Final report of Project Group 441
Project concubi*Net*

Norbert Basmaci, Edin Dedagic, Patrick Echtenbruck, Samuel Goossen, Serhad Ilgün, Sebastian Lehnhoff, Dominik Petzelt, Johannes Schneider, Holger Titze, Ömer Yildiz, Felix Yu

Jörg Platte, Peter Schramm, Edwin Naroska

Version 1.0, December 1, 2004

Contents

Abstract

This document presents the progress of our project group from October 2003 to September 2004. We improved and enhanced concubi*Net*, a development of our former project group.

The first section of this document gives a short description of concubi*Net* and describes some scenarios. For more information we refer to the final report of our former project group[1].

Section two surveys the given concubi*Net*. It describes its architecture and mentions some problems and features that were missing.

The following section applies to the optimization of concubi*Net*. It illustrates the new architecture and some optimizations we implemented. Furthermore, it describes new features, such as our new Wrapper-Factory, which allows the automated generation of wrappers, our new graphical interfaces and the new web-interface.

The next section is called Load Balancing. It is about a profiling tool we created to determine the size and CPU-usage of objects and about our mapping service.

As the first version of concubi*Net* had no security features, the following section illustrates our new security concept.

Afterwards we explain the handling and configuration of concubi*Net* and give some information about the provided CD.

1 What is concubi*Net*?

1.1 Motivation, ideas and philosophy

As the final report of our previous project group describes, they developed and implemented a distributed operating system for ubiquitous computing and ad-hoc networking called concubi*Net*. Concubi*Net* allows the movement of objects from one device to another, where any kind of device that is able to run a Java Virtual Machine can participate.

With regard to ubiquitous computing small computers and micro controllers in ordinary technical tools are used to make their services available in wired areas. The task is to get a better convenience and efficiency for the human user. To implement such an environment we had to create a distributed operating system, which is able to manage different types of devices. That means interoperability becomes a leading aspect, especially in the context of mobile and wireless communication. Looking at in-house networks as an example, the remote control of devices like lights or heaters using a cell phone or PDA as an interacting target is possible. Thinking further applications that handle some requests automatically are imaginable, for example an automated light control system, that dims the light in accordance to different user profiles. This goes hand in hand with the function of location awareness. In addition to that a type of content awareness is possible, if the system is able to decide between alternatives according to the time of day or the actual temperature outside for example. You can also think of some kind of intelligent resource management in this operating system. Just imagine the possibility of moving tasks from smaller devices (e.g. PDA) to bigger machines (PC) automatically. It is just a matter of being connected in the same net by this operating system. Even moving of complete objects is thinkable. Thus applications that need very much memory can be moved object by object to other machines and still fulfill their task together. As this works absolutely automatically, the user will never know about these things, which is the most convenience an operating system can provide.

1.2 Illustration: Sea of movable objects

This is the description of the concept of concubi*Net* when we started our work with PG441:

1.2.1 Sea of movable objects - Definition

To realize the former described systems, we decided to create some sort of "sea" of movable objects (figure 1), which will enable us to fulfill all the requirements. In this section, this concept of a "sea" of movable objects is going to be illustrated. To understand the idea of concubi*Net* one can imagine that every device which is capable of running CCN-services can be seen as a basin. This basin, as shown in the following figures, can house a number of bigger and smaller fish depending on its size. The fish, of course, represents any service or application running on that device. It is necessary that these basins can exchange the fish they are housing, because a fish might get bigger the longer it lives in its environment. That way it would be possible that bigger basins could "help out" smaller basins and every single fish is able to live - no matter which basin he lives in. The water of all basins combined would be concubi*Net* in our imagination. And there is another important aspect: because the fish does not care in

which basin he lives and he is able to swap to another basin, basins can be integrated into the accumulation of basins as well as being removed from it freely - the fish are distributed to the basins automatically. This underlines the capability of mobility in this concept. As a matter of fact there is no real "sea"; its an accumulation of basins as described before and the movable objects are the fish. In order to give the reader a more detailed overview of this whole scenario, a few figures might be helpful.

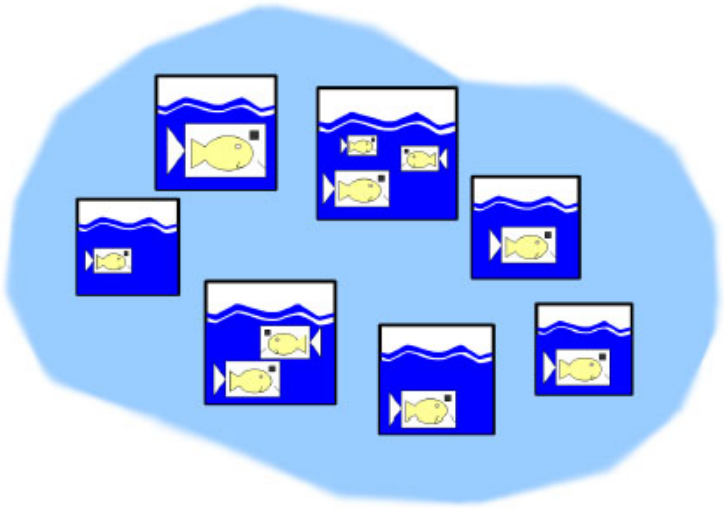


Figure 1: Sea of movable objects

As you can see, two basins of different size are joint in this case (figure 2). The smaller one runs out of space, because the trout it is housing has grown. The bigger basin has no space either; it is already housing a carp and a pike which need all the space.

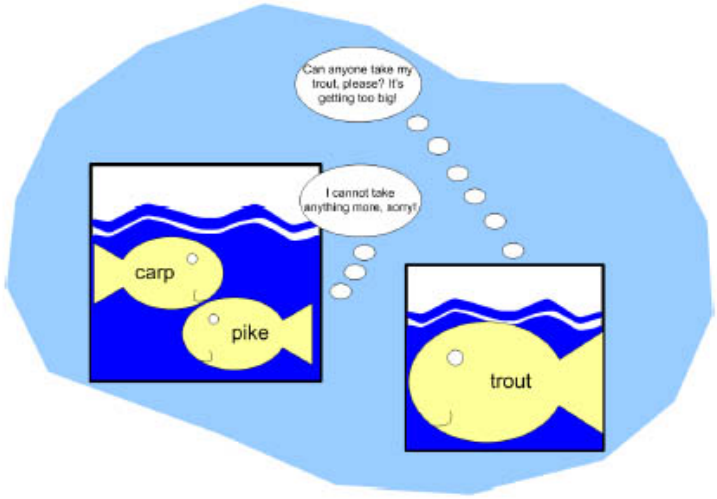


Figure 2: Sea of movable objects

Fortunately a much bigger basin joins these two basins which is only housing a small salmon (figure3).

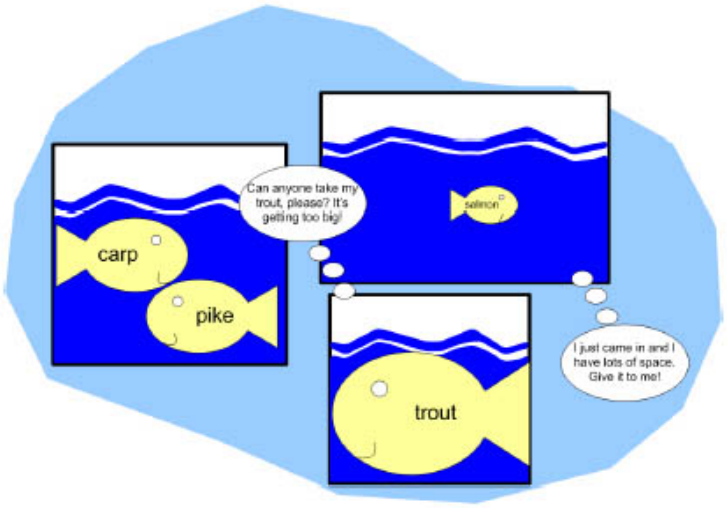


Figure 3: Sea of movable objects

Because the basin running out of space asked for it, the bigger basin takes the trout and there is enough space for every fish again (figure 4). In a more precise way the following scenario can be thought of: a user wants to go shopping using the concubi*Net* of the store. Therefore he takes his PDA with him/her, on which his profile, containing personal information such as age, gender, favorite interests and so on, are stored. When entering the store, the concubi*Net* of the store recognizes his/her PDA and fetches the profile in order to deliver current offers matching the personal favorites (e.g. home entertainment / DVD). It is possible for the user to choose whether to receive this kind of advertisements or not, of course.

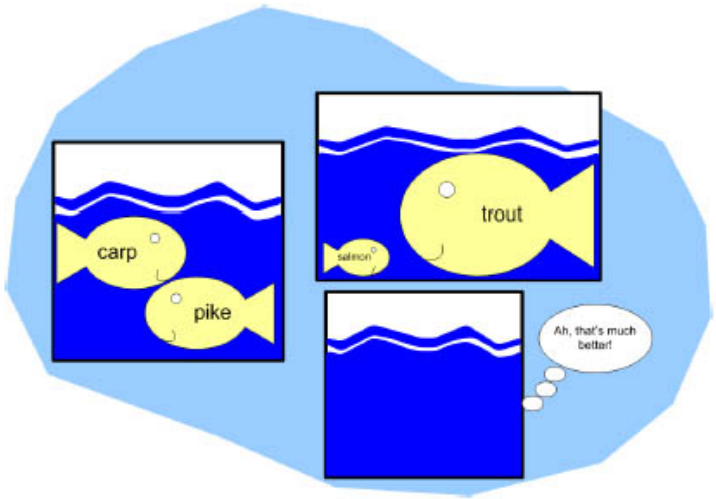


Figure 4: Sea of movable objects

1.3 Use-Cases

In order to give a better understanding of concubi*Net*, here are two scenarios that show its abilities:

1.3.1 Roaming profile

Usually, people who work in office buildings have their own office with own Desktop-PC. The idea of Roaming Profile using concubi*Net* is that each clerk can use every terminal in the building. As soon as he sits in front of a PC his complete desktop with all his data is transferred from the computer he last has used to that one he now wants to use. Running applications don't have to be closed.

Ideally, objects move to the new PC just before the user wants to use them. Therefore concubi*Net* could be expanded with location awareness, for example using RFID-Tags (radio frequency identification). Now, that concubi*Net* knows the position of each user, objects are able to follow their owners.

Imagine that each clerk uses his own P.I.M. (personal information manager). As he is wearing his RFID-Tag, concubi*Net* always knows his exact position and the computers close to him. If he now moves in front of a PC or any other device, immediately his P.I.M. pops up and he can check for appointments, receive or send e-mails, look for phone numbers or insert new notes. And if he moves away from this device, his P.I.M. follows him.

This simplifies the use of personal data in a network. The user doesn't have to pay any attention on carrying his data with him or synchronizing the data between two ore more devices, like he has to do today. This time-consuming operation is not required. The problem of inconsistent data does not exist any more.

1.3.2 Escape of objects

Another scenario, in which concubi*Net* can show its abilities, is when some devices run out of power or in some cases of an emergency, such as fire or an earthquake.

In a building with a UPS (uninterruptible power supply), concubi*Net* can be used to to save objects and applications, even if the UPS runs out of power.

In case of an emergency, the endangered peer can be notified by a message from the UPS, the fire detection system or by simply pressing a button. A lot of other triggers are thinkable, e.g. intrusion detection or proclamation of an abnormal system end. Now, the system has a certain time to react. A UPS should hold enough electricity to keep a system alive for more than 30 minutes, whereas a fire is able to destruct a device in a few seconds. In some cases, for example if a system like Microsoft's Windows wants to shut down, concubi*Net* might be able to delay the shutdown for a period, which should not be too long.

To be prepared for the extreme case, concubi*Net* should be able to react as fast as possible. Therefore, every host explores its neighborhood at frequent intervals and prepares some escape routes, regarding the security level of routes and of other devices. It decides under inclusion of security aspects, kind of emergency, available time and other facts, whereto applications and other objects should be moved, which route the messages should take, whether the messages should be encrypted (e.g. if the messages have to leave a defined security level), sent unencrypted (e.g because of lack of time) or even if some objects better should be destroyed, because the required

security measures couldn't be guaranteed. A host sequences the objects, to ensure that more important objects have a higher chance to survive.

1.4 Realization

Concubi*Net* is implemented in the programming language Java SE Version 1.4.1 from Sun Microsystems. This choice has the advantage that the resulting software product basically is platform independent.

The fundamental communication platform for concubi*Net* is UPnP. To realize the communication with movable objects, each object gets a unique ID which is independent from the peer on which that object is currently running. The ID stays the same, if an object moves to another peer. To communicate with a certain object you only have to know its ID. You don't have to care on which peer that object is running.

2 Analysis of the given concubi*Net*

2.1 Fundamental terms

As you can see in figure 5 on page 12, the basic concubi*Net* system was designed to consist of four central components:

- PeerManager
- LUS
- LBS
- Wrapper

To help you understand this report, these components and some other terms need to be specified. For more information see [1]. To extend concubi*Net* with security features, we added two new components which also need to be explained:

- Security (Authentication, Integrity, ...)
- User Management

PeerManager The PeerManager is the core application in concubi*Net*. He manages all applications running on a host and provides basic communication facilities, e.g. inter-object-communication of CCN-objects on different hosts or the transfer of CCN-objects to other peers. The PeerManager is controlled by a graphical interface, described in section 6.3. Up to now the PeerManager was called 'Agent' and has now been renamed. Thats the reason why there is no PeerManager mentioned in [1].

LUS The Lookup-Service is a high level service of concubi*Net*. It collects and holds a list of all objects in concubi*Net* so that a search for an object simply is done by asking the Lookup-Service. To keep concubi*Net* independent from any local services, the LUS is optional. All peers are also able to start a distributed search, which works without the LUS.

LBS The concubi*Net* Load Balancing Service is planned to be a high-level service of concubi*Net*. Its major task is in guaranteeing the stability and operational capability of concubi*Net* by predicting and preventing so-called Resource Bottlenecks. This goal will be achieved by distributing load equally over the peers in concubi*Net*. For distribution and load-transfer the standard transfer methods for CCN-objects will be used.

Wrapper The Wrapper is the proxy for communication between applications. It's main function is to decide whether a CCN-object can be called by reference or if a remote call has to be started in order to invoke the correct method of the CCN-object located on another peer.

conf.xml The file conf.xml is used to configure a participating peer. This file is read by the PeerManager when he starts. Concubi*Net* provides a feature called hot-config that allows the modification of a peer's configuration during runtime. This is done by periodically reading in the conf.xml.

Security Concubi*Net* is able to encrypt objects before they are moved to another peer. The user can choose between symmetric and asymmetric encryption with different algorithms. These algorithms also can be used to guarantee integrity and authentication.

User Management We provided a user management which allows to create users and limit their permissions, e.g. to write or read from the hard disk or to transfer objects. Each user gets a login name and a password. New users can login with a guest account and operate with very limited permissions.

2.2 Architecture

2.2.1 Conventions

Because of many ways a layer model can be interpreted we have to define an own standard. There are two basic elements used in figure 5. The first one is the alignment. Every block can only communicate with its horizontal and vertical neighbors. The second type of connection is indicated by hatching. The basic hard- and software of the host on which the concubi*Net* software is ought to run is hatched with thick dot-lines from bottom left to upper right. The concubi*Net* middleware is hatched by the triple line from bottom right to upper left and the applications running concubi*Net* programs or using them are single lined from bottom left to upper right. Now that the basic semantic of the layer model is explained we can go ahead by analyzing it.

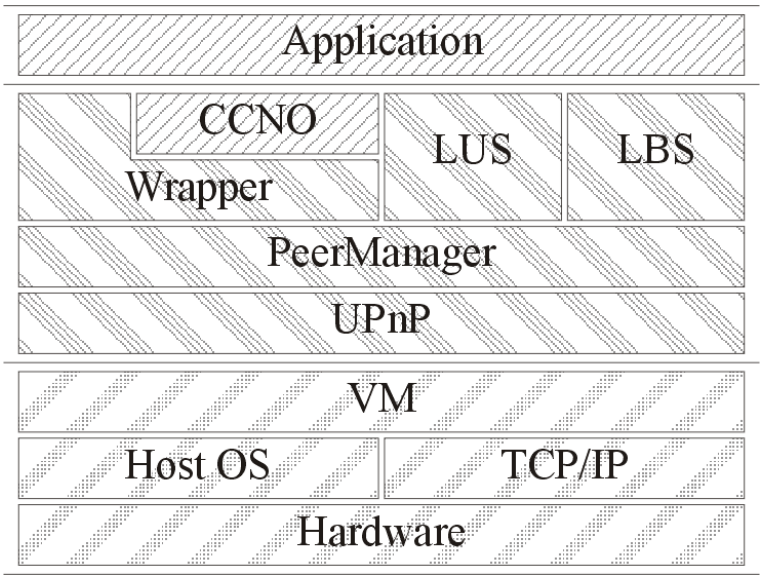


Figure 5: Layer Model of concubi*Net*

2.2.2 Analysis

Hardware Layer There is not much to say in this subsection. The concubi*Net* middleware requires a Java Virtual Machine. Because of being middleware for ubiquitous computing concubi*Net* has to be run on PC's as well as on mobile computers and PDA's. So the J2 Micro Edition is a part of the Java Virtual Machine layer in

our model. Unfortunately the KVM, a small Java Virtual Machine for limited devices, does not fulfill the requirements because of missing the foundation for downloading and execution of dynamic content and services. For the current JXTA implementation a TCP/IP stack is needed, but concubi*Net* only requires the JXTA interfaces. So if a future JXTA implementation will not be using TCP/IP, concubi*Net* would not need TCP/IP.

Middleware - concubi*Net* - and Application Layer The basis of concubi*Net* is built of the so called PeerManager. All the communication between devices is bundled over the PeerManager and PeerManager themselves use the JXTA protocol for communication. In the picture of our "sea" of movable objects the PeerManager takes the place of the basins. Every hardware device which wants to play an active role in concubi*Net* has to run a PeerManager.

Certainly the LUS needs to talk to all PeerManager in order to find the CCN-objects distributed on the different PeerManager. The LBS needs to talk to the PeerManager because it has to be able to talk to other LBSes on other devices and - like all communication between devices - also these are bundled over the PeerManager.

Wrappers are in connection with the PeerManager e.g. when on one machine there is a wrapper for a CCN-object running on another machine, again the communication is bundled over the PeerManager. A wrapper has connections to a CCN-object. The wrapper is surrounding the CCN-object indicating that the communication with it is only possible by accessing it by its wrapper. The wrapper can also be found in our "sea" of movable objects as the squared fish surrounding the real fish and acting as the communication object. The first vague idea of a "sea" of movable objects with fish swimming in basins is still represented in our layer model.

One could ask now, why the CCN-object is single lined and thus associated with the application layer. The reason is that the application developer certainly has to implement his own CCN-objects. In contrast to that the wrappers for the CCN-objects are exactly defined and and from now on automatically generated by a tool to avoid mistakes by the application developer. PG441 realized this so called Wrapper factory. For a detailed description see section 3.9.1 and following. The highest layer is the application layer. A concubi*Net* application consists of a number of CCN-objects.

2.3 Basic optimization approaches

2.3.1 Exceptions

First of all we were in charge of categorizing and visualizing the exceptions which were given by the program. That means that we have a graphical user interface which gives us information about the issued exceptions. Furthermore the remote exceptions got programmed in that way, so that they can be displayed on the computer on which the CCN-object is located which got activated by the exceptions. By visualization and cataloging of the exceptions it was easier to localize them.

2.3.2 Program code

Afterwards we tried to make the program code more concise by deleting unnecessary code or we divided big classes into several classes. For example the GUI got divided into a separate class so that it was not a part of the PeerManager class anymore.

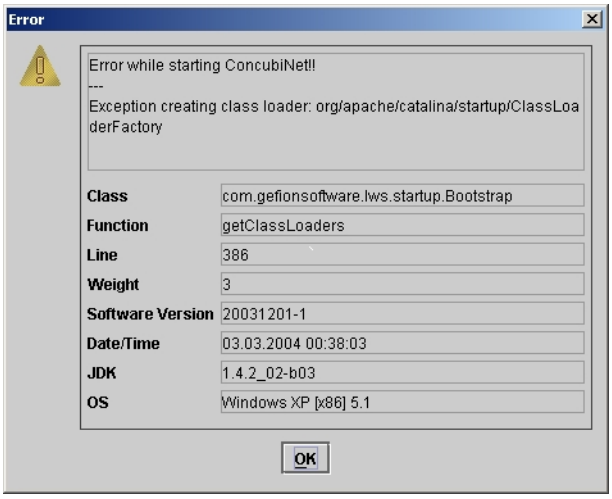


Figure 6: Exception-GUI

Because of that the program code got more concise. Also we chucked out libraries which were not used.

2.3.3 Performance

Then we had the problem that the performance of concubi*Net* was not very good. Because of that we had a new problem which was that we had to search for the reason of concubi*Net*'s slow run. Following that we started to occupy ourselves with JXTA and with the timeouts. Besides we recognized that concubi*Net* was not programmed by using Multithreading, so that the PeerManager was blocked during he accomplished an exercise. That means the PeerManager was not addressable, so that other PeerManagers had to wait for the complete timeout to communicate with the blocked PeerManager. This long waiting period was a reason why concubi*Net* became unnecessary slow. We solved this problem by reducing the timeouts without risking to get not all of the needed information because it is possible that the response of the distant peers might take longer. Even so it was not guaranteed that the PeerManager can communicate smoothly, because JXTA does not delete old advertisements so the PeerManager work with out-dated information. Because of that we implemented a list of PeerManagers, which holds information about in the net located PeerManagers. This list is updated automatically in the background, so that the list is always adjusted to the actual state of concubi*Net*. Nevertheless we decided to exchange JXTA with UPnP, because of the slow performance of JXTA. UPnP is a protocol which was made for Local Area Networks and which offers new useful services. But that also means we can use this version of concubi*Net* only for the LAN. Because of that two versions of concubi*Net* exists. The user can choose between concubi*Net* for the WWW by using JXTA or for the LAN by using UPnP. You can find a detailed description of the advancement of the performance of concubi*Net* at chapter 3.2.

2.3.4 Communication

Every principal in concubi*Net* can communicate with everyone else. That means a single object has a direct connection to every other object or PeerManager. So, if you

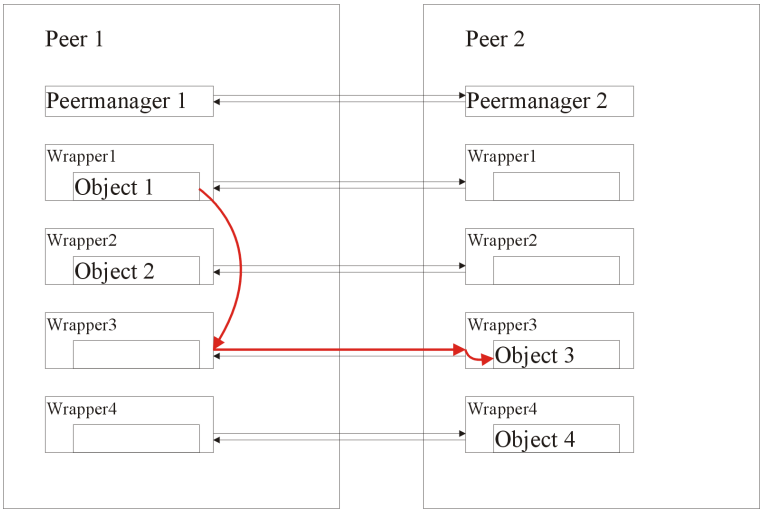


Figure 7: Communication of concubi*Net*

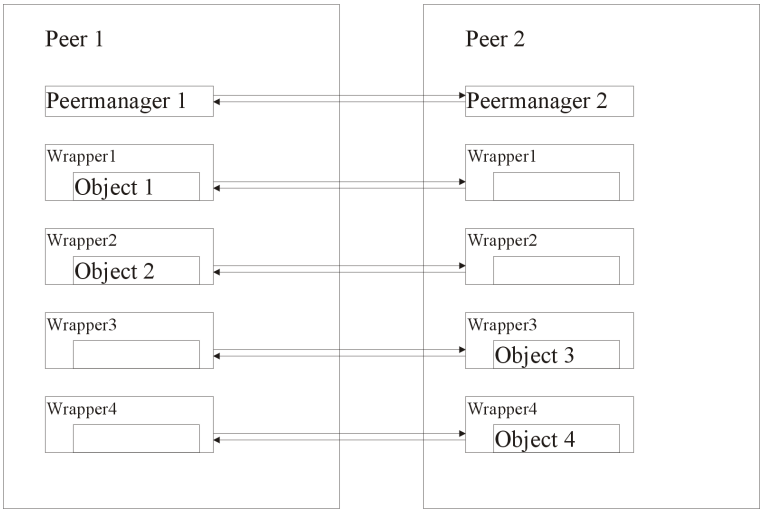


Figure 8: Connections of concubi*Net*

have a network with a relatively high number of objects, you have a lot of open connections to provide the communication. In our point of view this is a big disadvantage because the bandwidth of our network gets a lot of traffic and becomes overloaded. You can see this chaos in the figures 7 and 8. If a user wants to communicate with an object, the first thing the PeerManager does is to try if the object is local. If it is so, the wrapper of the wanted object signalizes that it is filled with the object. Now the user can communicate with the local object by the local wrapper. If the object is not local, the wrapper signalizes that it is not filled, but knows where the object should be. Now the user communicates with the remote object by the local wrapper with an own pipe especially opened for this purposes. So you can see that this way of communication has a big disadvantage: Every new communication attempt opens a new pipe. After a while we have lots of pipes which produce much overhead as you can see in figure (8).

Because of that we decided to change the communication architecture of concubi*Net*. Now the CCN-objects use the pipe of their PeerManager to communicate. That means that the new architecture does not allow a direct communication between the objects. Certainly the wrappers are very important components, which guarantee a smooth communication.

2.4 Missing features

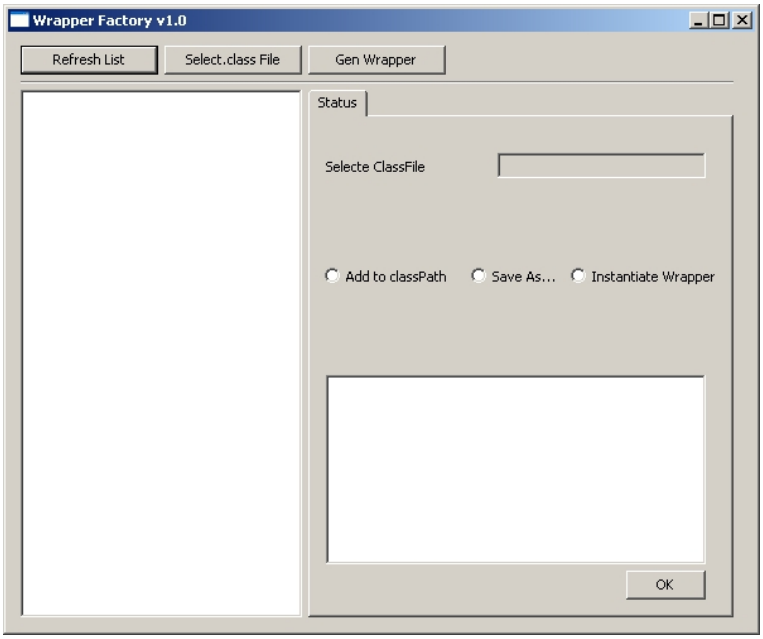


Figure 9: Wrapper factory

We decided that additionally to the SWING GUI we also should offer a SWT GUI to the user. We also installed a server so that PeerManager could addresses out of the Internet. You can read more about this later.

What we already recognized at the beginning was that the CCN-objects got moved targetless from one point to another. That means that it was not considered if a CCN-peer was overloaded or not. Furthermore we did not know enough about profiles of the moved CCN-objects. To solve this problem we decided to adopt a Profiling-Tool and a Mapping service. More about that in chapter 4.

Another problem is the security of concubi*Net* which did not exist until now. You could move CCN-objects without any problem to a wished PeerManager. Because of that we had to think about a system so that we could make concubi*Net* secure. While we were doing this we did not forget that the security should not have a huge influence on the performance so that also small devices with less computing power could use concubi*Net*. We will implement signatures so that the CCN-objects will not be moved to other computers without a specific signature in the future. An useful feature of security is the User Management. This Management enabled to create different types of users like administrator, guest etc. For more information please read chapter 5.

Every CCN-object is enclosed by a Wrapper. This Wrapper has a local references to this object. If a CCN-object does not locally exist, the wrapper delegates the call

remotely. As a result of this concept any communication looks like local communication. By using this kind of communication the user was always forced to program also a new Wrapper when he creates a new CCN-object. One of the new features of concubi*Net* is the Wrapper-Factory, which creates automatically a Wrapper for a new CCN-object, so that the user has not to care about programming a new Wrapper. You can find more detailed information in chapter 3.5.

To show the efficiency of concubi*Net* we program an sample application called ccnPIM. This Personal Information Manager consists of two tabs, an address tab to administrate contacts and a note tab to store memos. For more information read chapter 6.9.

3 Optimization and enhancements

The architecture has almost remained the same. It is now based on UPnP but we still have a PeerManager to manage everything. Every CCN-object has still a wrapper to communicate. So the figure of the layer model did not change. Also the view of concubi*Net* like a basin full of fishes is the same. But a few nevertheless important changes were made.

3.1 Code optimization

Before we could start to integrate new features and enhancements, we had to analyze the existing system. For this purpose we analyzed the source code in terms of understanding it, finding errors, optimizing the whole system and finally to get a stable concubi*Net*.

In many sections of the concubi*Net* source code we found methods with over 50 lines of code. This is disadvantageous in several ways: such constructs considerable complicate the understanding of a function because it is difficult to overview the whole source code. Also the testing is even harder, as a long source code mostly contains more loops and case differentiations. We also noticed, that this long methods often contained pieces of code that repeated several times. Thus it seemed to be suggestive to split methods into functional units and to put repeating units into own procedures. This made it easier to test and optimize the system.

Equally to long procedures, there were classes with up to 2000 lines of source code. An example is the central class org.concubinet.agent.Agent. Here it also made sense to create functional units and to put them into new classes. Thereby, the class 'JxtaConfig', which holds all JXTA-relevant procedures, as well as 'AgentConfigurator', which is used to configure the PeerManager, were generated.

The storage of same information in different variables in different places of the program is less desirable, as the effort to keep all variables up to date is unnecessary. It also creates the risk that variables, that ought to contain the same value, might be unequal at runtime. Here it is better to store all variables just once an to refer from other parts of the program. In several places, such variables have been replaced by references.

Expensive loops and iterations at different places of the source code could be replaced by suitable data structures. An example therefor is the search for a wrapper in a local list with an ID. It is obviously clear that iterating a vector and comparing the ID with the ID of the wrapper is much slower than accessing a hash table that uses the ID as a key and stores the wrapper as value.

3.2 Performance optimization

Performance of a system is defined by its capability and particularly the speed data are processed. In concubi*Net* speed could be increased in two ways:

- by decreasing the need of resources on the host concubi*Net* is running
- by optimizing the duration of internal processes

3.2.1 Optimization of resource requirement

To decrease the requirement of resources, a new graphical user interface has been created, using SWT-Libraries. SWT-Libraries have the advantage that they are able to revert to platform specific libraries, whereas Swing-libraries strain the available resources much more. As SWT-Libraries are available for many platforms (e.g. Windows, Windows-CE, Linux...), but not for all platforms, all relevant user interfaces have been implemented in SWT and in Swing. As the graphical user interface has been separated from the application layer, in concubi*Net* graphical user interfaces can be chosen or replaced arbitrarily, so that the platform independence finally isn't lost.

3.2.2 Optimization of processes

Besides the under 'code optimization' mentioned ways, which also serve the enhancement of performance, there were other ways to increase concubi*Net*'s efficiency.

On analyzing concubi*Net* we found out, that first of all JXTA influenced the systems speed and slowed down its performance in many aspects. So we decided to choose UPnP as an alternative for JXTA . The main disadvantage of JXTA is, that between the dispatch of a request and the collection of the responses has to be a certain idle time, to give distant JXTA-peers the chance to submit data. If you choose this time too short, you run the risk not to get all the needed information as the response of distant peers might take longer. If you choose a longer waiting period, the probability of getting all needed information increases, but therefor the program flow is delayed noticeably.

For many accesses on concubi*Net* it is enough, to revert to cached information. So, for example, a list of PeerManagers has been implemented, which holds information about in the net located PeerManagers. As very often in the program it is asked for the list of PeerManagers, this quick leads to a noticeable increase in efficiency. This list is updated in regular intervals automatically. The update is done in the background, so that the user does not notice it and the list always is adjusted to the actual state of concubi*Net*.

Outputpipes, which are used to transfer messages to distant peers, also get cached. That way, already established and tested pipes can be re-used. The creation of pipes is reduced to a minimum and only occurs if an outputpipe still hasn't been generated or an existing pipe is supposed to be faulty.

3.3 Communication enhancements

The existing communication concept of concubi*Net* seemed to be too unstructured and in many aspects unnecessary complex. Per PeerManager, thus for each Java Virtual Machine, there could be several active JXTA-instances with own JXTA-IDs, that could communicate independently with other JXTA-clients.

We expected a considerable performance enhancement from an optimization of JXTA and JXTA-related communication processes in concubi*Net*. To influence the communication in concubi*Net* we changed the communication processes to that effect, that not each CCN-object, each LUS, each class repository, ... owns an own JXTA-instance, but only the PeerManager so that he gets a central role in concubi*Net*'s communication. Thus only the PeerManager has the possibility to contact other PeerManagers. All other objects have to use the communication interface of the PeerManager if they want to communicate with other PeerManagers or with other objects.

3.3.1 callAgent

To simplify the communication interface within concubi*Net* for the developer, a central function has been implemented. This function is called 'callAgent' and is located in the PeerManager.

```
callAgent(String agentid, String methodToCall, Vector params)
```

The method 'callAgent' needs three parameters:

1. **agentid:** the ID of the PeerManager to communicate with
2. **methodToCall:** the method within the PeerManager to be called
3. **params:** a vector with parameter values according to the method to be called

Each class in concubi*Net* that now wants to communicate, being the PeerManager himself or another object, does this by the method 'callAgent'.

3.3.2 callWrapper

A special case in communication are the wrappers which possibly have to communicate directly to other wrappers. Therefore the according function 'callWrapper' has been implemented.

```
callWrapper(String Wrapperid, String methodToCall, Vector params)
```

This function is similar to the function 'callAgent' and differs only therein, that the ID of the CCN-object is specified and not the ID of the PeerManager. If the destination object is on a distant PeerManager a JXTA-connection gets automatically established that addresses the CCN-object.

3.3.3 Universal Plug and Play

As a big disadvantage of JXTA it turned out, that JXTA was over-strained with the dynamic changes of concubi*Net* during runtime. So, for example the discovering of other PeerManagers took up a relative long time.

An attempt to change the communication to UPnP and to do without JXTA seemed to show promise. UPnP has especially in finding other PeerManagers a big advantage, as UPnP-devices log on and off to the net when they start and terminate. Thus a dependable mechanism to find other PeerManagers was available.

For using UPnP within concubi*Net* the PeerManager on the one hand is an UPnP Control Point and on the other hand an UPnP Device. As an UPnP Device the PeerManager makes use of the UPnP's opportunity to assign to other UPnP Control Points which are PeerManagers within concubi*Net* as well. So when a PeerManager is started the other PeerManagers automatically will be informed about this process. At the same time the started PeerManager collects information about other running PeerManagers. Therefore they are able to send and receive UPnP messages from each other. Thus, three basic functions for communication within the concubi*Net* are met:

- A PeerManager can be found by other PeerManagers within the network.
- A PeerManager itself can find other PeerManagers.
- The PeerManagers are able to communicate with each other.

3.3.4 Conclusion

The simplification of communication processes was an obvious improvement to concubi*Net*. Thus the communication is more clearer to the developer and is replaceable easily by other technologies. Our experiences with UPnP showed good results and seem to promise a stable and dependable concubi*Net*.

3.4 Communication Architecture

As already described, communication within the concubi*Net* is now managed centrally via the local PeerManager. Therefore every PeerManager contains a communication module which establishes connections to the communication modules of other PeerManagers. The PeerManager uses the public interfaces of its own communication module, without any knowledge of the module's inner structure - for better comprehension the module can be understood as a black-box.

A main aspect for the development of the communication architecture has been the opportunity of interchanging the communication modules. Thus, different communication technologies can be used. So as an example for a local network it is a great advantage to use UPnP as a communication module whereas JXTA is the better choice for bigger networks where the opportunity to pass firewalls is needed.

Hereby development of new communication modules can be managed in the following way: At first, the main java-class of the new module has to be extended to the abstract class 'AbstractManager'. The abstract class automatically determines the functions that need to be implemented in the class of the new module. The following passage gives a short description of these functions:

- **getAgentID()** - returns the id of the local PeerManager .
- **getAgentList()** - returns a hash table of PeerManagers within the network. Hereby the key of the hash table is the id of the PeerManager and the value is its friendly name.
- **stop()** - is called when the PeerManager leaves the system. The network needs to be informed about this process.
- **getPeerName()** - return the friendly name of the local PeerManager .
- ***doQuery(String devID, String methodToCall, Vector params) throws Exception*** - central function within the communication module where a Remote Procedure Call to an other PeerManager is initiated. The 'devID' identifies the PeerManager to which a connection needs to be established. The 'methodToCall' and 'params' describe which function on the target PeerManager has to be called. Possibly a return value from the other PeerManager can be received. The function 'doQuery' will return this value or even 'null'.
- **registerCCNObject(String ccnid, String classname**) - when a CCNObject is created or even transmitted to a PeerManager it assigns via this function to network. Hence the network can make use of this information to find CCNObjects within the network.
- **deregisterCCNObject(String ccnid)** - informs the network that the object with the given id no longer exists.

- **getOwnerOfCCNObject(String ccnid) throws Exception** - establishes the particular PeerManager on which the object with the given ccnid is currently situated.
- **searchWrapperByClass(String classname)** - researches the network to find all CCNObject which class names are equal to the parameter 'classname'. A vector in which all IDs of ccnobject are listed is returned.

After a communication module is implemented only little change to the system is needed. Therefore the configuration file has to be edited and the name of the new module's class has to be inserted at place of the identifier 'ccnCommunicationmodul'.

Example: *<ccnCommunicationmodul> org.concubinet.core.net.upnp.UPNPManager </ccnCommunicationmodul>*

3.4.1 Communication review

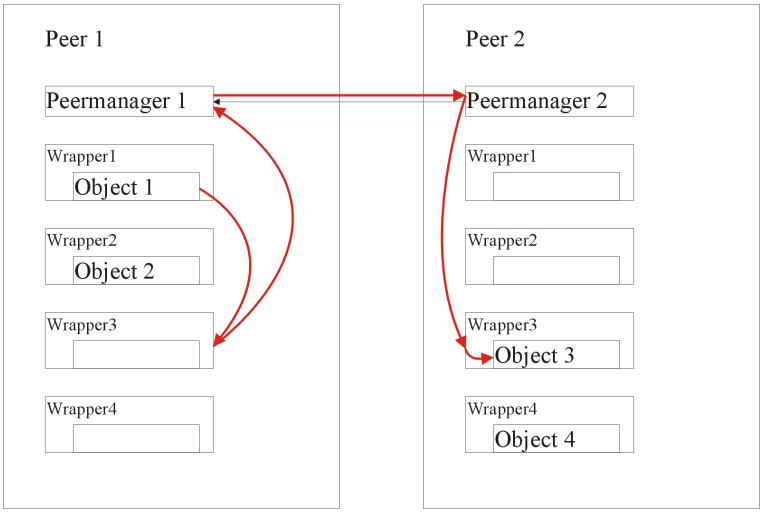


Figure 10: Communication of concubi*Net*

The point of communication is now the PeerManager. Every attempt to communicate has to pass the PeerManager first. That does not mean that objects cannot communicate directly with each other. It only means every CCN-object uses the pipe of its PeerManager to get a connection with an other object. So we have a relief of the network. Every PeerManager has a pipe to its neighbor PeerManager and to its own CCN-object. See the figures 10 and 11 which visualize the change. It should be compared with the both in the 'Architecture' section. If a user wants to communicate with an object, the PeerManager first tries if the object is local. If it is so, the wrapper of the wanted object signalizes that it is filled with the object. Now the user can communicate with the local object by the local wrapper. If the object is not local, the wrapper is not filled, but knows where the object has to be. Now the user communicates with the remote object by the local wrapper with the PeerManager as a mediator.

3.5 Transfer of CCN-objects

The transfer is nothing more than a remote call of a method, that takes the serialized object in form of a byte array by its parameters.

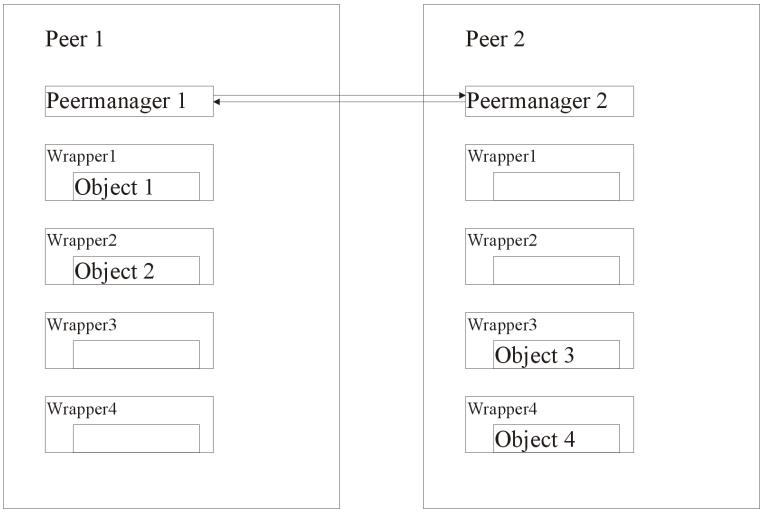


Figure 11: Connections of concubi*Net*

First `transferObject(...)` on the originator's site is called. It does the following:

1. Pack the CCN-object. Therefore the CCN-object's pipe is closed so that no more method-calls can be done, the CCN-object is sent to sleep (see `CCNObject.goSleep()`) and serialized.
2. The last step is the remote call of takeObject(...).

   Several failure handlings follow. These are necessary to guarantee, that the CCN-object cannot be lost. Either the CCN-object is transferred successfully and the local wrapper remains empty or the transfer fails (perhaps because the remote PeerManager is no longer available) and the CCN-object remains on the originator.

The called method is `Agent.takeObject(...)`. On the receiver side this method works as follows:

1. A class loader (repository) is created, that searches for class definitions in the own repository and in the originator's repository. For details see section 3.6.
2. Search for an already existing, empty wrapper for the transferred CCN-object by its CCNID. If no existing wrapper is available, it has to be created. To find the matching class, the former created class loader is used. The fully qualified classname of the wrapper is passed to `takeObject` by parameter.
3. Now the serialized object has to be implanted into the wrapper. This is done by the method `Wrapper.wakeUp(byte[] serializedObject, ClassRepository cr)`. The class repository has to be passed so that missing classes can be loaded during the deserialization by using the correct class loader.

   If everything succeeded the method returns true to tell the originator that the transfer is complete.

3.6 Class loader concept

Motivation We faced the following problem: Imagine a CCN-object has to be transferred to a PeerManager that does not know the corresponding class definitions (Java bytecodes) yet. In this case it is not possible to deserialize the object. So we had to find a way to get the correct bytecodes, load them on the recipient's VM and use them during deserialization.

Solution Possibly the easiest way to find the correct bytecodes is to ask the originator, because if he has the object he obviously must have the bytecodes too. To get the classes from the originator we created the class repository. The class repository performs two tasks: On the one hand it is a common class loader, on the other hand it is able to deliver the bytecodes of the classes.

On a PeerManager we have a so called ClassRepositoryFactory (classname is `org.concubinet.agent.classrepository.ClassRepFactory`), serving two methods:

1. `ClassRepository getLocalClassRepository()` - Returns a class loader, that searches for class definitions only on the local machine
2. `ClassRepository getClassRepository(CCNID)` - Returns a class loader, that searches for class definitions on the local machine and further on it is able to ask a remote PeerManager for bytecodes, defined by the passed CCNID

The process of byte code-transfer can be described as follows:

1. The originator of a CCN-object passes the recipient the CCNID of his local classrepository
2. The recipient asks his ClassRepFactory for the matching class loader
3. The deserialization method (`Wrapper.wakeUp(byte[] serObj, ClassLoader cl)`) uses the class loader just created by using an inheritor of the `java.io.ObjectInputStream`: `ObjectInputStreamForContext` (in package `org.concubinet.agent.classrepository`). In this class the protected method `resolveClass()` was overwritten so that it now uses our classrepository instead of the system class loader. By this technique inner classes and unknown parameter types are detected and loaded from the remote classrepository as well.

Abstract Factory design pattern The classrepository implements the Abstract Factory design pattern. The abstract factory is the class `org.concubinet.agent.classrepository.ClassRepFactory`, the abstract product is `org.concubinet.agent.classrepository.ClassRepository`.

The concrete factory we implemented is `org.concubinet.agent.classrepository.FileClassRepFactory`, the corresponding produced object is `org.concubinet.agent.classrepository.FileClassRepository`.

FileClassRepository First of all the FileClassRepository is a class loader and therefore has to implement the method `Class findClass(String)`, that is used by the class loader delegate concept of the Java 2 Platform. The method will be called by the inherited method `loadClass(String)` and encapsulates the local and remote search for class definitions by using the method `getBytecodeArray(String classname)`, that delivers the Java byte code of the class defined by the parameter.

Method `getBytecodeArray(String classname)` in detail does the following:

1. First it searches for the byte code in the local repository by calling `getBytecodeInputStream(String classname)`. If this method succeeds, a byte array (`byte[]`) is created and returned.
2. If the method throws a `BytecodeNotAvailableException`, it is tried to search for the byte code on a remote repository if it is set (This only takes place if the current FileClassRepository was created by a call of `ClassRepFactory.getClassRepository(CCNID remoteRep)`).
3. First step in remote search is to ask the remote repository, if there is a JAR available containing the class. This is made because the probability is high, that other classes, also contained in the JAR, will be needed in the near future and will then already be on the local machine.
4. If a JAR is available, it is transferred by the RPC of `getJarByteArray(String classname)` and stored in the local repository. Last step is to get the byte code out of the JAR and return it. If no JAR is available, the byte code of the class is transferred by the RPC of `getBytecodeArray(String classname)`, stored in the local repository and returned.

3.7 Multithreading

Every modern operating system tries to implement stability and performance as two primary goals. The crucial idea by the realization of the performance aspect is, to parallelize the execution of the processes. The suitable key word is multi-threading. The trick is to switch very fast between different processes, so that the user gets the feeling, all programs are running at the same time. From another point of view, a CCN object that is running, can be considered as a single process. The looking at the CCN from this perspective, brought us to the idea to implement the multi-threading function into CCN. Each locally stored object should be executed inside of a thread, thereby we hope to increase the overall achievement of the CCN.

Java gives us the opportunity to use the threads in a simple manner. The classes java.lang.Thread and java.lang.ThreadGroup defined in java API, are the central classes for our goal, to make the CCN multitaskable. Besides that, our local security architecture bases on the threads, and we use it to recognize the owner of a CCN Object, for further security decisions. In the following, we will give a detailed description of the multi-threading realization in CCN, so you can later more easy understand, how the local security does work.

Every machine has the set of locally known users. Whenever an CCN objet get moved to a machine, the locally installed Peer Manager finds out its owner and looks up, if the owner is already registered or not. The PeerManager creates for every user exactly one UserThread. The checking of the local familiarity with an user and the construction of his UserThread is the task of the UserThreadManager class contained in the local security package of CNN. UserThread is another class of this package. The problem is, that an user can locally posses several CCN objects. Creating only one thread per user would mean, that only one object of that user can be executed per PeerManager. Of course, this condition is not acceptable. Therefore we created another class called UserWorkThread, also saved in the same package as mentioned above in this chapter. The task of this class is to run a single object of an user: in other words, every object gets started in an instance of the UserWorkThread class.

Because of that, we demote UserThread class only for user-recognization purpose. We will describe it in the following of this chapter. The execution of an object occurs inside of an UserWorkThread class. The classes UserThread and UserWorkThread are the subclasses of the Java Thread class. Threads can be grouped into ThreadGroup (a Java class, mentioned at the beginning of this chapter). At the moment of creating of a new Thread, an instance of the ThreadGroup class can be passed as the constructor parameter. Giving the same ThreadGroup instance to different Thread objects, makes them belong to the same group. On the other side, if a thread creates a new thread, the child Thread inherits the ThreadGroup of the parent Thread. This characteristic makes it possible to identify the owner of every object and all its sub objects. In detail: An object X should be moved from PeerManager A to PeerManager B. The UserThreadManager instance of B, determines the owner of X by reading out its meta informations (we assume that the owners name is Otto). UserThreadManager looks up whether Otto is locally known or not. If not a UserThread for Otto will be created, and a new ThreadGroup named Otto will be created and passed to the Thread constructor. At the same time, an association between this User and his UserThread will be created for internal purpose. UserThread itselfs, creates a new UserWorkThread for X and lets it execute inside. If an UserThread instance for Otto already exists, then it will be used again, and this existing UserThread creates itselfs a new UserWorkThread for X. It means that for each locally know user, exactly one ThreadGroup and one UserThread will be created, both named with the name of the user.

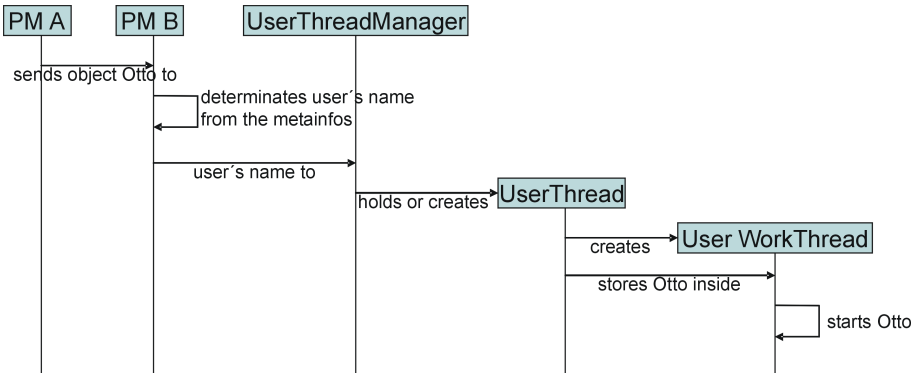


Figure 12: Multithreading

3.8 MetaInfo

3.8.1 PeerManagerInfo

First we used DOM to work with XML data files, for example for conf.xml. DOM was created by the W3C as a platform and language independent interface. That means, that DOM is a special interface for different programming languages to work with XML data files, for example W3C offers the package *org.w3c.dom* to manipulate or generate XML data files. DOM is adapt to the different programming languages, so that is in fact language independent, but also inefficient, because DOM does not use the strengths of the different programming languages. Thatt's why DOM has not a very well performance, but high memory requirements. JDOM is an open-source project, which is specially matched to Java. For example JDOM does not use the data structure of DOM, but of Java. For this reason we use JDOM to program the PeerMan-

gerInfo. This class consists of two methods, which allow us to read out information from every XML data file or to write new information in a XML data file. The method **readXML** consists of four parameters. First the user has to declare the destination of the XML data file, then the Name of the over element, in which the needed information is located. In a XML data file can appear several over elements from the same type. With the third parameter *Number* the user can declare, if we mean the first, second or etc. over element. The last parameter permitted us to access the needed information. By using the method **setXML** the user is able to write new information in a XML data file. With the first four parameters you can choose the information, which should be overwritten and with the last parameter you can replace the old information.

3.8.2 ObjectInfo

For this package we used DOM to work with XML data files. For further information on DOM see the chapter above. This is the reason why we stopped developing ObjectInfo with DOM. It remains in the package structure for review purposes only. For the functional XML interface see PeerManagerInfo. This ObjectInfo interface is not very dynamic and not compatible to different XML structures. We have a simple interface method called getProperty(String) with a parameter for example "SecurityLevel". The method will return a hash table containing all SecurityLevel parameters. But it is only compatible with this single XML file ObjectInfo.xml. So the use of the PeerManagerInfo interface is recommended.

3.9 Wrapperfactory

3.9.1 Wrapper-Factory - Introduction

Every wrapper encapsulates a CCN-object and prevents any other CCN-objects to have a local reference on it. If a reference does not locally exist, the wrapper delegates the call remotely. As a Result of this concept any communication looks like local communication. So the programmer can write his code without any difference from writing code for only local use.

How wrappers work Now let me explain what wrappers do. Every wrapper-class extends the class wrapper. This class holds an attribute as an reference to the CCN-object encapsulated in the wrapper-class. By moving the CCN-object to an other VM this reference becomes *null*. So we can specify if either the object is *local* or only *remotely* callable. The code which ensures this task is:

```
if (this.isFilled())
{
        //loacale call
        ...
}
else
{
        //remote call
        ...
}
```

How that works from a detailed, Java based, view should not be explained here. The important thing is that this code block is included in a counterpart wrapper method to

a public method in the CCN-object. So the wrapper fulfills his task to add either local or remote communication availabilities to any given CCN-object.

How wrappers were implemented so far The goal of providing transparency for the programmer in the way he writes code for remote communication was not supported completely, cause the writing of wrappers for every CCN-object is further an unpleasant task for him.
The goal of the Wrapper-Factory will be to switch off this overhead and provide concubi*Net* with features of creating new wrappers at design or runtime, and automatically fulfilling this task. The Wrapper-Factory creates new Wrappers for you at Design time by manually requesting it, or automatically when you wish to start an CCNObjects for which no wrapper exists at the moment.

3.9.2 Wrapper-Factory

The Wrapper-Factory is a factory which creates and provides a wrapper for any given CCN-object. At the same time the Wrapper-Factory is integrated in the PeerManager as a fundamental part.

LifeCycle of the Wrapper-Factory The functional structure of the Wrapper-Factory can be partitioned in 3 Levels.

1. Level: Analysis

   The first level analyzes the given CCN-object, for which a wrapper should be generated. All public methods of the object class are extracted using reflection. The method name, the return type and method parameters are the most important values. The particular implementation of the methods is irrelevant here, because a wrapper does, as explained before, only the delegation and not the implementation of Methods.

2. Level: Source-Generation

   In this level the source code will be generated, based on the results of level 1 on the one hand (creating of public methods for the main task of communication), and the generation of standard components on the other hand. These standards are:

   - Import statements
   - Standard constructors

3. Level: Compilation

   The compilation of the generated source code will be done in this last phase. As a result of compilation different types can be ordered as return values. They are:

   - Class-Object of the generated Wrapper-class
   - File Object of the generated Wrapper-class
   - byte-Array of the generated Wrapper-class

   In any case the generated .class-File will be added to the right place at the local directory and to the classpath, so it can be loaded without any problems.

How the Wrapper-Factory generates code The generation of code can also be partitioned in three phases.

1. Phase: Generation of package and import statements

   This phase is trivial. The package name of the given CCN-object will be extracted and added to the source file. The import statements are always the same so they can also be added directly.

2. Phase: Generation of the class header and standard constructors

   This phase is also trivial. The string "Wrapper extends Wrapper" is added to the extracted classname. The next step is adding the standard constructors. Here, the custom classname has to be considered, the rest is standard.

3. Phase: Generation of custom public methods for communication

   For every public method in the custom CCN-object a pendant in the wrapper will be generated. For this task the Wrapper-Factory uses reflection. The Wrapper-Factory is so programmed that only methods of the custom class were considered. Not methods, for example, of extended classes.

Class-Hirarchy All Classes of the Wrapper-Factory are in the package:

```
org.concubinet.wf;
org.concubinet.wf.exception;
```

The main class is:

```
org.concubinet.wf.WrapperFactory;
```

Any class of the Wrapper-Factory starts here in the method *createWrapper*. Additionally there are 3 classes which are used from the Wrapper-Factory to generate the source code:

```
org.concubinet.wf.WrapperClass;
org.concubinet.wf.WrapperConstructors;
org.concubinet.wf.WrapperMethod;
```

WrapperClass is a representation of the generated source. It includes all the needed attributes, such as a vector for the methods, the class and package name as string, the imports and so on. The class *WrapperConstructors* gets an PrintWriter and the classname as arguments. With these it generates the standard constructors to a given printstream. The printstream is used to write code into the .java-file. Finally the class WrapperMethod is the representation of any public method. This class can generate custom "Methodcode" for any possible public method in the CCNObject-class.

```
org.concubinet.wf.Compiler;
```

Finally we need a compiler to compile the generated source code. This task fulfills the compiler. It works as explained in the subsection "LifeCycle of the Wrapper-Factory", Phase 3.

3.9.3 Wrapper-Factory and Concubi*Net*

Now it will be explained, how the Wrapper-Factory is integrated in the Concubi*Net*.

Integration Objects can either be loaded from the conf.xml file or at runtime as a result of user-commands. In both cases the Wrapper-Factory takes action. That means: every attempt to load a CCN-object ends in a call to the Wrapper-Factory.

The Wrapper-Factory GUI The Wrapper-Factory GUI is the front end for the Wrapper-Factory. Here you can load CCN-objects by selecting their .class association. The .class files of course can be inside .jar-files. Additionally features are that the user can select which return-type he will get of the generated wrapper. They can be either a byte-array or a file-object which he can save in an user-specified directory.

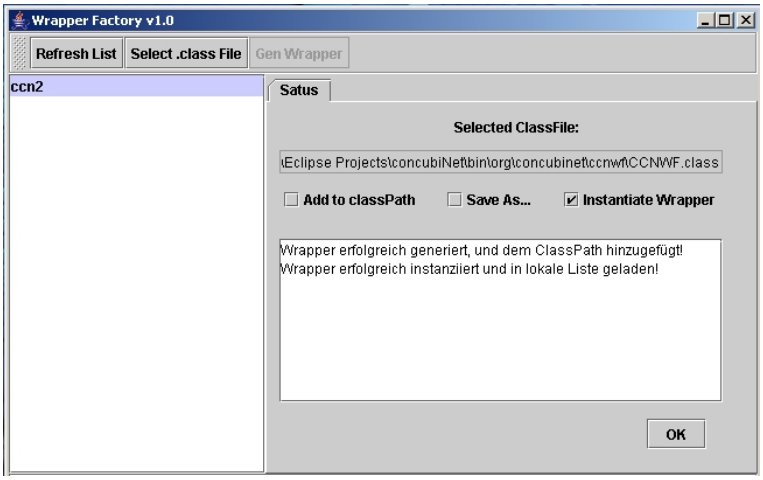


Figure 13: Die WF-GUI

The WF-GUI can **be** started from the XML-File, or by the Application Desktop.

3.10 (Graphical) User Interface

The advantages and disadvantages of SWT:

Advantages:

Compared to SWING SWT does not emulate the graphical user interface, but serves as an adapter for the respective operating system. That means that a "SWT Button" in Windows XP looks like a Windows XP Button and in Mac OS like a Mac OS Button. Concerning this the graphical user interface looks more professional. Because SWT serves as an adapter there is no difference with native applications. Due to that SWT is considerably faster than SWING. Furthermore it needs less resources.

Disadvantages:

For each popular operating system we need the SWT library. That means without the SWT library we cannot use the graphical user interfaces. The testing effort can be higher than in SWING because in SWT the graphical user interface can sometimes look different in other operating systems. In contrast to AWT SWT furthermore uses operating system resources to display colors and images which need to be explicit released if they are not required anymore. That is why SWT crashes down more than SWING.

Despite the disadvantages which we just mentioned we thought that the advantages over top the disadvantages and concerning we decided to program the GUI in SWT. By doing this we considered that the SWT GUI does not differ from the SWING GUI. That means that the user always can change from the SWT GUI to the SWING GUI presumed that the user has the essential library. What we recognized is that the SWING GUI did not get programmed without using the predefined layouts, so that each GUI element does not adjust to the window size. We avoided that in the new SWT GUI.

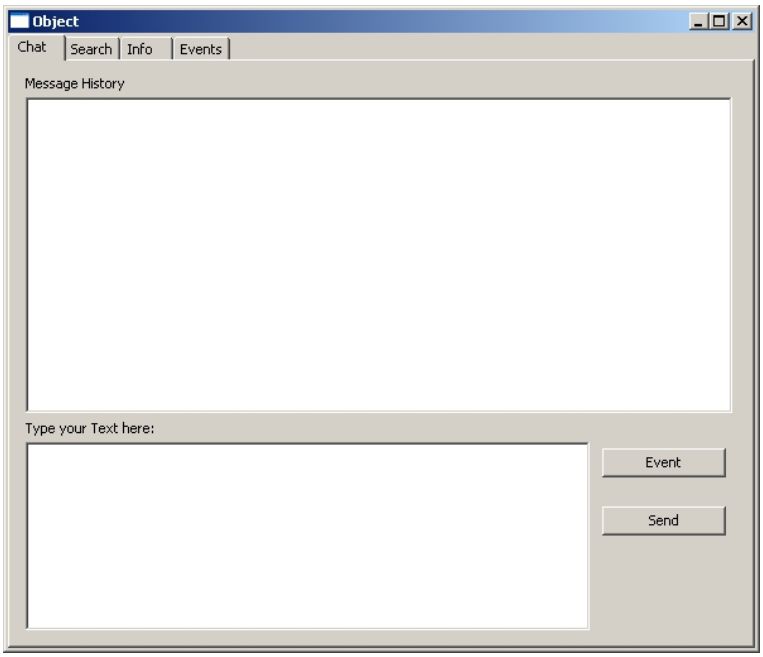


Figure 14: SWT GUI

3.11 Java WebStart

3.11.1 Java Web Feature

The Java Web Start software offers potential users of concubi*Net* the possibility to start the application from any computer connected to the Internet with just one mouse click.

3.11.2 What is Java Web Start?

The name Java Web Start as the revolutionary technology is misleading. In actuality Java Web Start is a reference implementation of the (Java Network Launching Protocol and API) JNLP specification. JNLP specification was developed and published by Sun as a result of the JSR 56. JNLP can be defined as a protocol that enables Java clients to deploy themselves on the client and run as if they were local applications. However for the purposes of introducing the Web Start functionality as a new feature of the concubi*Net* we shall discuss Java Web Start. For more information and complete JNLP API you may refer to http://java.sun.com/products/javawebstart/

How is this different from what Java introduced earlier as applets? Java applets were what Sun promised the world as a revolution. But after the initial interest of seeing animated pictures on the Internet they fell out of favor for their bulky size and slow execution. Users generally are not patient enough to wait for an applet to load from the network, check security permissions and then see the information. If the applet happens to be in Swing the wait is doubled. JNLP has overcome many of the difficulties of applets.

Java Web Start is a JNLP client that allows Java applications to be downloaded onto the client machine and runs them in the Java security Sandbox. So Java Web Start is a reference implementation provided by Sun to show the features of JNLP which allows a developer to create Java applications that can be run on a client without any installation procedures.

Java Web Start brings in all the security of the Java sandbox which means that you can run an application with the confidence that it will not over step and meddle with files on the client machine. It also brings the flexibility of using Java applications with Swing or AWT without the hassle of downloading bulky Swing jar files each time the application is run. Java Web Start is designed to download all the files required the first time it is invoked from a web page. After the first download the user has the option to include a shortcut to the application directly on the Windows start menu (option provided on Microsoft Windows only). Subsequent runs of the application can be done using the shortcut which would then run locally. The application can be designed to check for updates on the server to refresh the existing local copy with the changes made since the last access.

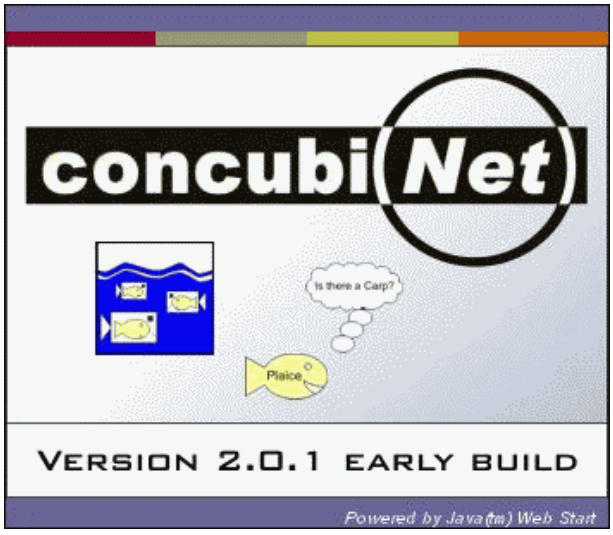


Figure 15: Java Web Start - CCN Splashscreen

3.11.3 Important features of Java Web Start

- It is a Web-based application architecture where applications can run locally using resources spread over the web.
- It provides for an installation free client application that can keep itself in sync with the updates on the server.

- It also provides for a facility to make incremental downloads and updates over a period of time.
- It offers the flexibility to run Java applications on different versions of JRE. The JRE can be can be downloaded if the version is not present on the machine.
- It offers a caching facility which can cache the application locally on the client which saves time the next time the application is run on the client.
- Applications can be run offline on the client machine after they are initially downloaded thereby decoupling them from the server.
- It offers a secure environment to execute applications like applets but it also provides flexibility within the API to do potentially unsecure actions with a warning to the user.
- The JNLP API and Java Web Start are now part of the J2SE 1.4 download.

3.11.4 JNLP Technology

JNLP is a XML based specification. The heart of JNLP technology is a JNLP file. It is a XML file that describes the different attributes used to describe the application. Shown below is the JNLP file used for the ccn.jnlp.

```
<?xml version="1.0" encoding="utf-8"?>

<jnlp
        spec="1.0+"
        codebase="http://www.concubinet.org/download/"
        href="ccn.jnlp">

  <information>

    <title>concubiNet2</title>
    <vendor>ein ubigitaeres Betriebssystem</vendor>
    <homepage href="http://www.concubiNet.org/" />

    <description>concubiNet2 - Die Rueckkehr</description>
    <description kind="short">concubiNet2</description>
    <description kind="tooltip">concubiNet2</description>

    <icon href="ccn32x32.gif"        width="32" height="32" />
    <icon href="ccn64x64.gif"        width="64" height="64" />
    <icon href="ccn-splash.jpg" kind="splash"           />
    <offline-allowed />

  </information>

  <security>
    <all-permissions />
  </security>

  <resources>
    <j2se version="1.4+" initial-heap-size="12m" max-heap-size="256m" />
    <jar href="ccn.jar" />
  </resources>

  <application-desc main-class="org.concubinet.core.PeerManager" />
</jnlp>
```

Figure 16: Diagram of the Java virtual machine process

The JNLP element is the root element that has a set of attributes that are used to specify information that is specific to the JNLP file. The information element describes meta-information about the application like title, description, vendor etc. The security element is used to request a trusted application environment, why applications need to be trusted will be discussed later in the security section.

The resources element specifies all of the resources that are part of the application, such as Java class files, native libraries, and system properties. The last part of the

JNLP file defines the kind of application. It could be one of the following four options: application-desc, applet-desc, component-desc, or installer-desc.

If an application-desc is defined in a JNLP file then it's an application descriptor. The application-desc element describes the application and the attributes required to invoke it.

The main-class is the Java class file which contains the main method that needs to be run. Any arguments that might be required may be passed in the argument element.

3.11.5 Updates and Caching

The JNLP application provides three different download protocols by which the application on the client can be made current. The first is the basic download protocol which downloads resources without any version information. The second is a version based download protocol which identifies all resources by a URL and version ID. In this case when the JNLP client starts up an application it sends the current version to the server as part of the request. If the server has a newer version it would download the newer version. The third protocol pertains to extensions where a URL or a URL and version ID can be specified to download the extension descriptor. If only a URL is specified the extension is downloaded using the basic download protocol. If a URL and a version is specified the version based download protocol is used with a few additional parameters. The extra parameters are used to identify the extension type and the platform for which it is needed.

JNLP also provides a facility of providing incremental updates. When the server finds that the client already has a version on the local machine and all it requires is a new version it sends an incremental upgrade instead of the whole application thereby reducing download time.

A JNLP client can cache the application to make it run faster in the subsequent runs. If an application is downloaded using the basic download protocol it would download the application without a version. When the application is downloaded a times tamp is downloaded on the client to keep track of updates. In the version based download protocol time stamps are not stored but the version would be part of the request.

When the request is made the JNLP client checks for the version already existing in the cache and the version in the request. If they match no download needs to be done.

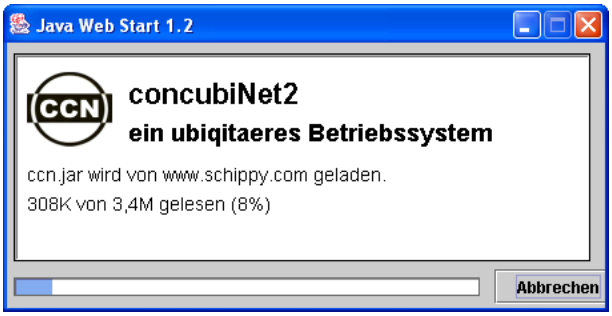


Figure 17: Downloading the appropriate components

3.12 Concubi*Net* and J2ME

Concubi*Net* designed as a ubiquitous operating system was also meant to support small devices like PDAs and smart phones. Thus in this final phase of our project group we wanted to review the possibility of porting the system on to the mobile edition of the Java Runtime Environment J2ME.
The implementations that originated from this process may be interpreted as feasibility studies rather than concrete, completely tested and applicable features of concubi*Net*.
To determine whether or not concubi*Net* will be realizable on a mobile Java edition, it is necessary to analyze the existing communication, trace down the underlying basic protocols and to find an appropriate implementation for the J2ME platform. Another basis of the concubi*Net* is the mobility of its objects which is primarily based on the serializable interface functionality provided by the standard Java Runtime Environment. Hence we need to investigate if realizations concerning this aspect exist as well.

3.12.1 Communication / Discovery

Although concubi*Net* doesn't utilize every aspect of UPnP (concubi*Net* only makes use of UPnP's very reliable discovery functionality - as mentioned earlier in this final report) there is still a heavy use of multicast messages. Though J2ME supports UDP, it doesnŠt support multicast and even while there are a few commercial workarounds (URL-1) which claim to run under Java's mobile edition, no open source forums attempt to add UPnP to J2ME.
Besides the use of multicast messages, concubi*Net* must be able to parse the XML description files which are vital to UPnP (concubi*Net* uses simpler XML descriptions due to its reduced UPnP capabilities). IBM published a web services API for J2ME quite recently (URL-2) which provides support for remote service invocation and XML parsing on a device level. Until then there were a few open source solutions on XML parsing with J2ME (URL-3).

3.12.2 Serialization / Object Transfer

A major drawback in our intention to port a fully functional concubi*Net* onto the J2ME is the lack of object serialization capabilities. Although an integration of serialization and RMI is demanded by the J2ME interest group (URL-4) for years it is very unlikely that this integration will be carried out in the near future.
However, a way of utilizing Java object serialization in a non standard environment is using a dedicated standalone VM (URL-5). These standalone virtual machines may be provided with different standard classes than the ones used by the KVM (URL-6). But unlike SunŠs KVM, only a very limited number of these dedicated virtual machines exist for more than one platform.
Reviewing all these facts we come to the conclusion, that by the time of this documents compilation there is no satisfying possibility of developing a J2ME implementation for the concubi*Net* containing all features of the standard J2SE version.

3.12.3 Possible Implementations

Nonetheless less extensive implementations are possible and reasonable. E.g. a concubi*Net* remote control application which uses a full fledged concubi*Net* proxy object to execute remote procedure calls. Or simply a concubi*Net* monitor that lists all available PeerManagers. Considering the proxy/substitute concepts, there may be

Figure 18: CCN on Palm OS 5

Figure 19: CCN monitor

numerous solutions to the problems mentioned above by using proprietary communication/transfer protocols between concubi*Net* and the J2ME implementation.

3.13 UPNP

3.13.1 What is UPnP?

UPnP is an architecture for pervasive peer-to-peer network connectivity of intelligent appliances, wireless devices and PCs of all form factors. It is designed to bring easy-to-use, flexible, standards-based connectivity to ad-hoc or unmanaged networks whether at home, in a small business, public spaces, or attached to the Internet. UPnP is a distributed, open networking architecture that leverages TCP/IP and the web technologies to enable seamless proximity networking in addition to control and data transfer among networked devices at home, office and public spaces.

UPnP is more than just a simple extension of the plug and play peripheral model. It is designed to support zero-configuration, "invisible" networking and automatic discovery for a breadth of device categories from a wide range of vendors. This means a device can dynamically join a network, obtain an IP address, convey its capabilities, and learn about the presence and capabilities of other devices. DHCP and DNS servers are optional and are used only if available on the network. Finally, a device can leave a network smoothly and automatically without leaving any unwanted state behind.

UPnP leverages Internet components, including IP, TCP, UDP, HTTP, and XML. Like the Internet, contracts are based on wire protocols that are declarative, expressed in XML and communicated via HTTP. IP internetworking is a strong choice for UPnP because of its proven ability to span different physical media, to enable real world multiple-vendor interoperation, and to achieve synergy with the Internet and many home and office intranets. UPnP has been explicitly designed to accommodate these environments. Further, via bridging, UPnP accommodates media running non-IP protocols when cost, technology or legacy prevents the media or devices attached to it from running IP.

What is "universal" about UPnP? No device drivers; common protocols are used instead. UPnP networking is media independent. UPnP devices can be implemented using any programming language and on any operating system. UPnP does not specify or constrain the design of an API for applications running on control points; OS vendors may create APIs that suit their customer’s needs. UPnP enables vendor control over device UI and interaction using the browser as well as conventional application programmatic control.

3.13.2 The UPnP Device Architecture

The UPnP Device Architecture (formerly known as the DCP Framework) contained herein defines the protocols for communication between controllers or control points and devices. For discovery, description, control, eventing and presentation, UPnP uses its own protocol stack.

At the highest layer, messages logically contain only UPnP vendor-specific information about their devices. Moving down the stack, vendor content is supplemented by information defined by UPnP Forum working committees. Messages from the layers above are hosted in UPnP-specific protocols, defined in this document. In turn, the

above messages are formatted using the Simple Service Discovery Protocol (SSDP), General Event Notification Architecture (GENA) and Simple Object Access Protocol (SOAP). The above messages are delivered via HTTP, either a multicast or unicast variety running over UDP or the standard HTTP running over TCP. Ultimately, all messages above are delivered over IP.

The foundation for UPnP networking is IP addressing. Each device must have a Dynamic Host Configuration Protocol (DHCP) client and search for a DHCP server when the device is first connected to the network. If a DHCP server is available, e.g. the network is managed, the device must use the IP addressed assigned to it. If no DHCP server is available, e.g. the network is unmanaged, the device must use Auto IP to get an address. In brief, Auto IP defines how a device intelligently chooses an IP address from a set of reserved addresses and is able to move easily between managed and unmanaged networks. If during the DHCP transaction, the device obtains a domain name, e.g., through a DNS server or via DNS forwarding, the device should use that name in subsequent network operations; otherwise, the device should use its IP address.

Given an IP address, Step 1 in UPnP networking is discovery. When a device is added to the network, the UPnP discovery protocol allows that device to advertise its services to control points on the network. Similarly, when a control point is added to the network, the UPnP discovery protocol allows that control point to search for devices of interest on the network. The fundamental exchange in both cases is a discovery message containing a few, essential specifics about the device or one of its services, e.g., its type, identifier and a pointer to more detailed information. The UPnP discovery protocol is based on the Simple Service Discovery Protocol (SSDP).

Step 2 in UPnP networking is description. After a control point has discovered a device, the control point still knows very little about the device. For the control point to learn more about the device and its capabilities, or to interact with the device, the control point must retrieve the device's description from the URL provided by the device in the discovery message. Devices may contain other, logical devices as well as functional units or services. The UPnP description for a device is expressed in XML and includes vendor-specific manufacturer information like the model name and number, serial number, manufacturer name, URLs to vendor-specific Web sites, etc. The description also includes a list of any embedded devices or services as well as URLs for control, eventing and presentation. For each service, the description includes a list of the commands or actions, the service responds to and parameters or arguments for each action; the description for a service also includes a list of variables; these variables model the state of the service at run time and are described in terms of their data type, range and event characteristics.

Step 3 in UPnP networking is control. After a control point has retrieved a description of the device, the control point can send actions to a device's service. To do this, a control point sends a suitable control message to the control URL for the service (provided in the device description). Control messages are also expressed in XML using the Simple Object Access Protocol (SOAP). Like function calls, in response to the control message, the service returns any action-specific values. The effects of the action (if any) are modeled by changes in the variables that describe the run-time state of the service.

Step 4 in UPnP networking is eventing. A UPnP description for a service includes a list of actions the service responds to and a list of variables that model the state of the service at run time. The service publishes updates when these variables change

and a control point may subscribe to receive this information. The service publishes updates by sending event messages. Event messages contain the names of one of more state variables and the current value of those variables. These messages are also expressed in XML and formatted using the General Event Notification Architecture (GENA). A special initial event message is sent when a control point first subscribes; this event message contains the names and values for all evented variables and allows the subscriber to initialize its model of the state of the service. To support scenarios with multiple control points, eventing is designed to keep all control points equally informed about the effects of any action. Therefore, all subscribers are sent all event messages, subscribers receive event messages for all evented variables that have changed and event messages are sent no matter why the state variable changed (either in response to a requested action or because the state the service is modeling changed).

Step 5 in UPnP networking is presentation. If a device has a URL for presentation, then the control point can retrieve a page from this URL, load the page into a browser and depending on the capabilities of the page, allow a user to control the device and/or view device status. The degree to which each of these can be accomplished depends on the specific capabilities of the presentation page and device.

3.14 UPNP Discovery within concubi*Net*

Discovery is Step 1 in UPnP networking. Discovery comes after addressing where devices get their unique UUID network address. Through discovery, PeerManagers find other PeerManagers.

Discovery is the first step in UPnP networking. When a device is added to the network, the UPnP discovery protocol allows that device to advertise itself on the network. Similarly, when a control point is added to the network, the UPnP discovery protocol allows that control point to search for devices of interest on the network. The fundamental exchange in both cases is a discovery message containing a few, essential specifics about the device.

When a new device is added to the network, it multicasts a number of discovery messages. Any interested control point can listen to the standard multicast address for notifications that new devices are available.

Similarly, when a new control point is added to the network, it multicasts a discovery message searching for interesting devices. All devices must listen to the standard multicast address for these messages and must respond since they meet the search criteria in the discovery message.

To reiterate, a control point may learn of a device of interest because that device sent discovery messages advertising itself or because the device responded to a discovery message searching for devices. In either case, if a control point is interested in a device and wants to learn more about it, the control point must use the information in the discovery message to send a description query message. Concubi*Net* makes very little use of the original and much more powerful features the UPnP descriptions have to offer. In concubi*Net* we simply use the description files to state the existence of UPnP-devices.

When a device is removed from the network, it should multicast a number of discovery messages revoking it's earlier announcements, effectively declaring that it will not be available any longer. To limit network congestion, the time-to-live (TTL) of each IP packet for each multicast message must default to 4 and should be configurable.

Discovery plays an important role in the interoperability of devices and control points using different versions of UPnP networking. The UPnP Device Architecture (defined herein) is versioned with both a major and a minor version, usually written as major.minor, where both major and minor are integers. Advances in minor versions must be a compatible superset of earlier minor versions of the same major version. Advances in major version are not required to be supersets of earlier versions and are not guaranteed to be backward compatible. Version information is communicated in discovery and description messages. In the former, each discovery message includes the version of UPnP networking that the device supports. As a backup, the latter also includes the same information. This section explains the format of version information in discovery messages and specific requirements on discovery messages to maintain compatibility with advances in minor versions.

The standard multicast address, as well as the mechanisms for advertising, searching, and revoking, are defined by the Simple Service Discovery Protocol (SSDP). The remainder of this section explains SSDP in detail as it is used in full fledged UPnP Networks (remember that concubi*Net* only uses the discovery functionality to advertise new devices and does not recognize anything like UPnP services or embedded devices), enumerating how devices advertise and revoke their advertisements as well as how control points search and devices respond.

3.14.1 Advertisement

When a device is added to the network, the UPnP discovery protocol allows that device to advertise its services to control points. It does this by multicasting discovery messages to a standard address and port. Control points listen to this port to detect when new capabilities are available on the network. To advertise the full extent of its capabilities, a device multicasts a number of discovery messages corresponding to each of its embedded devices and services. Each message contains information specific to the embedded device (or service) as well as information about its enclosing device. Messages should include duration until the advertisements expire; if the device remains available, the advertisements should be re-sent with (with new duration). If the device becomes unavailable, the device should explicitly cancel its advertisements, but if the device is unable to do this, the advertisements will expire on their own.

Device available - NOTIFY with ssdp:alive When a device is added to the network, it multicasts discovery messages to advertise its root device, to advertise any embedded devices, and to advertise its services. Each discovery message contains four major components:

- a potential search target (e.g., device type), sent in an NT header,
- a composite identifier for the advertisement, sent in a USN header,
- a URL for more information about the device (or enclosing device in the case of a service), sent in a LOCATION header, and
- a duration for which the advertisement is valid, sent in a CACHE-CONTROL header.

Choosing an appropriate duration for advertisements is a balance between minimizing network traffic and maximizing freshness of device status. Relatively short durations close to the minimum of 1800 seconds will ensure that control points have current

device status at the expense of additional network traffic; longer durations, say on the order of a day, compromise freshness of device status but can significantly reduce network traffic. Generally, device vendors should choose a value that corresponds to expected device usage: short durations for devices that are expected to be part of the network for short periods of time, and significantly longer durations for devices expected to be long-term members of the network.

Due to the unreliable nature of UDP, devices should send each of the above mentioned discovery messages more than once. As a fall-back, to guard against the possibility that a control point might not receive an advertisement for a device or service, the device should re-send its advertisements periodically. Note that UDP packets are also bounded in length (perhaps as small as 512 Bytes in some implementations) and that there is no guarantee that the messages will arrive in a particular order.

When a device is added to the network, it must send a multicast request with method NOTIFY and ssdp:alive in the NTS header in the following format. Values in italics are placeholders for actual values.

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age = seconds until advertisement expires
LOCATION: URL for UPnP description for root device
NT: search target
NTS: ssdp:alive
SERVER: OS/version UPnP/1.0 product/version
USN: advertisement UUID
```

Figure 20: ssdp:alive messsage

Device unavailable – NOTIFY with ssdp:byebye When a device and its services are going to be removed from the network, the device should multicast a ssdp:byebye message corresponding to each of the ssdp:alive messages it multicasted that have not already expired. If the device is removed abruptly from the network, it might not be possible to multicast a message. As a fall-back, discovery messages must include an expiration value in a CACHE-CONTROL header (as explained above); if not re-advertised, the discovery message eventually expires on its own and must be removed from any control point cache.

When a device is about to be removed from the network, it should explicitly revoke its discovery messages by sending one multicast request for each ssdp:alive message it sent. Each multicast request must have method NOTIFY and ssdp:byebye in the NTS header in the following format. Values in italics are placeholders for actual values.

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
NT: search target
NTS: ssdp:byebye
USN: advertisement UUID
```

Figure 21: ssdp:byebye messsage

Due to the unreliable nature of UDP, devices should send each of the above mentioned messages more than once. As a fall-back, if a control point fails to receive notification that a device or services is unavailable, the original discovery message will eventually expire yielding the same effect.

3.14.2 Search

When a control point is added to the network, the UPnP discovery protocol allows that control point to search for devices of interest on the network. It does this by multicasting a search message with a pattern, or target, equal to a type or identifier for a device or service. Responses from devices contain discovery messages essentially identical to those advertised by newly connected devices; the former are unicast while the latter are multicast.

Request with M-SEARCH When a control point is added to the network, it should send a multicast request with method M-SEARCH in the following format. Values in italics are placeholders for actual values.

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: seconds to delay response
ST: search target
```

Figure 22: ssdp:discover messsage

Due to the unreliable nature of UDP, control points should send each M-SEARCH message more than once. As a fall-back, to guard against the possibility that a device might not receive the M-SEARCH message from a control point, a device should re-send its advertisements periodically.

Response To be found, a device must send a response to the source IP address and port that sent the request to the multicast channel.

Responses to M-SEARCH are intentionally parallel to advertisements, and as such, follow the same pattern as with ssdp:alive (above) except that the NT header there is an ST header here. The response must be sent in the following format. Values in italics are placeholders for actual values.

```
HTTP/1.1 200 OK
CACHE-CONTROL: max-age = seconds until advertisement expires
DATE: when response was generated
EXT:
LOCATION: URL for UPnP description for root device
SERVER: OS/version UPnP/1.0 product/version
ST: search target
USN: advertisement UUID
```

Figure 23: response messsage

Due to the unreliable nature of UDP, devices should send each response more than once. As a fall-back, to guard against the possibility that a control point not receive a response, a device should re-send its advertisements periodically.

4 Loadbalancing

4.1 Profiling-Tool Introduction

4.1.1 Gatherable information

To predict bottlenecks it is necessary to collect quite a few information about the state of concubi*Net*, its peers, and its CCN-objects. One main gatherable information type is the amount of memory a given peer is temporarily occupying. Being able to access such information would make it possible to predict and eventually overcome possible memory lags by transferring critical objects - referring to objects with a huge memory load - just in time, so that the given peer would never run out of memory. It is also possible to guarantee a time-critical application to never run out of memory by predicting its memory load and freeing the needed amount of memory even before the application is launched. On the target peers we need the same mechanisms to ensure that the demanded space of memory is available or even to make assumptions on its near future state, being able to predict that a given peer is e.g. very likely to run out of memory on its own soon and will have to take advantage of the LBS itself.

4.1.2 JAVA internal methods to get memory information

Having explained the need to gather information on the state of a peer's memory-load, we must point out that usually it is not possible to access hardware information within JAVA due to its virtual-machine-running-in-a-sandbox principle. This principle allows programmers to write applications knowing to be executable on almost any hardware without the need to compile it separately and gives JAVA-applications a certain robustness in exception- and error-handling. The JAVA 1.4.1 API provides three rudimentary standard methods in its Class JAVA.lang.Runtime, namely totalMemory(), maxMemory(), and freeMemory(). The method totalMemory() returns the total amount of memory currently available for current and future objects, measured in bytes, maxMemory() returns the maximum amount of memory that the virtual machine will attempt to use, measured in bytes, and freeMemory() returns an approximation to the total amount of memory currently available for future allocated objects, measured in bytes. A load balancing service using these methods would be more than inaccurate because it would not distinguish between single objects but rather than this would sum up all memory allocated on the virtual machine making it impossible to determine a single memory bottleneck and therefore render itself useless.

4.1.3 Methods of memory profiling

- A way to get a more precise approximation on a memory load of a single object would be in allocating a large number of the given object in memory and measure the free memory before and after the process and thus averaging the load of the single object. To drop out the noise with this method in a big multithreaded environment we would have to allocate a huge number of a given object simultaneously which would reduce the practicability of this method in limited-resource environments such as handhelds, smartphones, etc.
- Another way to determine the amount of memory an object is allocating may be to serialize it, write it to the hard disk - or whatever permanent memory the current device is using - and to observe its file size in bytes. Although this method has shown to be quite practicable and easy to implement, the gathered

information is very inaccurate and may differ widely from the effective amount of memory an object allocates in memory. Thus we quickly discarded this method as our procedure of choice and did not pursue this approach any further.

- A third way is to use introspection. Introspection is a feature of the JAVA reflection API that allows us to take a look at the structure of a given class during runtime. Using introspection we are now able to count and identify any variables and attributes a JAVA class or object uses in its runnable state.

4.1.4 Using the JAVA virtual machine profiler interface

Next it would be desirable to know the exact size of a given variable- or attribute-type it allocates in memory. A way of achieving this goal is by using the JAVA virtual machine profiling interface. The JVMPI is a two-way function call interface between the JAVA virtual machine and an in-process profiler agent. On one hand, the virtual machine notifies the profiler agent of various events, corresponding to, for example, heap allocation, thread start, etc. On the other hand, the profiler agent issues controls and requests for more information through the JVMPI. For example, the profiler agent can turn on or off a specific event notification, based on the needs of the profiler front-end. Thus we could use a profiler agent that informs us on heap allocation or delivers a complete heap dump at any desired moment detailed enough to make a statement on how much memory is allocated in any type of variables. Combining this with the information on type and amount of the different attributes in the examined class it is quite simple to pick the desired values from the profiler output and sum them up.

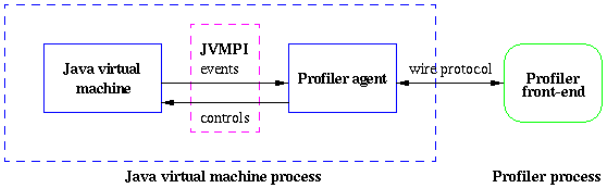


Figure 24: Diagram of the Java virtual machine process

4.1.5 Sun's HPROF profiling agent

To get a standardized profiler output a load balancing service can work with, we chose to use Sun's standard profiler HPROF which comes wrapped up with every JRE or JDK since version 1.2. The HPROF can be configured to profile memory allocation, CPU usage or to pull stack traces in a given interval. The hprof.dll may be launched at JAVA virtual machine startup or during runtime using the JAVA native interface (JNI) with parameters specifying the events at which the profiler agent should be notified.

4.1.6 Translating the profiler and introspection output

To combine the outputs from the introspection and profiler we also implemented a translation library. This became necessary since the naming these two instances use are quite different from one another. E.g. an array of character-type variables will be identified with '[C' by the profiler and with 'char[]' by introspection. This translation library yet fulfilled another purpose. It gathered statistical data on how much memory an attribute-type would allocate at an average. Thus we could build a heuristic that

comes in handy when running an LBS on an environment without profiling capabilities, such as handhelds or smartphones.

```
SITES BEGIN (ordered by live bytes) Mon Jan 05 17:18:12 2004
          percent          live       alloc'ed  stack class
 rank   self  accum     bytes objs   bytes objs trace name
    1 32.26% 32.26%    624224  669  624224  669     1 [I
    2 24.23% 56.49%    468872 6728  468872 6728     1 [C
    3 12.96% 69.45%    250744  595  250744  595     1 [B
    4  8.99% 78.44%    173856 7244  173856 7244     1 java.lang.String
```

Figure 25: Output from the HPROF profiling agent

4.1.7 Developing heuristics

As it is quite likely for our memory profiling tool to encounter a lot of variable-types which have yet to be translated in the 'JVMPI slang' the library was implemented to allow new and unknown types to be added into a special 'working' section of the library intending developers to enhance the library and close knitted the heuristics as well. This could be done directly after a session from within the memory profiling tool or at least once in a while with any arbitrary text editor. Of course the last profiler output will also be logged for further verification.

4.1.8 Integration and further design-space

Since a working concubi*Net* Load-Balancing-Service is not yet implemented and still fictional it is none the less reasonable to be concerned with the integration into the working concubi*Net*. The following scenario is imaginable:
Consider a mobile CCN-client running on a PDA or smartphone without the already mentioned profiling capabilities. This client detects that it is seriously running out of memory due to some heavy multitasking. Our client is now in the position to profile his CCN-object and receives a bunch of values for each of them. These values are approximate memory loads computed by each of the described methods and a calculated sharpness value, which expresses the accuracy of the collected data. The higher this sharpness gets the more sophisticated the methods were used to gather the memory load. On our mobile PeerManager we won't get a very high accuracy level, due to the limited profiling capabilities our virtual machine has to offer, so the most utilizable value comes from our heuristic. The high level LBS has to trade off memory load indications versus their accuracy and spots out the CCN-object that gives us the greatest benefit in balancing it out to a client that has the required resources.

4.1.9 Future steps

Although the memory load calculated this way for a given class or object is again an approximation we are convinced it is the most accurate one. One major disadvantage is the lack of taking loops in consideration. So if a while- or for-loop declares a variable and thus allocates the appropriate memory it will only be counted one time regardless on how many more times the loop will be executed.
Also yet to be implemented is a CPU profiling functionality which associates classes or tasks with the according amount of CPU-time which will be needed for execution. Since the HPROF agent is also capable of profiling CPU-usage it is conceivable that the same approach can be taken to accomplish CPU-profiling.
The heuristic is obviously a matter for future improvement. There are many possible alterations and enhancements thinkable to make the heuristically determined memory

load more accurate but it is our considered opinion that crucial improvement concepts will evolve during a more frequent usage and will take some time until a sophisticated Load Balancing Service exists.

4.2 Mapping service, visualization

It is planned to create a mapping service, that gives a visualization of all reachable peers and running objects. But it is too early to give more detailed information now. Up to now we just collected some ideas and proposals.

5 Security

5.1 The goals of cryptography

The goals of cryptography based on three basic requirements:

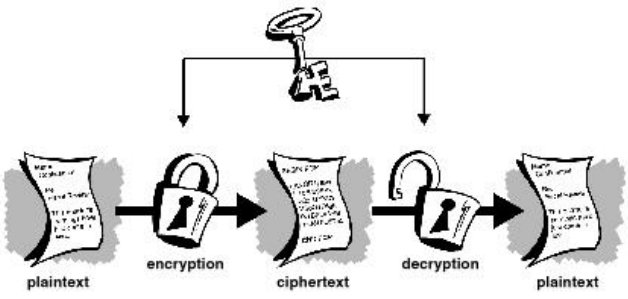
5.1.1 Confidentiality

The cryptography provides mechanisms to ensure the communication between two people, usually referred to as Alice and Bob, over an insecure channel in such a way that an opponent, Oscar, cannot understand what is said. This channel could be a telephone line or computer network, for example. The information that Alice wants to send to Bob, which we call "plaintext", can be an English text, numerical data, or anything at all - its structure is completely arbitrary. Alice encrypts the plaintext, using a predetermined key, and sends the resulting ciphertext over the channel. Oscar, upon seeing the ciphertext in the channel by eavesdropping, cannot determine what the plaintext was; but Bob, who knows the encryption key, can decrypt the ciphertext and reconstruct the plaintext.

Symmetric-key encryption Typical scenario: Alice and Bob want to communicate over an insecure channel like the Internet. Alice encrypts her message with a secret key and sends the ciphertext to Bob. Bob has the same secret key and he can decrypt the ciphertext and gets the plaintext again. If Oscar eavesdrops the ciphertext, he can’t encrypt the ciphertext without the secret key. We assume, that Alice and Bob have agreed on which algorithm and secret key they are going to use.

Common implementations of symmetric key algorithms are DES (Data Encryption Standard), 3-DES (triple DES), IDEA, RC5, Blowfish, and AES (Advanced Encryption Standard). AES is the new Federal Information Processing Standard (FIPS-197) algorithm endorsed for governmental use and chosen to replace DES as the de facto encryption algorithm. AES uses the Rijndael algorithm, chosen after a thorough evaluation of 15 candidate algorithms by the cryptographic research community.

Asymmetric-key encryption Typical scenario: Alice wants to send Bob a message over an insecure channel like the Internet. They have never met in real life, so they haven’t agreed to a certain key. Bob has a public and a private key. The private key is secret and only Bob knows the private key. Everybody knows the public key. It is assumed that the public key can’t be modified by Oscar and the public key is published in a kind of telephone book. Alice finds out the public key of Bob and encrypts the



message with this key and sends the ciphertext to Bob. Bob decrypt the ciphertext with his private key and gets the message. If Oscar eavesdrops the ciphertext of Alice, he can get the message, only when he has the private key of Bob, but only Bob owns the private key, so it is not possible for Oscar to get the plaintext. All Public-Key systems are based on the difficulty of calculating the discrete logarithm in a finite field or on the difficult factorize large numbers in prime numbers.

Algorithms such as RSA, Diffie-Hellman, and El-Gamal implement public key encryption methodology. Today a 512-bit key is considered barely adequate for RSA encryption and offers marginal protection; 1024-bit keys are expected to withhold determined attackers for several more years. Keys that are 2048 bits long become commonplace and rated as espionage strength.

5.1.2 Comparison of both systems

Both systems have advantages and disadvantages, which are compared in the next sections.

Advantage of asymmetric-key encryption

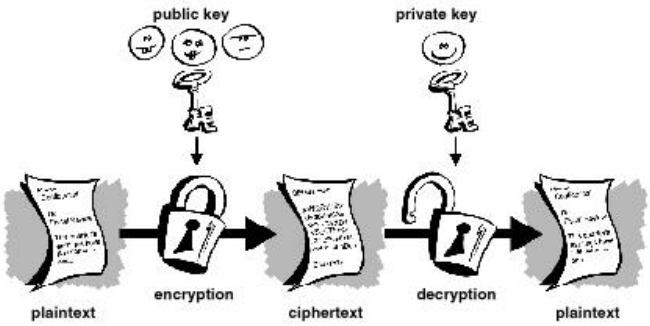
1. Secrecy is simple. The owner has only to keep his private key secret.
2. Key exchange or key agreement is not necessary.
3. It is possible to use asymmetric-key encryption for digital signatures.
4. The number of keys is reduced. Compared with symmetric encryption, it needs only one key pair per participant.

Disadvantage of asymmetric-key encryption

1. Asymmetric-key encryption is very slow and inefficient, because it uses very large numbers. Ciphertexts of small messages can be very big.
2. The key lengths of asymmetric-key encryption are much more larger than keys, which are used by symmetric-key encryption. Usually the length of a RSA-key is 1024 or 2048 bits.

Advantage of symmetric-key encryption

1. The key length is very small. Usually the length of a key is between 56 and 256.



2. Symmetric-key encryption is much more faster than asymmetric-key encryption. The main reason is that the key length are very short
3. It is possible to combinate two or more symmetric-key systems in order to increase the security.

Disadvantage of symmetric-key encryption

1. Key exchanging and key agreement is the main disadvantage of symmetric-key system. If Alice and Bob want to communicate over a secure channel, one of them has to send the secret key over a secure channel. It depends heavily on a secure channel to send the key to the partner.
2. For every participant an own key is needed.
3. The secret keys must be protected. By asymmetric encryption the owner has to protect one key, by symmetric encryption he has to protect many keys.

5.2 Authentication

An important region of cryptography is authentication. If Bob sends Alice a message, Alice can be sure that the message is from Bob and no one else has sent the message. Bob signs his message and Alice on the other side verifies this message. With the tool of cryptography Oscar can't write a message to Alice in the name of Bob. Alice would always find out, that the message is not from Bob. Bob uses a digital signature. Authentication assures that people you deal with are not imposters. Usually Alice and Bob use digital signatures. When Bob wants to send Alice a signed message, he encrypts his message with his private key and send the ciphertext to Alice. Alice decrypts the ciphertext with Bob‘s public key. So Alice can be sure, that the message is from Bob, because only Bob has the private key which corresponds to Bob‘s public key.

5.3 Integrity

Integrity ensures that unauthorized manipulations at data are recognized. If Alice wants to send a message to Bob. In order to be sure that the message won‘t to be manipulated, she calculates the message digest of her message and sends the message and the digest value to Bob. He calculates the message digest of the message and compares it to the message digest value. If both values are equal, Bob knows that the message is not manipulated on the way to him. To calculate the digest value of the message, Alice and Bob use a cryptographical hash-function f. The main properties of a hash function are:

1. Collision resistance: It is computationally impossible to find two message m and mt' with the same hash value (f(m)=f(mt'))
2. Preimage resistance: If the hash value y is known, it is computationally impossible to find a message m with $f(m) = y$

The most well-known hash-functions are MD5 and SHA-1. The length of hash values of MD5 and SHA-1 are 128 bits.

5.4 Java and Cryptography

Java supports the most common cryptographical algorithms, but cryptography is not implemented directly in the libraries of JDK. Java cryptography software comes in two pieces. One piece is the JDK itself, which includes cryptographical classes for authentication. The other piece, the Java Cryptography Extension (JCE), which includes so-called "strong cryptography". Sun provides an architecture in order to develop cryptographical services. The architecture is called Java Cryptography Architecture (JCA). Because of the export restrictions of the U.S.A. for "strong cryptography" this architecture comes in two pieces. The Java Cryptography Extension (JCE) is the part of the JCA, which is limited by the export restriction. Thus with the JCA it was considered that it is possible for Java developers outside of the U.S.A. to integrate their own implementation of strong cryptography into the JCA. Because of the loosened export restrictions Sun changed the architecture. The export of the new JCE implementation is possible only, if the framework plans mechanisms, to announce only determined providers. One can use the JCE only with authorized providers. The authorization happens by digital signature. A further change of the architecture arose as a result of the fact that it was possible to export the JCA and particularly the JCE into each country of the world. Special files called "Jurisdiction Policy Files" bound the strength of cryptographical algorithms to the locally given laws. If a country permits key length of max. 1024 bits e.g. for RSA, nobody can produce keys of larger length up to authorized applications.

It is possible for Java developers to implement security agencies on application level by the JCA. The JCA was introduced with JDK 1.1. It contained APIs for digital signatures and hash functions. In the following versions additional security services were added. The packages are arranged below java.security.*. By the export restriction for "strong cryptography" APIs for en- and decryption, key exchange and MACs were collected in a separate collection of classes. It is called "Java Cryptography Extension". The appropriate packages are arranged below javax.crypto.*. The following illustration shows the connection of the different packages.

Due to the export restriction there exist certain requirements to the design of the architecture, in order to guarantee that developers can implement outside of the U.S.A. strong cryptography algorithms into the architecture of JCA. Thus concepts developed such as engines class and Cryptography service provider. The requirements to the architecture were for example:

1. Support of different from each other independent implementations
2. Interoperability of the different implementations must be ensured
3. Algorithm agility: Independent from specific algorithms
4. Extensibility of the API

The fist two points may be summarized as "Provider agility". The independence of

certain implementation is a result of the export restriction of the U.S.A.. Thus developers outside the U.S.A. can develop their own implementation and integrate it into the architecture. In this case it is important that different implementations can work together. The framework ensures for the fact that in the crucial places strict rules ensure this in the form of given interfaces. The implementations are integrated as Cryptography Service Provider (called JSP or provider). The independence of algorithms is ensured by Engine Classes. These classes provide security services, which are independent of the cryptographical algorithm that is realized inside. For example the programmer doesn't need to determine a special algorithm for the encryption (i.e. Rijndael, DES, 3DES, ...) when he works with engine classes. The result is that he can change the cryptographical algorithm in the engine class by changing a string. The following list of some engine classes describe a cryptographical methods in an abstract way:

Class	Description
MessageDigest	Calculates hashvalues
Signature	Generates and verifies digital signatures
KeyPairGenerator	Generates keypairs for asymmetric algorithms
SecureRandom	Generates random numbers
Cipher	Encrypts and decrypts messages
KeyGenerator	Generates secret keys
Mac	Calculates MACs

To clarify how these classes work, let's have a closer look at the class Cipher. Every class above has two static methods:

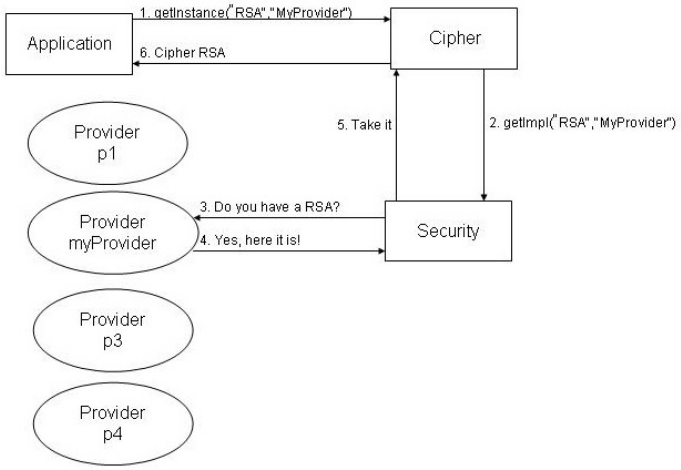
```
public static <Typ> getInstance(String algorithm, String provider)
  throws NoSuchAlgorithmException, NoSuchProviderException }

public static <Typ> getInstance(String algorithm) throws
  NoSuchAlgorithmException
```

The first method requests an instance of the class Cipher for the special algorithm, as implemented by a special provider. The name of the algorithm and of the provider are specified as method parameters. The providers are announced to the class java.security.Security. The Security-class asks the provider for the implementation of the requested algorithm. If the provider has no implementation of the algorithm, then an instance of NoSuchAlgorithmException is thrown. If the specified provider doesn't exist or isn't installed, then an instance of NoSuchProviderException is thrown. This method searches first for the provider and then for the algorithm.

```
try {
    Cipher RSAcipher =Cipher.getInstance("RSA", "MyProvider");
    }
    catch(NoSuchProviderException)
        {System.out.println("Provider is not installed");}
    catch(NoSuchAlgorithmException)
        {System.out.println("There exists no implementation
        of the specified algorithm");}
```

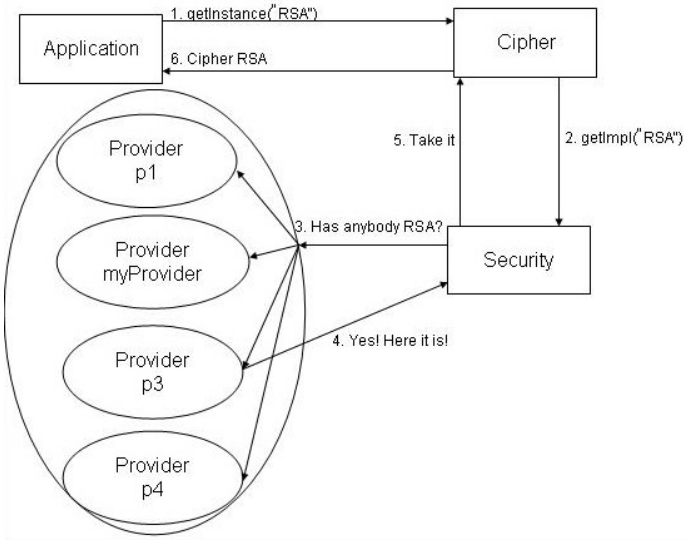
The interaction is clarified in the illustration:

The second method requests an instance of the class Cipher for a special algorithm. In this case all installed providers were asked by the Security-class, whether they



have an implementation of the algorithm. If no provider has an implementation of the algorithm, then a NoSuchAlgorithm Exception is thrown.

The interaction is clarified in the following illustration:



Installing a provider The actual version of JDK comes with a default provider called "SUN". This provider supports implementation of algorithms, which are not limited by the export restriction of the U.S.A. These algorithms are some hash-functions like MD5, algorithms which support DSA, etc. First the programmer has to know which algorithms he will need and which provider supports these algorithms. Not every provider supports every algorithm. To integrate other providers, the programmer has to download jar-files from the website of an appropriate provider like Cryptix, Entrust, JCSI, etc... After the download of the jar-files, they should be added to the Java classpath. Providers can be configured in two ways. The programmer can edit a properties file, which is called static provider configuration, or he can manage providers at runtime, which is called dynamic provider configuration. To configure the provider

statically, he needs to edit the java.security file, which is found in

`jre/lib/security`

directory under the JDK home directory. Inside this file, each installed provider is represented by a line in the following format:

`Security.provider.n=providerclassname`

n is the number that determined the preference order of the providers. Usually the first provider is the default provider SUN:

`Security.provider.1=sun.security.provider.Sun`

To add a provider dynamically the programmer can call

`Security.addProvider()`

or

`Security.insertProviderAt()`

with an instance of the provider class. The first method adds the provider at the end of the provider preference list and returns the position of the provider in the list. The second method adds the given provider at the given position and returns the position of this provider. If the provider is already installed, the return value is -1. In order to examine, whether the program is correctly installed, he can use the static getProviders()-method in the Security class and getName()-method in Provider class. The first method returns an array of all installed providers and the second method returns the name of the provider as a string. Here is an example how to check the installed providers:

```
Provider[] p=Security.getProviders();
for(int i=0;i<p.length;i++){ System.out.println("Installed provider
  at position "+i+":"+p[i].getName());
}
```

5.5 Cryptography and Authentication in Concubi*Net*

At the beginning of the project no security features were installed. The communication between agents and objects ran unencrypted. Every object could be moved to an arbitrary agent. Now a part of the communication works encrypted. Concubi*Net* works with the provider Cryptix, because Cryptix supports most cryptographical algorithms like RSA, DES, 3DES, Rijndael, MD5, SHA-1, etc. The technical realization is completed. This means, that every agent supports symmetrical and asymmetric algorithms, they are able to sign and to verify digital signatures and they can generate fingerprints of serialized objects. Every agent is able to communicate with each other over a secure channel. At present the agents can transport encrypted CCNObjects. The objects are symmetrically encrypted in order to minimize the time of en- and decryption. In order to agree on a secret key the agents use RSA. For example: If agent A wants to transfer an object to B, A requests the public RSA-key of B. B sends his public key to A, A generates a secret key an encrypts this key with B's public key. A sends the encrypted secret key to B, B decrypts the encrypted key and at this moment A and B know the secret key, which was generated by A. A and B can use this key to encrypt and decrypt objects with symmetric algorithm. In Concubi*Net* as soon as an agent starts, a RSA keypair is produced in the constructor. The public key can then be accessed by a get-method. If an agent A wants to transport an object to B, then A

obtains the public key of B. A gets the key of B by the method getRemoteKey(). The method contains the ID of the agent B among other things as parameter. Afterwards A creates a symmetrical key. This key is encrypted by the public RSA key of B. A sends it by means of the method transferKey() to B. The method transferKey() calls on the side of the agent B the method takeKey(), with which the symmetrical key is decrypted and set as session key. After this procedure A and B know the secret key and are able to en- and decrypt with this key symmetrically. A serializes and encrypts the object. The result of the encrypted object is a byte array. This byte array is transferred at B. B decrypts it and may continue to work with the object normally.

5.6 Introduction into the public key infrastructure and their components

Typical scenario without use of certificates: Alice and Bob would like to communicate with each other over an unsecure channel. Out of this reason they use an asymmetrical cipher. Alice has to get Bob's public key, in order to build up a secure communication channel. The aggressor Mallory is able to manipulate Bob's public key. He exchanges the public key of Bob by his own. Furthermore it can intervene actively in the communication. He can intercept messages. Alice uses the key of Mallory, although she thinks that she uses the public key of Bob. She encrypts the message and sends it to Bob. Mallory catches the message decrypts the message and manipulates it. Then it takes the real public key of Bob, encrypts the manipulated message and sends it to Bob. Alice and Bob draw no suspicion that the message is manipulated by a third person and not authentic. This kind of attack is called man-in-the-middle-attack. The main problem which is shown by this attack is, that the authenticity of the keys cannot be ensured.

5.6.1 Term introduction

Certificates A certificate is a digital identity card. By certificates it is possible to determine the authenticity of data. That means it that one can be sure that the data comes from the person which one assumes and that the data were not manipulated during the transmission knowingly or unknowingly. More near described it certificates an authentication of the public key, which is stored in the certificate. Certificates guarantee that the public and thus also the private key can be used only by one certain person.

Certificates of the standard X.509v3 Certificates of the standard X.509v3 are used at most. The first version has been used 1988 and gradually been extended.
A certificate, which corresponds to the standard X.509v3, contains the following data:

1. Version number: Information of the X.509 version
2. Serial number: The serial number must be uniquely
3. Signature of the exhibitor: Signature of the issuer
4. Exhibitor of the certificate: The name of the issuer
5. Validity period of the certificate: The date for the validity of the certificate at
6. Certificate owner: The owner's name of the certificate
7. Public key and key information of the owner: A part from the entries of the public keys, it indicates which algorithm is used

8. Identificator: If the name of the owner is not clearly, an additional identifier indicates it
9. Additional extension: Arbitrary extensions can be inserted, which one is to certify by the issuer.

Properties and use of certificates A certificate has two important safety characteristics:

1. The first security characteristic is the signature. It is generated by an issuer. An asymmetrical algorithm is used. Thus the authenticity of the certificate is guaranteed.
2. The second security characteristic is a unique hash value. With the hash value it can be examined whether the certificate was changed later. The purpose of a certificate is obvious: It is to create in public key infrastructure confidence between users. With certificates the authenticity of the keys is guaranteed among other things. In addition a user cannot deny that a public key does not belong to him. That means that by certificates commitments are guaranteed. Unfortunately certificates cannot prevent that someone abuses the confidence of an other user.

TrustCenter The TrustCenter is an instance that issues certificates. Every user which has a certificate of an instance of a TrustCenter trusts the instance. The TrustCenter has the task of generating certificates and to publish the public information of an user like the public keys or certificate of an user. It can likewise generate pairs of keys for RSA and DSA. Furthermore the TrustCenter has the task of the authenticity examination.

Generating a certificate A user requests a certificate by a TrustCenter. He has to give personal data. This is usually examined on the basis of his identity card. If the user has a private and public key, then this key pair is used for generating the certificate. Otherwise a pair of keys is generated by the TrustCenter. The certificate is then stored on data medium and/or on the operational area corresponding object and handed over to the user. The TrustCenter publishes the certificate.

Signing and verifying data Assume Alice knows Bobs public key and Bob knows Alice public key. The authenticity of the keys is ensured. Alice would like to send a signed message m to Bob. That means that if Bob receives the signed message, he can be sure that the message m is from Alice. Alice takes the message m and computes a cryptographical hash value h(m) from m. This procedure has the advantage that Alice needs to sign only the hash value. It does not need to sign the complete message. The hash function, which is used, is public, so Bob and Alice know, which hash function is used. Alice takes her private key and encrypts h(m) and receives the signature m '. Alice sends tuple (m,m ') to Bob. Bob has to verify. Bob takes the public key of Alice and decrypts m' and gets m". Now he computes h(m) and compares it with m". If both values agree, then Bob can be sure that the message comes from Alice. If they do not agree, then Bob knows that the message is not from Alice. If they user certificates, the whole protocol works as follows: The certificates of Alice and Bob are signed by a TrustCenter. Alice sends the tripel (m,m ', Cert) consisting of the message m, the signed hash value of the message and the certificate of Alice to Bob. Bob examines the certificate of Alice. Bob computes the hash value of Alice data in

the certificate and decrypts the signature of the TrustCenter with the help of the public key of the TrustCenters. If both computed hash values agree, then Bob knows , that the certificate is not manipulated by Mallory. Now Bob can be sure that the public key of Alice, which is stored in the certificate, is authentic. He uses the public key of Alice, in order to verify Alice's message.

5.7 Description of PKI infrastructures

Three concepts will describe the formation of trust between users, which use asymmetrical ciphers. These concepts describe different trust models. In the following descriptions it concerns the persons Alice, Bob and Vera. Alice knows Bob and trusts him. Alice does not know Vera and so Alice doesn't trust Vera (e.g. Alice knows the public key of Vera, but Alice is not sure whether the key of Vera is authentic). Alice wants to send a mail to Bob and/or to Vera, whereby she uses the public keys of the respective communication partner. Alice does not know however whether the respective public key is authentic.

Direct trust This confidence model is a very simple model. Bob confirms the authenticity of his key to Alice. Bob sends the key to Alice. Both compute a cryptographical hash value of Bobs's public key and compare these values with one another over the telephone. Thus it can be prevented that someone can accomplish a man-in-the-middle-attack. If Alice wants to send a mail to Vera, first Alice has to contact Vera over a secure channel (fore example by telephone) in order to get the public key of Vera. She assumes that she really speaks to Vera. No additional infrastructure is needed. The direct trust can be used only in simple cases. If Alice wants to communicate with several people which she does not know, then she must make the procedure she did with Vera with every different one.

Web of trust Alice would like to send a Mail to Vera. Bob knows the key of Vera and he ensures the authenticity of Vera's public key. Alice trusts Bob. Bob signs the public key of Vera and sends the signed key to Alice. Alice verifies the signed key and knows that the authenticity of Vera's key is ensured by the signature of Bob. Vera can sign a key of another person and send it to Alice. By web of trust a trust chain can be provided. Web of trust is more efficient than direct trust and can be used without problems in companies. In infrastructures like the Internet this concept is not practicable.

Hierarchical trust In this structure an independent instance is introduced. Generally this instance is called TrustCenter. Here one assumes Alice and Bob have a certificate that is signed by the TrustCenter. So Alice and Bob trust this instance. Alice has to get the public key of the TrustCenter. Now Alice can verify Bob's certificate by the key of the TrustCenter. If the verification is successful, she trusts Bob and knows the authentic key of Bob. She trusts every user, if he has a certificate which was signed by the TrustCenter which signed Alice's certificate. This concept hierarchical trust is used in practice.

5.8 PKI in CCN

In CCN the hierarchical infrastructure is used. The hierarchy consists of two levels. The PeerManager forms the first and also lowest level. The second and highest level

forms the TrustCenter. In this case every root user (also called administrator) is the TrustCenter. He decides whether a PeerManager of a user may enter the CCN network or not. In the positive case, the PeerManager gets a file, in which the root certificate and the root key pair are stored. The root key pair is encrypted by a password. Every PeerManager trusts this root certificate and it is self signed. The root certificate contains the public root key and it is signed by the appropriate private root key. When a PeerManager starts, it looks for the certificate of the PeerManager. When the certificate does not exist, the PeerManager started for the first time. A root user has to login, because the password the root user agrees with the password, which the root key pair is encrypted. After the root user logged in, RSA- and DSA-keypairs are generated, if they don't exist. Afterwards a certificate of the PeerManager is generated, which is signed by the private root key. The name of the owner is the ID of the PeerManager. The public DSA-key of the PeerManager is the key, which represents the authentic key in the certificate. The PeerManager signs always with the DSA key. So every PeerManager can be identified by the certificate. An object transfer between a PeerManager A and B works as follows: A initializes the transfer and would like to send an object to B. A and B exchange their public data. Public data are the certificate of the PeerManager and the signed RSA-key of the PeerManager. A gets the public data of B and sends its public data to B. The exchange is initialized by a remote call by A. Then a verification of the data takes place on both sides. A requests the verification of the public data of A from B by a remote call. If on both sides the verification is successful, the transfer can take place. It is examined whether A and B know each other. That means that A searches in a hash table for to the ID of B. The hash value is a secret key which was used in another session with B. B does the same. If A and B do not know each other, e.g. they communicate for the first time with each other, a key agreement takes place. A generates a secret key, encrypts it with the public RSA-key of B and signs the encrypted key. A sends the tuple consisting of the encrypted secret key and the signed encrypted secret key to Bob. B verifies and decrypt the secret key. If the verification on B's side passes, A and B know the secret session key. A encrypts the serialized object secret key and sends it to B. B decrypts directly. Because only B and A know the secret key, in this case signing/verification is not necessary. A and B store this session key in a hash table. The hash table consists of tuples (ID of the PeerManager, secret key). If A and B know each other, then they take directly the key from the hash table and communicate directly with the symmetrical cipher.

5.9 Local Security

The task of the local security is to manage local resource permissions, based on the authentification and authorization of the users and programs. The meaning of local resource is wide-spreaded and represents all imaginable resources of a machine, like hard disc, network, installed software etc. Some computers should be reachable and usable for every single user and process, but this state is not always meaningful. For example, a storage driver could contain directories for different users that work toward the same goal, but as opponents. Each of those persons would sleep better, if they know, that nobody can read, write or even delete his or her own files. To perform above mentioned actions, specific permissions are needed. An operating system has to insure that only authorized persons is allowed to assign permissions to a certain user.

The task of the local security, in cooperation with the user management, is to give or take away the permissions. In above mentioned example, to access to the specific files or directories. Referring to CCN and Java as the programming language, we have to deal with, user and CCN object accesses to the resources, because a logged

on user can try to perform some actions on the Peer Manager GUI, on the other hand, a running CCN object can also try to perform some security critical accesses.
java.security.Permission is the central abstract class of the Java API to represent the Permissions of the local resources. Java represents each kind of resource permissions with matching subclass of the Permission class. Those subclasses describes, more or less precisely, the permission for that type of the resource. As an example let us take a look at java.io.FilePermission, that represents the permissions for the file access. The constructor needs two String parameters, one for the file name and one for the sort of access (read, write, etc.). The first parameter can describe the file more or less exactly. For example you can precisely specify a single file, on the other handy you can set the permission for the whole directory. A user or CCN object, that wants to access a file, specified in the constructor, has to possess this Permission. You can of course construct the new types of Permissions for new kinds of resources, for example CCNPermission, that represents CCN resources (descriptions follows later in this chapter).
The task of detecting if a user or an object possesses the needed permission belongs to java.lang.SecurityManager. The SecurityManager defines a method called checkPermission(Permission perm). The functionality is the following:
A class that defines a resource (e.g. java.io.File represents a File and has the methods for reading writing on that File etc.) specifies the Permission perm needed for the access. A security critical part of the class that requires perm, gets the following "if" construct as the header:

```
if(System.getSecurityManager==true)
    System.getSecurityManager.checkPermission(Permission perm);
```

This insures that the security call-ups only get performed if the Security Manager is activated. Permission perm, is the needed Permission to run the rest of the code. When a caller tries to access the security critical code, he indirectly calls the checkPermission method. CheckPermission identifies the caller, checks if his set of given Permissions implies the needed Permission perm, and if so, checkPermission returns quietly, otherwise a Security Exception will be thrown. We overwrote the SecurityManager class with our CCNSecurityPermission class, which is better adjusted to our needs. Our checkPermission method calls the Thread.getCurrentThread.getThreadGroup() to determinate the name of the caller (review the multithreading chapter for the details), looks in its internal lists to ascertain the permissions set of the caller and then makes the decision whether or not to allow the access. As mentioned in the user management chapter, we have four types of user: root, administrator, user, guest. CheckPermission determines to which group the user belongs and then checks the static set of permissions defined for this sort of users. The sets are defined in the following classes stored in the localsecurity package: RootPermissions, AdminPermissions, UserPermissions, GuestPermissions. As mentioned, CCN Permissions are defined with CCNPermission class. This class only needs one String parameter that describes the CCN Resource. An example of a CCNPermission there is the CCNPermission("EditPerm"). An user has to possess this permission in order to access the Permission Editor.

Permission Editor is an administration tool for editing the permissions of each locally known user. On the left side there is the user list and on the right the users permissions. With the "new Permission" and the "remove Permission" buttons, you can edit users permission set. By adding a permission you have to write down the full java name, e.g. java.io.FilePermission for a FilePermission. You must also be careful

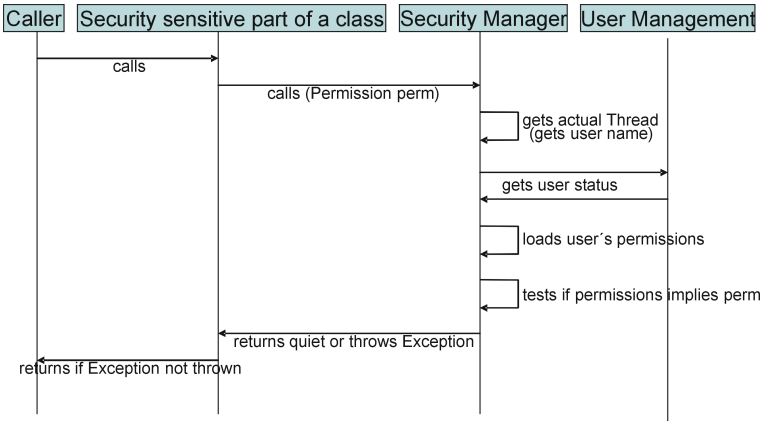


Figure 26: Security check

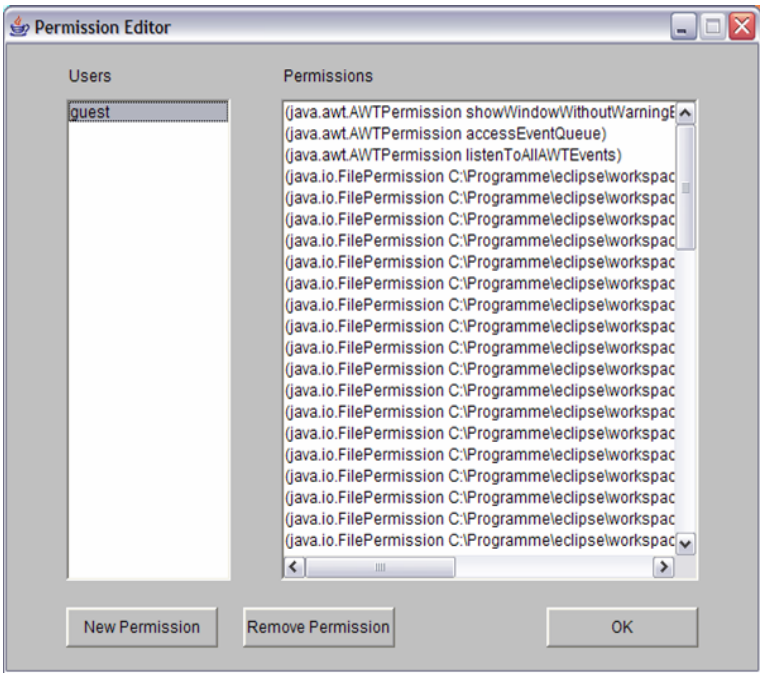


Figure 27: Permission Editor

about the parameters. CCNPermission has only one string parameter. For further permission details we refer to the java API.

5.10 An example

At the end of the security chapter, we should examine an example. We will focus our point of view on a single PeerManager and describe the security mechanism of concubi*Net*. Now we are going to take a look at the following picture and describe all the steps mentioned on the image.

PeerManager B is the central figure in this example. At first we need to introduce a

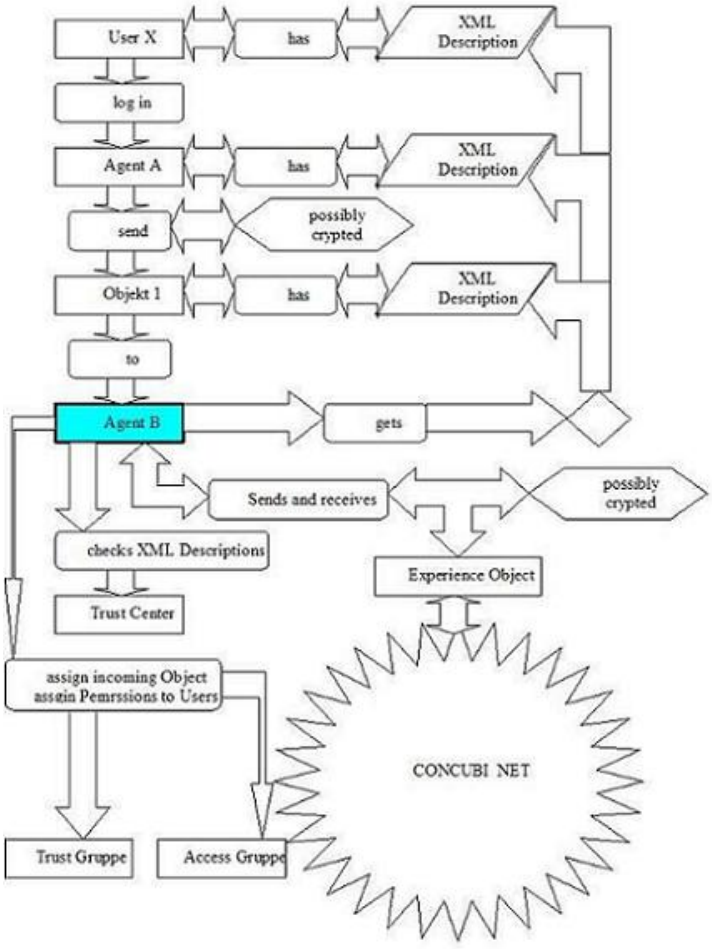


Figure 28: Security check

new component in concubi*Net*, a CCN User. A CCN User has to log in on a PeerManager he wants to use. We have to insure that the user, the PeerManager and the object can be clearly authenticated. This will be probably made with a XML description of all of them and they must be certificated by a trust center instance. Trust center is another new component and will play the key role in recognizing concubi*Net* elements. It is also possible that the trust center contains a kind of an artificial intelligence to control or suggest local security decisions of a PeerManager.
After the process of logging in, the user can decide to send an object to another one. In our example, the user X is announced on the PeerManager A and he wants to send the Object 1 to the PeerManager B. PeerManager B has first to check the identity of all of them by getting the description files of them and consulting the trust center. After that he can receive (or reject) the object. Descriptions and objects have to be secured of any kind of manipulation during the transport and PeerManagers could therefore use an encrypted connection. The process of authentification and encryption has been described in subsections before.
After the PeerManager B determined the identity of other elements, he has to decide

weather he should accept or reject that incoming object. The decision could be based on his local policy or on the suggestion of a neutral instance like trust center. One way is to integrate a local database for every PeerManager. A PeerManager can now store his experiences he has with different users, PeerManagers or objects. Based in his experiences he can decide how to handle an incoming object. Additionally a PeerManager can create a new Object holding his database and send it sporadically to exchange his experiences with other PeerManagers.
Another possibility is to let the trust center make suggestions for those critical decisions. We have to think about, if and how to implement those functions.
Once the PeerManager B got the Object 1, he has to make some local decisions. First he has to give a set of permissions to that object and he has to find out to what locally stored objects the incoming object is allowed to communicate. These decisions could be also made by checking the local data base or by consulting the trust center, but as mentioned this part is still not implemented and has to be done in the next time.

5.11 Usermanagement

5.11.1 Introduction

Distributed systems like concubi*Net* are environments where end users initiate actions. These kinds of actions inevitably create a need for authenticating the initiator. Authenticating can be done in many different ways, including with passwords and certificates. Once the initiator is authenticated, it is necessary to verify that this principal is authorized to perform the requested action. This authorization can only be decided by the operator of the concubi*Net*, and thus requires administration. The User Management Tool provides this type of functionality. The local security service can use the User Management service to authenticate an initiator and his security context. The UM differentiates between three security context modes:

1. Admin: The user in this context has all permissions in the System.
2. User: The user in this context has limited permission. He is only allowed to manipulate owned data.
3. Guest: Only read permission.

5.11.2 The previous state of concubi*Net*

The previous concubi*Net* presented itself as a system without any mechanisms of security and control. Everyone hasn't any restrictions on the system. This could be used to writes objects which have all permissions. So we start to work in order to provide concubi*Net* with security services. For this purpose the User Management (UM) was introduced, that integrates the term of confidence into the system. Concubi*Net* now becomes acquainted with users, security contexts in which they interact and a root-user who can modify any user of the system.

5.11.3 Detailed look at the UM

Now we take a detailed look at the UM. You will see that any features, which are required from local management services, are implemented. The UM is also able to provide distributed management services.

5.11.4 Local UM Management Services

The first thing to do is to authenticate themselves at the local system. See next chapter. Once the user is logged in, he can use the UM-Module to manipulate any registered user account. See chapter UM-Module.

Login The previous system has been changed so far, that any action in concubi*Net* needs authorization. In order to make that possible we introduced the login. Only user already registered may start the PeerManager. After successful login, the associated principal object are loaded from the local user database, and registered as the logged in user in the system. The associated principal object contains following information:

- Name of the user
- Password
- Group affiliation
- HomePM-ID

Group affiliation As we said, a user can only act in the context of a security level. This security level is associated by the group membership. At this time there are three levels (see introduction). Additionally there are root-user who can manage this users.

HomePM-ID This unique identification number is associated with the concubi*Net* ID of that PeerManager, in which this user was created the first time. This ID cannot be manipulated!

5.11.5 Login-Runtime

The first action on a PeerManager is the login. At the first instantiation of the PeerManager, the systems request the root password. after that, the login screen appears. With **root** as user name and the password typed in, the system can be started. The following files are generated at the background.

1. The directory **uadmin**: Stands for User-Administration. This is the root directory
2. The directory **uadmin/admins**: All users with admin permissions are stored here.
3. The directory **uadmin/user**: All users with user permissions are stored here.
4. The directory **uadmin/HOME**: Here the owned user directories are created, in which the user can store owned data. The name of these directories are equal with the name of the users. Only the owner and admin have permission on it.
5. The file **pplist.ccnpp**: In this file all user passwords are stored. The association is the user name as *key* and the password as *value*. This file is ciphered with root password, so only he can encrypt and use it. The purpose of this file is to enable the root to edit the ***.ccnp** files of the users and therefore he needs the passwords cause these files are encrypted with the user passwords.

Any attempt to login at the system, ends in a call to the *Login Verifier*, which determines the identity of the initiator/user. Systems in which the user name and password were stored in databases are a very suitable model for this purpose. However we

decided to develop the storage described above, cause that produces not so much overhead. In our system at a glance, every user must proof his identity by trying to encrypt his associated principal file (*.ccnp). If he is not the one he pretends to, the encryption will throw a *WrongPasswordException*, which results in an *Access Denied* Message.

5.11.6 The UM-Module

With this module you can do any operations for local user management. Following features or operations are available:

- Creation of new user accounts
- Editing of user accounts
- Removing of user accounts

Pay attention to the fact that only the root-user can load this module.

The UM-Module Runtime Also here we take a look at the runtime and the background processes.

1. Create new User: If the creation of an user is successfully, following directories and files will be created:
   (a) The file *.ccnp into the *admins* or *user* directory. *ccnp* stands for *concubiNetPrincipal* and holds any information about this principal. See Subsection login. This file further is encrypted with the password of the user.
   (b) A directory with the name of the user in the directory *HOME*.
   (c) Into this, a X509-user certificate.
   (d) A *Key-Value* entry into the *pplist.ccnpp* file.
2. Edit User: Any Information about the Principal to be edited will be removed and replaced with new entries and files.
3. Del User: Any Information about that Principal will be removed from the local Machine. Distributed Principal Accounts stands alive.

5.11.7 Distributed Management Features

We assume Principal *X* has an account at PeerManager *Y*. But in the environment there is one more PeerManager called *Z*, at which X has no account. What can be done? *X* can login at *Z* as a guest! Then open the application desktop and perform an update principal request. What happens? A Dialog appears where the guest has to type a *username* and the *password*. These informations is passed on to the **root**-User by adding a file called **uprequest.txt** to his *HOME* directory which contains these request. The next time the root logs in, he has to work on it. He performs a remote principal search by any known PeerManager. If a principal with the given username exists remotely, an associated principal object is send to the requester. Is further the given password equal to the remote one, the given principal object is created locally, so the user can login!

5.11.8 How to use it

The main interface to the User Management Services is the UM-Module Graphical User Interface. It can be started over the application desktop, if you are logged in as the root-User.

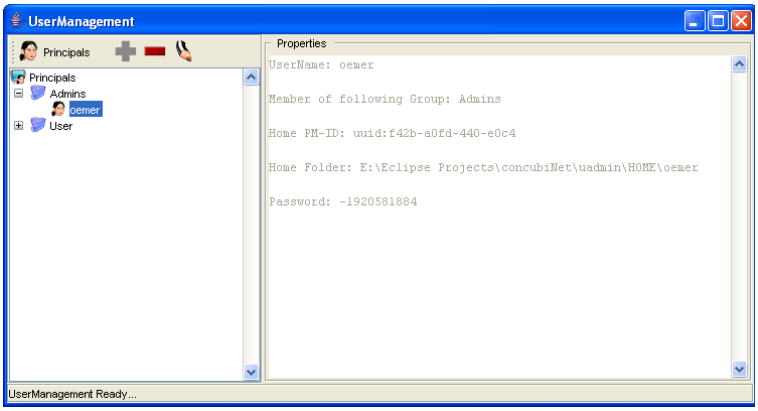


Figure 29: The UM GUI

UM-Module-GUI You see two main panels. On the left side is the navigator, while at the middle the main principal informations are displayed. The principal navigator holds two main directories. The group of *admins* and *users*. If you wish to create an user which is member of the group of the *admins* click the *admins* node and then the + symbol. The new principal wizard will appear where you can type specific data for this principal. At the same way you can delete or edit principals.

6 Appendix

In this chapter the installation and the usage of the PeerManager-GUI will be described. In the first two chapters you will learn to install concubi*Net* and start it the first time. A small programming guide will help you to develop your first simple CCN-object. After that we will give you an introduction on how to use the PeerManager-GUI. The usage of the Remote-Manager will also be shown. Afterwards this chapter gives an overview on how to configure the concubi*Net* system. Last we will describe all the sample applications which exist by now.

6.1 Installation

In this chapter you will learn to install and run concubi*Net* for the first time.

6.1.1 Download from the web

First, download the file ccn2.zip from our home page:
http://www.concubinet.org.
Go to "Resources - Documents" and click on "ConcubiNet" to start the download. You need some kind of unzip program to open this file. Just unzip all the files into a folder, for example "C:\ccn2".

6.2 Getting Started

There are some small things to do to start concubi*Net*.

6.2.1 Modify the conf.xml

After you have unzipped all the files you need to make some modifications on the configuration file "conf.xml". The following modifications have to be made to run the concubi*Net* without troubles. Open the configuration file with any desired text editor. First, modify the tag <pathtosrc> to the value "C:\ccn2" (or whatever folder you have chosen to unzip the file ccn2.zip). Second, modify the tag <javacpath>. This value should be the home folder of your java compiler. (for example "C:\j2dsk1.4.2\bin\javac"). To learn more about all the different tags, refer to chapter 6.4.1 .

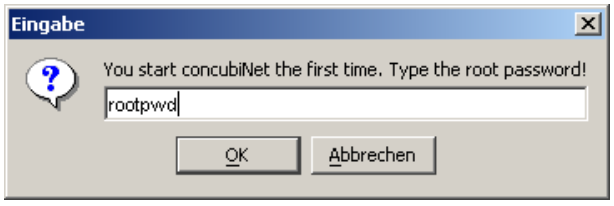


Figure 30: PeerManager-GUI - first login

6.2.2 Start concubi*Net*

After you have unzipped the files and modified the configuration file, you are ready to start the concubi*Net*. Simply double-click the file start.bat. Because you are starting concubi*Net* the first time, you will be prompted to type the root password. Just select

a root password and remember it, to login as the root user. After that, another dialog appears and asks for a username and password. For now there only exists the root user. Type root for the username and your chosen password into the second text field. After you have confirmed your input, the concubi*Net* will begin to initialize, indicated by the splash screen. After a short time a little tray icon will appear which indicates that concubi*Net* is now running. Right click the tray icon to show up the PeerManager. For more information about handling the PeerManager take a look on chapter 6.3.

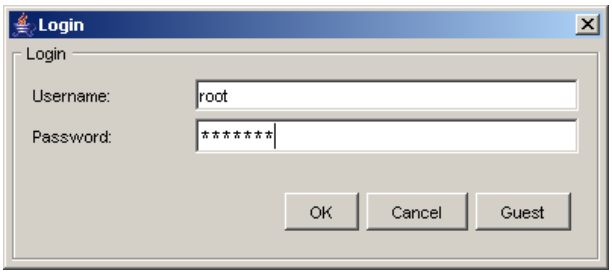


Figure 31: PeerManager-GUI - login

6.3 Handling

The PeerManager-GUI gives the user a graphical interface to retrieve communication related information about the running PeerManager, such as Peer-groupname, PipeID and so on. In addition, the PeerManager-GUI offers the option to browse the wrappers in the wrapper list of the users PeerManager. Another feature of the PeerManager-GUI is the possibility to select any CCN-object and a PeerManager in the neighborhood of the users PeerManager to execute an object-transfer. The PeerManager-GUI shows the functionality of the existing methods of the system and acts as an presentation tool.

6.3.1 Information about the PeerManager

The first tab "About Agent" in the GUI offers information about the running PeerManager associated with the current user. The user can see Peergroupname, GroupID, Peername, Peer-ID and the PeerManager-ID. The user may also start a central LUS or check if a central LUS is already running. Another button is added, called Emergency. In this case the running PeerManager will try to evacuate running objects to other PeerManagers in the neighborhood.

6.3.2 Transfer CCNObject

With this tab the user has the possibility to transfer an object to any other PeerManager. First the user has to use the button "update lists". In this case the PeerManager will start to search for other existing CCN-objects. All movable CCN-objects and all PeerManager in the neighborhood will then be shown and are ready to be transferred. Now the user may select any CCN-object and transfer it to a PeerManager of his choice. This is done by pressing the button "transfer ccnobject". The Output windows shows additional information about the running object-transfer.

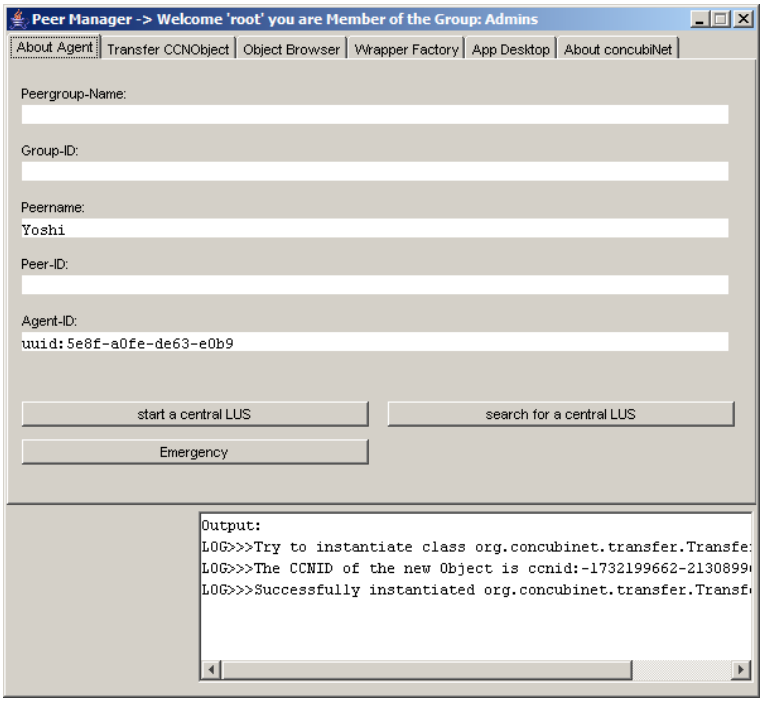


Figure 32: PeerManager-GUI - About Agent

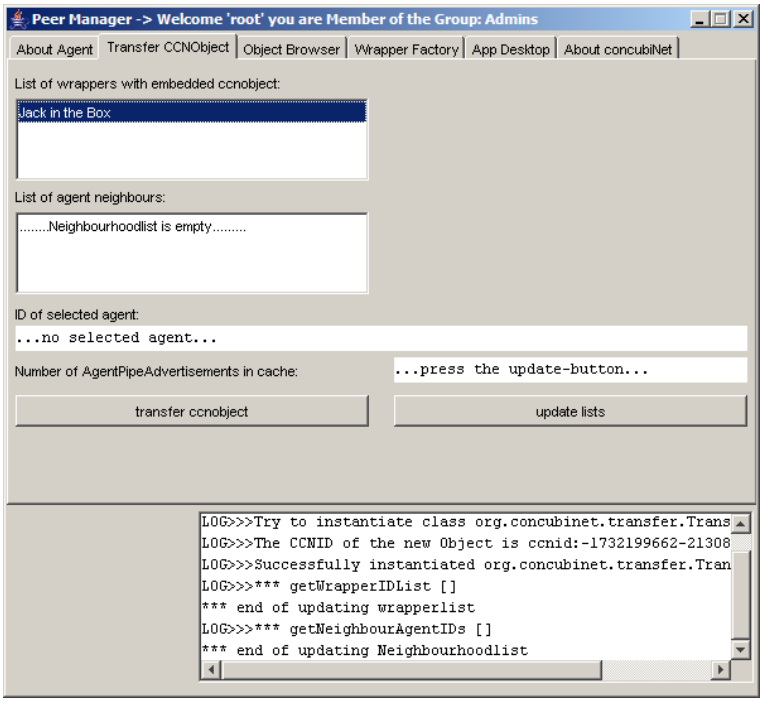


Figure 33: PeerManager-GUI - Transfer CCNObject

6.3.3 Object Browser

On the tab "Object Browser" the user may start existing CCN-objects. To start a CCN-object the user has to press the button "Add JAR" and choose an existing JAR file of a CCN-object. The Application, for example the chat-application, will then be shown and the user could start the chat by selecting it and pressing the button "Start Application". The Object Browser was added to start CCN-objects at runtime making it more comfortable for the user.

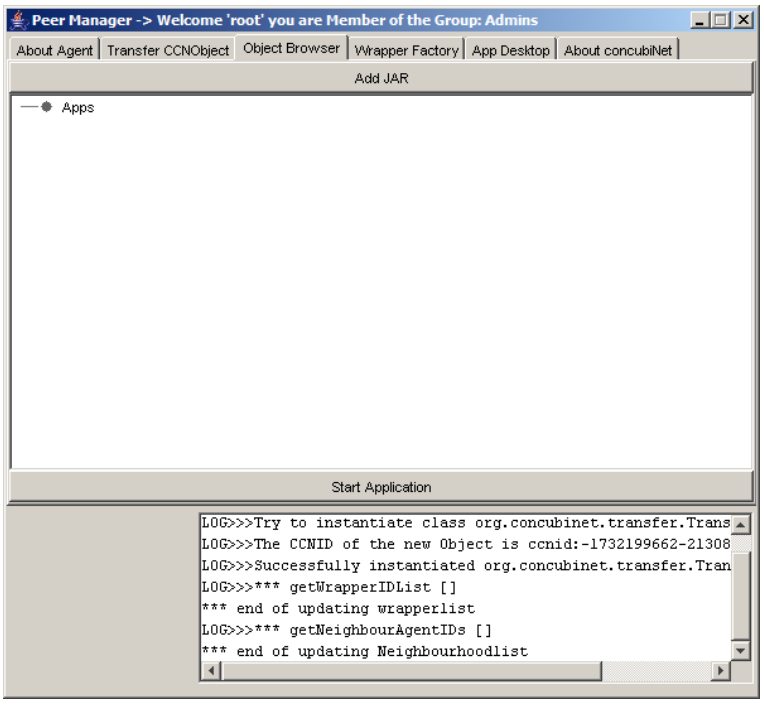


Figure 34: PeerManager-GUI - Object Browser

6.3.4 Wrapper-Factory

As you can see in figure 35 on page 69, we integrated a graphical user interface for our Wrapper-Factory into the PeerManager GUI so that every developer easily can create a wrapper for each of his classes. To create a wrapper first click on "select .class" file and chose a file. You can tell the Wrapper-Factory to add the wrapper to the classpath or to directly instantiate the wrapper by selecting the accordant options. Confirm your selected file and your options by clicking the OK button.

To create a wrapper for the selected .class file chose a PeerManager from the list on the left side (after refreshing the list) and click on "perform request".

6.3.5 Application Desktop

There are several ways to start a concubi*Net* application. As explained before, one is to use the configfile and start the application when concubi*Net* starts. Another is to use our application desktop shown on figure 36 on page 69, which holds links to often used applications, e.g. the user management, personal information manager or the concubi*Net* chat. To start an application just double click its icon and it gets started.

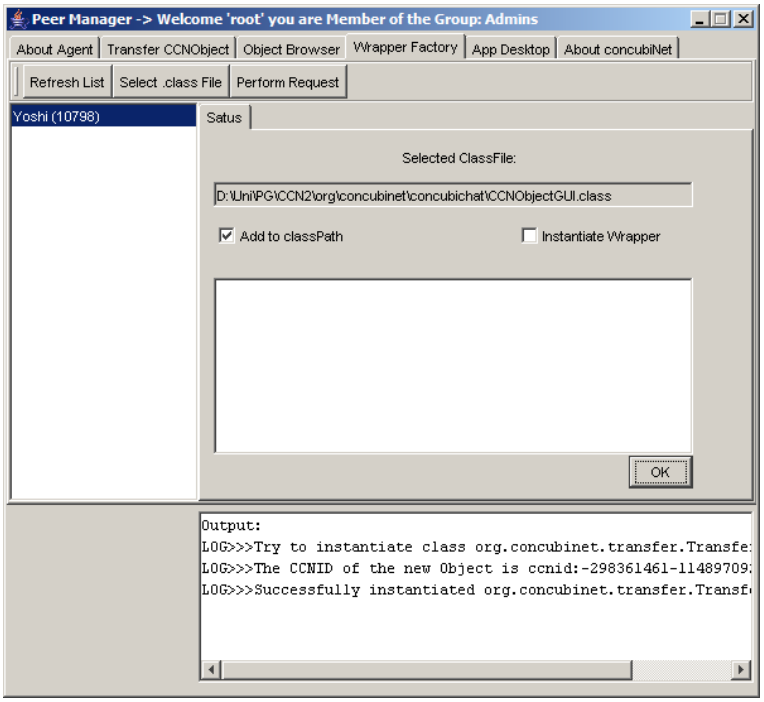


Figure 35: PeerManager-GUI - Wrapper-Factory

Figure 36: PeerManager-GUI - Application Desktop

6.3.6 About concubi*Net*

This tab shows the user some information about the concubi*Net* project, such as the used version of the java-VM, the vendor or the website address.

Figure 37: PeerManager-GUI - About concubi*Net*

6.3.7 Concubi*Net*-Manager

The user may choose to start the concubi*Net*-Manager-GUI instead of the normal PeerManager-GUI. The concubi*Net*-Manager-GUI differs a little bit from the PeerManager-GUI. After the concubi*Net*-System has been initialized the user has to press the button "Refresh". Then all existing PeerManager in the neighborhood will be shown and the user has the ability of remote access to all running PeerManager. On the left it will show the PeerManager in a tree structure. By clicking a running PeerManager all running CCN-objects on this PeerManager will be shown. By double-clicking the PeerManager the user will then get a display of any PeerManager. The user then has full remote access on this PeerManager and may run any operation described in the above chapter. By double-clicking a running CCN-object the user will get additional information on the selected CCN-object, such as CCNID, classname, owner and so on.

6.4 Configuration

A great part of the configuration is done by modifying the configuration file, named conf.xml, which is located in the working directory of the concubi*Net* system. You can edit the configuration file by any text-editor or XML-editor. A detailed description of the configuration tags in the conf.xml follows.

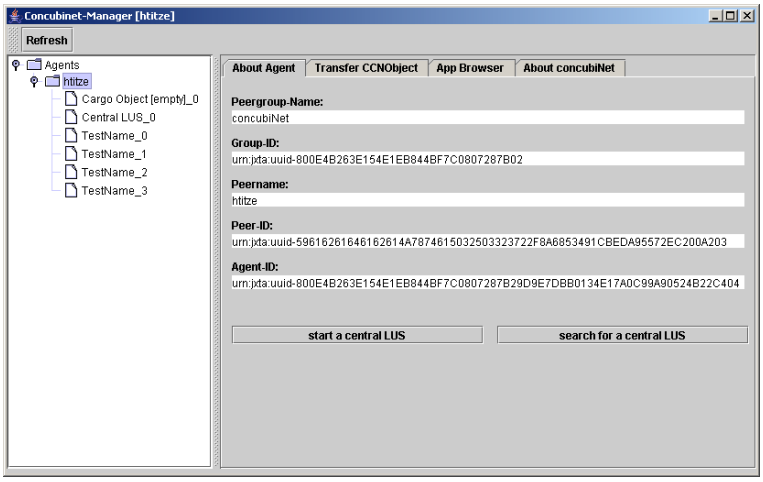


Figure 38: PeerManager-GUI - Concubi*Net*-Manager

6.4.1 Configuration Tags

- **<ccnagent><classesDirectory>**
  The value of this element specifies the path, where the FileClassRepository will store bytecodes received from other PeerManager. This may be Java archive files (.jar) containing class files or plain class files in a subdirectory, representing the class's package structure. The path is interpreted as a subdirectory of the current working directory.

  You also can put classes (or JARs) of a new CCN-object into this directory to start it on a running PeerManager. But if you want to do so, don't forget to edit the **<ccnagent><ccnobjects>**-tag.

- **<ccnagent><systemproperties>**
  All elements inserted here will be available during runtime by the properties container accessible via System.getSystemProperties(). A detailed description of the system properties is given later.

- **<ccnagent><ccnobjects>**
  This tag is to define the CCN-objects that shall run on this PeerManager. You could increase the count of instances of a CCN-object by changing the value numberOfInstances.

  If you want to start a CCN-object not known by the PeerManager yet, you have to put its classes (or alternatively a JAR file containing its classes) into the classesDirectory and insert a corresponding tag into the configfile.
  In **<ccnagent><ccnobjects><ccnobject>wrapperclassname** you have to specify the name of your CCN-object's wrapper, in numberOfInstances the number of instances of CCN-objects you want the PeerManager to start up.

- **Caution:** If you want to instantiate classes of a CCN-object not known to the virtual machine yet during runtime, be sure you have inserted a complete ccnobject-tag with both sub tags (numberOfInstances and wrapperclassname) before you save the configfile.

6.4.2 System-Properties

Furthermore the configuration file gives the possibility to add or modify system properties. The following system properties are used by the current system.

- **<guiclass>**
  org.concubinet.agent.gui.remote.Mainframe</guiclass>
  In this place you can define which GUI-class will be used if you start the PeerManager. By now there are three possible values.
  Remote-GUI:
  org.concubinet.agent.gui.remote.Mainframe
  Swing-GUI:
  org.concubinet.agent.gui.swing.Mainframe
  SWT-GUI:
  org.concubinet.agent.gui.swt.Mainframe
- **<useTray>true</useTray>**
  Enables (true) or disables (false) the Tray Icon. The Tray Icon is only available for Windows.
- **<showSplashScreen>true</showSplashScreen>**
  Enables (true) or disables (false) the starting splash screen.
- <**stdAttributeLocation>**
  PG-Room </stdAttributeLocation>
  <stdAttributeTimestamp>
  June, 2003 </stdAttributeTimestamp>
  <stdAttributeName>
  Test Peer Manager </stdAttributeName>
  <stdAttributeOwner>
  PG441</stdAttributeOwner>
  <stdAttributeLastAgent>
  None </stdAttributeLastAgent>

  Some standard attributes for the PeerManager.
- **<pathtosrc>C:**
  concubiNet
  src</pathtosrc>
  Points to the place where the source files are located. This entry is used by the Wrapper-Factory.
- **<javacpath>C:**
  j2sdk1.4.2
  bin
  javac</javacpath>
  Points to the place where the Java Runtime is located. This entry is used by the Wrapper-Factory.
- **<ccnCommunicationmodul>**
  org.concubinet.core.net.upnp.UPNPManager</ccnCommunicationmodul>
  In this place you can define which Communication Modul will be used by the PeerManager. By now there is one possible values.
  UPnP:
  org.concubinet.core.net.upnp.UPNPManager

- **<classesdirectory>classes</classesdirectory>**
  Points to the place where temporary files can be stored by the PeerManager.

6.5 Sample Application - concubiChat

This manual is intended as an overview of all functions the chat GUI is capable of. The chat GUI itself is an application which extends the CCN-object and therefore supports all of its functions, e.g. movability through the network.

6.5.1 Chat Tab

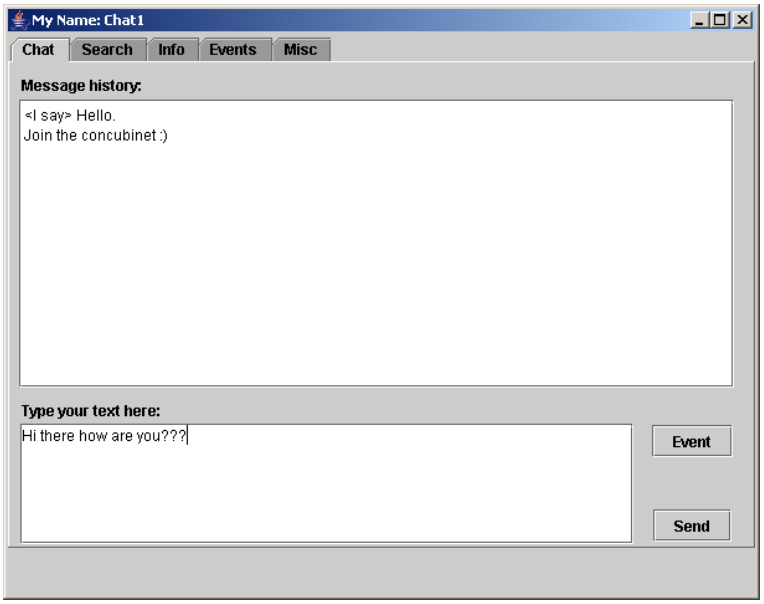


Figure 39: ChatGUI - Chat Tab

Before the chat tab can be used for chatting, a connection has to be established. See subsection “Search Tab” on page 74 first.

The main part of the chat tab consists of the “Message history” field, which contains all the text typed by oneself, indicated by the prefix `<I say>`, plus the received text sent by another user of another chat GUI, displayed in red color and additionally indicated by the prefix `<NameOfOtherUser>`.

Below the “Message history” a text field is placed where ones own text can be entered. By clicking on the button “Send” or by pressing CRTL+Enter the text is sent to every other connected chat GUI.

The second button “Event” is used for eventing (see page 76). When clicked an event is fired and every connected listener will be noticed.

6.5.2 Search Tab

The search tab contains four text fields and a check box for searching and two fields containing the search results and further information. Furthermore a connection to another chat GUI can be established.

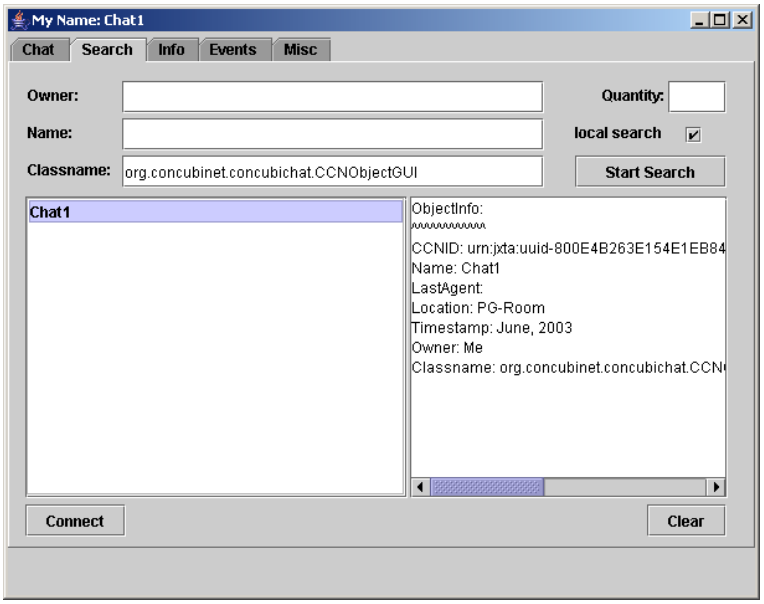


Figure 40: ChatGUI - Search Tab

The fields “Owner”, “Name” and “Classname” specify the attributes set in the ObjectInfo, which is a part of every CCN-object. The field “Quantity” limits the number of objects which are found and displayed. At a last option the check box “local search” indicates whether the search is performed local or not. If a “local search” is checked, only the own PeerManager is searched for the requested objects. When not checked, two ways of search are possible:

1. Distributed search: if no Central LUS is running, a distributed search is performed, asking all neighbor PeerManager.
2. Central search using Central LUS: if the Central LUS is already running, it will be asked and the search time will be usually should shorter.

Of course the search itself is started by pressing the “Start Search” button.

The two fields below this “input mask” display the results of the search. In the left field the name(s) of the specified object(s) is(/are) shown. When one of these list entries is selected, the ObjectInfo belonging to this object is displayed in the list on the right hand side as well.

At the bottom of this tab the “Connect” button is placed. When a search result in the list above is selected, the connection to this chat GUI will be established. The “Clear” button clears all displayed search results and the displayed ObjectInfos.

6.5.3 Info Tab

The info tab displays additional information belonging to one’s own chat GUI. Furthermore, it may be used to modify several attributes.

The upper two text fields display the ObjectInfo (left text field) and the ResourceDescription (right text field), which can be updated by gaining focus, which means to click into

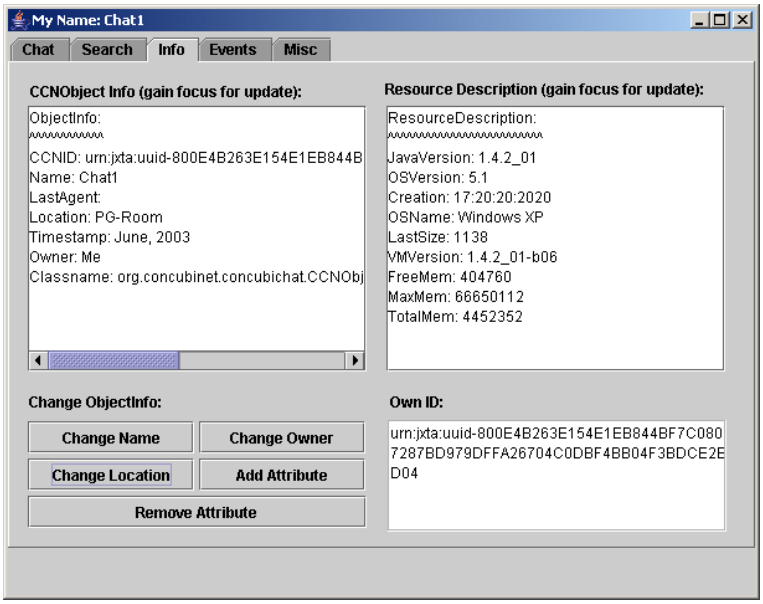


Figure 41: ChatGUI - Info Tab

the desired text field using the mouse. The bottom right text field displays the own CCNID, which is part of the ObjectInfo, again.

At the bottom left several buttons can be used to change the corresponding attributes in the ObjectInfo. With the buttons “Add Attribute” and “Remove Attribute” one is able to add and remove new attributes which may be needed later.

6.5.4 Event Tab

The event tab is needed for displaying subscribed listeners as well as connected wrappers.

In the left-hand list, connected wrappers are displayed by name. When selecting one, it is possible to either subscribe for eventing by pressing “Subscribe” or disconnect from it by pressing “Disconnect” (no more chatting is possible until a new connection is established). Furthermore the former connected chat GUIs are informed by a short message. The updated list of connected wrappers is displayed by pressing the left “Update” button.

In the right-hand list, subscribed wrappers are displayed. When selecting a wrapper of this list, it is possible to unsubscribe it by pressing the “Unsubscribe” button. Here the former subscribed chat GUIs are informed by a message as well. The “Clear” button clears this list and the right “Update” button displays the updated list.

For instructions how to fire an event see section “Search Tab” on page 73.

6.5.5 Misc Tab

The misc tab only displays the concubi*Net* logo. It has just been included for the ease of adding more features to the GUI in the future.

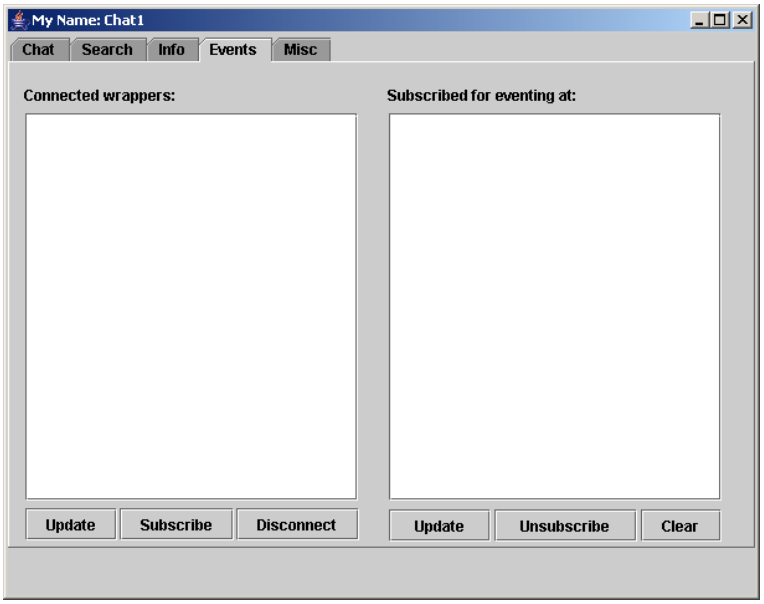


Figure 42: ChatGUI - Event Tab

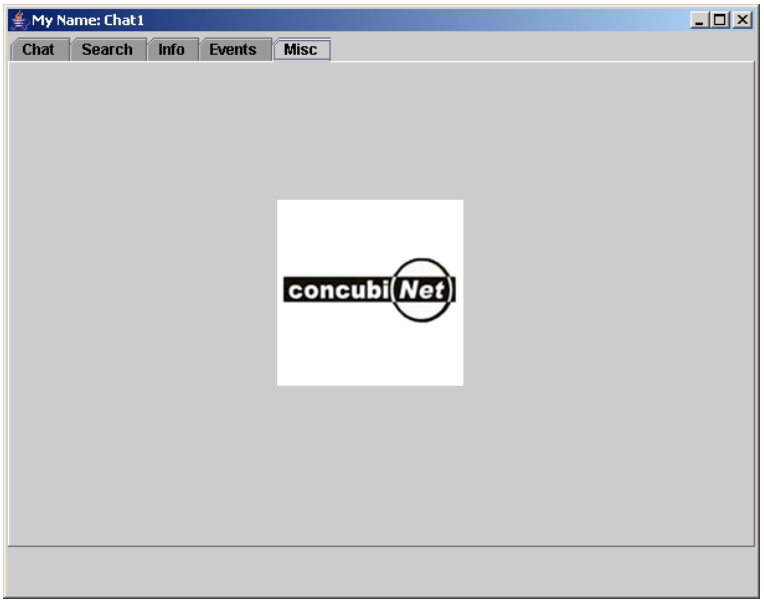


Figure 43: ChatGUI - Misc Tab

6.6 Sample Application - Cargo Object

The CCNCargo-object is another small sample application. The user could load this object with any file. This is done by pressing the button "Load". Figure 44 shows a screens-hot of the CCNCargo-GUI. The user then chooses any file he wishes that the CCNCargo-object should carry. The information of the selected file will be displayed. The user has now the possibility of transferring this object to another PeerManager in

the neighborhood by using the PeerManager-GUI.

Figure 44: CCNCargoObject-GUI

6.7 Sample Application - ccnPIM

Another sample application for Concubi*Net* is our Personal Information Manager called ccnPIM. It consists of two tabs, an address tab to administrate contacts and a note tab to store memos. In both tabs you can export and import your list of contacts or notes, add, change or delete items. Of course you can also send contacts or notes to other users and they can insert them into their list.

6.7.1 Address Tab

The first tab is the address tab. Here you can manage your contacts including forename, surname, address (street, zipcode, city, ...), telephone number, cell phone number and other items.

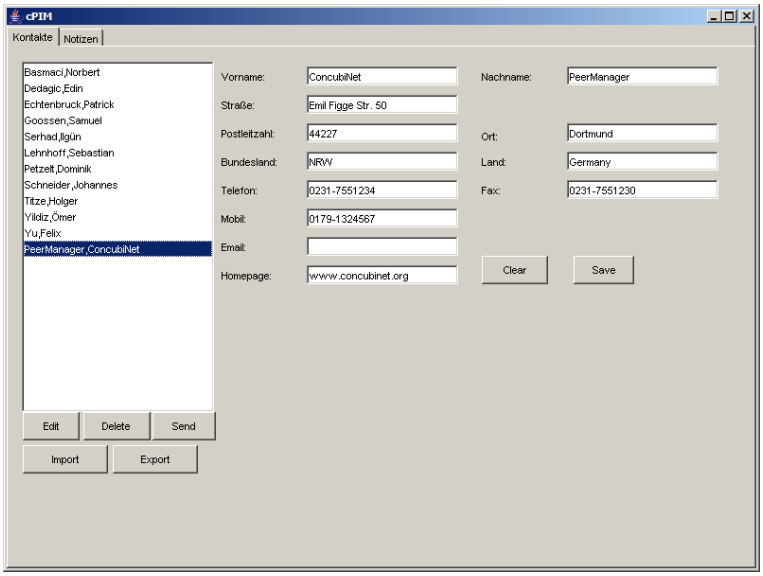


Figure 45: Address Tab

On the left you have a list of stored contacts. When you select a contact and click on the edit button you can see all items on the right side. When you change some entries and click the save button a new contact will be created and shown in the list on the left side. To delete a contact just select it by clicking it once and then hit the delete button.

To import or export your contact list just click the accordant button below the contact list. In the following dialog you can select a file to load or choose a location where to save your list.

To send a selected contact to a Personal Information Manager running on another PeerManager just click the send button. A new window opens which includes all items of the selected contact. Click on update to get a list of neighboring PeerManagers. Select a PeerManager and click on send to transfer the send window including all items to the selected PeerManager. It is also possible to modify some items before sending them. When you want to insert a received contact into your contact list, just click on "save to PIM".

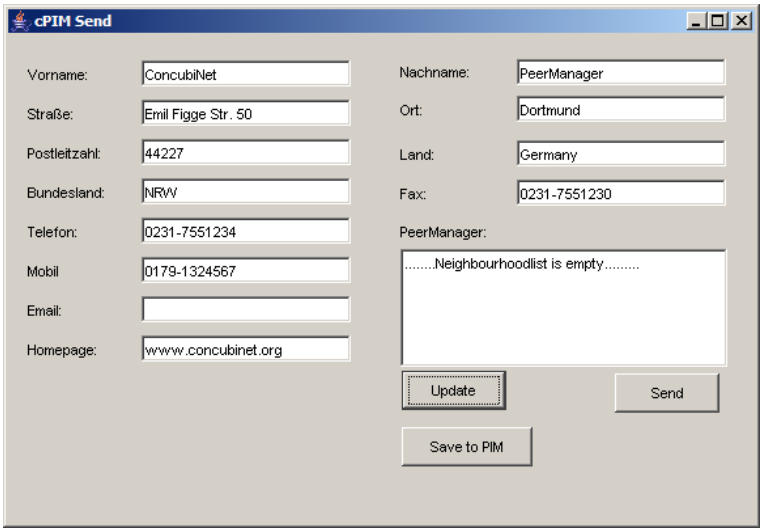


Figure 46: Address sendGUI

6.7.2 Note Tab

The second tab is the note tab. Here you can write down memos and tasks. Each note contains a caption, a date and an additional text.

Again, on the left side you have a list and on the right side all items of a selected note can be shown. To create a new note click on the new button. Now you can enter all items on the right side. The date is automatically set to the current date but it also can be modified. Click on save to save your note or click on discard to discard it. When you select a note in the list on the left, you can chose to view all items, to change them or to delete a complete note.

By clicking on Export your list gets saved to your Concubi*Net* user directory. With a click on Import you can load the list from your Concubi*Net* user directory.

To send a note to another user chose one from the list and click on view to view all

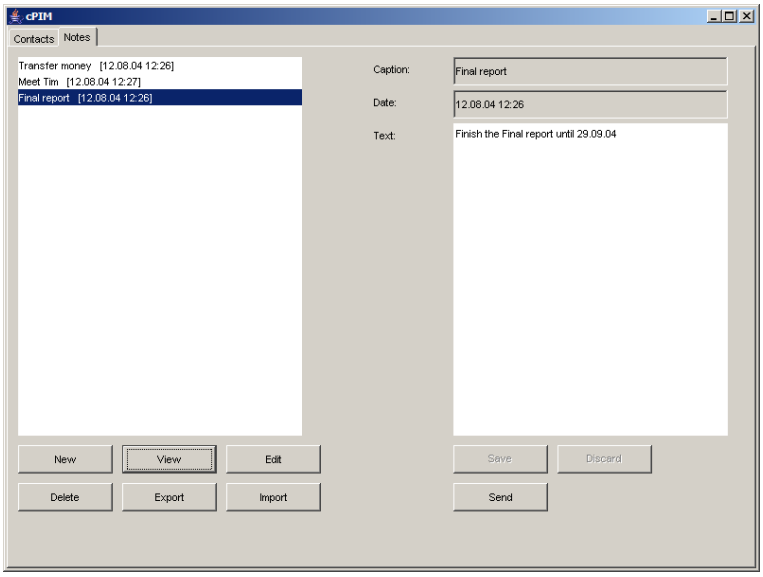


Figure 47: Note Tab

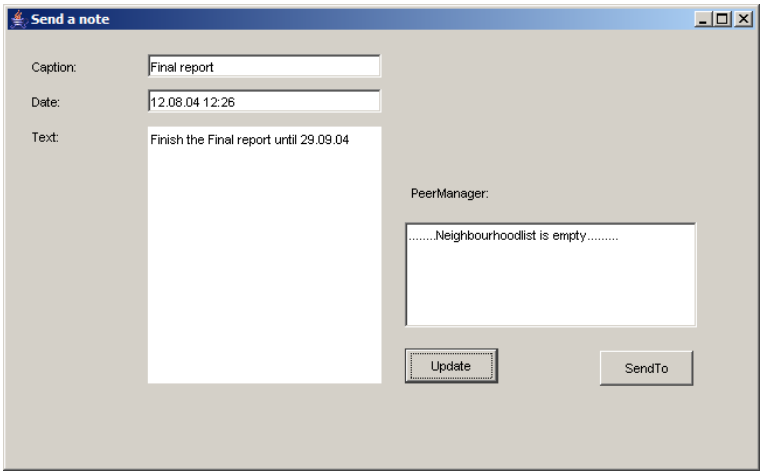


Figure 48: Note sendGUI

items on the right side. Now you can chose the send button to bring up the note-SendGUI. The rest is the same as described for the address tab.

6.8 Sample Application - TransferObject

We created an application that moves CCN-objects targetless from one PeerManager to another and exactly determines how much time it takes. To solve this problem we used already implemented classes and methods from CCN-object and PeerManager.

In order to start the TransferObject, need you to add the follow lines to the "config.xml":

```
<ccnobjects> <ccnobject>
    <ccnclassname>org.concubinet.transfer.TransferObject</ccnclassname>
```

```
        <numberOfInstances>1</numberOfInstances>
    </ccnobject>
</ccnobjects>
```

This tool automatically starts in your PeerManager. If you press the shoot button the application moves from the current PeerManager to an arbitrary neighboring PeerManager witch is currently running. From now on the application jumps from one PeerManager to the next, rests there for 3 seconds and then again jumps to another PeerManager. To stop its movement just press the stop button. After this movement you can see the following information:

1. Number of PeerManagers
2. How much time the transfer takes

If you want this application to run on its own, you may uncomment the lines 274 and 279:

```
//autoTransfer();
//timer.wait();
```

and comment line 272:

```
transferObject();
```

References

[1] Final report of Project Group 421, University of Dortmund, September 2003