ACE

a collaborative editor

Report Evaluation Network

Berne University of Applied Sciences
School of Engineering and Information Technology

Date:	14.06.2005
Version:	0.6
Projectteam:	Mark Bigler (biglm2@hta-bi.bfh.ch) Simon Räss (rasss@hta-bi.bfh.ch) Lukas Zbinden (zbinl@hta-bi.bfh.ch)
Receivers:	Jean-Paul Dubois (doj@hta-bi.bfh.ch) Claude Fuhrer (frc@hta-bi.bfh.ch)
Location:	Subversion Repository

Contents

List of Tables

List of Figures

1 Introduction

The goal of the semester project is to create a fully functional operational transformation algorithm. We are implementing the *Jupiter* algorithm. In the diploma project we will (most likely) build a collaborative text editor that uses this algorithm. One of the most central things to implement will be a network layer.

The purpose of this document is to examine existing networking solutions and to gather information so that we can decide how to implement the network layer of *ACE* . It is not a goal to select a network technology for *ACE* yet. This decision is delayed further.

1.1 Requirements

The network layer will need to provide the following two basic features:

- discovery
- communication

Let us explain these two basic points in greater detail.

1.1.1 Discovery

In *ACE* a user will be able to share a document so others can join him editing this document. Without a way of first advertising a shared document and then discovering it, the application is pretty useless. There are several technologies that allow automatic discovery of services, for instance *Jini* (see section 5), *Bonjour* (also known as zero-conf networking, see section 4) and diverse peer-to-peer frameworks (e.g. *JXTA*, see section 6).

1.1.2 Communication

When a user discovered a shared document and decided to join that document, there must be a way to communicate between the document publisher and the user that wants to join the document. There are different technologies ranging from plain socket communication over remote method invocation to complex peer-to-peer frameworks. They differ in several aspects such as ease of use and platfrom independence.

The *Jupiter* algorithm does not need a network layer that is capable of multicast. Each operation is sent to the server only. Clients must not be explicitely aware of other users editing the same document from the viewpoint of the algorithm. That is why some network technologies targeting multicast communication are most likely not an appropriate choice (although they would be for an algorithm that requires multicast messaging).

1.2 Overview

In the following sections we will discuss several network technologies:

- JGroups (see section 3)
- Bonjour (see section 4)
- Jini/RMI (see section 5)

- JXTA (see section 6)
- BEEP (see section 7)

Each section will introduce the general details of the technology. If there is a prototype for this technology, it will be explained. Next an analysis of the technology and its suitability for *ACE* is given, containing good, bad and open points. The open points (if there are any) describe things we should investigate further before we decide to use that particular technology. Finally, there is a section with links to more information in the resources section.

2 General Aspects

Ideally, every user could publish a document and that document could be discovered by any other users no matter where they are. However, the Internet poses some issues that make this ideal world hard to accomplish. In the following sections we will first analyze the fundamental problems we face when deploying a distributed application on the Internet and then propose some ways how one could deal with them.

2.1 Firewalls/NAT

Most Internet users today do not have public IP addresses. They are behind a NAT (network address translation) router and sometimes behind firewalls that they cannot control too (e.g. corporate users). If a user wants to publish a document, there must be some kind of server that accepts connections for the given document. If the server is behind a NAT router, there is usually no way an external client can connect to that document server. If the server is behind a firewall, the situation is even more problematic as the firewall will most likely block most incoming traffic. The basic dilemna with NAT routers and firewalls is depicted in figure 1.

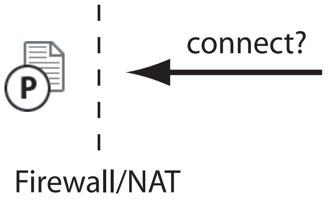

Figure 1: Basic Problem with Firewalls and NAT Routers

2.2 Tunneling Firewalls

It would be possible to tunnel firewalls, although this technique is a bit delicate. Firewalls are there for a reason. Tunneling firewalls is most likely against the will of the security policy of a company. For this reason this is not considered to be a requirement for our application.

2.3 Dealing with NAT Router

The problem with NAT routers is, that nobody from the outside can connect to a computer on the inside because the computer on the inside does not have a public IP address. Of course it would be possible to set up the router in a way that it delegates requests for certain services to one particular computer, but we do not consider this as an acceptable technique for an end user application. We cannot expect end users to do this setup. The solution to this problem is, that first the client must establish a connection to a host outside of the local network.

2.3.1 Proxy Host

Let us say there is a host X, called *proxy*, and a host A, called *publisher*. Host A sits behind a NAT router and wants to publish a document. Host A connects to the proxy host X which publishes the document in stead of A. Every other client that wants to join the document published by A connects to X. X acts as a proxy and forwards all messages for that published document to A (using the connection opened by host A). X also forwards any responses/messages from A to all the other clients of this editing session. This solution to the NAT problem is depicted by figure 2.

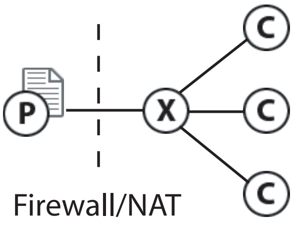

Figure 2: Proxy Host

2.3.2 Substitute Host

Let us say there is a host X, called *substitute*, and a host P, called the publisher. Host P again sits behind a NAT router and wants to publish a document. It connects to the substitute host X, which publishes the document and hosts the document itself. So host P is no longer the publisher of the document, but merely an ordinary client. Host X plays the substitute of host P because he has a public IP address whereas host P has not.

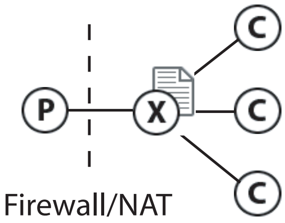

Figure 3: Subsitute Host

The substitute host solution to the NAT problem is depicted by figure 3. It is very similar to the proxy host solution. The only difference is, that the substitute host is itself the server whereas the proxy host is only a proxy forwarding all requests back to the publisher host P.

2.4 Document Publishing

When a user publishes a document, he wants other users to find it. There must be some kind of discovery mechanism that allows users to discover published documents. There are several

solutions to this problem.
On the LAN clients could use *IP multicast* to discover documents (see section 5 and section 4). IP multicast usually fails in WANs because routers block multicast traffic. A common and relatively straightforward solution for WANs would be to have a (or several load-balancing) *central server* that manage all the document advertisements. A pure *P2P solution* could build upon a P2P framework (see section 6) to avoid central servers at all.

3 JGroups

JGroups is a toolkit for reliable multicast communication. It can be used to create groups of processes whose members can send messages to each other. The main features include:

- Group creation and deletion (members can spread across LANs or WANs).
- Joining and leaving groups.
- Membership detection and notification about joined/left/crashed members
- Detection and removal of crashed members
- Sending and receiving of member-to-group messages (point-to-multipoint)
- Sending and receiving of member-to-member messages (point-to-point)

One of the most powerful features of *JGroups* is its flexible protocol stack, which allows developers to adapt it to exactly match the application requirements and network characteristics. The benefit of this is that one has only to pay for what is actually used. By mixing and matching protocols, various differing application requirements can be satisfied. *JGroups* comes with a number of protocols, for example

- Transport protocols: UDP (IP multicast), TCP, JMS
- Fragmentation of large messages
- Reliable unicast and multicast message transmission. Lost messages are retransmitted.
- Failure detection: crashed members are excluded from membership
- Ordering protocols: atomic (all-or-none message delivery), FIFO, causal, total order (sequencer or token based)
- Membership
- Encryption

3.1 Concepts

In order to join a group and send messages, a process has to create a channel. A channel is similar to a socket. When a client connects to a channel, it gives the name of the group it would like to join. A channel is always associated with a particular group (in it's connected state). The protocol stack takes care that channels within the same group find each other. Whenever a client connects to a channel with group name G, it tries to find existing channels with the same group name and joins them. If no members exist, a new group will be created. When a channel is created, it is first in the unconnected state. The clients connects to the channel by calling `connect` supplying the group name of the group it wants to join. Once the channel is in the connected state, messages can be sent/received. A channel can be disconnected from a group by calling `disconnect` and it can be closed by calling `close`. A closed channel cannot be used anymore. Any attempt to do so results in an exception. The channel can only be connected to one group at the time. To communicate with multiple groups, multiple channels have to be created.

3.1.1 Creating a channel

To create a channel we can use the public constructor of `JChannel`.

```
public JChannel(Object properties) throws ChannelException
```

The `properties` argument defines the composition of the protocol stack, that is the number and types of layers, their parameters and their order. For `JChannel` this has to be a string. For details about the composition of this string, please refer to the *JGroups* users guide.

3.1.2 Sending messages

One of the two `send` methods of a channel is used to send messages to group members. The message payload can be any serializable object.

```
JChannel channel = ...;
channel.send(null, null, "test");
```

The above code fragment sends the string "test" to all the other members in the group.

3.1.3 Receiving messages

The `receive` method is used to receive messages. A channel receives messages asynchronously from the network and stores them in a queue. When `receive` is called, the first message in the queue is removed and returned. If there is no message in the queue, the method invocation blocks.
There is a so-called *building block* that provides a listener interface for receiving messages. The building block is called `PullPushAdapter`. This relieves the application developer from the task to create a separate reception thread.

3.1.4 Transport Protocols

A transport protocol refers to the protocol at the bottom of the protocol stack which is responsible for sending and receiving messages to/from the network. There are a number of transport protocols in *JGroups*.

- The transport protocol UDP uses IP multicast by default but it can be configured to use multiple unicast messages instead of one multicast message. For discovery of initial members, the PING protocol can be configured to use a well-known server (*GossipServer*).
- The TCP transport protocol can be used where UDP is undesired, most likely over WAN, where routers discard IP multicast. The TCPPING protocol can be used to determine initial group membership. No external gossip server is needed with this protocol, but other members have to be known in advance. The TCPGOSSIP protocol is essentially the same as the PING protcol for UDP but only for TCP.
- The TUNNEL transport protocol can be used to tunnel firewalls. It needs a router server running outside of the firewall to tunnel the firewall.
- The JMS transport protocol can be used to send messages via JMS.

3.2 Prototype

The simple chat client (see `ch.iserver.ace.net.jgroups.SimpleChat`) is based on *JGroups*. When started, it asks the user for its nick name. Then it displays the GUI. The GUI has a *join* button that allows the user to join a group. The button prompts the user for a group name. The application connects to the group and enables the text field at the bottom of the window. The user can then send messages to the group. Other members of the group receive these messages.
The prototype does not have any discovery mechanism for existing groups. The users must agree on a group name outside of the chat application. For a real chat application, this would be of course unacceptable.

3.3 Analysis

JGroups provides a very simple API to create groupware applications. A shared document in *ACE* could be represented as a separate group in *JGroups*.
It is used in famous open source projects like Tomcat (session replication), OSCache (J2EE caching solution) and Jetty (session replication). *Glicpse*, a collaborative editor plugin for *Eclipse* uses *JGroups* too for multicast communication among sites. The main reason why we examined *JGroups* was because of *Gclipse*.
JGroups provides both discovery and communication, the requirements for the network layer. Although, the discovery part would need some tweaking before it can be used, as one has to know the group name to join a group.

3.3.1 Positive Points

JGroups provides a very simple API to do multicast messaging. Furthermore it's configurable protocol stack allows a great deal of flexibility.

3.3.2 Negative Points

For an operational transformation algorithm that needs to broadcast operations to all other sites, *JGroups* would certainly be suitable (e.g. *GOTO* or *adOPTed*). Note however that our chosen algorithm (*Jupiter*) does not need to broadcast messages. Multiple unicast connections are used between one server and several clients. Because of this aspect, we think to implement the network layer, *JGroups* is not the best solution. We would need to teach *JGroups* not to do group communication (and that is the prime feature of *JGroups*).
The *JGroups* library is not programming language independent. It uses serialized objects as message payload. This can be seen as and advantage or as a disadvantage depending on the requirements of the application.

3.3.3 Open Points

The current implementation of the group membership service is very simplistic. Joining a group is always successul. There is no way to forbid a particular member to access the group. This is a serious problem as the confidentiality of the document content cannot be guaranteed. A custom protocol implementation could solve this particular problem. This would need to be investigated in more detail.
As mentioned, to join a group, one has to know the group name beforehand. This is a serious shortcoming. There are several solutions possible. One would be to create a well-known group (e.g. a group named "ACE") that every application instance is a member of. Peers in this group could then be queried about available published documents. Another solution would be to have a central server that keeps track of which documents are published. The second solution seems to be a bit clumsy as one has to setup a separate server.

3.4 Resources

- *JGroups* website
- *JGroups* users guide

4 Rendezvous / Bonjour

Bonjour, formerly known as *Rendezvous*, enables automatic discovery of computers, devices, and services on IP networks. It uses industry standard IP protocols to allow devices to automatically discover each other without the need to enter IP addresses or configure DNS servers. It is a technology developed by Apple that is submitted to the IETF as part of an ongoing standards-creation process. The technology is more generally known as zero-conf networking.
There are opensource *Java* libraries available that allow to use this technology in any *Java* application. The library needs a native library that is available for all major platforms.
DNS service discovery is a way of using standard DNS programming interfaces, servers, and packet formats to browse the network for services. It is compatible with, but not dependent on, multicast DNS. Multicast DNS is a way of using familiar DNS programming interfaces, packet formats and operating semantics, in a small network where no conventional DNS server has been installed.

4.1 Prototype

The technology is best described by a simple prototype. We've created a simple echo server and echo client that show off the basic features of this technology.

4.1.1 Echo Server

The echo server is implemented in the class `ch.iserver.ace.net.bonjour.EchoServer`. It is very similar to normal echo server implementation. First a `ServerSocket` is created listening on a particular socket. Then the service is registered with DNS service discovery system represented by the `DNSSD` class. A listener (`RegisterListener`) is used to inform the application when the registration was successful. Once the registration was successful, the echo server starts to service clients as any echo server would do.

4.1.2 Echo Client

The echo client (see `ch.iserver.ace.net.bonjour.EchoClient`) is a simple GUI that allows to send echo requests to echo servers. It browses the network for available echo servers and lists them in a combo box. The user can select one of those echo servers and send an echo request to them.

The method `browse` of the `DNSSD` class is used to issue a browse request to the DNS service discovery system. We have to supply a `BrowseListener` implementation that gets called back whenever a service of specified type is found.

Whenever the user sends a message to the currently selected echo server by entering a message in the bottom textfield and hitting enter, the service has to be resolved. This happens by calling `resolve` on the `DNSSD` class. The application is called back over a `ResolveListener`. The method `serviceResolved` is called on this listener when the service is resolved. This provides us with the service name, host name and port of the service. As a last step we have to get the IP address of the service. This is done by calling `queryRecord` on the `DNSSD` class. We get called back by a method call to `queryAnswered` in the `QueryListener` we supplied to the `queryRecord` call.

At that point we do have enough information to contact the echo server and send the echo request over a plain socket connection. That is one important point to observe. This technology enables us to discover services on the network but does not impose any limitations on the way we communicate from client to server.

4.2 Analysis

Bonjour technology provides discovery of other services on a LAN. It leaves the communication protocol completely up to the service. So for instance a client can discover a HTTP server and communicate by sending HTTP requests with the server. He can discover a SSH server and then communicate by talking the SSH protocol. So *Bonjour* does not provide any support for communication.

SubEthaEdit, the only commercially available collaborative editor for OS X, uses *Bonjour* to discover other published documents. Once other documents are discovered, SubEthaEdit uses *BEEP* (see section 7) using a proprietary protocol to communicate with other instances of the application. This combination could also be a viable alternative for our network layer implementation.

4.2.1 Positive Points

Bonjour provides a platform and programming language independent way of discovering other services on the network. It is on its way of becoming an IETF standard. This technology will remain available for the years to come.
This would allow other applications programmed in other programming languages to interoperate with *ACE* . Depending on our requirements, interoperability could be a very important point.

4.2.2 Negative Points

Bonjour discovery gets you only a host name plus a port. Of course, this information can be used to open a socket, but other technologies provide you simpler ways of communicating with each other (e.g. *RMI*).
The discovery is further limited to the LAN. Although it would be theoretically possible to extend this behavior, it needed fairly deep understanding of the technology.

4.2.3 Open Points

Implementing discovery using *Bonjour* would be pretty straightforward. There are no open points to examine.

4.3 Resources

- Developer site for *Bonjour* technology
- Article about Java *Bonjour*
- Java API documentation
- Multicast DNS Internet Draft
- DNS Based Service Discovery Internet Draft

5 Jini/RMI

In a running *Jini* system, there are three main players. There is a service, a client which would like to make use of services and a lookup service (service locator) which acts as a broker/locator between services and clients. Clients can query lookup services to dynamically discover services. Services register themselves with a lookup service so that they can be discovered by clients. The communication from client to service happens usually through a service proxy that communicates with RMI to the real service (although it could use any protocol).

5.1 Analysis

Jini/RMI fulfills all our requirements, discovery and communication. Services are discovered through the lookup service. The lookup service is discovered either through unicast connections (in case that the lookup service is known in advance) or through multicast messages. A shared document could be represented as a document service that is registered in the lookup service.

5.1.1 Positive Points

We studied *Jini* and *RMI* in a semester course. So these technologies are known well enough to implement the network layer with them.
A network layer based on *Jini/RMI* would be relatively straightforward. The whole system would be based on Java.

5.1.2 Negative Points

Jini relies on a JVM at each site, although not the whole system has to be implemented in *Java* (e.g. services can be implemented in other programming languages). The need for a JVM at each site is a drawback if we want to make *ACE* interoperable.

5.1.3 Open Points

Jini needs a lookup service to discover services. We must decide which machines run a lookup service. This decision should be transparent to the end user.

5.1.4 Jini vs. Bonjour

At that point, it would be interesting to compare *Jini* with *Bonjour* (a.k.a. zero-conf networking). Both provide ways to dynamically discover services. One obvious difference is that *Bonjour* does not rely on a JVM. Programs written in almost any programming languages can discover services with it, as it is an extension to the domain name system (DNS). This is also one of the most obvious limitations of *Bonjour* technology. *Bonjour* can only discover that what could be stored in a traditional *DNS* server. Usually all that is discovered through *Bonjour* is an IP address and port number. It is then up to the client to know what to do with this information. If it does not know the protocol used to talk with that particular service, the address/port information is pretty useless.
Jini on the other hand downloads a service object (usually a proxy object). The service object itself either does the requested work itself or in case of a proxy knows how to communicate with the real service. All the client has to know is the service interface (a Java interface).
It is this transfer of service object that makes *Jini* so much more powerful than *Bonjour* technology. Interesting enough, *Bonjour* seems to be much more widely adopted than *Jini*, most likely because it does not depend on a JVM.

5.2 Resources

- Jini Website

6 JXTA

JXTA technology is a set of open protocols that allow any connected device on the network ranging from cell phones and wirless PDAs to PCs and servers to communicate and collaborate in a P2P manner.

JXTA peers create a virtual network where any peer can interact with other peers and resources directly even when some of the peers and resources are behind firewalls and NATs or are on different network transports.
The project objectives of *JXTA* are:

- Interoperability - across different peer-to-peer systems and communities
- Platform independence - multiple/diverse languages, systems and networks
- Ubiquity - every device with a digital heartbeat

6.1 Concepts

6.1.1 Peers

A *peer* is any networked device that implements one or more of the *JXTA* protocols. Peers can include phones, PDAs, as well as PCs, servers and supercomputers. Each peer operates independently and asynchronously from all other peers, and is uniquely identified by a peer ID.
Peers are not required to have direct point-to-point network connections between themselves. Intermediary peers may be used to route messages to peers that are separated due to physical network connections or network configuration (e.g. NATs, firewalls, proxies).
Peers are typically configured to spontaneously discover each other on the network to form transient or persistent relationships called peer groups.

6.1.2 Peer Groups

A *peer group* is a collection of peers that have agreed upon a common set of services. Peers self-organize into peer groups, each identified by a unique peer group ID. Each peer group can establish its own membership policy from open (anybody can join) to highly secure and protected (sufficient credentials are required to join).
Peers may belong to more than one peer group simultaneously. By default, the first group that is instantiated is the Net Peer Group. All peers belong to the Net Peer Group. Peers may elect to join additional peer groups.
There are several motivations fro creating peer groups:

- *To create a secure environment*: Groups create a local domain of control in which a specific security policy can be enforced. The security policy may be as simple as a plain text user name/password exchange, or as sophisticated as public key cryptography. Peer groups form logical regions whose boundaries limit access to the peer group resources.
- *To create a scoping environment*: Groups allow the establishment of a local domain of specialization. For example, peers may group together to implement a document sharing network or a CPU sharing network.
- *To create a monitoring environment*: Peer groups permit peers to monitor a set of peers for any special purpose (e.g. traffic introspection, accountability).

A peer group provides a set of services called peer group services. *JXTA* defines a core set of peer group services. Additional services can be developed for delivering specific services. In order for two peers to interact via a service, they must both be part of the same peer group.

6.1.3 Core Peer Group Services

The core peer group services include the following:

- *Discovery Service* - The discovery service is used by peer members to search for peer group resources, such as peers, peer groups, pipes and services.
- *Membership Service* - The membership service is used by current members to reject or accept a new group membership application. Peer wishing to join a peer group must first locate a current member, and then request to join. The application to join is either rejected or accepted by the collective set of current members. The membership service may enforce a vote of peers or elect a designated group representative to accept or reject new membership applications.
- *Access Service* - The access service is used to validate requests made by one peer to another. The peer receiving the request provides the requesting peers credentials and information about the request being made to determine if the access is permitted.
- *Pipe Service* - The pipe service is used to create and manage pipe connections between the peer group members.
- *Resolver Service* - The resolver service is used to send generic requests to other peers. Peers can define and exchange queries to find any information that may be needed (e.g. the status of a service).
- *Monitoring Service* - The monitoring service is used to allow one peer to monitor other members of the same peer group.

Not all the above services must be implemented by every peer group. A peer group is free to implement only the services it finds useful, and rely on the default net peer group to provide generic implementations of non-critical core services.

6.1.4 Network Services

Peers cooperate and communicate to publish, discover, and invoke network services. Peers can publish multiple services. Peers discover network services via the Peer Discovery Protocol. The JXTA protocols recognize two levels of network services:

- *Peer Services*: A peer service is accessible only on the peer that is publishing that service. If that peer should fail, the service also fails. Multiple instances of the service can be run on different peers, but each instance publishes its own advertisement.
- *Peer Group Services*: A peer group service is composed of a collection of instances (potentially cooperating with each other) of the service running on multiple members of the peer group. If any one peer fails, the collective peer group service is not affected (assuming the service is still available from another peer member). Peer group services are published as part of the peer group advertisement.

6.1.5 Pipes

JXTA peers use pipes to send messages to one another. Pipes are an asynchronous and unidirectional non reliable (with the exception of unicast secure pipes) message transfer mechanism used for communication, and data transfer. Pipes are indiscriminate; they support the transfer of any object, including binary code, data strings, and Java technology-based objects.
The pipe endpoints are referred to as the input pipe (the receiving end) and the output pipe (the sending end). Pipe endpoints are dynamically bound to peer endpoints at runtime. Peer endpoints correspond to available peer network interfaces (e.g., a TCP port and associated IP address) that can be used to send and receive message. JXTA pipes can have endpoints that are connected to different peers at different times, or may not be connected at all.
Pipes are virtual communication channels and may connect peers that do not have a direct physical link. In this case, one or more intermediary peer endpoints are used to relay messages between the two pipe endpoints. Pipes offer two modes of communication, point-to-point and propagate. The JXTA core also provides secure unicast pipes, a secure variant of the point-to-point pipe.
A *point-to-point pipe* connects exactly two pipe endpoints together: an input pipe on one peer receives messages sent from the output pipe of another peer, it is also possible for multiple peers to bind to a single input pipe.
A *propagate pipe* connects one output pipe to multiple input pipes. Messages flow from the output pipe (the propagation source) into the input pipes. All propagation is done within the scope of a peer group. That is, the output pipe and all input pipes must belong to the same peer group.
A *secure unicast pipe* is a type of point-to-point pipe that provides a secure, and reliable communication channel.
Since pipes provide unidirectional, unreliable communication channels, it is necessary to implement bidirectional and reliable communication channels. The platform provides the following to address the level of service quality required by the applications.

Reliability Library: Ensures message sequencing, delivery and exposes message and stream based interfaces.

JxtaSocket and JxtaServerSocket: Provides `Socket` and `ServerSocket` implementations built on top of pipes and the reliability library. It exposes the well knwon stream based interface for communication. Furthermore it provides bidirectional and reliable communication channels.

JxtaBiDiPipe and JxtaServerPipe: Is built on top of pipes and the reliability library. It provides bidirectional and reliable communication channels and exposes a message based interface.

6.1.6 Messages

A message is an object sent between *JXTA* peers. It is the basic unit of data exchange between peers. Messages are sent and received by the Pipe Service and by the Endpoint Service. Typically, applications use the Pipe Service to create, send and receive messages.

A message is an ordered sequence of named and typed contents called message elements. Thus a message is essentially a set of name/value pairs. The content can be an arbitrary type.
The JXTA protocols are specified as a set of messages exchanged between peers. Each software platform binding describes how a message is converted to and from a native data structure such as a Java technology object or a C structure.

6.1.7 Advertisements

All *JXTA* network resources - such as peers, peer groups, pipes and services - are represented by an *advertisement*. Advertisements are language-neutral meta-data structures represented as XML documents. The *JXTA* protocols use advertisements to describe and publish the existence of peer resources. Peers discover resources by searching for their corresponding advertisements, and may cache any discovered advertisements locally.
Each advertisement is published with a lifetime that specifies the availability of its associated resource. Lifetimes enable the deletion of obsolete resources without requiring any centralized control. An advertisement can be republished (before the original advertisement expires) to extend the lifetime of a resource.

6.1.8 Security

Dynamic P2P networks such as the *JXTA* network need to support different levels of resource access. *JXTA* peers operate in a role-based trust model, in which an individual peer acts under the authority granted to it by another trusted peer to perform a particular task.

6.1.9 IDs

Peers, peer groups, pipes and other *JXTA* resources need to be uniquely identifiable. A *JXTA* ID uniquely identifies an entity and serves as a canonical way of referring to that entity.

6.2 Prototype

The prototype is a very simple sample application from the *JXTA* programmers guide. There are two classes, `ch.iserver.ace.net.jxta.Server` and `ch.iserver.ace.net.jxta.Client`. The server first advertises a service, which the client discovers and sends a simple message to it. The communication is strictly unidirectional (and unreliable, as it is using pipes). The example could certainly be extended to provide a bidirectional and reliable service (e.g. to create an echo server, client).
The server creates first a new module class advertisement and publishes it both locally and remotely (using the discovery service). Afterwards it creates a module specification advertisement. A pipe advertisement is read from the filesystem and attached to the module specification advertisement. This module specification advertisement is then also published locally and remotely.
The server then creates an input pipe from the pipe advertisement and waits for messages on this pipe in an endless loop.
The client first tries to discover the module specification advertisement published by the server. It queries for this advertisement by specifying a value ("JXTASPEC:JXTA-EX1") for the name attribute of the specification. The advertisement can be already cached locally. In

this case, the client does not need to send a discovery request. If the advertisement is not cached, it sends a discovery request to the network.
Once the client has retrieved the desired module specification advertisement, it can extract the pipe advertisement from the specification advertisement. The pipe advertisement is then used to create an output pipe. The output pipe provides a way of sending a message to the server. When the message is sent successfully, the client shuts down.

6.3 Analysis

JXTA does certainly sound very promising. It provides some features that seem very interesting for our application.
JXTA satisfies both requirements, discovery and communication. It could be used on its own to create the whole network layer for *ACE* .

6.3.1 Positive Points

One of the most interesting features of *JXTA* is its capability to penetrate firewalls and pass through NAT routers. This means that a user can share a document even if he is behind a NAT router and/or a firewall. The advertisement of a shared document can still be found from the outside. No central server is required to achieve this. *JXTA* employs so called router and rendezvous peers, some special type of peers, to achieve this characteristic.
By avoiding the use of a central server, there is no single point of failure. If our editor's network implementation would be based on *JXTA*, no additional setup would be required for users (setup of server). A user could download an application and would be capable of directly using it.
Further *JXTA* is programming language independent. There is an implementation of the *JXTA* protocols for *Java* as well as there is one for *C*. XML is used a lot in protocol messages, although this is not strictly necessary for custom services.

6.3.2 Negative Points

Although *JXTA* exists since 2001 there are not many applications using it. Only a handful applications are listed on the jxta.org homepage.
Another discomforting point is that we have stumbled across a serious bug in a so called stable release (2.3.3). A clone method caused a stack overflow exception and we had to use the latest build. The bug is even observable in an official sample application presented in the programmers guide. It seems to us that *JXTA* is still in some way a research project and not rock solid and mature.

6.3.3 Open Points

Before we could decide to use *JXTA* for the implementation of the network layer, we would need to explore this technology in greater detail:

Performance: We don't have much experience with this technology. Although the capabilities seem great, the question remains whether the overhead introduced by them is acceptable. Two separate aspects would have to be tested: (i) discovery performance and (ii) communication performance. The first point concerns itself with the question, how long it would take to find a published advertisement (e.g. a shared document). The second point concerns the communication throughput and latency once a connection (i.e. a pipe) has been connected.

Application: How would we map the desired network functionality of *ACE* (publish document, join document, leave document) to *JXTA*?

Membership Service: The *JXTA* documents mention that a custom implementation of the membership service can be used to decide who is allowed to join a group (a document) and who is not. Different possibilities exist, e.g. voting in a group or decision of the document owner.

Security: *JXTA* seems to provide security services (e.g. transport layer security). It would certainly be interesting to make all communication within a group confidential.

6.4 Resources

- *JXTA* project website
- free book, slighlty outdated
- Java World article
- article from developer.com

7 BEEP

BEEP stands for Blocks Extensible Exchange Protocol. It is a protocol framework for building application protocols. Many Internet protocols reinvent a set of basic functions. The most common include:

- Framing: separating each request from the next
- Matching responses to requests
- Pipelining: permitting multiple outstanding requests
- Multiplexing: permitting multiple asynchronous requests
- Reporting errors
- Negotiating encryption
- Negotiating authentication

BEEP specifies reusable tools for all these functions, instead of requiring the same decisions to be made over again for each new application. It provides a framework that integrates existing Internet standards for encryption and authentication and new standards for connection management. A networked application needs merely to supply the *important* part, the part that distinguishes it from other applications, as a *BEEP* profile. The mundane parts are the same for all protocols, and therefore can be coded as a library, freeing the application designer to focus on the interesting bits.

The good thing about *BEEP* is that it solves common problems of most connection-oriented application protocols just once inside a framework. Further it is very efficient, requiring only a minimal overhead.

BEEP is an inappropriate protocol if the protocol requires multicast or real-time support (for instance multicast video streaming).

7.1 Concepts

BEEP is a peer-to-peer protocol in the sense that there is no notion of client or server. For convenience we'll refer to the peer that starts a connection as the *initiator*, and the peer accepting the connection as the *listener*. When a connection is established between the two, a *BEEP* session is created.

7.1.1 Channels

All communication in a session happens within one or more *channels*. The peers require only one IP connection, which is then multiplexed to create channels. The nature of communication possible within that channel is determined by the *profiles* it supports (each channel may have one or more).

The first channel, channel 0, has a special purpose. It supports the *BEEP* management profile, which is used to negotiate the setup of further channels. The supported profiles determine the precise interaction between the peers in a particular channel. Defining a protocol in *BEEP* comes down to the definition of profiles.

7.1.2 Profiles

After the establishment of a session, the initiator asks to start a channel for the particular profile or set of profiles it whishes to use. If the listener supports the profile(s), the channel will be created. Profiles themselves take one of two forms: those for initial tuning and those for data exchange.

Tuning profiles, set up at the start of communication, affect the rest of the session in some way. For instance, requesting the TLS profile ensures that channels are encrypted using Transport Layer Security. Other tuning profiles perform steps such as authentication.

Data exchange profiles set expectations between the two peers as to what sort of exchanges will be allowed in a channel, similar to the was Java interfaces set expectations between interacting objects as to what methods are available. A profile is identified by a URI.

7.1.3 Types of Data

BEEP puts no limits on the kind of data a channel can carry. It uses the MIME standard to support payloads of arbitrary type.

7.2 Prototype

7.2.1 Echo Server

The *BEEP* implementation of an echo server can be found in `EchoServer`. The code is straighforward. First a profile registry (`ProfileRegistry`) has to be created. All the supported profiles (`Profile`) have to be added to the created profile registry by calling `addStartChannelListener`. The profiles are registered with an URI that is used as a key to specify a specific profile when creating a new channel.
Once the registry is set up and the profiles added, the server can start listening for client connections. The class `TCPSessionCreator`'s `listen` method is used for this purpose. The server does not need to do anything special with the returned session as all the communication is handled by the selected profile.
The echo server uses the existing `EchoProfile` profile. This profile implements the echo behavior.

7.2.2 Echo Client

The echo client (see `EchoClient`) connects to the echo server and sends echo requests using the BEEP echo profile. First a session has to be created to the server. The method `initiate` in `TCPSessionCreator` is used for that purpose. Once we have a session, we can start a channel by specifying the desired profile URI in a call to `startChannel` on the session.
The `sendMSG` method on the channel allows us to send a message on the channel. We need to specify a `OutputDataStream` and a `ReplyListener`. The `Reply` class implements this interface and allows to retrieve the reply from the server.

7.3 Analysis

BEEP provides a framework for creating new application protocols. It fulfills our communication requirement pretty well, but not the discovery requirement. Other technologies could be used to discover shared documents, for instance *Bonjour*.

7.3.1 Positive Points

BEEP makes it relatively easy to create new protocols. It solves a lot of common problems, for instance multiplexing multiple logical channels onto one connection.
The message format used in channels is completely up to the application. It could be serialized Java objects (whereby one would loose language independence), XML messages or some binary format. There are implementations for Java, C, TCL, Ruby and others. So *BEEP* would allow us to achieve interoperability with other collaborative editors programmed in other programming languages (unless, of course, we would use serialized Java objects as messages).

7.3.2 Negative Points

Creating *BEEP* profiles is a low-level task. One has to deal with streams, bits and bytes.
We would need to develop a protocol specification. All the messages and possible responses would need to be defined (e.g. as XML messages).

7.3.3 Open Points

There are no open points, besides defining the protocol and its messages.

7.4 APEX

APEX is a *BEEP* profile. On top of *BEEP* it adds service discovery, application-level addressing, presence information, and permission management. In other words, it defines how to find a person with whom you are interested in communicating, regardless of where they are, not unlike an electronic mail address does. It also allows discovery of presence information, meaning that one can be notified when a coworker becomes available for a video conference. Finally, it provides a standard mechanism for permissions, specifying a service and database for defining permissions associated with user names, applications and transactions.
From the general description this sound very much like what we need. It covers both discovery and communication. Unfortunately there does not seem to be a *Java* implementation for the *APEX* profile. The corresponding protocol specification is not finalized.

7.5 Resources

- official beepcore website
- A bird's-eye view on beep
- A worm's eye view on beep

8 Conclusions

We have discussed several technologies in this report. This information will be useful when selecting a network technology for the diploma project.

8.1 Selection Criterias

In order to select a particular network technology or a combination of several technologies we have to answer some basic questions.

Open Architecture: Do we want to have an open or a closed protocol. If we decide to have an open, public protocol, other applications could collaborate with our application. We would have to define the protocol and make this information publicly available. An open protocol would greatly benefit if it were not tied to any programming language, that is in our case *Java*. *Jini/RMI* for instance would pretty much limit the collaboration with other applications to *Java* applications. A network layer based on either *JXTA* or a combination of *Bonjour* and a *BEEP* profile would be programming language independent and thus more broadly applicable.

Centralization: We have to decide whether we want to have some kind of central server (possibly several loadbalancing servers) or not. *JXTA* does work without any central server, but does it work good enough (performance, latency, discovery time, ...)? A central server would mean that a central, publicly available server must be maintained by somebody.