web.

Once a peer successfully entered the network, it will be possible to reconnect without major problems (except after a really long time), as the known peers cache can be stored persistently and re-used later with a high probability that at least some of the peers stored are still online.

Of course one might argue that this approach completely negates the claim of creating a completely serverless and de-centralised system, but unfortunately there is no other practical way. The only other possibility would be to randomly scan parts of the internet for other peers, an approach which is definitely not acceptable for users, ISPs and developers.
Unfortunately as it is, more decentralisation than in MURRAY is simply not possible with the technologies used by the internet. In LANs the situation is different, and different possibilities for entering an existing MURRAY-Network in a LAN are thinkable, but neither part of the protocol specification nor of the reference implementation.

4.2. XML schema

For instant messaging much communication between the different clients is necessary. Information about the clients has to be exchanged as well as the messages themselves.

In the MURRAY protocol all this data is sent from one client to another as XML. Via the use of XML it is possible to define as well document structures as data structures which are ideally suited for the respective use[1]. Another advantage is that XML can be easily parsed. A disadvantage of XML is that XML files are larger than files with pure data. This disadvantage has to be accepted, because in return the maintainability is much better. The reason for this is that XML files are not only machine-readable, but also human-readable.

The structure of the XML messages used in the MURRAY protocol is defined by the usage of XML schemas *[sic]*[2]. "The purpose of a schema is to define a class of XML

[1] [Mintert, 2002], page 32
[2] In this document the incorrect notation "schemas" is used, because it is used by the World Wide Web Consortium (W3C).

documents, and so the term 'instance document' is often used to describe an XML document that conforms to a particular schema."[3]

The usage of XML schemas has been preferred to the usage of Document Type Definition (DTD) which can be used for the same purposes as XML schemas, because in DTDs it is not possible to define data types[4]. This is a feature that is needed for the MURRAY protocol.

The XML schemas that have been defined for the MURRAY protocol are described in section 5.5 on page 54 and can be found in appendix A on page 83. In appendix B on page 91 there are examples for XML according to these XML schemas.

[3] [Fallside and Walmsley, 2004]
[4] [Mintert, 2002], page 32

4.3. Crypto layer

You don't have to distrust the government to want to use cryptography.

Phillip Zimmerman, creator of PGP

4.3.1. Introduction

One central issue in modern communication is the transmission of sensitive information over insecure communication-paths. This also holds for the scope of this project.

The very nature of a chat implies a certain privacy between the chat-participants, as a chat is a conversation between a limited number of conversation partners. To ensure privacy between the partners, a way to transform the original message into a form an attacker cannot understand but is easily readable by the recipient has to be used. This transformation is called encryption and is the main responsibility of the crypto-layer.

Another aspect of privacy is the question of identity: How can the sender be sure that the person receiving the message is really the intended recipient? How can the recipient be sure that the message he received really originates from the sender? The crypto-layer also ensures that this aspect of privacy is taken care of.

4.3.2. Encryption

4.3.2.1. What is encryption?

Encryption is a way to transform information (referred to as *plaintext*) into a form an attacker could not understand (this form is referred to as *cyphertext*). The reversion[5] of encryption (turning cyphertext back into plaintext) is called decryption. The pair of corresponding encryption and decryption algorithms form a *crypto-system*(also often referred to as *cipher*).

The requirement of being hard to decrypt for the attacker and easy to decrypt for the intended recipient implies that the recipient has something the attacker has not. In the beginning of the use of encryption, this something was the algorithm used to encrypt the plaintext: without knowledge about the used algorithm, the attacker could not decrypt

[5] The reversibility of the encryption is not optional.

the cyphertext[6]. In the context of software-based encryption, this approach would be useless and dangerous: Most software written today is intended to be used by as many people as possible. This means that the attacker has easy access to a copy which he can reverse-engineer to identify the used algorithm. This is why all modern encryption algorithms follow the Kerckhoffs-principle[7], which states that the security of a crypto-system may not rely on the secrecy of the used algorithms.

Modern encryption algorithms utilize a secret information only known by the sender and the recipient of the message. This secret information is called *key*. The security of the encryption relies fully on the secrecy of the key.[8]

4.3.2.2. The two types of encryption-algorithms

Encryption mechanisms can be divided into two categories by the way they handle their keys: symmetric and asymmetric.

- Symmetric encryption algorithms use the same key for encryption and decryption. This means the key has to be exchanged through a secure channel.
- Asymmetric encryption algorithms use different keys for encryption and decryption. Each communication partner has a pair of keys: a public and a private one. The public key is used to encrypt the messages for a certain partner; it can be distributed freely, as it does not endanger the security of the message. The private key is used to decrypt messages encrypted with the public key; the secret key has to be kept secret in order to maintain security.

4.3.2.3. Advantages and disadvantages

Symmetric encryption requires that the sender and the recipient have exchanged the key over a secure channel prior to initialising encrypted communication. This defeats the purpose of using encryption in MURRAY, as we want to use encryption to establish a secure communication channel.

[6] The security of the Enigma-machine used by the Germans in WW2 relied heavily on the secrecy of the inner workings of the machine. The British military was only able to decrypt German radio-messages because Polish researchers had secured a unit of the predecessor-model in 1920. (The Enigma also used keys. The breaking of the keys was the hard part in breaking the Enigma.)

[7] The Kerckhoffs-principle was stipulated by the dutch linguist Auguste Kerckhoffs in 1883 in the essay *La Cryptographie militaire*, which was published in the french journal of military science

[8] If the algorithm doesn't contain errors.

Asymmetric encryption solves the problem of the key-exchange: The information the attacker could gain by eavesdropping on the key-exchange is of no worth for him: the intent of the attacker is to decrypt cyphertext, not to create it. This makes asymmetric encryption the only viable solution for the project.

However, even asymmetric encryption has a huge drawback: all asymmetric algorithms known today require enormous processing power compared to their symmetric cousins. The performance-discrepancy between the two types of crypto-systems is roughly by the factor of 1000. Even on todays most fastest processors, asymmetric encryption uses 100% of the available resources for whole seconds. In the intended usage-pattern of M.U.R.R.A.Y, which implies the possibility of recieving several messages the second, this performance-penalty is unacceptable.

The solution to this problem is the usage of a hybrid of both types. A hybrid scheme encrypts the plaintext symmetrically using a randomly generated key. This key is then encrypted using an asymmetric algorithm. The encrypted key and the encrypted plaintext together form the cyphertext. This approach combines the performance advantage of symmetric encryption while including the solution to the key-exchange-problem. Please see figure 4.2 on the next page for a graphical representation.

4.3.2.4. The chosen algorithms

Having decided to use a hybrid encryption-system, the actual algorithms to be used have to be chosen.

Symmetric cipher Today, following well-known symmetric encryption algorithms are considered to be fairly secure:

- Twofish
- Blowfish
- Serpent
- AES/Rijndael
- CAST5
- RC4
- TDES/DES

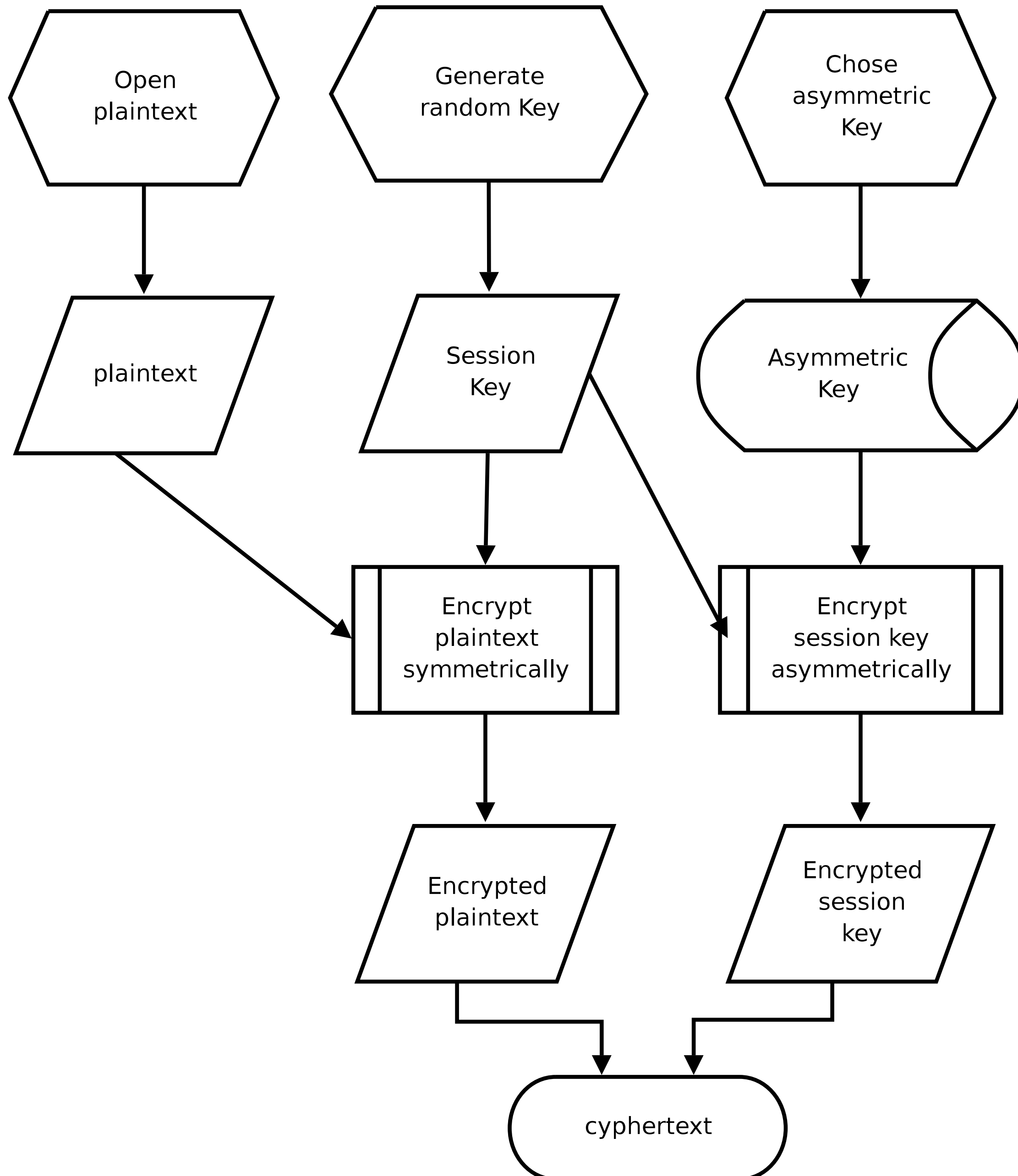

Figure 4.2.: Hybrid encryption scheme

- IDEA

RC4 is is a stream-cipher, meaning that it encrypts every bit of the plaintext as a individual unit. The other algorithms are block-ciphers, which means that the algorithms handle blocks of bits (usually 64 or 128 bits form a block) as units. The implications of block-ciphers are discussed in the implementation-chapter (5.4.2.1 on page 53).

The IDEA-algorithm could not be used for this project, as it is patent-encumbered (European patent EP-B-0482154, US patent #5,214,703 and japanese patent JP 3225440) and acquiring a license to use the algorithm would have required vast amounts of monetary resources.

TDES (Triple DES) was not used because it is one of the slowest ciphers on all-purpose processors. It was specifically designed to be fast when implemented in hardware, which leads to the weak performance in a software-implementation. While TDES still is considered to be secure, the other algorithms offer higher security margins and its usage was thus rejected.

RC4 (named after nothing in particular) has been proven to be susceptible to several attacks depending on the implementation[9] of the algorithm. As an analysis of the available implementations of RC4 would have been impossible to complete during the time of the project, RC4 was also rejected.

CAST5 is also a patented cipher (US patent #5,511,123). While patent-licenses are available royalty-free from the owner Entrust Inc., the legal ramifications were considered to be too complex to handle in a team without access to lawyers. As the rest of the algorithms were unencumbered by patents and have been placed into the public domain, it was decided that trying to get a license would have been a waste of resources.

Twofish, Blowfish and Serpent were all candidates which were considered for standartisation into the Advanced Encryption Standard by the US National Security Agency but were ultimately rejected because of performance and security concerns. While none of them were really broken, each of the ciphers has an attack which reduces the security of them. They are thus not used by MURRAY.

Rijndael is the cipher which the NSA chose for AES. AES offers superior performance and at the time of writing this work no known weaknesses exist. While the absence of known weaknesses does not imply absence of weaknesses, the usage of AES by the US government for its highest secrecy-level[10] implies a good security. As it is unencumbered

[9] The most notable weak implementation is WEP, the now unsecure way to encrypt wireless networks.

[10] Actually, AES is used for the secrecy-level "secret". For the highest level ("top secret"), AES has to be used with a key-length of 256 bit.

by patents and has been placed into the public domain, it was chosen as the symmetric cipher for MURRAY.

Asymmetric cipher The list of well-known asymmetric encryption algorithms is substantially shorter than the list of symmetric ciphers:

- RSA
- ElGamal

Other asymmetric encryption algorithms exist, but they are either too new to be considered well-known (like the Paillier crypto-system) or have too many implementation-shortcomings (like the Rabin crypto-system, which decrypts each cyphertext into 4 possible plaintexts of which only one is the correct plaintext).

Both RSA and ElGamal are patent-free and in the public domain. They offer comparable security and are both succible to the same type of weaknesses. This meant for the team that the algorithm could be chosen completely based on implementation-related criteria.

RSA was chosen because ElGamal requires a random number following certain rules to encrypt a plaintext; generating such a number would have been too cumbersome.

A note on key-length As longer keys mean more security, MURRAY uses AES with it's biggest key-size of 256 bit and RSA with a key-size of 2048 bit.

4.3.3. Identity

4.3.3.1. Why identity matters

While communicating, it is very rare that the recipient of the message is not important to the sender. Normally, the sender wants to communicate with a certain recipient. In order to be able to do that, the sender must have a way to identify the recipient. MURRAY does this by associating a so called MURRAY-ID to each participant of the network.

The other issue of identity is the question whether a message really originates from the sender. This issue is handled by the use of so called signatures in MURRAY.

4.3.3.2. The MURRAY-ID

In order to be able to refer to a certain participant of the MURRAY network, a unique identifier (ID) for each participant had to be invented. As the MURRAY network does not have a central authority which regulates the participants, each participant needs to generate an ID independently of the other members of the network.

This is a difficult feat, as duplicate IDs have to be avoided for the network to work. Duplicate IDs cannot be avoided completely; the risk of having a duplicate ID can only be minimised to a certain degree.

MURRAY minimises this risk by using a 256-bit long ID. This means that there are 2^{256} $(115,792,089,237,316,195,423,570,985,008,688 \times 10^{77})$ IDs[11]. The probability of having no duplicate ID when every internet-user today would participate in MURRAY can be calculated by calculating

$$\frac{(n_m)}{(n_m)} \times \frac{(n_m)-1}{(n_m)} \times \frac{(n_m)-2}{(n_m)} \times \cdots \times \frac{(n_m)-(n_i)}{(n_m)}$$

where n_m is the number of possible MURRAY-IDs and n_i is the number of internet participants.

As of June 2006, `internetworldstats.com` estimates the number of internet users as 1.133.408.294. We can approximise the propability by calculating[12]

$$\frac{(n_m)-(n_i)}{(n_m)}^{n_i}$$

The result of this calculation is bigger than 0.9999999999999999999999999999999999999 meaning that there is a 99.99999999999999999999999999999999999% chance that everything will work fine[13].

Realising this, the project-team decided to use the SHA-256 hashing-algorithm to generate the MURRAY-ID by hashing the public-key of the participant. As SHA-256 only uses lowercase characters and numbers, there still exist 36^{32} $(63,340,286,662,973,277,706,$-$162,286,946,812 \times 10^{49})$ possible MURRAY-IDs. It is more probable that the reader of

[11]For reference: Earths surface consists of 5.1×10^{20} square *millimeters*
[12]The result of this approximation is significantly lower than the real probability.
[13]Calculation of the actual value can only be done on a high-precision calculator to which the authors did not have access. Working with 32-bit double precision real arithmetic did result in rounding-errors (rounding to one).

this text wins in the lottery ten times in a row[14] than that two participants of the MURRAY network have the same ID. The team is sufficiently optimistic that an ID-collision will not occur and wishes the reader much luck with the lottery.

4.3.3.3. Signatures

The definition of a crypto-system is that two functions $f_e()$ for encryption and $f_d()$ for decryption of the message m exist, and $f_d(f_e(m))$ equals m. If $f_e(f_d(m))$ also equals m, then the crypto-system can also be used to generate signatures. The chosen RSA crypto-system is such a system.

Using his own secret-key, the sender uses the decryption function $f_d()$ on the plaintext. The recipient can now use the encryption function $f_e()$ with the public key of the sender on the cyphertext. If that results in the plaintext, the recipient can be sure that the sender really sent that plaintext.

This approach of course contains the huge performance-penalty asymmetric ciphers infer. To escape this performance-trap, MURRAY uses the SHA-256 hashing-algorithm to generate a 256-bit long hash of the plaintext. The hash is much shorter than the original message and can thus be ciphered much faster than the message. All the recipient has to do to check the signature is calculating the hash of the plaintext and checking if the sender also arrived at the same hash.

4.3.4. Known vulnerabilities

> *More people are killed every year by pigs than by sharks, which shows you how good we are at evaluating risk.*
>
> Bruce Schneier, famous cryptologist

The crypto layer only uses algorithms which have no known weaknesses or require the attacker to have full controll of the victims computer in order to exploit the weaknesses (which means that the attacker could also just read the input directly from the keyboard thus bypassing the complete crypto process). If the team did not make an implementatio error, the system should be secure. However, one attack of the crypto system still is possible:

[14] The chance of winning ten times in a row in the german lottery is $0,0000071511^{10}$

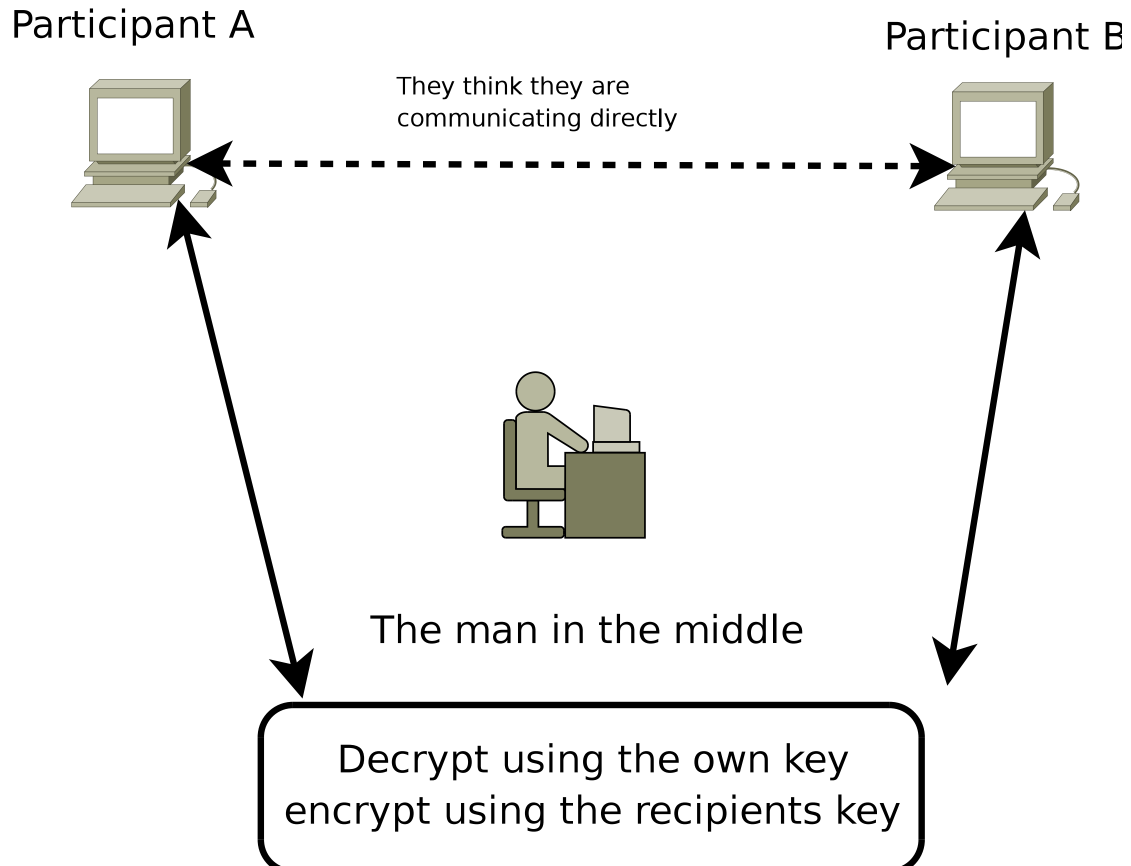

Figure 4.3.: A man in the middle attack

4.3.4.1. The man in the middle

MURRAY is vulnerable against a so called man-in-the-middle attack.

A man-in-the-middle attack works the following way: The attacker routes the whole communication of two participants over his own computer. The participants think they are communicating directly with each other, but are both communicating with the attacker, which relays the messages to the other participants. During the key-exchange, the attacker substitutes the participants keys with his own key. When the participants now try to encrypt the messages, they encrypt the messages with the key the attacker already has. The attacker then decrypts the messages, reads them/stores them, and encrypts the messages using the real key of the recipient. Please refer to figure 4.3 for a graphical representation of the attack.

This attack only works if the keys between the recipients have not been exchanged prior to the attacker gaining control of the communication.

Remedies against this attack The technical remedy against this attack would be to implement a public key infrastructure with a web of trust, where participants would vow for each other. As this does not solve the initial problem of a new participant to the network, this technical approach was considered to be too cumbersome to implement.

The only way this attack can be detected is by confirming the MURRAY-IDs and IP addresses of the recipients manually. This cannot be done automatically, as the attacker could write a filter which automatically substitutes the real IDs and IPs with his own. The chances of having every user of the network perform these checks is slim.

4.4. Application data

Although the protocol is designed in a way that arbitrary data can be transmitted using the very generic `User data` package, it was to decided to use Jabber/XMPP as application layer in the reference implementation.

4.4.1. Jabber/XMPP protocol basics

The terms "Jabber" and "XMPP" (see section 4.4.1.2 and 4.4.1.3 on the next page respectively) are used here interchangeably (unless specifically stated otherwise) to refer to XMPP, just like in the real world.

4.4.1.1. XML

Jabber is based completely on XML and as such inherits XML's capabilities, e.g. multiple namespaces in one document.

4.4.1.2. Jabber

The Jabber Protocol is an instant messaging protocol created in 1998 and released in 2000 by Jeremy Miller. Contrary to then-established IM protocols it is based on open standards such as XML, easily extensible and permits everyone owning a domain to participate in the network, just as SMTP does for email[15]. Thus it does not depend on the goodwill or availability of a central instance. Just as with Email if there is a server

[15]And just like email it is possible create a SRV record in DNS to make clear which Jabber server is responsible for which domain.

outage the affected domain is cut off of the Jabber network, but the rest of the network is unaffected.

Jabber is standardised in the form of the Extensible Messaging and Presence Protocol (XMPP).

4.4.1.3. XMPP

Starting in 2002 an XMPP working group was formed by the Internet Engineering Taskforce (IETF). This XMPP WG produced four Request for Comments (RFCs) in 2004 namely RFC 3920–RFC 3923[16] which are as of writing these lines reworked to be promoted from Proposed Standard to Draft Standard.

XMPP forms the basis of many independent and – thanks to the openess of the protocol – mostly interoperable client and server implementations. On the client side there are e.g. the Pidgin multi protocol IM client (formerly Gaim)[17], GoogleTalk[18], Apple's iChat[19] or Psi[20].

Many references to Jabber survive in the standards documents and in daily usage. E.g. the term "Jabber ID" (JID) is used in the RFCs. Also a "connection" always refers to a server-client connection, never to a server-server connection.

4.4.1.4. Fundamentals of XMPP

XMPP is made up of four "building blocks" one of which, the `<stream:stream/>` tag, will be discussed in section 5.7.3.1 on page 69 (see also [Adams, 2002], chapter 5).

The <presence/> tag is used to announce one's presence (or lack thereof) to the world/a single person and to subscribe to/probe another person's presence. The server adds a `from` attribute, and the client may add a `to` attribute in case the presence is directed.

There are subelements/attributes to signify the type, status, text and/or priority of the presence.

A simple example would look like this:

[16] See [Saint-Andre, 2004a], [Saint-Andre, 2004b], [Saint-Andre, 2004c], and [Saint-Andre, 2004d].
[17] See `http://pidgin.im/`.
[18] See `http://www.google.com/talk/`.
[19] See `http://www.apple.com/macosx/features/ichat/`.
[20] See `http://www.psi-im.org/`.

```
                    Sample <presence/> stanza
1 <presence>
2   <show>dnd</show>
3   <status>Studying!</status>
4 </presence>
```

Due to design decisions there is no presence subscription in MURRAY yet, so the next example serves solely to illustrate Jabber's presence subscription mechanism and the possibilities of the `<presence/>` tag.

```
                    Sample <presence/> stanza
1 <presence to="kirk@ncc-1701.ent.sfc.ufp.org" type="subscribe"/>
```

The <message/> tag handles, as can be deducted from the name, messaging and is thus crucial. Via its optional `type` attribute the type of the message is specified (such as `normal`, `chat`, `groupchat`, `headline` or `error`) and via the `<body/>` subelement the message text is carried. As with the `<presence/>` tag the `from` attribute is set by the server and the `to` element is set by the client in all but unusual cases and of course specifies the receiver.

```
                    Sample <message/> stanza
1 <message from='kirk@ncc-1701.ent.sfc.ufp.org'
2          to='scotty@ncc-1701.ent.sfc.ufp.org'>
3   <body>We need more power!</body>
4 </message>
```

This example shows a simple message from `kirk@ncc-1701.ent.sfc.ufp.org` to `scotty-@ncc-1701.ent.sfc.ufp.org`.

The <iq/> tag The Information/Query tag is sort of a "girl friday" tag for everything not fitting in the two categories "message to someone" or "presence announcement." It facilitates such things as roster management and user registration/authentication.

Its `<query/>` subelement, which makes heavy use of namespaces signified by the xmlns attribute[21], in conjunction with the attribute `type` signify what action is to be taken by the server or the client.

The next two examples from the client's point of view are the communication necessary to add `spock@ncc-1701.ent.sfc.ufp.org` to the roster.

[21] N.b.: these namespaces should not be confused with real XML namespaces!

Sample <iq/> stanza: SEND

```
<iq id="adduser1" type="set">
  <query xmlns="jabber:iq:roster">
    <item jid="spock@ncc-1701.ent.sfc.ufp.org" name="Spock"/>
  </query>
</iq>
```

Sample <iq/> stanza: RECEIVE

```
<iq type="set">
   <query xmlns="jabber:iq:roster">
     <item jid="spock@ncc-1701.ent.sfc.ufp.org" name="Spock"
           subscription="none"/>
   </query>
</iq>
```

4.4.2. Why Jabber?

The question arises "Why use the Jabber protocol?" The answer should be obvious taking the considerations in the last two sections into account:

- Jabber is an open, well documented protocol.
- Jabber is simple.
- A plethora of clients exist freeing the authors of MURRAY from the burden of programming yet another chat client.

Having to create a chat *client* would have resulted in much work not directly related to the core functionality or respectively in a very rudimentarily implemented, hardly platform independent chat client not nearly up to expectations.

5. Implementation

Aim of the implementation phase was to create a prototype of a reference implementation which would be able to demonstrate the central protocol features and perhaps serve as a basis for possible further development.
During the course of developing a reference implementation many things, which are only generally defined in the protocol specification, become a lot more tangible.

5.1. Development process

The open source development model was used as the development framework for the implementation.

This model – although not formally specified – basically desribes a means for geographically distributed, highly independent and parallel development of software. In the case of MURRAY this means that each team member took responsibility for one part of the later prototype, which made greatly independent development possible.

The central idea of the development process is to split up the software into largely independent modules. Each of these modules has a clearly defined purpose and is largely developed by one person (or a group of people) which are responsible for the whole development process, from early design through implementation until test and maintenance. Whereas no clear “phases” of development can be defined and the result itself is never really considered “finished”.
All the different modules are working together by using a set of clearly defined interfaces, making the product itself very modular and highly extendable.

Adapting this development process for the MURRAY implementation meant to split up the whole of the program into four modules:

Network core Contains all the network logic like known peers cache, request/response coordination etc.

XML converter This Module is responsible for parsing the incoming messages and creating the ones going out.

Cryptography module Covers the complete crypto-layer, from creating a UID to encrypting and decrypting application data

XMPP module This is the connector for the – until now – only supported application: XMPP.

In order to make the modules work together properly, also interfaces between them had to be defined which was done on function-call level. Adhering to this model made it surprisingly easy to plug all the modules together and make them work once they were functional, although miscommunication at some points complicated the process. But these problems could also be dealt with surprisingly fast without the need to do a major rewrite of the affected modules.

5.2. Programming language and tools

Especially when following a highly dynamic development process like it was done during the creation of the MURRAY prototype, the tools used for programming, the programming language itself and the means of communication used within the team are extremely important to the success of the development.

This section will give a short overview of the tools used in creating the prototype.

5.2.1. Python

The prototype is written in Python[1], a free, dynamic object-oriented programming language. Python emphasises concepts like programmer productivity, clear syntax and platform independence.
It also features an extensive standard library, making the language a powerful tool enabling the programmer to achieve working results very fast but still being flexible enough to make bigger projects possible.

Python is a bytecode language like Java[2], the sourcecode is compiled into a bytecode-format which is then executed in a virtual machine.

[1] `http://www.python.org`
[2] `http://java.sun.com`

Python syntax is very clean and is focused on readability, which makes it extremely easy to learn and understand the language. A very simple Python example would look like this:

A simple Python program

```
for i in range(99,0,-1):
    bword = 'bottles' if i>1 else 'bottle'
    print str(i), bword, 'of beer on the wall'
    if i == 0:
        print 'Go to the store and buy some more'
```

Another important advantage of Python is its platform independence: Python implementations are available for most Unix-like systems, Microsoft Windows, Apple Mac OS and also a number of more exotic systems like Palm OS[Lutz, 2005].
The platform independence also holds true for the extensive standard library. A python program which is not using some exotic, platform-dependant modules will most likely run on all supported systems.

The standard library itself was the main reason to use Python: The complete prototype could be implemented in relatively short time by just using standard modules or modules readily available and themselves written in Python. This made the whole process a lot easier.

Base for the implementation was version 2.5 of the language.

5.2.2. Tools used

To enable the team to use the highly dynamic development model described in 5.1 on page 47, several freely available tools were used for source code management and communication.

Version control Subversion[3] was used as a system for managing and versioning of the source code.

Wiki As a central means for communication, documentation and specifications, a mediawiki[4]-based wiki was used extensively throughout the development process.

[3] http://subversion.tigris.org/
[4] http://www.mediawiki.org

Bug tracking For keeping track of requirements and bugs in the software the bug tracking system Flyspray[5] was used. This system also proved extremely helpful in distributing tasks and communication between the team members.

5.3. Network logic

This section describes the implementation of the logic handling the MURRAY protocol, as specified in 3.2 on page 18 and 4.1 on page 23

5.3.1. Incoming packages

The general approach for handling incoming packages is relatively simple: A TCP server listening for incoming MURRAY connections was implemented using Python's `ThreadingTCPServer` class, which is a basic TCP server starting a new Thread for each incoming connection[Martelli, 2006].
The incoming packages are put through the XML parser described in 5.6.3 on page 59, which extracts the necessary information from the MURRAY-packages and returns them in data dictionaries, dynamically extendable hashtables[Lutz, 2005], which are then processed according to the specifications formulated in 4.1.6 on page 27.

5.3.1.1. Checks

Before the information from the package is actually used, some basic checks are performed to ensure that the information given concerning receiver and sender fits to the real circumstances. The program checks if the given `destination` UID is the actual own UID, i.e. ensuring that the package was really meant to arrive at this host. Additionally, it is checked whether the given `source` IP-address is the one from which the package was received, in case the IP of the sender has changed without him noticing (because of NAT or a Firewall). In both cases, the original request is ignored and instead, a `Hello` package is sent containing the correct information.

5.3.1.2. Package handling

The functions handling the packages themselves are basically what was already defined in 4.1.6 on page 27, implemented in program code.

[5] `http://www.flyspray.org`

Thus, the `source` information is first used to update the own known peers cache. Following this step, appropriate responses for the received responses are constructed. In the case of received `user data` packages, the encapsulated data is decrypted, verified (see 4.3.3.3 on page 41) and then passed over to the application layer described in 5.7 on page 65.

As an example, here the program code handling incoming `search requests`.

Function handling incoming search requests

```
1  receivesearchrequest(self, argsIn_dict):
2      """Responsible for incoming search requests. At current state, it is only
3      possible to search for a Murray ID
   
5      returns nothing"""
6      self.__knownPeers.update(argsIn_dict['source']['uid'],\\
7          argsIn_dict['source']['ipversion'], \\
8          argsIn_dict['source']['ip'], \\
9          argsIn_dict['source']['port'])
10     knownNodes = Search().search(argsIn_dict['searchfields'], \\
11         argsIn_dict['operation'])
12     if knownNodes:
13         argsOut_dict = {'destination' : argsIn_dict['source'],
14                 'source' : my_identity,
15                 'responseto' : argsIn_dict['requestid'],
16                 'searchfields' : argsIn_dict['searchfields'],
17                 'foundnodes' : knownNodes}
18         self.sendsearchresponse(argsOut_dict)
19     else:
20         if argsIn_dict['ttl'] >= 10:
21             pass
22         elif argsIn_dict['ttl'] > 0:
23             ttl -= argsIn_dict['ttl']
24             self.sendsearchrequest(argsIn_dict['searchfields'], ttl = ttl)
25         else:
26             pass
```

5.3.2. Outgoing packages

Outgoing messages are constructed by a number of functions which are either called to fulfil requests from other modules or started as response to a received request.

Once the process is started, all necessary information is collected, transformed into a

XML package using the XML-creator-functions described in 5.6.2 on page 59 and finally sent to the receiver via a standard TCP socket connection.

5.3.2.1. Sending requests

When sending a request, the contents of it are stored in a *Waiting List* together with its request ID and a timestamp. In case of receiving a response, the program is able to assign the response to a request and act accordingly.

5.3.2.2. Unknown UIDs

In case a package is to be sent to a UID which is not available in the known peers cache, the original data is stored and a search is started for this UID instead.

Here an example of the function sending an echo request:

Function creating an echo request

```
def sendechorequest(self, murray_id):
    """sends an echo request and adds it to the waiting buffer

    returns nothing"""
    if self.__knownPeers.known(murray_id):
        peer = self.__knownPeers.getByMurray_ID(murray_id)
        destination = {'uid':murray_id, 'ipversion':peer[0], \\
            'ip':peer[1], 'port':peer[2]}
        argsOut_dict = {'type':'echorequest', 'destination':destination, \\
            'source':my_identity}
        argsOut_dict['requestid']=self.__waitingList.add('echo', argsOut_dict)
        out('echorequest', argsOut_dict)
    else:
        destination = {'uid':murray_id}
        argsOut_dict = {'type':'echorequest', 'destination':destination,\\
            'source':my_identity}
        self.sendsearchrequest(search = {'murray_id' : murray_id}, \\
            searchtype = 'murraysearch', request = argsOut_dict)
```

5.3.3. Service functions

Additionally to the functions handling the specified package types, there are several “service functions” running in their own threads which take care of regularly exchanging known peers information, checking after peers which have not been contacted for a while or cleaning up the known peers cache.

5.4. Cryptography

5.4.1. The library

In order to have a secure implementation of RSA, AES and SHA-256, it was decided to use an existing library instead of re-implementing the algorithms on our own. The library which was chosen was the Python Cryptography Toolkit[6]. This particular library was chosen because it is available for all operating systems used in the project (Debian GNU/Linux, Kubuntu 7.04, Windows XP, Windows Vista) and is the library recommended by the Python developers.

5.4.2. Interesting code fragments

5.4.2.1. Padding with AES

As AES is a block-cipher, the cleartext needs to be of a length which is a multiple of 16 bytes. This means that in $\frac{15}{16}$ of the cases the cleartext needs to be padded.

```
1  def _aes_encrypt(text, password):
2  """Encrypts the text with AES. The password has to be a 256 bit long string."""
3   aes_key = Crypto.Cipher.AES.new(password)
4   paddingchar="X";
5   if text[-1]=="X":
6    paddingchar="Y"
7    text = text + paddingchar
8   else:
9    if text[-1]=="Y":
10    paddingchar="X"
11    text = text + paddingchar
12    if (len(text) % 16) <> 0:
13     while (len(text) % 16) <> 0:
14      text = text + paddingchar
15  return aes_key.encrypt(text)
```

5.4.2.2. How random is random?

The normal system-functions to get random bytes does not really return randomness but pseudorandom byte-sequences, which repeat after a certain time. Such pseudorandom

[6] http://www.amk.ca/python/code/crypto

bytes pose a threat for the security of the crypto-layer, as the AES-encryption uses a randomly generated key. An attacker could try to reproduce the pseudorandom byte-stream and thus guess the password.

To avoid this, the MURRAY-prototype uses a safer way to get random bytes: the time-device with the highest precision on the system is regularily polled and the least significant byte is taken as random byte. As the high-precision time-devices in modern systems have a resolution which is below milliseconds, the sequence of bytes gained through this way is very random and does not repeat itself. The big disadvantage of this method is that is relatively slow; gaining the necessary bytes for key-creation can take up to 10 seconds.

```
1 randomnessgenerator = Crypto.Util.randpool.RandomPool(8192)
2 #getting a cryptographic "good" randomnessgenerator
```

5.5. XML schemas

The XML schemas have been defined according to the specification of the Peer-to-Peer layer (see section 4.1.6 on page 27). Because the definition of peers (or “clients”) that consists of several properties is used in every XML schema, this definition is put into an own XML schema that is included into the other XML schemas.

In the MURRAY protocol there are ten XML schemas which define the format of the XML stream that is sent in packages between the clients:

- cacherequest
- cacheresponse
- echorequest
- echoresponse
- hello
- pkrequest
- pkresponse
- searchrequest

- searchresponse
- userdata

All XML schemas can be found in appendix A on page 83 and are described in the following. In appendix B on page 91 there are XML examples for all XML schemas.

5.5.1. client

All XML schemas include a definition of a `source` client (peer) and a `destination` client. The `source` client is the sender and the `destination` client the receiver of the package. Some XML schemas also contain a list of clients. Because the client definition is used several times, it is put into the XML schema `client`.

The `client` XML schema contains four elements: An unique identifier (`uid`) serves as identification of the client. Also information about the IP address (`ip`), the IP version 4 or 6 (`ipversion`) and the used `port` are transmitted.

5.5.2. cacherequest

If a client asks for a list of other available clients, an XML stream according to the XML schema `cacherequest` is sent. According to this XML schema a cache request includes a `destination`, a `source`, a `count` value and a request ID (`requestid`). The `count` value defines how many available clients at the most shall be transmitted in the response to the cache request. The `requestid` is used to identify each single `cacherequest` and to assign a `cacheresponse` to a `cacherequest`.

5.5.3. cacheresponse

A cache response transmits as reply to a `hello` or a `cacherequest` package a list of available clients. This XML stream is defined in the XML schema `cacheresponse`. Besides the `destination` client and the `source` client the `cacheresponse` package contains a `list` of available `nodes` and a request ID (`requestid`) to identify each single `cacheresponse` and to assign it to the respective `cacherequest`. The `list` is an element that contains a number of `nodes` not exceeding the `count` value defined in the respective `cacherequest`.

5.5.4. echorequest

An echo request is used to find out whether a specific `client` is available (comparable to ICMP ping). The elements that are defined in the `echorequest` XML schema are an `destination` element, a `source` element and a `requestid` element that contains a request ID to track the echo request.

5.5.5. echoresponse

The XML schema `echoresponse` defines the XML stream that is the reply to an `echo-request`. This XML schema includes the `destination` client, the `source` client and the request ID (`requestid`) that was used in the respective `echorequest` package.

5.5.6. hello

If one client wants to register itself, it needs to send a `hello` package. This package type is also used in the unlikely case that a client receives a package that should have been sent to another client. In this case the client sends a `hello` package containing the correct user data to the client that has sent the package.

The `hello` XML schema contains the `destination` client, the `source` client and a `count` value that defines how many `nodes`' data shall be transmitted in the replying `cacherequest` package. Differing from the `client` definition described above in this package it is not necessary to specify the IP address (`ip`) and the IP version (`ipversion`) of the `source` client. This is important in the case that the `source` client does not know this data because of masquerading, the usage of NAT etc.

5.5.7. pkrequest

Because the MURRAY protocol uses encryption it is necessary for the clients to get to know the public key of the clients it wants to communicate with. In `pkrequest` packages a client asks another client for its public key. The respective XML schema contains the `destination` client, the `source` client and a request ID (`requestid`) to track each request.

5.5.8. pkresponse

By the usage of a `pkresponse` package a client replies to a `pkrequest` and sends its public key to the respective client. Besides the `publickey` element in the `pkresponse` XML schema the elements `destination`, `source` and `requestid` are defined. The `requestid` is the one transmitted in the respective `pkrequest` package.

5.5.9. searchrequest

If a client searches for specific other clients, it can send a search request. The `searchrequest` XML schema includes the `destination` client, the `source` client, a `ttl` (time to live) value, a list of `searchfields`, a `relay` value and the `requestid`. The `ttl` value specifies after how many hops the search shall end. The list of `searchfields` contains a list of `field`s and the searched `value` of each field. Also the information is transmitted whether it is sufficient that a client matches one of the search fields of if all search fields have to be met (default value). The `relay` value is optional and defines whether a client that forwards the request is also interested in the respective `searchresponse`. The `requestid` is used to track the `searchrequest`.

5.5.10. searchresponse

A searchresponse package is a reply to a `searchrequest`. In the respective XML schema the elements `destination`, `source`, `requestid`, `searchfields` and `foundnode` are defined. The `requestid` is taken from the respective `searchrequest`. In the optional `searchfields` element the search fields from the respective `searchrequest` may be transmitted. The element `foundnode` may occur several times and contains a `client` that matches the specifications of the `searchfields`.

5.5.11. userdata

The `userdata` XML schema defines the packages including the communication itself. It contains the `destination`, the `source` and the `data` which is a message from one client to another.

5.6. XML Conversion

The packages that are sent from one peer to another contain XML. Thus it is necessary to create XML and parse it. In this section the necessary mechanisms are explained. A description of the concrete implementation of these mechanisms for this project is another part of this section.

5.6.1. DOM and MiniDOM

“The Document Object Model [DOM] is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page.”[7]

DOM documents are structured hierarchical. Beginning with the root of the document all constructs in the document are stored as a tree of node objects in the way that each node object may have childe nodes.

DOM is of interest, because it provides an XML feature set that “adds additional interfaces used to represent entity and notation declarations provided by the document type declaration (though not the document type itself), and some lexical information helpful in generating a modified document”[8]. Thus by the usage of DOM it is possible to create and read XML.

For Python several DOM implementations exist. MiniDOM provides “a simple implementation and smaller memory footprint than a full DOM implementation”[9]. Nevertheless it is faster than a fully compliant DOM and suits the needs of most users.[10]

“Overall, miniDOM may be best for loading simple (not necessarily small) configuration files for your applications, dealing with form submissions from web pages, handling user authorization, and using it anywhere a ’little’ bit of XML is needed. You can reduce memory and time overhead by using miniDOM. These are two elements of significant importance in web application development.”[11]

[7] `http://www.w3.org/DOM/#what`
[8] [Jones and Fred L. Drake, 2002], page 87
[9] [Jones and Fred L. Drake, 2002], page 90
[10] [Jones and Fred L. Drake, 2002], page 89
[11] [Jones and Fred L. Drake, 2002], page 90

5.6.2. Creating XML with Python and MiniDOM

To create a XML string in Python the module `xml.dom.minidom` has to be imported. This module contains all necessary functions. The steps for creating XML are the following: First a miniDOM `Document` is created. Then the function `createElement()` is used to create a root element in the `Document`. By means of the `setAttribute()` method it is possible to add an attribute to an element. For each child element the functions `createElement()` and `append()` have to be called to create another element and to append it to the parent element. Finally the function `toxml()` is called to convert the root element (including all child elements and attributes) into XML.

5.6.3. Parsing XML with Python and MiniDOM

To retrieve data that is included in XML it is necessary to parse the XML. In Python there is the module `xml.dom.minidom` that contains functionality for parsing XML. This module provides two functions: `parse` and `parseString`. `Parse` takes either a file name or an open file object as parameter, `parseString` takes a string. "Both functions return a `Document` object representing the content of the document."[12] In this project a string from the MURRAY packages shall be parsed, thus the function `parseString` is used.

As next step the data has to be retrieved from the `Document` object. This can be done by the usage of the function `getElementsByTagName`. This function returns a list of elements with a certain tag name from a certain `Document` object. By means of the function `data.strip` the data of one element can be fetched. An attribute of one element can be retrieved using the function `getAttribute(attributename).strip`.

5.6.4. Parsing XML with SAX

The Simple API for XML (SAX)[13] is a cross-language, event driven serial access parser for XML streams which need not necessarily be complete or documents in the form of files. SAX is not formally standardised, but the original Java implementation serves as the normative one.

An XML stream is fed to the parser incrementally and for every

- text node

[12] [van Rossum, 2006]
[13] See `http://www.saxproject.org/`.

- element node
- processing instruction
- comment

an event is raised. This is a stark contrast DOM, which needs to hold the complete document in memory. As can be seen in section 5.7.3.1 on page 69 this simply is impossible using the Jabber protocol[14].

Important to MURRAY's Jabber server are only the first two, text nodes and element nodes. See section 5.6.5.3 on page 63.

5.6.5. Implementation

5.6.5.1. Implementing XML creation

Because MURRAY packages contain an XML string, it is necessary to create this string. This functionality is implemented in MURRAY's `core_to_xml` module and uses miniDOM (see sections 5.6.1 on page 58 and 5.6.2 on the preceding page).

As soon as an instance of the class `CreateXml`, which is part of the `core_to_xml` module, is created, a miniDOM `Document` is created.

Portions of xml_to_core module

```
1 from xml.dom.minidom import Document
2 class CreateXml:
3         def __init__(self):
4                 self.doc = Document()
```

The function `incoming()` is used to receive a dictionary, call the function regarding to the package type (parameter `type_str`) and return an XML string.

incoming()

```
1 def incoming(self, type_str, args_dict):
2         try:
3                 function = getattr(self, 'createXml_' + type_str)
4                 return function(args_dict)
5         except AttributeError:
6                 raise NotImplementedError
```

[14] Of course, if one regards every XML stanza as a complete XML document it would be possible to abuse DOM to parse XMPP, but every conceivable advantage such as validation would be lost.

The XML string itself is created in the respective functions, which receive the dictionary and receive the created XML string.

For example by means of the function `createXml_userdata()` the XML string for a `userdata` package is created.

Portions of createXml_userdata()

```
def createXml_userdata(self, argsIn_dict):
        udata = self.doc.createElement("userdata")
        [...]
        data = self.doc.createElement("data")
        dataText = self.doc.createCDATASection(argsIn_dict['data'])
        data.appendChild(dataText)
        udata.appendChild(data)
        str_xml = "<?xml version='1.0'?>" + udata.toxml()
        return str_xml
```

First an `udata` element is created as root element in the miniDOM `Document`. Then the `destination` element and the `source` element are created and appended to the `udata` element. This is done analogous to creating and appending the `data` element, which is done next. For this purpose after creating the `data` element this element has to be filled with data: A CDATA[15] section is created in the miniDOM `Document` which contains the respective value from the dictionary. Then this section is appended as child node to the `data` element, which itself is appended as child node to the root element `udata`. As next step an XML string is created which contains the XML declaration that specifies the used XML version and the udata element that is converted to an XML string. Finally this XML string (cf. appendix B.10 on page 98) is returned to `incoming()`.

5.6.5.2. Implementing parsing XML with MiniDOM

Incoming XML packages have to be parsed. For this functionality the function `incoming()` of MURRAY's `xml_to_core` module is called, which uses miniDOM (cf. sections 5.6.1 on page 58 and 5.6.3 on page 59). This function parses the XML string, which is a parameter of the function, and reads the type of the transmitted package. Then `incoming()` calls the respective function to convert the whole XML string into a dictionary, which is returned to `incoming()`.

[15] "CDATA sections may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup. CDATA sections begin with the string '<![CDATA[" and end with the string "]]>'." ([Bray et al., 2006b])

incoming()

```
def incoming(self, str_xml):
    try:
        dom = xml.dom.minidom.parseString(str_xml)
        function = getattr(self, 'readXml_' + dom.documentElement.nodeName)
        return function(str_xml)
    except AttributeError:
        raise NotImplementedError
```

For example `incoming()` discovers that the XML string contains data about a search request. In this case the function `readXml_searchrequest()` is called. This function parses the XML string which is a parameter of the function. Then all data included in the XML string is stored into variables. Finally a dictionary containing the values of these variables is created and returned to `incoming()`.

readXml_searchrequest()

```
def readXml_searchrequest(self, str_xml):
    dom = xml.dom.minidom.parseString(str_xml)

    destination_dict = self.get_client(dom, 0)
    source_dict = self.get_client(dom, 1)
    ttl = self.get_data(dom, 'ttl', 0)
    requestid = self.get_data(dom, 'requestid', 0)
    relay = self.get_data(dom, 'relay', 0)
    operation = self.get_attribute(dom, 'searchfields', 'operation', 0)

    searchfields_dict = { }
    nodelist = dom.getElementsByTagName('value')
    for i in range(len(nodelist)):
        value = self.get_data(dom, 'value', i)
        field = self.get_attribute(dom, 'value','field',i)
        searchfields_dict[field] = value

    argsOut_dict = {'type':'searchrequest', 'destination':destination_dict,
                    'source':source_dict, 'ttl':ttl,
                    'searchfields':searchfields_dict, 'operation':operation,
                    'requestid':requestid, 'relay':relay}
    return argsOut_dict
```

This function is very short even though it is quite complex. Everytime data (or attribute data) is stored into a variable, the function `get_data()` (or `get_attribute()`) is called which reads the data out of the miniDOM `Document`. A special case is the `searchfields`

XML tag, because it may occur several times. Therefore a `for-loop` is used to store the data and attributes into a separate dictionary which is finally embedded into the returned dictionary.

A separate dictionary is also used for `destination` peers and `source` peers. In this case the function `get_client()` is called which uses also the function `get_data()` to store the data into variables which are merged into a dictionary. This dictionary is returned to `readXml_searchrequest()` and embedded into the dictionary which is the return value of the latter function.

get_attribute()

```
def get_attribute(self, dom, tagname, attrname, number):
    nodelist = dom.getElementsByTagName(tagname)
    node = nodelist[number]
    return node.getAttribute(attrname).strip( )
```

The function `get_attribute()` receives the miniDOM `Document`, the name of the respective tag, the name of the respective attribute of the tag and a number as parameters. This number is needed, if there are several tags of the same name. Three steps are needed to get the value of the attribute and to return it to the calling function. First a list of nodes is created which contains all nodes with the specified tag name of the particular miniDOM `Document`. Secondly the requested node is separated. As third and last step the value of the respective attribute of the node is returned. The function `get_data()` works similar. In this case instead of the function `getAttribute().strip()` the function `firstChild.data.strip()` is called.

5.6.5.3. Implementing parsing XML with SAX

Parsing XML with SAX involves the creation of a SAX parser and passing it a handler. Only the SAX handler has to be implemented on its own if it is to do something sensible. This SAX handler is implemented in the MURRAY XMPP Server (see section 5.7.2) in the class MurrayHandler and is derived from ContentHandler. It handles the four basic XML stanzas of the Jabber protocol (see section 4.4.1.4 on page 44).

If the SAX parser encounters an opening XML tag it calls `startElement()` with the name of the element and its attributes in a dictionary:

Portions of startElement()

```
def startElement(self, name, attrs):
    # Prepare an array element to hold the current element's text.
```

```
3       # If there was anything in it now it is not.
4       self.__currentElement = name
5       self.__currentCharacters[self.__currentElement] = ""

7       if name == "stream:stream" and not self.__inStream:
8           self.__handleStreamStart(attrs)

10      elif name == "message":
11          self.__handleMessageStart(attrs)

13      elif name == "body":
14          self.__handleBodyStart(attrs)
```

As can be seen in this example the `startElement()` method "branches" into different private methods which handle the details. Of course the opening of a XML tag is not alone of interest, so there is the equivalent method, `endElement()`:

Portions of endElement()

```
1   def endElement(self, name):
2       if name == "stream:stream" and self.__inStream:
3           self.__handleStreamEnd()

5       elif name == "message":
6           self.__handleMessageEnd()

8       elif name == "body":
9           self.__handleBodyEnd()
```

The "real work" is done by these private methods, as it can only be determined what todo with an XML stanza when it is complete.

But this again is not sufficient, as of course there can be text between the XML tags which has to be recorded somehow. This is what the `characters()` method is there for:

Portions of characters()

```
1   def characters(self, characters):
2       if self.__recordCharacters:
3           self.__currentCharacters[self.__currentElement] += characters
```

Due to the nature of the SAX parser the `characters()` method has to be written to possibly append the text, as it is not guaranteed that e.g. the text "Omicron-Omicron-

Alpha-Yellow-Daystar-2-7" in the example `<authorization-code>Omicron-Omicron--Alpha-Yellow-Daystar-2-7</authorization-code>` is passed to the `characters()` method in one piece.

Of course a method pair has wanting to record the text between two XML tags has to indicate so and has to set `self.__recordCharacters` to true in the `__handleFooStart()` and to false in the `__handleFooEnd()` method. What to do with the recorded text and/or attributes is up to the `__handleFooEnd()` method.

5.7. The XMPP Server

5.7.1. External architecture

By definition the MURRAY network is serverless and so the different semantics have to be translated. For example the Jabber ID (JID) which is constructed like this: `<username>@<host>[<resource>]` serves to uniquely identify a user. As with an ordinary email address the combination of username and hostname guarantees the uniqueness. But in the serverless P2P world a hostname or IP address is nothing which can be relied on. So the UID (see section 4.3.3.2 on page 40) has to serve this purpose. For this the UID has to fit the abovementioned schema of `<username>@<host>[<resource>]`. The following two functions serve for translating in both directions:

MID ↔ JID translation

```
1  def midToJid(self, hashvalue, alias, resourcep):
2      """Converts a Murray ID (hash, alias) to a Jabber ID.  If
3      resourcep then append Murray as a resource."""
4
5      jid = alias + "@" + hashvalue + ".murray.p2p"
6      if resourcep: jid += "/Murray"
7
8      return jid
9
10 def jidToMid(self, jid):
11     """Converts a Jabber ID (alias@hash.murray.p2p/resource) to a
12     Murray ID."""
13
14     jidRE = re.compile(r'@([^.]*)')
15     alias, hashvalue, throwaway = jidRE.split(jid)
16
17     return alias, hashvalue
```

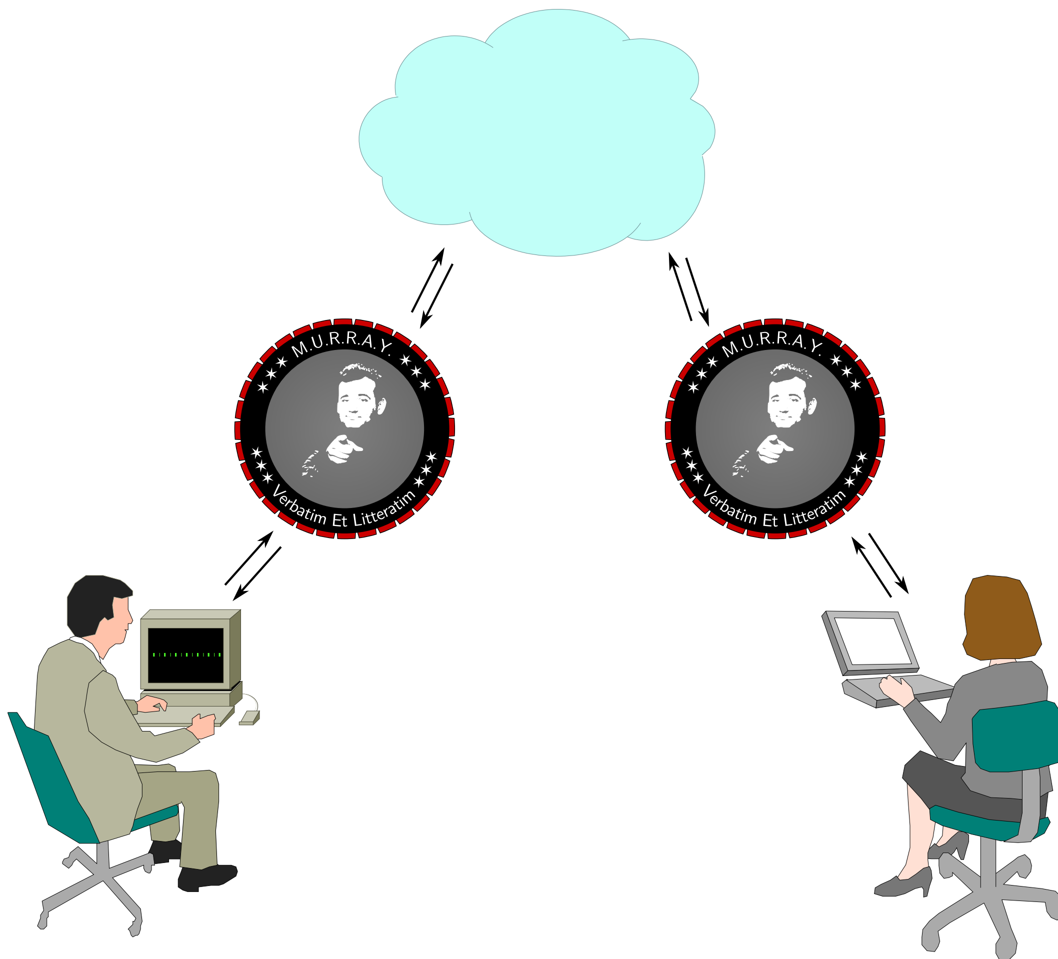

Figure 5.1.: Architecture of the Jabber/MURRAY translation

Of course this is just one example of required adaption.

5.7.2. Internal architecture

5.7.2.1. Overview

Figure 5.2 on the following page shows the internal architecture of the MURRAY XMPP server. The communication can be split into six steps:

1. The user sends some XMPP data to the server
2. This data is fed to the parser
3. When the XML stanza is closed the parser calls the appropriate action of the server

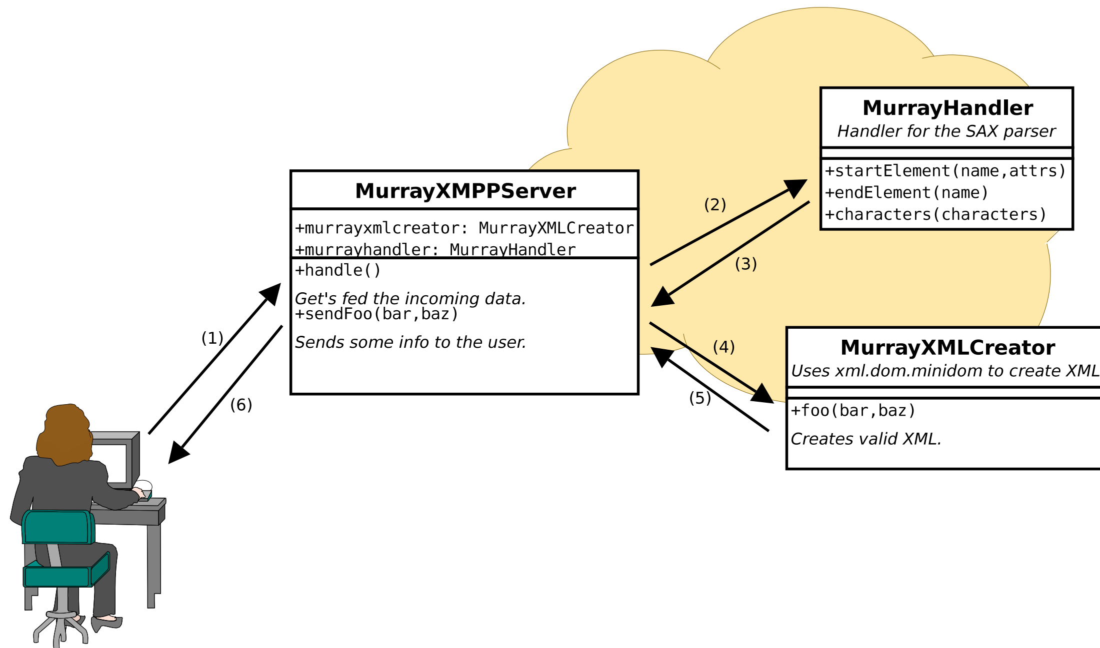

Figure 5.2.: The internal architecture of the MURRAY XMPP Server

4. The server calculates its response and calls the XML creator
5. The XML creator wraps the response in valid XML and passes it back to the server
6. The server sends the XML to the client

Of course if the received data is to be passed to another peer this is done. Depending on the data there will be feedback which has to be sent to the user's client. The "internet cloud" in the back of the diagram symbolises this.

5.7.2.2. Details

The class `MurrayXMPPServer` is main class of the XMPP server part. It is derived from `SocketServer.StreamRequestHandler` (see e.g. [Martelli, 2006], chapter 20). As such it is constructed to be bound to port and listen for incoming connections. The next example shows a possibility to do so:

Usage of MurrayXMPPServer

```
1  if __name__=='__main__':
2      server = None
3      try:
4          # Create an instance of our server class.  The empty hostname is
5          # equivalent to localhost.
```

```
6         print "Creating TCP server..."
7         server = SocketServer.TCPServer(('', PORT), MurrayXMPPServer)

9         # Enter an infinite loop to serve.
10        server.serve_forever()
```

This code snippet is actual code from the `MurrayXMPPServer` class which served to start the XMPP server without having to have the other code running. Using `TCPServer` exactly one connection is possible. It would have been possible to use `ThreadingTCPServer`, but this would not only have added to the complexity but also would have been contrary to the design that exactly one client can connect to the XMPP server and no more.

Any incoming communication is handled in inherited the `handle()` method which looks like this:

handle()

```
1  def handle(self):
2      print "Connection from " + self.client_address[0] + ", port " \
3          + self.client_address[1]
4      print "Initiating session..."

6      saxparser = make_parser()
7      saxparser.setContentHandler(self.murrayhandler)

9      while True:
10         recvDat = self.request.recv(8192)
11         if not recvDat: break
12         saxparser.feed(recvDat)
```

If a connection is established some info is printed, then the instance attribute `murrayhandler` is created as the content handler for the SAX parser (see section 5.6.4). An infinite while loop is then entered which gets any incoming data in chunks of 8,192 bytes maximum[16], though of coure less data may actually arrive. This data is then fed to the SAX parser. If no data is received because the connection has been closed the while loop is broken.

[16]Which is the upper limit for sockets.

5.7.3. Implementation

5.7.3.1. The communication stream

A Jabber communication stream is a sequence of well-formed XML stanzas exchanged by the client and the server and may start like this:

Jabber XML stream start

```
1  <?xml version="1.0"?>
2  <stream:stream
3      xmlns:stream="http://etherx.jabber.org/streams"
4      to="jabber.org"
5      xmlns="jabber:client">

7  <?xml version="1.0"?>
8  <stream:stream
9      xmlns:stream='http://etherx.jabber.org/streams'
10     id='DEADBEEF'
11     xmlns='jabber:client'
12     from='jabber.org'>
```

The first XML stanza is generated by the connecting client, and the second is generated by the server in reply. As can be seen `to` and `from` have been exchanged and an `id` has been added. This is the so called connection ID and allows the server to more easily track a connection.

5.7.3.2. Connection

This exchange of headers opens up a connection and allows further communication between the server and the client, so that all valid communication is encapsulated in a valid XML stream.

Now that a connection is established the question comes to mind what needs to be done to disestablish it. If everything works perfectly the client emits `</stream:stream>` to which the server answers itself with `</stream:stream>` and closes the TCP communication. Of course the internet being fundamentally an unreliable network this does not always hold true, so if a client disconnects the TCP connection without prior closing of the stream this is acceptable also. And if a client does not respond for a (configurable) timeout a session is also regarded as closed.

All communication between the client and the MURRAY XMPP server takes place between the formal opening described in section 5.7.3.1 and this section.

5.7.3.3. Authenticating with the MURRAY XMPP server

If one does disregard the registration of a new Jabber account (and the other 47 little things a Jabber client asks for but does not necessarily want answered) for a moment the next step is to a authenticate to the MURRAY XMPP server. There are numerous possibilities and combinations enabling multiple levels of security and/or encryption, but as shown in section 5.7.2 on page 66 the communication between Jabber client and the MURRAY XMPP server never leaves the local computer unencrypted and therefor does not need any of them[17].

[17] If the user does not trust the other users on a system encryption would *not* solve the problem as another user determined enough could nevertheless snoop on the communication between the server and the client. Enforcing trust on an untrusted system is impossible.

6. Testing

Formal testing procedures had a very low priority during the project, as the team used the bazaar-approach[Raymond, 2001]. This means, that the work was split up into thematic congruent pieces which only communicated through well-defined interfaces. The module-based design of the python framework enabled the team to focus on the the work at hand without having to worry about the other parts of the code-tree.

To test the work, only two types of tests had to be done regularly: unit-tests and integration-tests.

6.1. Unit tests

Unit tests were done on a per-module[1] basis.

Python has a construct which makes unit testing really easy: the variable `__name__` is a read-only string which the python runtime sets at load-time of the module. If the module is called from within another part of code, `__name__` is set to the name of the calling code-fragment. If the module is executed directly, it contains the string `__main__`. It now is as simple as comparing the content of `__name__` to `"__main__"` to branch between production-procedures and testing-code.

Here an example from the crypto-layer:

```
1  """TEST-Routine
2  If the module is executed directly, the following code will be executed
3  Basically, we use every function once.
4  """
5  if __name__ == '__main__':
6          errors = 0;
7          print '=============================================='
8          print 'Testing Crypto-module'
```

[1] One module corresponds exactly to one file

```
 9          print '=============================================='
10          print ''
11          print 'Testing randomnessgenerator'
12          print '=============================================='
13          print 'randomnessgenerator-size:', randomnessgenerator.bytes
14          print 'randomnessgenerator-entropy:', randomnessgenerator.entropy
15  #        print 'random bytes of the day:', randomnessgenerator.get_bytes(8)
16          print 'randomnessgenerator-entropy:', randomnessgenerator.entropy
17          print
18          print 'Testing randomnessgenerator'
19          print '=============================================='
20          print 'sha1hasher will now hash the string "bacon"'
21          print 'hashed bacon:',sha1hash('bacon')
22          print 'should be   : 8abf15bef376e0e21f1f9e9c3d74483d5018f3d5'
23          if sha1hash('bacon')<>'8abf15bef376e0e21f1f9e9c3d74483d5018f3d5':
24                  errors = errors +1;
25          print
26          print 'Testing RSA, AES + our identity'
27          print '=============================================='
28          print 'Our Murray-ID is:', murrayid()
29          print 'Have we saved our Public Key in the keyring:', \
30          known(murrayid()), '(should be True)'

32          if known(murrayid())<>True:
33                  errors = errors + 1;
34          print 'I will now sign+encrypt \"bacon\" :', \
35          decrypt(murrayid(), encrypt(murrayid(),'bacon'))
36          print 'This should of course be        : bacon'
37          if decrypt(murrayid(), encrypt(murrayid(),'bacon'))<>'bacon':
38                  errors = errors + 1
39          if errors <1:
40                  print
41                  print '=============================================='
42                  print 'ALL TESTS COMPLETED SUCCESSFULLY'
43                  print '=============================================='
44          else:
45           print
46           print '=============================================='
47           print 'Something went horribly wrong.'
48           print '=============================================='
```

6.1.1. Integration test

The integration tests were done regularly between individual members which had defined interfaces between each other. As the interfaces were defined well in advance, the integration tests did not offer any surprises.

7. Additional Aspects

Other aspects concerning the MURRAY protocol were only considered shortly and it was decided to keep them out of further development for various reasons. This chapter will give a short overview of these aspects.

7.1. Security

Although the transmission over MURRAY itself is strongly encrypted, cryptographically verified and can be considered secure, the mere usage of a P2P system alone creates certain security problems.

7.1.1. Opening up firewalls

Since most “normal” internet users do not have the proper equipment to securely open services towards the internet, and so need to shut down Firewalls and other security equipment at least partially. This is necessary because of the very basics of the network’s architecture require a method for receiving incoming TCP connections.

Unfortunately, opening up to the internet in such a way is always a security problem. Even if the MURRAY implementation used is 100% bug free (which nobody will ever be able to prove), there are still ways for possible attackers to do harm on operating system or network level.

But since this basic problem virtually always exists when the user decides to offer some service to the general internet community, it was not considered a critical issue – also because there simply *is* no real solution anyway.

7.1.2. DDoS

Theoretically, MURRAY (like other P2P systems) offers an ideal possibility for staging a Distributed Denial of Service (DDoS) attack. A possible attacker would only have to

infect one peer and would get the information and possibility needed to infect a great number of others for free.

7.1.3. Disrupting the network

Although the network should operate stable under normal conditions and the protocol is able to cope with occasional errors in the management data exchanged between peers, it will most probably not be capable of working in case an attacker decides to “flood” the network with false information on purpose (and has the resources to so).

7.2. Features left out

Besides the protocol features which had been implemented or specified, several other possible features are conceiveable and seem useful, but were not implemented because of time constraints.

The architecture is designed in a way that is very open and extendable, making the implementation of these features easy in the future.

7.2.1. Further checks on incoming data

In order to overcome the problem mentioned in 7.1.3, additional checks could be performed on incoming packages. This could even include the propagation of blacklists inside the network.

7.2.2. File transfer

The MURRAY protocol is designed in a way that any kind of data can be transmitted over the network, which of course includes file transfer. But a method for actually doing this is not part of the current prototype.

7.2.3. Full contact list management

Because of the relatively complex way the XMPP protocol handles buddy-list management, it was not possible to implement this to a full extend. Although the online status of

the user can be transmitted over MURRAY and displayed, there is no method for adding or managing contacts on the buddy list. The internal logic for contact list management is implemented but the communication with the XMPP-client is not. So any modifications to the own contact list have to be done manually – a somehow unsatisfactory limitation.

7.2.4. Zeroconf

As described in 4.1.7.4 on page 31, it is nearly impossible to enter the network without any prior knowledge of any peers which are already participating.
While this is a unchangeable truth in the internet, there are methods for things like that in LANs. These technologies are summarised under the term “zeroconf”[1] and are designed to enable users to use network resources without any configuration. This is usually achieved by announcing the services a device offers via broad- or multicasts.

One – or several – of these technologies could very well be used by MURRAY peers to announce the network and connect to it.

This possibility is a very useful feature which could be implemented with ease, but stays an extension after all. So it is not part of the specification or implementation.

7.2.5. More compatible encryption scheme

In the current implementation, the cryptographic methods used in the protocol are quite Python-specific. Although all techniques used are openly available and well documented, it would still be a major handicap for everybody deciding to write an implementation in any other programming language.

So it would be nice to make the encryption compatible to a widely-used standard for PK-encryption, like OpenPGP[2]. This would make the whole project even more platform-independent.

7.2.6. Multi user chat

Until now, it is possible to send messages to *single* peers.
A method for conducting chats with multiple users would be a nice feature and the core functions necessary to track multi user chats could be implemented quite fast. But within the given time frame, other things had to take precedence unfortunately.

[1] `http://www.zeroconf.org`
[2] `http://www.openpgp.org`

8. Conclusion

8.1. The project

In the six months of working on Murray there were ups and downs and even miscommunication. All in all coordination of the team members worked surprisingly well due to no small amount of – sometimes new – tools such as SVN for versioning, MediaWiki for documentation, FlySpray for bug tracking or LaTeX for writing this document.

The three phases of the project, from the beginning brainstorming sessions over the writing down of concepts to the actual coding of the prototype, all worked out very well, with a little bias towards the coding, of course.

Some incompatibilities between the Unix® and the Microsoft® Windows™ way of doing things had to be overcome, but these did not pose serious obstacles in any way thanks to the free software nature of all parts of the project.

8.2. The programming language

Even though only one author had prior exposure to Python on a more than merely a superficial level Python was chosen because of its cleanlines, conciseness, well documentation and large module library.

The fact that the whole code written for the murray prototype is only using modules provided by the Python base library speaks volumes.

8.3. The result

In principle the prototype does what it was designed for and only time constraints kept it from being better tested or more deployed.

The Murray prototype should enable the implementation of a final version rapidly as the envisioned core functions of the project are already implemented. Some minor optional points were left out, but due to the modularity of the code should prove easy to implement if one choses to do so.

9. Bibliography

[Adams, 2002] Adams, D. (2002). *Programming Jabber.* O'Reilly Media, Inc. ISBN 9780596002022.

[Bray et al., 2006a] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2006a). Extensible markup language (xml) 1.0 (fourth edition), chapter 1. http://www.w3.org/TR/2006/REC-xml-20060816/#sec-intro.

[Bray et al., 2006b] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2006b). Extensible markup language (xml) 1.0 (fourth edition), chapter 2.7. http://www.w3.org/TR/2006/REC-xml-20060816/#sec-cdata-sect.

[Bray et al., 2006c] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2006c). Extensible markup language (xml) 1.0 (fourth edition), chapter 4. http://www.w3.org/TR/2006/REC-xml-20060816/#sec-physical-struct.

[Fallside and Walmsley, 2004] Fallside, D. C. and Walmsley, P. (2004). Xml schema part 0: Primer second edition, chapter 2. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/#PO.

[Fielding et al., 1999] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Rfc2616: Hypertext transfer protocol – HTTP/1.1. http://tools.ietf.org/html/rfc2616.

[Jones and Fred L. Drake, 2002] Jones, C. A. and Fred L. Drake, J. (2002). *Python & XML.* O'Reilly Media Inc., first edition. ISBN 0596001282.

[Klensin, 2001] Klensin, J. (2001). Rfc2821: Simple mail transfer protocol. http://tools.ietf.org/html/rfc2821.

[Lutz, 2005] Lutz, M. (2005). *Python, kurz & gut.* O'Reilly, third edition. ISBN 389721511X.

[Martelli, 2006] Martelli, A. (2006). *Python in a Nutshell.* O'Reilly Media, Inc., second edition. ISBN 9780596100469.

[Maymounkov and Mazières, 2001] Maymounkov, P. and Mazières, D. (2001). Kademlia: A peer to peer information system based on the xor metric. Technical report, New York University. http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf.

[Mintert, 2002] Mintert, S. (2002). *XML & Co - Die W3C-Spezifikationen für Dokumenten- und Datenarchitektur*. Addison-Wesley. ISBN 3827318440.

[Raymond, 2001] Raymond, E. S. (2001). *The Cathedral and the Bazaar*. O'Reilly Media, Inc., revised edition. ISBN 9780596001087.

[Saint-Andre, 2004a] Saint-Andre, P. (2004a). Rfc3920: Extensible messaging and presence protocol (xmpp): Core. http://tools.ietf.org/html/rfc3920.

[Saint-Andre, 2004b] Saint-Andre, P. (2004b). Rfc3921: Extensible messaging and presence protocol (xmpp): Instant messaging and presence. http://tools.ietf.org/html/rfc3921.

[Saint-Andre, 2004c] Saint-Andre, P. (2004c). Rfc3922: Mapping the extensible messaging and presence protocol (xmpp) to common presence and instant messaging (cpim). http://tools.ietf.org/html/rfc3922.

[Saint-Andre, 2004d] Saint-Andre, P. (2004d). Rfc3923: End-to-end signing and object encryption for the extensible messaging and presence protocol (xmpp). http://tools.ietf.org/html/rfc3923.

[van Rossum, 2006] van Rossum, G. (2006). Python library reference. http://docs.python.org/lib/module-xml.dom.minidom.html.

10. Abbreviations

AES	Advanced Encryption Standard
AIM	AOL Instant Messanger
API	Application Programming Interface
DDoS	Distributed Denial of Service
DNS	Domain Name System
DOM	Document Object Model
DSL	Digital Subscriber Line
DTD	Document Type Definition
HTTP	HyperText Tranfer Protocol
ICMP	Internet Control Message Protocol
ID	Identifier
IETF	Internet Engineering Taskforce
IM	Instant Messaging
IP	Internet Protocol
ISP	Internet Service Provider
JID	Jabber ID
LAN	Local Area Network
MSN	Microsoft Network
MURRAY	M.U.R.R.A.Y.
NAT	Network Address Translation

NSA	United States National Security Agency
P2P	Peer-to-Peer
PAT	Port Address Translation i.e. Masquerading
PK	Public Key
RFC	Request for Comment
SAX	Simple API for XML
SGML	Standard Generalized Markup Language
SIP	Session Initiation Protocol
SMTP	Simple Mail Transfer Protocol
SRV	DNS Server record
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol over Internet Protocol
TTL	Time To Live
UDP	User Datagram Protocol
UID	MURRAY Unique Identifier
VoIP	Voice over IP
W3C	World Wide Web Consortium
WG	Working Group
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

A. XML schemas

A.1. Client

```
1  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2  <xsd:complexType name="client">
3    <xsd:sequence>
4      <xsd:element name="uid" type="xsd:string" minOccurs="1" maxOccurs="1"/>
5      <xsd:element name="ip" type="xsd:string" minOccurs="1"
6        maxOccurs="1"/>
7      <xsd:element name="port" type="xsd:nonNegativeInteger"
8        minOccurs="1" maxOccurs="1"/>
9      <xsd:element name="ipversion" minOccurs="1" maxOccurs="1">
10         <xsd:simpleType>
11           <xsd:restriction base="xsd:nonNegativeInteger">
12             <xsd:enumeration value="4"/>
13             <xsd:enumeration value="6"/>
14           </xsd:restriction>
15         </xsd:simpleType>
16     </xsd:element>
17   </xsd:sequence>
18 </xsd:complexType>
19 </xsd:schema>
```

A.2. Cache request

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:include schemaLocation="client.xsd"/>
<xsd:element name="cacherequest" type="cache"/>
<xsd:complexType name="cache">
  <xsd:sequence>
    <xsd:element name="destination" type="client" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="source" type="client" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="count" type="xsd:nonNegativeInteger" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="requestid" type="xsd:nonNegativeInteger" minOccurs="1"
      maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

A.3. Cache response

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:include schemaLocation="client.xsd"/>
<xsd:element name="cacheresponse" type="cacheres"/>
<xsd:complexType name="cacheres">
  <xsd:sequence>
    <xsd:element name="destination" type="client" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="source" type="client" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="list" minOccurs="1" maxOccurs="1">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="node" type="client" minOccurs="0"
            maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="requestid" type="xsd:nonNegativeInteger"
      minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

A.4. Echo request

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:include schemaLocation="client.xsd"/>
<xsd:element name="echorequest" type="echoreq"/>
<xsd:complexType name="echoreq">
  <xsd:sequence>
    <xsd:element name="destination" type="client" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="source" type="client" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="requestid" type="xsd:nonNegativeInteger" minOccurs="1"
      maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

A.5. Echo response

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:include schemaLocation="client.xsd"/>
<xsd:element name="echoresponse" type="echores"/>
<xsd:complexType name="echores">
  <xsd:sequence>
    <xsd:element name="destination" type="client" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="source" type="client" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="requestid" type="xsd:nonNegativeInteger" minOccurs="1"
      maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

A.6. Hello

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:include schemaLocation="client.xsd"/>
<xsd:element name="hello" type="hi"/>
<xsd:complexType name="hi">
  <xsd:sequence>
    <xsd:element name="destination" type="client" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="source" minOccurs="1" maxOccurs="1">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="uid" type="xsd:string" minOccurs="1"
            maxOccurs="1"/>
          <xsd:element name="ip" type="xsd:string" minOccurs="0"
            maxOccurs="1"/>
          <xsd:element name="port" type="xsd:nonNegativeInteger" minOccurs="1"
            maxOccurs="1"/>
          <xsd:element name="ipversion" minOccurs="0" maxOccurs="1">
            <xsd:simpleType>
              <xsd:restriction base="xsd:nonNegativeInteger">
                <xsd:enumeration value="4"/>
                <xsd:enumeration value="6"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="count" type="xsd:nonNegativeInteger" minOccurs="1"
      maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

A.7. Public key request

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:include schemaLocation="client.xsd"/>
<xsd:element name="pkrequest" type="pkreq"/>
<xsd:complexType name="pkreq">
  <xsd:sequence>
    <xsd:element name="destination" type="client" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="source" type="client" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="requestid" type="xsd:nonNegativeInteger" minOccurs="1"
      maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

A.8. Public key response

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:include schemaLocation="client.xsd"/>
<xsd:element name="pkresponse" type="pkres"/>
<xsd:complexType name="pkres">
  <xsd:sequence>
    <xsd:element name="destination" type="client" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="source" type="client" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="publickey" type="xsd:string" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="requestid" type="xsd:nonNegativeInteger" minOccurs="1"
      maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

A.9. Search request

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:include schemaLocation="client.xsd"/>
<xsd:element name="searchrequest" type="search"/>
<xsd:complexType name="search">
  <xsd:sequence>
    <xsd:element name="destination" type="client" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="source" type="client" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="ttl" default="4" minOccurs="1" maxOccurs="1">
      <xsd:simpleType>
        <xsd:restriction base="xsd:nonNegativeInteger">
          <xsd:max-inclusive value="255"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="searchfields" minOccurs="1" maxOccurs="1">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="value" type="xsd:string" minOccurs="1"
            maxOccurs="unbounded">
            <xsd:attribute name="field" type="xsd:string" use="required"/>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="operation" use="optional" default="and">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="and"/>
              <xsd:enumeration value="or"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="requestid" type="xsd:nonNegativeInteger" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="relay" type="client" minOccurs="0" maxOccurs="8"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

A.10. Search response

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:include schemaLocation="client.xsd"/>
<xsd:element name="searchresponse" type="searchres"/>
<xsd:complexType name="searchres">
  <xsd:sequence>
    <xsd:element name="destination" type="client" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="source" type="client" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="requestid" type="xsd:nonNegativeInteger" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element name="searchfields" minOccurs="0" maxOccurs="1">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="value" type="xsd:string" minOccurs="1"
            maxOccurs="unbounded">
            <xsd:attribute name="field" type="xsd:string" use="required"/>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="operation" use="optional" default="and">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enmueration value="and"/>
              <xsd:enmueration value="or"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="foundnode" type="client" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

A.11. User data

```
1  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2  <xsd:include schemaLocation="client.xsd"/>
3  <xsd:element name="userdata" type="udata"/>
4  <xsd:complexType name="udata">
5    <xsd:sequence>
6      <xsd:element name="destination" type="client" minOccurs="1"
7        maxOccurs="1"/>
8      <xsd:element name="source" type="client" minOccurs="1" maxOccurs="1"/>
9      <xsd:element name="data" type="xsd:string" minOccurs="1" maxOccurs="1"/>
10   </xsd:sequence>
11 </xsd:complexType>
12 </xsd:schema>
```

B. XML examples

B.1. Cache request


```
<?xml version='1.0'?>
<cacherequest>
        <destination>
                <uid>014b4b04988fb2422465b6f4bf96b3e2d29760ae</uid>
                <ip>82.123.231.174</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </destination>
        <source>
                <uid>1c5e9a51e5d33b006ee1a70bb0adffa75f3d4065</uid>
                <ip>217.224.12.12</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </source>
        <count>100</count>
        <requestid>28940</requestid>
</cacherequest>
```

B.2. Cache response


```
<?xml version='1.0'?>
<cacheresponse>
        <destination>
                <uid>1c5e9a51e5d33b006ee1a70bb0adffa75f3d4065</uid>
                <ip>217.224.12.12</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </destination>
        <source>
                <uid>014b4b04988fb2422465b6f4bf96b3e2d29760ae</uid>
                <ip>82.123.231.174</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </source>
        <list>
                <node>
                        <uid>c01c06ab4a22edf5f71ae51a9695b4759078ef9a</uid>
                        <ip>182.168.6.1</ip>
                        <port>4711</port>
                        <ipversion>4</ipversion>
                </node>
                <node>
                        <uid>796155745ade5a9a771a19abafa89ed7599c99c9</uid>
                        <ip>84.38.47.42</ip>
                        <port>4711</port>
                        <ipversion>4</ipversion>
                </node>
                <node>
                        <uid>(string)</uid>
                        <ip>87.165.137.80</ip>
                        <port>4711</port>
                        <ipversion>4</ipversion>
                </node>
        </list>
        <requestid>46497</requestid>
</cacheresponse>
```

B.3. Echo request

```
<?xml version='1.0'?>
<echorequest>
        <destination>
                <uid>014b4b04988fb2422465b6f4bf96b3e2d29760ae</uid>
                <ip>82.123.231.174</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </destination>
        <source>
                <uid>1c5e9a51e5d33b006ee1a70bb0adffa75f3d4065</uid>
                <ip>217.224.12.12</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </source>
        <requestid>24148</requestid>
</echorequest>
```

B.4. Echo response

```
<?xml version='1.0'?>
<echoresponse>
        <destination>
                <uid>1c5e9a51e5d33b006ee1a70bb0adffa75f3d4065</uid>
                <ip>217.224.12.12</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </destination>
        <source>
                <uid>014b4b04988fb2422465b6f4bf96b3e2d29760ae</uid>
                <ip>82.123.231.174</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </source>
        <requestid>21963</requestid>
</echoresponse>
```

B.5. Hello

```
1  <?xml version='1.0'?>
2  <hello>
3          <destination>
4                  <uid>014b4b04988fb2422465b6f4bf96b3e2d29760ae</uid>
5                  <ip>82.123.231.174</ip>
6                  <port>4711</port>
7                  <ipversion>4</ipversion>
8          </destination>
9          <source>
10                 <uid>1c5e9a51e5d33b006ee1a70bb0adffa75f3d4065</uid>
11                 <port>4711</port>
12         </source>
13         <requestid>34346</requestid>
14         <count>100</count>
15 </hello>
```

B.6. Public key request

```
1  <?xml version='1.0'?>
2  <pkrequest>
3          <destination>
4                  <uid>014b4b04988fb2422465b6f4bf96b3e2d29760ae</uid>
5                  <ip>82.123.231.174</ip>
6                  <port>4711</port>
7                  <ipversion>4</ipversion>
8          </destination>
9          <source>
10                 <uid>1c5e9a51e5d33b006ee1a70bb0adffa75f3d4065</uid>
11                 <ip>217.224.12.12</ip>
12                 <port>4711</port>
13                 <ipversion>4</ipversion>
14         </source>
15         <requestid>42461</requestid>
16 </pkrequest>
```

B.7. Public key response


```
1  <?xml version='1.0'?>
2  <pkresponse>
3          <destination>
4                  <uid>1c5e9a51e5d33b006ee1a70bb0adffa75f3d4065</uid>
5                  <ip>217.224.12.12</ip>
6                  <port>4711</port>
7                  <ipversion>4</ipversion>
8          </destination>
9          <source>
10                 <uid>014b4b04988fb2422465b6f4bf96b3e2d29760ae</uid>
11                 <ip>82.123.231.174</ip>
12                 <port>4711</port>
13                 <ipversion>4</ipversion>
14         </source>
15         <publickey>
16                 8614 bc2d 3f09 d614 e413 8c9c 1399 424f 564a 163e a25f 673b
17                 1314 a3bd 9864 d012 5c61 b2cf d772 6c2c 0d8d bd18 fc2c 7769
18                 15f2 9ca8 6da3 9e7c d559 65ee 5873 f0cf c48a 227d ca5f b31d
19                 79ff 4c87 435e 4c2e 2e26 54c6 ac8c 6cb8 3208 1ed5 d8ea dbf0
20                 bd8a bd57 ca50 98b5 afd7 9c8d c9b1 7109 2c38 05e1 f59e 8295
21                 271d bb79 e9a0 abc3 dd60 a3e2 6e50 891d fb41 4fc6 0ae2 4420
22                 b7cf 645a e10a 5806 f44c af59 1af4 42b5 49ff 5fda 3929 4a17
23                 4a7a 2b46 4cc2 e2a4 87ef 135c 5025 97f8 0728 3591 eac8 4d44
24                 e3a4 6e18 9389 4ab1 a5fe e036 2e91 27f2 7a4b 528d 5c3d 7ded
25                 c19b 77ba ad0b bd45 2b31 11dd 2729 24d1 3994 3ed3 70e9 a5ee
26                 5ad7 f31a d0bf 8aa3 f570 9b63 edfa 0355 97e1 7be9 3a94 f191
27                 df8b 11e4 d282 2428 5420 72af d38a d965 a925 8c75 d33a 3d89
28                 1796 a702 48c2 6dd3 7265 e1d0 74d5 f1d7 1307 e050 9d3b 029c
29                 4bf6 6f9c ccfa 5a3a 0306 351e 6ce3 cc07 3fb5 3c61 0af5 2f64
30                 c341 4816 3d35 c54b dc79 01c6 32ea 4fce 9aa7 ed78 cbe7 894d
31                 9ac8 e9f4 a9d0 995d ff2d 55b4 e5b9 5447 5ec7 9b9f 8ecd da97
32                 816b 0677 7505 6717 4380 9ed4 e108 f8b3 99e0 fba6 f058 f670
33                 4732 d83f ea44 7338 8248 8525 e2e2 fb02 244f 8cef df60 a876
34                 5dd8 9d57 c035 cc85 bd9c df2b 41ca 9ee0
35         </publickey>
36         <requestid>41416</requestid>
37 </pkresponse>
```

B.8. Search request


```
1  <?xml version='1.0'?>
2  <searchrequest>
3          <destination>
4                  <uid>014b4b04988fb2422465b6f4bf96b3e2d29760ae</uid>
5                  <ip>82.123.231.174</ip>
6                  <port>4711</port>
7                  <ipversion>4</ipversion>
8          </destination>
9          <source>
10                 <uid>1c5e9a51e5d33b006ee1a70bb0adffa75f3d4065</uid>
11                 <ip>217.224.12.12</ip>
12                 <port>4711</port>
13                 <ipversion>4</ipversion>
14         </source>
15         <ttl>4</ttl>
16         <searchfields operation="or">
17                 <value field="murrayid">
18                         796155745ade5a9a771a19abafa89ed7599c99c9
19                 </value>
20                 <value field="alias">
21                         Picard
22                 </value>
23         </searchfields>
24         <requestid>12697</requestid>
25 </searchrequest>
```

B.9. Search response


```
<?xml version='1.0'?>
<searchresponse>
        <destination>
                <uid>1c5e9a51e5d33b006ee1a70bb0adffa75f3d4065</uid>
                <ip>217.224.12.12</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </destination>
        <source>
                <uid>014b4b04988fb2422465b6f4bf96b3e2d29760ae</uid>
                <ip>82.123.231.174</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </source>
        <requestid>12697</requestid>
        <foundnode>
                <uid>c01c06ab4a22edf5f71ae51a9695b4759078ef9a</uid>
                <ip>182.168.6.1</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </foundnode>
</searchresponse>
```

B.10. User data

```
<?xml version='1.0'?>
<userdata>
        <destination>
                <uid>014b4b04988fb2422465b6f4bf96b3e2d29760ae</uid>
                <ip>82.123.231.174</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </destination>
        <source>
                <uid>1c5e9a51e5d33b006ee1a70bb0adffa75f3d4065</uid>
                <ip>217.224.12.12</ip>
                <port>4711</port>
                <ipversion>4</ipversion>
        </source>
        <data>
          1-7-3-4-6-7-3-2-1-4-7-6-Charlie-3-2-7-8-9-7-7-7-6-4-3-Tango-7-3-2-
          Victor-7-3-1-1-7-8-8-8-7-3-2-4-7-6-7-8-9-7-6-4-3-7-6-Lock
        </data>
</userdata>
```

C. GNU Free Documentation License

Version 1.2, November 2002

Copyright ©2000,2001,2002 Free Software Foundation, Inc.

51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image

format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents,

unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.