Uppsala Master's Theses in
Computing Science No. 299
Examensarbete DV3
2005-06-02
ISSN 1100–1836

Using Reliable JXTA P2P Communication Between Mediator Peers

Teppo Siirilä

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Abstract

The main objective of this thesis is to investigate how JXTA technology can be used for reliable communication between Amos II mediator peers.

JXTA technology is a set of Peer-To-Peer protocols designed that multiple of devices can communicate with each other in a Peer-To-Peer fashion. The idea of JXTA is that any kind of device that can communicate in some way will be able to participate in a JXTA network.

Amos II is an object-oriented mediator/wrapper database system developed at UDBL. Via the wrapper capabilities Amos II can access and query many external data sources including other Amos II nodes. By using mediator systems application design is simplified because the application need only to communicate with the mediator instead of all the external data sources.

A series of evaluation tests were made that measured latency and throughput using both JXTA communication methods and the Amos II communication methods, which uses TCP/IP sockets.

Supervisor : Tore Risch
Examinator: Tore Risch

Passed:

Sammanfattning

Detta examensarbete gick ut på att undersöka hur man kan använda JXTA teknologi för kommunikation mellan olika Amos II noder.

JXTA teknologi är en mängd "Peer-To-Peer" protokoll som möjliggör att man kan koppla ihop väldigt många olika apparater som kan kommunicera på ett "Peer-To-Peer" sätt. Meningen med JXTA är att alla apparater som kan kommunicera på något vis ska kunna vara en del i ett fungerande JXTA nätverk.

Amos II är ett objekt-orienterat mediator/wrapper databas system som utvecklas i UDBL gruppen. En "wrapper" är ett litet program som översätter mellan den externa data källans format och ett gemensamt format som i det här fallet Amos II använder. Genom att använda "wrappers" kan Amos II utföra frågor mot många externa data källor samt andra Amos II noders data. En "mediator" är ett program använder sig av olika "wrappers" för att utföra frågor mot data i externa datakällor. Genom att applikationer använder sig av "mediatorer" kan applikationen göras enklare då den inte behöver ställa separata frågor mot flera datakällor och sedan kombinera deras resultat utan det räcker med att kunna kommunicera med "mediatorn".

Test mätningar gjordes när systemet var implementerad för att mäta och jämföra latensen och "throughput" mellan JXTA metoderna och Amos II metoderna som använder sig av TCP/IP sockets.

Contents

1 Introduction

The use of P2P[1] networks and applications are increasing and P2P technology is used more and more for communication between applications. P2P applications started with instant messaging systems followed by resource sharing applications. To make it easier for developers to write P2P applications, Sun released JXTA[18] technology in 2001 and it has since then developed further as an open source project[9]. JXTA technology is a set of P2P protocols designed in such a way so that a multiple of different devices can communicate with each other in a P2P fashion. The idea is that any device with some kind of communication capabilities will be able to participate in a JXTA P2P network in some way.

Databases are used together with different applications either in a distributed manner or as a stand alone database. Amos II[2][15] is an object-oriented mediator/wrapper[19] database system, developed at UDBL[3], which can act both as a single-user database and as a multi-user server to applications and other Amos II peers. It is light weight and very extensible because of its wrapper capabilities. A wrapper is a piece of software that translates requests and data to an other data format. So via wrappers a database can access data from a multiple of different external data sources. Mediators use these wrappers to access external heterogeneous data sources and make it possible for applications to use these data sources. The use of mediators simplifies applications which needs to access several external data sources by letting the mediator access and query the data. The application needs only to communicate with the mediator in order to access all the external data sources.

The main objective of this thesis was to investigate how JXTA technology can be used for reliable communication between Amos II mediator peers and to evaluate if the JXTA communication is efficient and reliable enough for use in a database system compared to Amos II communication as it is now, i.e. TCP/IP sockets

The approach to this was first to do some research on how JXTA technology works, what kind of methods it supports, how to use JXTA, etc. and to learn about how Amos II works and how to use its query language AmosQL. After that the core JXTA communication methods were implemented incrementally and when those were ready they were made available to Amos II as foreign functions. When the JXTA communication was made available to Amos the evaluation test were made.

[1]Peer-to-Peer
[2]Active Mediator Object System
[3]Uppsala DataBase Laboratory, http://www.it.uu.se/research/group/udbl/

1.1 Report overview

The rest of this report is arranged in the following way: In section 2 the background information is presented. The background information is followed by section 3 in where the architecture of the new JXTAAmos system is described. Section 4 describes the evaluation tests that were made and the results of the tests. Section 5 contains a brief summary of the conclusions along with a brief discussion of future work.

The background section there is a brief description of databases and query languages (section 2.1), Mediator/wrapper systems and Amos II (section 2.3) and finally P2P and JXTA technology (section 2.5).

2 Background

2.1 Databases and Query languages

Databases are simply a collection of data of some kind. This data can be accessed and modified usually via a DBMS[4]. A DMBS is a set of programs and meta-data for accessing the data in a database in an efficient and fast way. Central to the DBMS approach is that every database contains meta-data, called schema, that describes the structure of the database. A data model of a DMBS is the concepts used to describe its schema, i.e. model its data.

Databases can have different kinds of data models[16], which is how the database models the meta-data. The following are data models are the most common ones:

- Relational data model
- Entity-Relational data model[5]
- Object-Oriented data model
- Object-Relational data model

The most widely used data model is the relational data model, in which the data and the relation ship between different data is represented by tables. The E-R model is also widely used. In the E-R model data can be seen as entities and relationship is simply associations between different entities.

There are also other data models that are used, like Object-oriented data model which extends the E-R model with methods and more. When using an object-oriented data model there is no need to flatten out objects into tables when storing them, the objects can be stored directly. Object-relational data model combines ideas from object-oriented data model and relational data model.

Query languages are used to send queries to the DBMS for some particular data. In the early ages of the databases there were many different kinds of query languages, but there has emerged a standard, SQL, which many of todays relational- databases use. SQL which stands for Structured Query Language. Even though SQL is called a query language, but it can also update and manage the database schema. It was designed at IBM in the late 1970's and since then has spread to many other databases. Although it has been made a standard by both ISO[6] and ANSI[7], many database application from companies have added proprietary features to their version of SQL but they all support a subset of SQL.

[4]database-management system
[5]The E-R model
[6]International Organization for Standardization
[7]American National Standards Institute

2.2 Mediator/Wrapper

In order to make it easier to access and query data from different kinds of data sources the mediator/wrapper architecture was thought of in 1992[19].

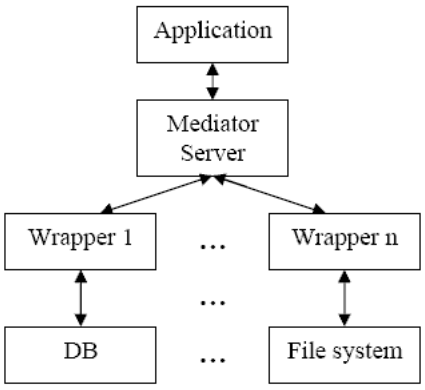


Figure 1: Mediator/Wrapper Architecture

The purpose of the mediator/wrapper architecture is to make it possible for applications to make use of many heterogeneous data sources in a simple way. This way applications only need to communicate with one mediator in order to gain access all the different external data sources and this simplifies the application design.

The mediator/wrapper architecture has one mediator server and may use one or possibly more wrappers (see figure 1). The mediator server has a CDM[8] in which the mediator handles the data. The following part illustrates how a query might go through a mediator/wrapper system.

In the application program, queries are written and sent down to the mediator. Depending on the query the mediator can split up the query to smaller parts and send the smaller queries to one or several of its wrappers or send queries to other mediators which can split it further if there is a need to do that.

When a query arrives at a wrapper, the wrapper translates the query to a data specific form and obtains the wanted data from its data source. After that the wrapper translates the answer back to the CDM and returns it to the mediator. The mediator then assembles the answer from all the wrappers/mediators it has gotten an answer from and sends the whole answer back to the application.

[8]Common Data Model

2.3 Amos Mediator System

Amos II is a distributed object-oriented mediator/wrapper system, developed at UDBL, it can connect to other Amos II clients using TCP/IP sockets and to many external data sources using different wrappers. Amos II consists of a complete lightweight DBMS that is extensible and has a complete query language called AmosQL which is similar to the object-oriented parts of SQL:99. Amos II runs on Windows and Linux systems and has a graphical user-interface written in JAVA which is called GOOVI[9][2].

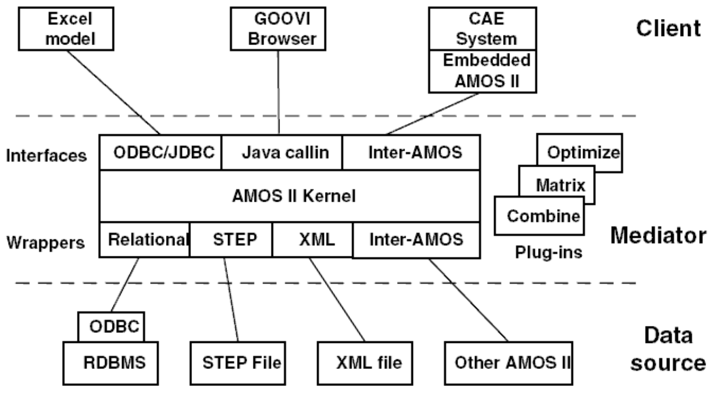


Figure 2: Amos Architecture[14]

Figure 2 shows the architecture of an Amos II server. It is split into three levels. The top level is the application layer, where different applications can access Amos II via its callin interface or other embeddings. The middle level is the mediator layer which consists of the main Amos II functionality such as wrappers. The kernel contains the basic DBMS functionality. The third and lowest layer consists of the external data sources Amos II can handle.

Amos II can access data from three different kind of sources due to its mediator/wrapper capabilities:

- from Amos II internal database
- from external data sources
- from other Amos II clients.

By being able to do this Amos II can access data from many sources, mainly because the ability to access data from external data sources. This makes Amos II appear as a single database system to the user rather than many single databases. It is also this that makes Amos II very extensible. To be able to access and query data from a new data source a new wrapper

[9]Graphical Object-Oriented View Integrator

has to be defined that translates between Amos internal data format and the new external data format. For example wrappers for XML files[11], Internet search engines[10], MIDI music files[7] and others[1] have already been implemented.

2.3.1 External Interfaces

Amos II has external interfaces so it can communicate with external programs written in JAVA, C or lisp, with the the callout interface[4]. External applications can also call Amos II via the callin interface.

Callin The callin interface is used if an external application wants to access Amos II. As mentioned earlier Amos II can be either embedded in the application or have a client/server connection to the application. While the client/server connection can handle several applications at the same time it is also hundreds of times slower than the embedded Amos connection.

Amos II can be accessed in two ways via the callin interface, either by an embedded query method or the fast-path method. In the embedded query method a string is passed to Amos II to be dynamically parsed and evaluated. In the fast-path method specific Amos II methods are called directly by the application. This method is faster than the embedded query method due to the need of parsing the query.

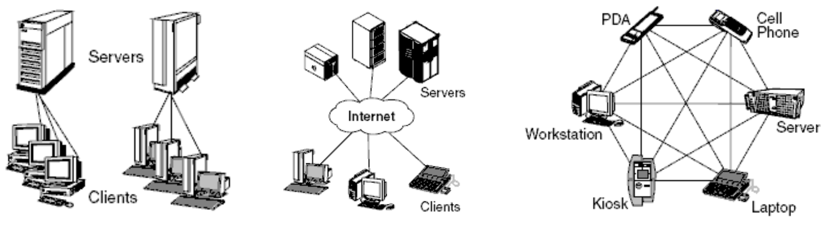
Callout The callout interface is used to access external methods in other languages than AmosQL. The things needed for this to work, is the implemented function, a foreign function in AmosQL that describes how to access the external method and an optional cost hint of the method. After defining the foreign function in a similar way then the call to the external method is as easy as any other call to an Amos function. An example of how to create a foreign function, assumed that you have the correct JAVA classes:

```
create function jlisten() -> Charstring cs as
foreign "JAVA:JxtaAmosAPI/listen";
```

2.4 Peer-to-Peer

The evolution of Internet has gone from homogeneous client-server architecture (see figure 3(a)) to heterogeneous client-server architecture (see figure 3(b)) and now is taking going more and more towards peer-to-peer (see figure 3(c)) architecture.

P2P applications communicate differently with each other than client-server applications. In client-server architecture there is a centralized server to which many clients can send requests for data, i.e. web server and Internet browsers. But in P2P architectures the peers communicate directly with other peers, so there is no need for a central server. This can amongst other things reduce bandwidth in the net since peers can obtain the information from other peers and not from a central server.



(a) Client-Server Architecture (b) Web-based Client-Server Architecture (c) Peer-to-Peer Architecture

Figure 3: Evolution of Internet architecture[13]

The regular client-server model, where several clients connect to one server has dominated the Internet structure since the dawn of the Internet, but P2P applications are used more and more in different applications such as instant messaging clients have used P2P communications to send messages to other clients.

In the recent years P2P applications like Freenet[3], Gnutella[5] and many others have made it possible for users to share resources directly with each other without the need of a central server. JXTA technology wants to change this so that not only desktop computers use P2P networks and applications but also other smaller devices as PDAs[10], phones and others as seen in figure 3(c) can be connected together in a P2P fashion.

2.5 JXTA Technology

JXTA technology is a series of P2P protocols designed so that as many devices as possible can communicate with each other as easily as possible. When using JXTA technology many different devices can make use of it, like

[10]Personal Digital Assistant i.e. a hand held computer or a personal organizer

PDAs, mobile phones, regular workstations, servers and many other types of devices.

JXTA technology was first developed at Sun microsystems, Inc, to make P2P networks and applications more easier to develop. In 2001 the JXTA project became an open source project when Sun released JXTA version 1.0 with many participants all over the world, either researching about it or developing application that uses JXTA technology. Anyone who wants can become involved with the project[9].

JXTA provides methods for peers on the network to find other peers, communicate with them, forming peer groups, searching for resources, etc. Lets begin by looking in more detail at a couple of these things. At the moment JXTA technology defines six different protocols that peers can use[6].

- Peer Discovery Protocol, peers use this protocol to find different advertisements from other peers, and thus join peer groups, etc.
- Peer Resolver Protocol, this allows peers to search for different resources in the JXTA network, like peers, pipes etc.
- Peer Information Protocol, peers can use this protocol to obtain information of other peers, to check that the other peer is alive etc.
- Peer Membership Protocol, is used to assure that a peer is allowed to join a peer group etc.
- Pipe Binding Protocol is used when binding a pipe advertisement to a pipe endpoint.
- Endpoint Routing Protocol is used by peer routers to answer queries from peers that want to know routes to a destination peer.

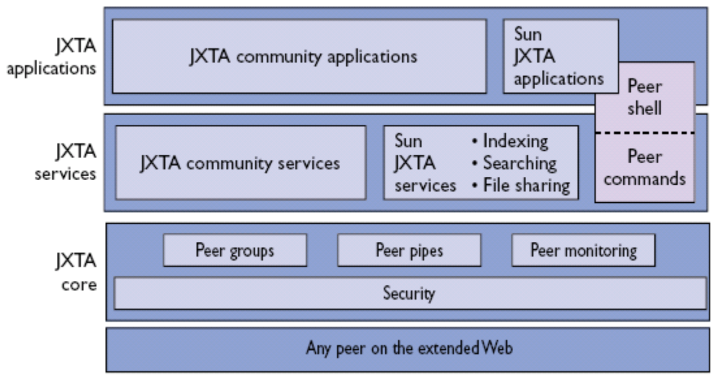


Figure 4: Jxta Architecture[6]

Figure 4 on the previous page show the architecture for JXTA systems. The bottom layer is the core layer. This layer has some basic methods which are needed for basic P2P applications such as creation of peers, peer groups, discovery of other peer and communication methods. The second layer is the service layer. This layer includes network services that might be desired but not necessary for P2P applications, such as searching, indexing, file sharing. The application layer is where the developer applications are such as instant messaging, resource sharing and other applications.

2.5.1 Peers

A peer is a node that communicate in some form with other peers or services using a JXTA protocol. The idea is that any type of device that can communicate in some way can become a peer in the JXTA networks. The peers in a JXTA network does not have to understand all six JXTA protocols in order to participate and function in a JXTA network. Many peers with same interests build up peer groups.

2.5.2 Peer groups

Peer groups are a gathering of peers that have similar interests or similar needs. JXTA provides methods for creating, joining and leaving peer groups.

Any peer can create a new peer group that either are open to all or open only to those that meet a certain criteria. JXTA technology does not describe how or when a peer group should be created. Figure 5 shows an example of how peer groups can be formed and used.

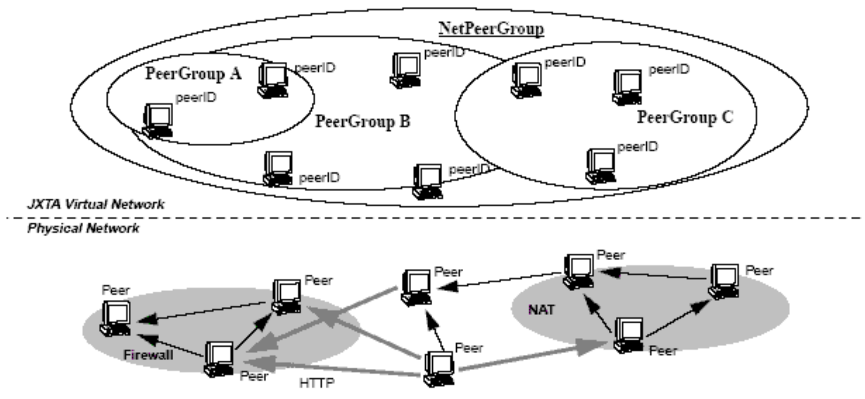


Figure 5: peer group example[18]

So to communicate with other peers, the peers must be members of the same peer group. which leads us into JXTA communication.

2.5.3 JXTA communication

There are several ways for peers to communicate with each other in peer groups. JXTA Pipes are the basic way of communication. They are used to communicate with other peers and with JXTA services. Pipes are channels with which peers can send messages to other peers or more exactly to other peers endpoints. Endpoints are actual input and output channels of a peer. There are two different kinds of pipes[12]:

- Unicast pipes are pipes where a message can only go in one direction. Therefore there are two kinds of pipes, input pipes (the receiving end) and output pipes(the sending end) (see figure 6). Messages sent out on an output pipe will end up in the listening input pipe.
- Propagated pipes are similar but instead of having one input pipe it has many. This means that messages which are sent out on the output pipe will propagate out to all the input pipes that are listening to that output pipe.

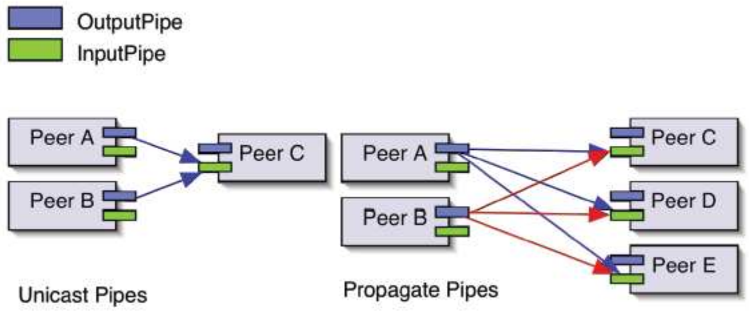


Figure 6: pipe example[8]

As mentioned earlier pipes are the base of JXTA communications, this means that they can be extended to allow other forms of communications on top of the pipe service. An example of this are bidirectional pipes which are built on top of the input/output pipe infrastructure. These function in a similar way that input/output pipes. The difference is that you do not need two separate pipes to send and receive data to/from. Another difference is that bidirectional pipes are reliable. Bidirectional pipes use a message based interface like the regular pipes. There is a limit on the message sizes when using bidirectional pipes, messages can at most be 64KB large.

Another example of extension of the pipes are JXTA Sockets. These sockets behave and work almost the same as regular JAVA sockets i.e. they use a stream based interface to send messages and they also are reliable. JXTA sockets use automatic message chunking which enables to send larger messages than 64Kb through JXTA sockets.

2.5.4 Advertisements

In order to be able to join a peer group, create a pipe or service etc, the peer must have access to an advertisement. Advertisements can be seen as the blueprints of different JXTA objects. These advertisements describe the resource, what name and id it has, how to find it and everything else needed to be able to create an instance of the object or join a peer group. They are represented as XML documents with all the meta information about the resource. An example of an advertisement for a regular input pipe in JXTA:

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
    <Id>
        urn:jxta:uuid-05C3A089241E4C91A3D4AA3345548C7110
3E127EDABF437087CFCF1803F6FE3904
    </Id>
    <Type>
        JxtaUnicast
    </Type>
    <Name>
        AmosPeerGroup:AmosPipe:InputPipe:Amos
    </Name>
</jxta:PipeAdvertisement>
```

The *uuid* string is an identification string for a specific advertisement. When creating an advertisement there is a choice of creating the advertisement with a specific *uuid* number or with a random *uuid* number. But how do other peers create pipes to others if they do not have its advertisements? After creating an advertisement for a pipe, by using the advertisement the peer can create one instance of the object from the advertisement. The peer can also publish the advertisement so that other peers can find it when they do a search for advertisements. Publishing means that the advertisement is propagated out to other peers or to a rendezvous peer so that other peers that only know about the rendezvous peer can find more advertisements.

2.5.5 Services

Peers of peer groups can provide services to other peers. These services are simply a set of methods that the provider offers. Examples of services can be a resource sharing service provided by a resource sharing peer group.

JXTA Services can be divided into:

- Peer Services
  are accessible only on the peer that provides the service. If that peer

fails in some way the service fails with the peer. The provider must publish the service advertisement in the peer group to make it possible for other to make use of it.

- Peer group Services
  are running on multiple peers in the peer group, so it does not matter if one peer fails, the service is accessible from the other peers.

3 Architecture

This section describes how the new module that was implemented during this project works. The following figure (figure 7) gives an overview of the system and how it is built up. The parts that were developed in this project are the JXTAAmosAPI and JXTAAmos that can be seen in the figure 7.

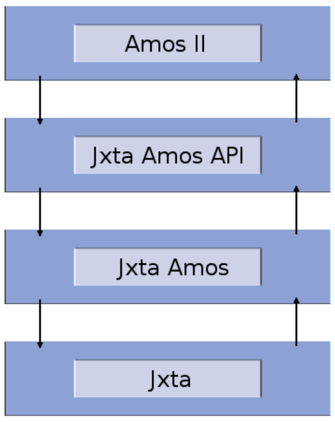


Figure 7: Architecture of the system

The architecture is divided into 4 parts. The top layer is the Amos II level which has been described in section 2.3 on page 8. Via the callout interface and foreign functions Amos II can call functions in the JXTAAmosAPI layer. The API layer calls the next layer, JXTAAmos level, via standard JAVA calls. The bottom layer is the JXTA level, which has been described in section 2.5 on page 10. It is this layer that handles the actual communication between different JXTAAmos peers, like sending and receiving messages, searching for other peers and more.

3.1 API layer

The API layer consists of all the foreign functions that Amos II can call, so this layer only directs the traffic between the Amos II layer and the JXTAAmos layer. The reason for this is explained in section 3.4 on page 19.

3.2 JXTAAmos layer

Most of the important methods are in the JXTAAmos layer. It contains methods like:

- Initialization method. This method initializes the JXTAAmos layer. Creates or joins the peer group, creates all pipes, sockets that are needed and publishes the advertisement so other peers can communicate with this peer.

- Listening. These methods listen for incoming traffic from other peers, and may send back a result depending on the message. There are some "listen" methods that listen on the different JXTA communication channels, i.e JXTA sockets and JXTA bidirectional pipes. These methods block until they are canceled by pressing ENTER in the terminal.
- Send methods. There are two kind of send methods:
  - Send, this is the basic send command. Simply sends a message to another peer.
  - Ship, this method is similar as the basic send command except this method waits for the receiver to send back a result.
- Search for other peers.
- Message chunking for JXTA bidirectional pipes.

3.3 Example

To explain how the system works, lets look at how the send and ship command goes through the system and what happens where. Steps 1 – 10 are the same for the send, ship and broadcast commands, steps 11-16 applies only to the ship command where a reply is sent back to the sending end. Figure 8 shows how the different layers communicate with each other.

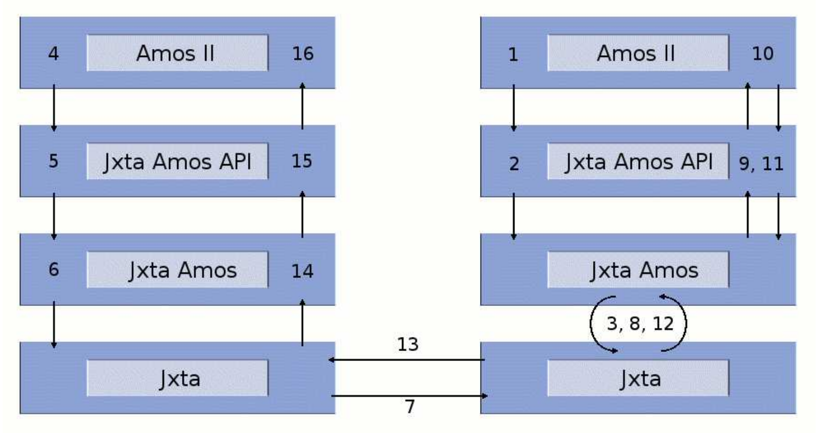


Figure 8: Example of a send command

The different steps of the communication process:

1. The receiving end starts listening to incoming messages by calling the foreign function `listen`, which blocks until canceled. The foreign function is in the API layer.

2. The API layer redirects the command to the JXTA Amos layer.

3. In the JXTA Amos layer the application loops and checks for messages from the JXTA layer until the user cancels ENTER is pressed.

4. The sending end either issues one of the send commands to send an AmosQL command to one or more receiving ends.

5. A foreign function is called in the API layer that calls the the correct communication method in the JXTA Amos layer.

6. An XML document is created that contains the AmosQL command and what kind of message it is (i.e. if a reply is needed for the message). Depending on which communication method is used; a pipe, socket or bidirectional pipe that is connected to the receiving end is created [11] and the message is sent down to the JXTA layer.

7. The message is sent to the receiver using a JXTA protocol.

8. The message is received on the receiving end. The AmosQL command is fetched from the XML document and depending on if a reply is needed[12] then the command is sent up to JXTA Amos API layer and the listen loop waits for the result from the API layer. Otherwise if no reply is needed then the command is sent up to the API layer and ignoring any result.

9. The API layer here executes the AmosQL command by using the callin interface to Amos II.

10. Amos II executes the AmosQL command.

11. If a reply was needed then Amos II sends back the result of the AmosQL command to the API layer where a result XML document is created and is sent down to the JXTA Amos layer.

12. The result document is sent on back using the correct pipe, socket or bidirectional pipe.

13. The message is sent back using some JXTA protocol.

[11]this happens only the first time when sending to another peer, the rest of the time the already created pipe, socket or bidirectional pipe is used.

[12]only for the ship command

14. The sending peer receives the result document and goes through the XML file and emits[13] the results to Amos II.

15. The results are emitted to Amos II using the Callout interfaces result method.

16. The results reach Amos II and are shown in the usual way.

3.4 Implementation Alternatives

The reason why I decided to use an extra API layer in the middle, as seen in figure 8 was so that I could develop and test the JXTAAmos part on its own without having to go via Amos II to test the different methods. This lead to much easier and faster testing and of the communication methods to make sure they work as they are intended to.

JXTA is available in many different programming languages, and the initial plan was to use JXTA-C[17] which allows to write P2P applications that use JXTA technology in the programming language C. But after some research about JXTA-C, it turned out that JXTA-C did not support all the desired features and was not as complete as the original JAVA version of JXTA. It was this and the fact that there was not as much documentation for the JXTA-C version than for the JAVA version when this project began, led to that the implementation was done in the JAVA version of JXTA. The JXTA version used to implement the system was JXTA J2SE 2.3.2.

[13]To send a result back to Amos II from an external application

4 Evaluation

The evaluation of the JXTA system was done by making three different types of evaluation tests. These tests were made first on the JXTA system and afterwards on the Amos II, then the test results were compared with each other. The idea of these tests were to evaluate how efficient the JXTA based communication is compared to the communication methods used in Amos II, which is TCP/IP sockets. As mentioned there were three different tests:

- Reliability of JXTA communication
- Latency
- Throughput

Because TCP/IP sockets are reliable by default only the two lasts tests (that is latency and throughput tests) results were compared with the JXTA communications methods.

4.1 Test set-up

The tests where set up as follows. The computers were connected with each other using one Ethernet LAN as figure 9 shows. Computer B was set up as a rendezvous peer in JXTA so that the different JXTA Peers could find each other and form a peer group.

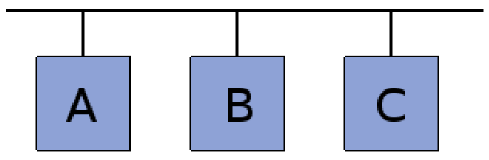


Figure 9: The test set up

Before the actual tests were made, a "warm up" of the communication was performed. The reason why this was done is simply so that the time needed for a peer to create a connection to another peer would not interface with the measurements of the actual tests. The "warm up" consisted of sending a few round trip messages using both JXTA sockets and JXTA bidirectional pipes to all other peers. The reason why to send round trip messages was so that the other peer would also create a connection to the sender.

The latency tests and the throughput tests were mainly measured between computer A and computer C, with some control measurements between computer C and computer B. The reason for this was that computer

C and computer A had similar CPU speeds whereas computer B was a bit slower than the other two. The reliability tests were measured using all computers because in these tests the time was not a critical moment, only the order of the messages were important and that no messages were lost during the send phase of the test.

To measure the performance of the JXTA communication, I made a slight modification to the JXTAAmos layer. Instead of sending/forwarding messages up to the JXTAAPI layer from the JXTAAmos layer, as seen in figure 8 on page 17, the messages only reach the JXTAAmos layer, and processed there, and depending of the message type the messages were directly sent back to the sender (Steps 1–8, 12–16 in section 3.3 on page 17).

All the measurements were done at the JXTA Amos layer(see figure 8 on page 17), which means that all messages have go through JXTA, and processed at the JXTAAmos layer and then sent back to the sender depending on if the message was sent using a send or a ship method.

4.2 Reliability

These tests were made to test the reliability of the different JXTA communication methods. The following JXTA methods were tested

- Pipes
- Broadcast socket
- JXTA Sockets
- Bidirectional Pipes

The first two methods: pipes and socket broadcast are unreliable but were tested to check how many messages were lost and/or received in the wrong order.

Pseudo code for the reliability tests:

```
TestReliability() {
    Send("Initialize Reliability Test");
    SendMessages(100); // Send 100 Messages numbered 1...100
    Send("End Reliability Test");
    Results = GetResults();
}
```

A short explanation of the different rows in the pseudo code:

1. a message was sent to the receiver to start storing messages.
2. 100 messages are sent to the receiver, which stores the messages it gets in an array

3. a message was sent to the receiver to stop storing messages.

   Thereafter the received messages are analyzed so that they are in the correct order, the number of messages that were received in the wrong order and the number of lost messages are reported back.

4. A message to retrieve the results from the receiver is sent.

	In wrong order		**Lost messages**	
	10	**100**	**10**	**100**
Pipe	0	7	4	2
Broadcast Socket	3	9	5	45
JXTA Socket	0	0	0	0
Bidirectional Pipes	0	0	0	0

Table 1: Reliability tests table

Table 1 shows the result from the reliability tests. The tests were first done by sending 10 numbered messages and then analyzing the results. After that the tests were re-run but now by sending 100 numbered messages. From table 1 we can see that regular pipes and the broadcast socket communication methods are not suitable for reliable communication between two peers, because when using these methods messages are lost and/or arrive in wrong order. This is can be very serious especially when working with databases since the database is supposed to be consistent. JXTA sockets and Bidirectional pipes in JXTA are reliable and they should be used if reliable communication is to be used.

4.3 Latency

Latency was measured between two peers to see if there was any difference in latency when using TCP/IP sockets for communication and when using the different JXTA implemented methods.

Latency was calculated by first measuring the time it took to send a number of round trip messages, then by using that time to calculate latency like this.

From the test we get the total time it took to send a number of round trip messages:

$$Totaltime = NumberOfMessages * RoundTripTime \tag{1}$$

Since the round trip time consist of latency in both ways and a small message process time on the receiver. Therefore the round trip time can be written as:

$$RoundTripTime = 2 * Latency + MessageProcessTime \tag{2}$$

which changes equation 1 on the facing page into

$$Totaltime = NumberOfMessages * (2 * Latency + MessageProcessTime) \tag{3}$$

But when the message is small then the message process time is negligible compared to the latency. Then latency can be calculated the following way:

$$Latency = \frac{Totaltime}{NumberOfMessages * 2} \tag{4}$$

In pseudo code the latency calculations looked like this:

```
CalculateLatency() {
    start = currentTime();
    SendRoundtrip(N); //Send N roundtrip messages
    end = currentTime();
    Latency = (end - start ) / (N*2); //Calculate latency
}
```

A short explanation of the different rows in the pseudo code:

1. Store the starting time.
2. Send N round trip messages.
3. Store the ending time.
4. Calculate the latency using equation 4.

Latency was measured for JXTA communication methods when using JXTA sockets and JXTA bidirectional pipes, and also for Amos II communication, which uses TCP/IP sockets.

JXTA Sockets	**Bidirectional pipes**	**TCP/IP sockets**
163	50	2

Table 2: Latency in milliseconds

Table 2 shows the results of this test and all the values are in milliseconds. As can be seen from that table, JXTA sockets had the highest latency (163 ms) while JXTA bidirectional pipes only had $\frac{1}{3}$ (50 ms) of the latency for JXTA sockets. The lowest latency value had TCP/IP sockets with a value roughly 2 ms.

4.4 Throughput

Throughput was measured for the two reliable communication methods for JXTA (i.e. JXTA sockets and JXTA bidirectional pipes) and then compared with TCP/IP sockets which is the send method that is used in Amos II. The measurements were done as follows:

1. a message was sent to the receiver to start a timer.
2. lots of messages were sent to the receiver with a specific message size.
3. a message to stop the timer on the receiving end was sent and the time it took to receive all the messages was returned.
4. throughput for the message size is calculated.

In pseudo code the test looked like this:

```
CaclulateThroughput(size) {
    msg = createMessageWithSize(size);
    Send("Start timer");
    repeat N times
        Send(msg);
    Send("Stop timer");
    time = getReceivedTime();
    throughput = (N*size) / (time);
}
```

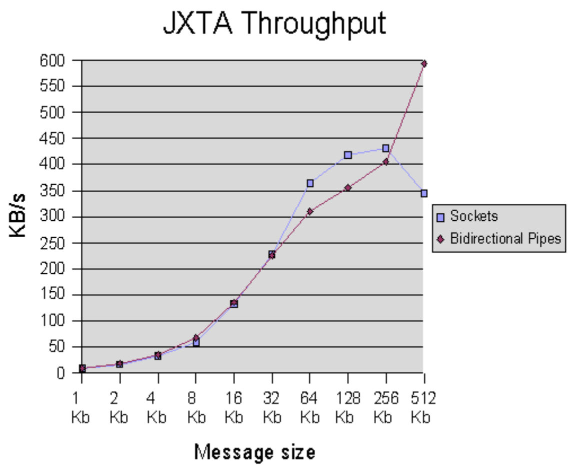


Figure 10: Throughput for JXTA methods

This was done for 10 different message sizes. The results of the JXTA tests can be seen in figure 10. The bidirectional pipes originally only support a message size of up to 64 KB so in order to send messages that were larger than 64 Kb message chunking was implemented. From the figure we can see that JXTA Bidirectional pipes and JXTA sockets have very similar throughput up to message sizes around 256 Kb. There after the throughput for JXTA bidirectional pipes becomes higher than the throughput for JXTA sockets.

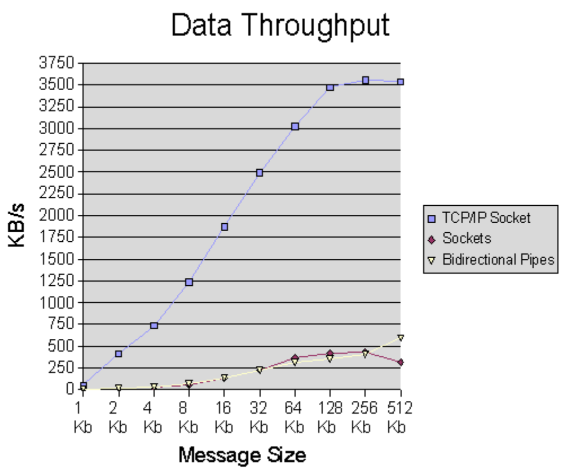


Figure 11: JXTA throughput compared to TCP

Measurements in Amos were done to compare how the JXTA throughput is compared with the throughput of TCP/IP sockets. Figure 11 shows the throughput for TCP/IP sockets compared to the two different JXTA communications methods. From figure 11 we can clearly see that the throughput for TCP/IP sockets is much much higher than the two JXTA methods throughput.

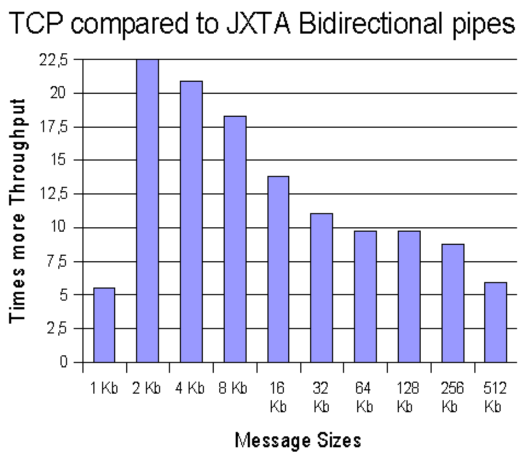


Figure 12: TCP compared with JXTA Sockets

Figure 12 on the preceding page shows the ratio between TCP/IP socket throughput and the throughput of the JXTA communication methods. From that figure we can see that message sizes up to 8 KB have roughly about 20 times or higher throughput in TCP/IP sockets than in JXTA. For larger message sizes than that, 16KB and higher, TCP/IP sockets still have roughly about 10 times higher throughput than the JXTA methods.

So compared to TCP/IP sockets JXTA communication methods seems to have very low throughput. This means that if the application which uses JXTA for communication is used to send a large amount of data at a given time, it might be better to use TCP/IP sockets instead of JXTA because of the higher throughput and much lower latency.

4.5 Factors

There were different factors with some of the JXTA methods of communication. Firstly the JXTA broadcast method for pipes and sockets are unreliable, meaning that there is no guarantees that messages will arrive, and if they do there is no guarantees that they will be in the correct order. To work around this I implemented some broadcast methods that are based on reliable communication which use JXTA sockets and JXTA bidirectional pipes.

These methods were not tested as thoroughly as the regular send methods because they use the underlying JXTA sockets/bidirectional pipes for communication. The reason for this is that the throughput for the tested methods can be seen as an upper limit for the throughput. This would lead to that the broadcast methods would have less throughput than the simple send methods.

4.6 Comments

The reason why the other JXTA methods, that is sockets and bidirectional pipes, are slower than the JXTA pipes are because that they are built on top of the pipe service so they inherit the pipe services processing time and add new processing time themselves by ensuring reliability etc.

As mentioned in section 4.4 on page 24 the bidirectional pipes used in JXTA, support message sizes up to 64KB compared to JXTA sockets which support any message sizes. The reason for this is that JXTA sockets uses message chunking and JXTA bidirectional pipes do not[8]. To send messages with size greater than 64Kb, message chunking can be implemented manually on top of bidirectional pipes. Before sending the message a simple check is made if the message is greater than 64KB, and if it is then the message is split into 64KB messages and sent one after another. On the receiving end the content of the split messages are merged before sending them further.

5 Summary & Conclusions

5.1 Summary

In this project a new communications module for Amos II was implemented which uses JXTA P2P Technology. The different basic communications methods that were implemented are:

- Two send methods that uses the two reliable communication methods in JXTA. The methods sends a command to an other Amos II peer without waiting for a result.
- Two "ship" methods that uses the reliable communication methods in JXTA. The methods sends a command to an other Amos II peer and waits for the receiver to send back a result.

A simple message chunking for JXTA bidirectional pipes were implemented also in order to send larger than 64 Kb using the pipes and compare them with JXTA sockets.

The methods were first tested so that they worked as they were supposed to and then the performance of these new methods were tested using three different tests,

1. Reliability for the different JXTA communication methods was tested.
2. Latency was measured to see if there was any difference in latency when using the different methods.
3. Throughput of the methods was tested and compared.

The results for the JXTA based communication were then compared to TCP/IP sockets which Amos II uses.

5.2 Conclusions

From the evaluation phase of this project we can draw some simple conclusions. If JXTA based communication is to be used for reliable communications then there are two suggestions. Either use JXTA Sockets or JXTA bidirectional pipes since they are the only ones which are reliable.

From the latency test results (table 2 on page 23) and throughput test results (figure 10 on page 24) we can clearly see that both JXTA bidirectional pipes and JXTA sockets have very similar throughput but JXTA bidirectional pipes seems to have slightly better throughput for larger message sizes. This gives us that JXTA bidirectional pipes are better suited for reliable communication when using large message sizes.

But when the throughput of JXTA is compared with TCP/IP socket communication that Amos II uses (figure 11 on page 25) the results shows that JXTA has still a long way to go to reach the same values as TCP/IP sockets. The latency for TCP/IP sockets are also very low, only roughly 2 ms as can be seen in table 2 on page 23. Because of this TCP/IP sockets are still the best thing to use.

5.3 Limitations

There are some limitations on the system that was implemented in this project. Some of the limitations to name a few:

- Objects cannot be sent from one peer to another. This was not needed to make the simple tests that were made during the evaluation phase of this project.
- At the moment there is no check for duplicate peer names when joining the peer group. A way around this would be to use Amos II name server to ensure that peers have unique names in the peer group.
- Peers, when initialized, become members in AmosPeerGroup automatically. Peers can't create any other peer group.

5.4 Future Work

JXTA technology is available in many languages. It might be a good idea to look more into the C version of JXTA[14]. This due to the fact that the communication between extensions written in C and Amos II is faster than between JAVA extensions and Amos II and that extensions written in C might run faster than in JAVA.

So it might be a good idea to evaluate that as well because it seems that JXTA-C has been developed further since this project began and seems to have more documentation about the JXTA-C version project now than before.

Otherwise a good idea might be to wait for a while so that JXTA becomes more effective in its communication handling.

[14]JXTA-C

References

[1] Amos II Wrappers. http://user.it.uu.se/˜udbl/amos/wrappers.html.

[2] Kristofer Cassel and Tore Risch. An object-oriented multi-mediator browser. In *User Interfaces to Data Intensive Systems*, pages 26–35, 2001.

[3] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. Protecting Free Expression Online with Freenet. *Internet Computing, IEEE*, 6(1):40–49, Jan/Feb 2002.

[4] Daniel Elin and Tore Rish. Amos II Java Interfaces. Technical report, Dept. of Information Science,Uppsala University, Uppsala, Sweden, Aug 2000.

[5] Gnutella website. http://www.gnutella.com/.

[6] Li Gong. JXTA: a Network Programming Environment. *Internet Computing, IEEE*, 5(3):88–95, May/Jun 2001.

[7] Martin Jost. A Wrapper for MIDI files from an Object-Relational Mediator System. Technical report, Department of Information Science,Uppsala University, Uppsala, Sweden, Aug 2000. http://user.it.uu.se/˜udbl/publ/MidiWrapper.pdf.

[8] JXTA v2.3.x: Java^{TM} Programmer's Guide. www.jxta.org, Apr 2005.

[9] Project JXTA homepage. www.jxta.org.

[10] Timour Katchaounov, Tore Risch, and Simon Zürcher. Object-Oriented Mediator Queries to Internet Search Engines, Sep 2002. Presented at International Workshop on Efficient Web-based Information Systems (EWIS-2002), Montpellier, France.

[11] Hui Lin, Tore Risch, and Timour Katchaounov. Adaptive Data Mediation over XML Data. *Journal of Applied System Studies, (JASS)*, 3(2), 2002. Special Issue on : "WEB Information Systems Applications".

[12] Scott Oaks, Bernard Traversat, and Li Gong. *JXTA In a Nutshell.* O'Reilly, first edition, Sep 2002.

[13] Project JXTA: An Open Innovative Collaboration. www.jxta.org, April 2001.

[14] Tore Risch and Vanja Josifovski. Distributed data integration by object-oriented mediator servers. *Concurrency and Computation: Practice and Experience*, 13(11):933–953, 2001.

[15] Tore Rish, Vanja Josifovski, and Timour Katchaounov. Functional data integration in a distributed mediator system. In Peter M.D. Gray, Larry Kerschberg, Peter J.H. King, and Alexandra Poulovassilis, editors, *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, chapter 9, pages 211–238. Springer, 2003.

[16] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, fourth edition, Jul 2001.

[17] Bernard Traversat, Mohamed Abdelaziz, Dave Doolin, Mike Duigou, Jean-Christophe Hugly, and Eric Pouyoul. Project JXTA-C: Enabling a Web of Things. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, Jan 2003.

[18] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA 2.0 Super-Peer Virtual Network. www.jxta.org, May 2003.

[19] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *Computer, IEEE*, 25(3):38–49, Mar 1992.

A APPENDIX: Test Explanation

This appendix goes through what methods were used to make the tests and what they do.

A.1 Reliability tests

The reliability tests are simply AmosQL functions that were called from Amos II. Here is an example of how the test for regular JXTA pipes looked like:

```
1   create function testReliabilityPP(charstring node,
2                                     integer x)
3   -> charstring as
4   begin
5           jsendp(node, "begin="+itoa(x));
6           select jsendp(node, itoa(i))
7                   from integer i
8                   where i=iota(1,x);
9           result jshippp(node, "end");
10  end;
```

The tests for the other communication formats were very similar. The only difference was the use of a different `send` and `ship` method. The `send` method sends a string to the receiver without waiting for a result whereas the `ship` method sends a string and waits for the receiver to reply.

The first send command on row 4 initializes the receiver so it will store the next X messages. Rows 5–7 sends out X messages to the receiver. Row 8 sends a `end` command to the receiver which will analyze the X messages to check if any messages were lost or arrived in the wrong order.

The code for the analyze method:

```
private String calculateSendTestReply(int []sendTest) {
  ArrayList error = new ArrayList();
  for (int i = 0; i < sendTest.length-1; i++) {
    lostMessages.remove(new Integer(sendTest[i]));

    if ( !(sendTest[i]+1 == sendTest[i+1]) ) {
      boolean b1=false, b2=false;
        if (i-1>=0 && (sendTest[i-1]+1 == sendTest[i])){
          b1=true;
          error.remove(new Integer(sendTest[i-1]));
          error.remove(new Integer(sendTest[i]));
        }
        if ((i+2 < sendTest.length) &&
              ( sendTest[i+1]+1 == sendTest[i+2] ) ){
            b2=true;
            error.remove(new Integer(sendTest[i+1]));
            error.remove(new Integer(sendTest[i+2]));
          }
          if(!(b1 && b2) &&
              ! error.contains(new Integer(sendTest[i+1]))){
            error.add(new Integer(sendTest[i+1]) );
          }
      } else {
        error.remove(new Integer(sendTest[i+1]));
        error.remove(new Integer(sendTest[i]));
      }
    }
    lostMessages.remove(
      new Integer(sendTest[sendTest.length-1]));
    if (sendTest[sendTest.length-1] == sendTest.length) {
      error.remove(new Integer(sendTest.length));
    }

    // Create reply String
    ...
  }
```

Since the messages consists of numbers only and should be ordered, the method simply loops through the messages and checks that message *i* is one number smaller than message *i+1*. Any deviations from that are reported and presented. The method can report that messages are in wrong order in some cases where they are not. The method was written to help me find most of the messages that are in the wrong order not all and by looking at the message order manually afterwards the number of errors can sometimes be reduced if the method has reported too many errors.

A.2 Latency tests

Latency was measured by using the following methods, for TCP/IP socket measures this lisp method was used:

```
(defun latency()
        (setq start1 (clock))
        (rptq 1000 (reval@nameserver 1))
        (setq end1 (clock))
        (osql-result (/ (/ (- end1 start1) 1000) 2))
)
```

Row 2 stores the starting time while row 3 sends 1000 short round trip messages to the nameserver. Row 4 stores the ending time and finally row 5 calculates the latency.

The JXTA socket version of the Latency calculations:

```
1   public double latSocket(String peer, int numOfMessages) {
2     JxtaSocket socket = getSocketToPeerDisc(peer);
3     double latency = 0;
4     if ( socket != null ) {
5       try {
6         PrintWriter out =
7           new PrintWriter(socket.getOutputStream(), true);
8         BufferedReader in =
9           new BufferedReader(
10            new InputStreamReader(socket.getInputStream()));
11
12        String pingPongMsg =
13          createMessageString(JxtaAmos.SHIP, SOCKET,
14                              "ping-pong");
15        long start1=System.nanoTime();
16        for (int i = 0; i < numOfMessages; i++) {
17          out.println(pingPongMsg);
18          String reply = in.readLine();
19        }
20        long end1 = System.nanoTime();
21        latency = (double) ((end1-start1)/1000000000) /
22                           (double) numOfMessages;
23      } catch (IOException e) {
24        // TODO Auto-generated catch block
25        e.printStackTrace();
26      }
27    } else {
28      System.out.println("Problem connecting to peer.");
29    }
30    return latency;
31  }
```

Row 2 gets the socket to the wanted peer, rows 6–10 creates some streams to use with the socket. Rows 12–14 creates the message string that is sent as a round trip message. Row 15 starts the timer and rows 16–19 sends the round trip message and receives the answers. Row 20 stops the timer and finally row 21 calculates the latency according to equation 4 on page 23.

The JXTA bidirectional pipes version of the latency calculations:

```
1   public double latBidi(String peer, int numOfMessages) {
```

```
2     JxtaBiDiPipe outPipe = getBiDiPipeToPeerDisc(peer);
3     double latency = 0;
4     if( outPipe != null ) {
5       try {
6         long start1, start2, end1, end2;
7         start1 = System.nanoTime();
8         for (int i = 0; i < numOfMessages; i++) {
9           Message pingPongMsg = createMessage(JxtaAmos.SHIP,
10                                              BIDIPIPE,
11                                              "ping-pong");
12          outPipe.sendMessage(pingPongMsg);
13          Message result = outPipe.getMessage(rtimeout);
14        }
15        end1 = System.nanoTime();
16        latency = (double) ((end1-start1)/1000000000) /
17                            (double) numOfMessages;
18      } catch (IOException e) {
19        e.printStackTrace();
20      } catch (InterruptedException e) {
21        e.printStackTrace();
22      }
23    } else {
24      System.out.println("Problem connecting to peer.");
25    }
26
27    return latency;
28  }
```

This method is very similar as the JXTA socket version. Row 2 gets the bidirectional pipe to the wanted peer. Row 7 starts the timer while rows 8–14 handles the creation of the messages and sending them to the receiver and getting the result back. Row 15 stops the timer and finally row 16 calculates the latency according to equation 4 on page 23.

A.3 Throughput tests

To tests the throughput in Amos II for TCP/IP sockets the following method were used:

```
1   (defun kbps(os ls)
2     (setq start1 (clock))
3     (rptq os (send-form (buildstring ls) *nsp*))
4     (reval@nameserver 1)
5     (setq end1 (clock))
6     (osql-result (/ (* os (/ (calculateBytes ls) 1024 ))
7                   (- end1 start1)))
8   )
```

The `buildstring` methods simply builds a string with a inputted length, while `calculateBytes` calculates how many bytes the string has. Row 2 stores the starting time and row 3 sends the list to the receiver `os` times. The round trip message sent on row 4 is simply to make sure that the receiver has gotten the list that was sent on row 3. Row 5 stores the ending time and row 6 – 7 calculates the throughput.

The JXTA socket version of the throughput measurements:

```
public double kbpsSocketSend(String peer, int numOfMessages,
                             int numOfBytes) {
  JxtaSocket socket = getSocketToPeerDisc(peer);
  double bytesPerSecond = 0;
  if ( socket != null ) {
    try {
      PrintWriter out =
        new PrintWriter(socket.getOutputStream(), true);
      String msgString = createString(numOfBytes);
      long bytesSent = 0;
      String initMesg =
        createMessageString(JxtaAmos.SEND, NOREPLY,
                            "Start measure");
      out.println(initMesg);
      for (int i = 0; i < numOfMessages; i++) {
        String msg = createMessageString(JxtaAmos.SEND,
                                         NOREPLY,
                                         msgString);
        bytesSent += msg.length();
        out.println(msg);
      }
      String endMesg =
        createMessageString(JxtaAmos.SHIP, SOCKET,
                            "End measure");
      out.println(endMesg);
      InputStreamReader isr =
        new InputStreamReader(socket.getInputStream());
      BufferedReader in = new BufferedReader(isr);
      reply = in.readLine();
      double receivedTime = Double.parseDouble(reply);
      bytesPerSecond = (double)bytesSent/receivedTime;
    } catch (IOException e) {
      e.printStackTrace();
    }
35     } else {
36       System.out.println("Problem connecting to peer.");
37     }
38     return bytesPerSecond;
39   }
```

Row 9 creates a string with the specified length. Rows 11–14 creates and sends the initialization message to the receiver which starts a timer on the receiver. Rows 15–21 sends a string with the specified length the given number of times. 22–25 creates and send the stop message to the receiver to stop the timer. In row 29 the reply is gotten which contains the time it took to receive the messages and finally row 31 calculates the throughput of the receiver.

The JXTA bidirectional pipes version of the throughput measurements, which is similar to the JXTA socket version:

```
 1   public double kbpsBidiSend(String peer, int numOfMessages,
 2            int numOfKBytes) {
 3     JxtaBiDiPipe outPipe = getBiDiPipeToPeerDisc(peer);
 4     double bytesPerSecond = 0;
 5     if( outPipe != null ) {
 6       try {
 7         String msgString = createString(numOfKBytes*1024);
 8         long bytesSent = 0;
 9         Message initMsg =
10           createMessage(JxtaAmos.SEND, NOREPLY,
11                         "Start measure");
12         outPipe.sendMessage(initMsg);
13         for (int i = 0; i < numOfMessages; i++) {
14           Message msg[] =
15             createBiDiPipeMessages(JxtaAmos.SEND, NOREPLY,
16                                    msgString);
17           bytesSent += sendBidiPipeMessages(msg, outPipe);
18         }
19         Message endMsg =
20           createMessage(JxtaAmos.SHIP, BIDIPIPE,
21                         "End measure");
22         outPipe.sendMessage(endMsg);
23
24         Message result = outPipe.getMessage(rtimeout);
25
26         double receivedTime =
27           Double.parseDouble(
28             result.getMessageElement("Reply").toString());
29         bytesPerSecond = (double)bytesSent/receivedTime;
30       } catch (IOException e) {
31         e.printStackTrace();
32       } catch (InterruptedException e) {
33         e.printStackTrace();
34       }
35     } else {
36       System.out.println("Problem connecting to peer.");
37     }
38
39     return bytesPerSecond;
40   }
```

Row 7 creates a string with the specified length. Rows 9–12 creates and sends the initialization message to the receiver which starts a timer on the receiver. Rows 13–18 sends a string with the specified length the given number of times. The method `createBiDiPipeMessages` handles the message chunking in a simple way, when a string is larger than 64 Kb the method splits it int 64 Kb chunks. The `sendBidiPipeMessages` method sends all the messages to the receiver. The receiver then

19–22 creates and send the stop message to the receiver to stop the timer. In row 24 the reply is gotten which contains the time it took to receive the messages and finally row 29 calculates the throughput of the receiver.