IJCSET |December 2012 | Vol 2, Issue 12, 1510-1515 www.ijcset.net | ISSN:2231-0711

Design and Implementation of a Network Layer for Distributed Programming Platforms

G. Srinivas Reddy[#1], Dr. K. Venkateswara Reddy[*2], Prof. T. Venkat Narayana Rao[#3]

[#1] *Mahatma Gandhi Institute of Technology, Hyderabad, India*

[#2] *Principal, MLR Institute of Technology and Management, Hyderabad, India*

[#3] *Professor Department of CSE , Guru Nanak Institutions Technical Campus, Hyderabad, A.P, India*

***Abstract*—This paper discuss the design, implementation and evaluation of a new network layer for Mozart. Mozart is one of the distributed programming platform which is based on the multi paradigm language namely Mozart supplies a factual network transparent implementation by maintaining network awareness, openness, and also fault tolerance. Its network layer provides message passing service to its higher layers which run protocols to uphold the state of distributed entities. In the study of the old network layer problems, a new model was designed and implemented. With this the Solutions achieved includes less fragmented sending of data, more efficient usage of file descriptors and similar resources, leaner memory usage, improved multiplexing over communication channels and a monitor mechanism for error handling. This paper also focuses on some other languages distribution models and the message passing services. It come out that the many different aspects of a multi paradigm language also require more of such service. When we compare the performance of the network layer of Mozart with Java's RMI it is competitive , or even higher than the performance of Java's RMI, which uses a more simplistic message passing service.**

***Keywords*— Distributed, layer, ByteBuffer , message passing, binding, remote.**

I. INTRODUCTION

Distributed environment has become demand of the day now a day's. This distributed application involves much extra programming which is not of interest for the application to run. In simple terms it still involves getting into low-level networking or perhaps using some abstraction that scarcely hides the networking. Mostly this is because network tools and abstractions have been designed with the idea to provide a nice interface directly to the low-level communication. A different perspective is to design an environment which distributes a centralized application without openly defining any communication. From the application programmers' point of view, Mozart is a multi paradigm language that provides distribution [1]. Still here also the low-level networking needs to be taken care and the creative parts of this paper is about the design and implementation of new networking support for Mozart which is described in sections below.

All distribution models need some means of communication in the form of a message passing system. This is referred to as a network layer throughout this paper. The design of network layer usually can be done with the help of operating system or some internet protocols which provide directly, but they do not provide all that is expected of the communication. Typical issues here are use of synchronous or asynchronous message passing and how to uphold connections. This is dealt with respect to single or multiple connections between each virtual machine and limited resources such as file descriptors or port numbers, what communication protocol to use and lastly how to reflect errors to higher layers.

1) *The network layer of Mozart:* Unlike RMI the network layer of Mozart does not have one single protocol to serve, it has several protocols to handle variety of entities as shown in figure 1. These protocols include sending messages to one or more sites, synchronous or asynchronous. To efficiently serve the various needs, the network layer of Mozart implements an asynchronous message passing service [2].

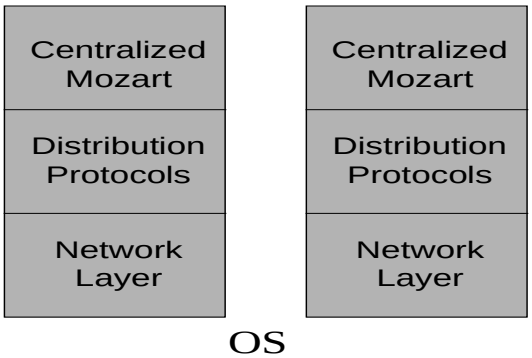

Fig 1. Architecture of distributed Mozart

For example in Mozart to utilize allocated network resources as well as possible, there will always be at most one virtual connection to every site. This is depicted in **Error! Reference source not found.**2. Here communication is multiplexed over the shared connections.

Since a typical Mozart application runs several threads, several messages to one site can be collected and sent in one chunk, which will lower the overhead on each message. When resources are low, physical connections can be temporarily closed to let other connections be opened [3]. To upper layers, this is only visible as higher latency as shown in figure 2.

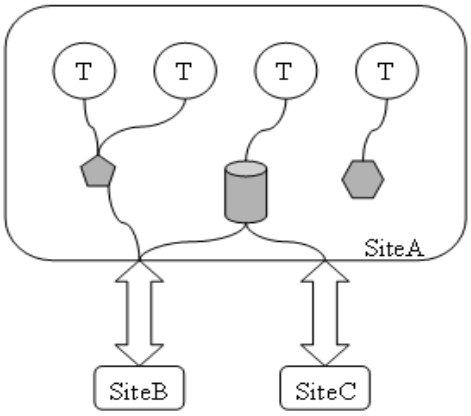

Fig 2. Utilization of Resources in Mozart

2) Design and implementation of the new Network Layer:

In the previous model, the distributed subsystem is divided into a distribution layer and a network layer. The distribution layer maintains information on where different sites are physically located. The network layer passes the messages. This structure is kept for the new model, but the network layer is now divided into one communication layer and one transport layer as shown in Figure 3.

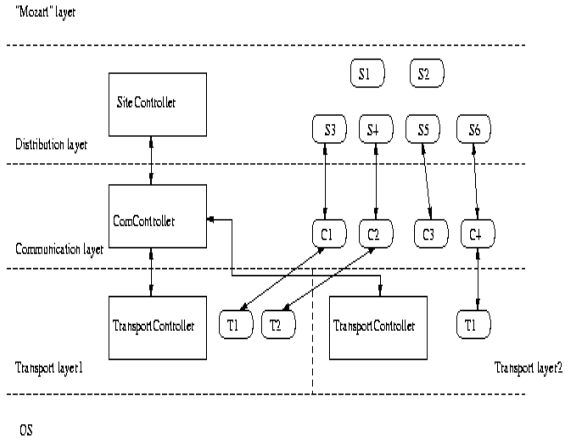

Fig 3. Architecture of the new model.

In Figure 3 the boxes represents controllers that exist whenever the distribution is running. Controllers are responsible for creating and garbage collecting the multiple objects of each layer. The oval represents object instances on each layer that are exclusive for every remote peer.

3) *Distribution layer:* The objects S1-S6 represent site objects. In Mozart the word site is used to denote processes that might reside on different machines, and in this model site-objects are used as references to remote sites that are known to the own site[3]. A site object contains all the necessary information to connect to a remote site, such as the physical location and what transport media to use when connecting to it. When a reference is passed to some other site, the information in the site is marshaled and transmitted. When information about a site is received and unmarshaled, a site-object is created in the system if it does not exist.

4) *Communication layer:* Every ComObject takes care of the communication for one site-object. It gets a channel, sets up properties for the communication over that channel, keeps track of when a channel can be closed, queues messages, and assures that messages reach their destination or are reported unsent and provides probes for network monitoring.

5) Transport layer: A TransObject T1, T2 provides an abstraction for some means of a reliable physical communication channel. Several different types of TransObjects can coexist in the system, providing different types of communication between sites depending of their relative physical location. When a ComObject needs a TransObject it requests it from a special "Physical Media Mediator", that with the help of the site object decides on what type of transport media to use. The different TransControllers are kept by this mediator and used to extract the TransObject. The connection is then established with the connection procedure as described in [7]. Resources are also controlled by TransControllers. If too many are in use, ComObjects will be queued to get a resource when there is one free. A ComObject should always be sure to sooner or later get a TransObject, i.e. resources are handled fair in the same aspect as threads [5]. To begin with, one TransObject is implemented to use TCP, and one to use shared memory, but it is left open to add communication via UDP or similar protocol.

6) *Messages and MsgContainers*: Messages are stored in MsgContainers that act as transporters of information. MsgContainers are created by the distribution protocols when a message is to be sent or by the transport layer when a message has been received. Except for storing the type and content of the message, a MsgContainer can also store information on when a message was sent. All the knowledge of how to marshal and unmarshal a message of a certain type and what to do during garbage collection is also kept in the MsgContainer.

- Connecting to a Remote site

ComObjects "connect" to remote sites by getting a TransObject (physically connected) and running an open protocol to set some properties for the communication. Connections are originally initiated when a ComObject is ordered to send. A remote peer will accept the connection. Being an initiator or acceptor are two cases that need to be treated in different ways. The reason for this is that an accepting ComObject cannot know who is trying to connect to it, until the remote peer has introduced itself [4].

7) *Building the architecture as an accepting site*: Whenever a connection is accepted a TransObject is created and handed over to the ComController. The ComController creates an anonymous ComObject since the remote site is not yet known. This ComObject gets to run the open protocol that will determine if another ComObject to this site exists. In those cases where the remote site is known prior to the accept, a ComObject may or may not exist for

that site. If an object does exist, that ComObject is to be used since it may contain queued messages and valuable information from prior connections. The anonymous one is discarded. If the old ComObject also has a TransObject this means the two sites must be trying to connect to each other simultaneously and the open protocol described below determines which channel is to be used. The old ComObject then adopts the TransObject of that channel.

8) *Connect and accept with the Physical Media Mediator:* This design is based on “Physical Media Mediator" for the establishment of a physical channel. The idea of the "Physical Media Mediator" is to make it achievable to control what transport media to use from oz-level. It should work for both standard and added on transport layers. This architecture is described in the following Fig 4.

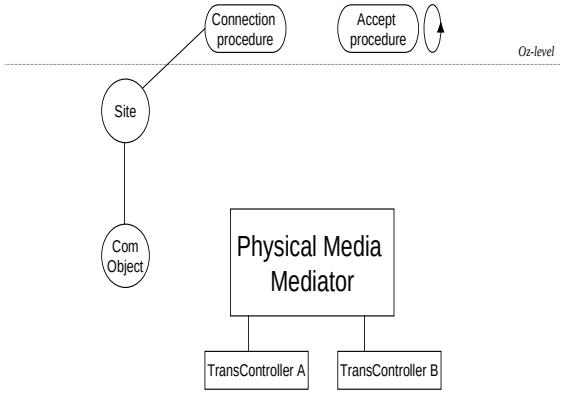

Fig **4.** G*raphical view of architecture with a "Physical Media Mediator".*

Every site object has one connection and one accept procedure. When a reference to a site is given, the connection procedure to be used to connect to that site is included. Locally every site runs an accept procedure dedicated to accept connections from the connection procedure. The interface to the network layer is given through a built-in.

9) Open protocol: The open protocol is used to negotiate some communication properties and to assure that both sites are using the same period version and know whom they are talking to. It also determines what channel to use in case both sites initiate communication simultaneously. It consists of three regular messages created and sent by the ComObjects .

- The anonymous ComObject sends PRESENT(Version, Site).
- The initiating ComObject sends NEGOTIATE(Version, Site, ChannelInfo)

At any point, either side can close the channel without sending any prior warning. Sending a specific abort message has been considered, but whenever there is a version mismatch (the other site might not even be an ozengine) that technique cannot be used [5][8]. Therefore the transport layer must always detect a lost channel, and a lost channel during the open protocol will be interpreted as aborting the entire connection. The first four are definite, but the buffer size needs to be equal for good performance with big messages. Therefore the initiating side gives a suggestion; the accepting side can either accept this or choose a smaller size, but never a larger (to make a natural end of the protocol). When the open protocol is run the ByteBuffer is already in use. An effect of this is that the ByteBuffer must always be large enough to fit a received open-protocol message, and the size has to be possible to change during runtime. Decisions on what size the buffer should have, is completely up to the used transport layer, and the corresponding TransObject should be asked for this.

10) Close protocol: A channel can be closed either because it is no longer needed which is detected at garbage collection, or because one site is running out of resources and needs to give the resources to another ComObject for a while. When garbage collection is run the other side is given a chance to keep the channel, but when resources are out, the connection must be closed. This implies two different close messages:

- CLOSE_WEAK, sent at garbage collection
- CLOSE_HARD, sent at lack of resources

A ComObject that is being closed will also be understood and one of the close messages as CLOSE_ACCEPT. After sending a CLOSE_HARD or CLOSE_WEAK the state of the ComObject changes to closing hard or closing weak. No more messages will be sent until a CLOSE_REJECT is received or the channel has been closed and reopened. At the remote peer the state will be closing wait for disconnect if it accepts the close. If the close is accepted the close initiator will close the channel and the remote peer should notice the disconnection as a lost channel. On both sides, the TransObject is handed back to the TransController, but the ComObject will stay until removed from the site [9].

11.) Passing Messages

The responsibility of transport layer is the actual transportation of messages. This can be done in different ways depending on what type of media used and thus on what type of TransObject is used. The communication layer expects the transport layer to provide a entirely reliable service. Messages may only be lost when a channel is lost. Messages sent should be received at their remote peer in the same order. Some priority levels could be defined non-FIFO. In this case, an extra parameter needs to be added to the communication that allows these messages to be received out of order. When a ComObject has something to send, it will order its TransObject to deliver. The TransObject should then pull messages when it is allowed to send. The scheduler in the Mozart Virtual Machine could be used to schedule when each TransObject can run [5]. A more naive approach is used, where it is checked at each thread switch whether input or output is possible, and TransObject will be invoked. When a message is received by the TransObject it should be put in a MsgContainer and handed up to the ComObject. The TransObject is also responsible for sending acknowledgement numbers that will be provided when a message to be sent is pulled.

12) Byte Buffers: As most of the media are assumed to transmit serialized consecutive data, a MsgContainers provides one method that marshals a message into a ByteBuffer and one that unmarshals it from a ByteBuffer. To make marshaling and unmarshaling independent of

when data is sent from, or received to the ByteBuffer, it is convenient to have a continuous ByteBuffer. Therefore, the ByteBuffer is implemented as a circular structure. The marshaler and the read handler, who write to the bytebuffer, must assure that the ByteBuffer is not overfilled. This requires a suspendable marshal. A suspendable marshaler can stop when the ByteBuffer is occupied, and continue later where it left off. This assumes the unmarshaler can handle unmarshaling of the fragments produced by such a marshaler.Each TransObject use exactly one ByteBuffer for incoming and one for outgoing data which confines the amount of memory used by each TransObject.

13) Big Messages: One of the problems of the old distribution engine was that one large transfer was allowed to monopolize the channel [11]. The complete message had to be sent before any other message could be sent. In addition, the complete message had to be received before unmarshaling could be done which made the memory of the distribution engine to grow temporarily. Therefore, the new model introduces the concept of marshaling and sending only parts of a big message at a time. This is made possible through the suspendable marshaler mentioned in the previous section. The unmarshaler will require a full fragment to be received before beginning to work, and therefore the ByteBuffer of the receiving side, must be at least large enough to fit the complete contents of the ByteBuffer at the sending side [11].

14) Specifics of the TCP-transport layer: When the TCP TransObject is initialized, it registers a read handler with the I/O-handler and when it is ordered to deliver, it registers a write handler. The I/O-handler will then offer the control back to the TransObject at certain times. These are the times that messages will be marshaled and written, and when messages will be read and unmarshaled. Messages are marshaled from MsgContainers into frames. The contents of the frames are stored in ByteBuffers. One message can be put in one or more frames. One or more frames can be put in each ByteBuffer. On every chance the TransObject gets to write, it will try to write up to one full ByteBuffer. The chosen structure means the first nine bytes must always be received before trying to use any data. Then the frame size can be compared to the received amount and a frame can possibly be unmarshaled. The acknowledgement is received earlier since it is independent of the message, and might wait for the probe usage.

- *Priorities*

In the delivery of messages five different priority levels are used, where level 1 and 5 are reserved for special system messages and levels 2 through 4 are used for all other messages. Levels 2-4 are scheduled like threads, but on number of bytes sent instead of on time. The relation between these priority levels can be altered from application level [10].

Level 5: Send as fast as possible, Level 4: High priority, Level 3: Medium priority, Level 2: Low priority, Level 1: Send when no messages are waiting on levels two through five. At regular intervals, messages on this level will be moved to level two to ensure throughput.

- *Acknowledgements and retransmission*

Since the transport layer is expected to provide a reliable transfer, the only case when messages can be lost is when the connection is lost [10]. Therefore, the only time that messages need to be retransmitted is at reconnection but for this scenario messages must be stored at the sender until they are acknowledged. Messages from the protocol layer are implicitly numbered by the ComObjects at the sending and at the receiving side. Whenever such a message is sent, an acknowledgement number telling the number of the last received message is attached to the frame. Messages produced by the ComObject are not desirable to acknowledge. In this approach a big message that was almost completely transmitted in a number of frames needs to be resent if one frame is lost since only the complete message is acknowledged. However with a reliable transport media, this will rarely happen and the implementation of the TransObject is free to add an extra acknowledgement schema if the protocols or media used are likely to cause lost messages.

15) Reference protocol: The only message exchange needed is when a channel is opened and for clearing a reference. Therefore, a field in the ChannelInfo declares the need when a channel is opened, and one message, the CLEAR_REFERENCE message clears a remote reference.

16) Garbage collection: When Mozart does garbage collection, the distributed subsystem needs to be checked in two phases.:

- All queued or unacknowledged MsgContainers need to be traversed since they can hold references to oz-entities and therefor are roots for garbagecollection. Especially they can hold references to sites, which imply the two phases. MsgContainers are found through the ComController that maintains a list of all (including anonymous) ComObjects.
- The site table is traversed. All sites that were not marked need to check if they can be garbagecollected. This is done by asking the collection of garbage collection through the instruction. If the ComObject still has a TransObject but no need, it will send a C_CLEAR_REFERENCE message. If it also thinks that the remote peer has no need, it will send a C_CLOSE_WEAK message. Then it will answer no. If it is closeable the close protocol will close the connection and at the next garbage collection, the ComObject will be collected to [4].

17). *Resource caching:* Using a transport media is often connected to some limited resource. In the case with TCP, this resource is file descriptors; with UDP it is port numbers and with shared memory it is number of shareable memory pages. In order not to put a limit on the number of connected sites, the available resources have to be shared fairly between connections. In this model, this is done by a mechanism called resource caching. A positive side effect of resource caching in this way is that the memory usage is also limited. TransObjects contain ByteBuffers that may be large, and this way there will always be a limited number of ByteBuffers. The number of resources available can be set from oz-level as a weak and a hard limit. The weak limit may be temporarily exceeded for incoming requests, but the hard limit is definite. When the resources available run out,

the TransController will start a timer and try to preempt the resources of some ComObjects. The ComObjects then have to nicely close their connections and hand back the TransObject to the TransController who can grant it to the first waiting ComObject.

18) *Preemption:* A big issue for efficiency with e.g. servers is whom to preempt, how many and when [5]. This is a scheduling problem. For now a naive model is adopted:

- Whenever a resource for accept is granted, and the weak limit is exceeded, or else if a connect is requested and the weak limit is reached; start a timer.

- The timer will preempt the resources of one ComObject at regular intervals. The decision on which one to preempt is done as follows: list of running ComObjects is traversed and the first one that can be closed (currently defined as being in state working) will be chosen with the following prioritizing:

1.Has non-empty buffers.

2.Has empty buffers but queued messages.

The basic advantage of this model is its disadvantages are plenty as given in the following:

- No respect is given to incoming messages. Thus the one connection where a message resolving a suspension in this engine is to come might be closed, or a server might close an active client instead of a client that just forgot to disconnect.
- The empty buffer criteria might not be appropriate for all transport mediums.
- Clients behind firewalls should be treated separately since it might be unfeasible to reconnect from the outside.
- No guaranties are given on how long a connection can be up prior to it will be closed again.

19) Error handling

In the cases where it can be definitely known that a remote site is permanently down, this needs to be reported to upper layers. Those cases are when the open protocol determines the remote site to run the wrong version or to be a different site than expected, or possibly if the transport media reports that the site is permanently down. The new fault model will give the application programmer some way to close the communication with a remote site. The importance in being able to do so is to free resources such as memory and any other resources connections use.

II. PERFORMANCE EVALUATION

For the complete evaluation of the performance changes, one would have to study how the performance has changed for each of the distributed entities as well as for larger applications performing various different tasks. The performance can be evaluated based on different parameters, throughput, simultaneous connections, multiplexing and the memory usage. A generic test measuring roundtrip times is constructed. It consists of one server and a number of clients. The server listens to a port for incoming messages, and simply sends these messages back. It also triggers the clients to begin and measures the total time from the trigger to when all clients are done. The server can also poll its memory usage. Every client sends lists of a specified size and waits for it to come back. They measure the total time for a number of iterations. Specifiable parameters are number of clients, size of lists, and number of iterations. All tests are run over a LAN at off peak hours. The server runs on a workstation and all clients on a CPU server. Comparative tests are run in a sequence of new, old, new, old to verify reproducible results.

A. Throughput

Throughput is measured by starting one client and using larger and larger lists to send data for a number of iterations. The average roundtrip time is recorded and plotted in **Error! Reference source not found.**5 . This indicates that the performance has improved for all sizes i.e. the size of the list (x) in elements versus the average roundtrip time (y) in milliseconds. It should be noted that these results depend on the number of iterations. At 1000 number of iterations the differences vanish for list sizes below about 300 elements, but remain for larger lists. At less iterations the differences increase.

B. Simultaneous connections

The list size is set rather small, and the number of clients is gradually increased. The total times at server and clients are measured and plotted. It is also attempted to find an upper limit on the number of clients one server can handle simultaneously. The engine per default allows thirty file descriptors to be used simultaneously .Results are displayed in **Error! Reference source not found.**5. And a comparative study for old and new engine is also shown in table 1.

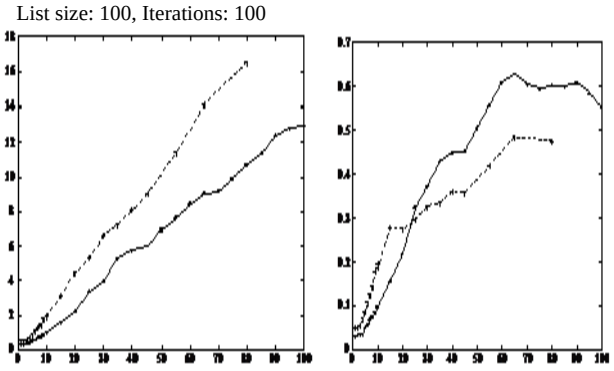

Fig 5. The average round trip at each client(y) in ms verses Number of clients (x). The total number of servers(y) Verses Number of clients (x).

TABLE 1. TEST ON MULTIPLEXING

Number of clients/threads	Old engine	New engine
10	0,93	0,75
30	1,3	1,4
60	1,3	1,5

With many clients, the old engine test only rarely came to a result, explaining the lack of data in that graph. With the new engine, tests with up to 170 clients were successful. Beyond that, the operating system did not permit more processes on the CPU server. Further testing with more machines will be conducted in the future. When viewing the total time of the server in Figure 5, it is evident that the performance has improved, The strange shape of the curves can be explained by the way the test is written. All clients introduce themselves and then wait for the server to react before they start their timer. This is to make sure that they really run simultaneously. With many clients, the response from the server will be delayed until there is a connection to a particular client. This means a client can starve for some time and then still measure a short time. The fact that the clients running with the old engine get a lower time comes from the preemption decision. The old engine has a more sophisticated algorithm that closes the least recently used connection. Hence we can conclude that performance has improved, but resource scheduling can be further refined. It should be considered that different applications might favor clients getting done fast, while others might want to let clients use resources fairly.

C. Multiplexing

Roundtrip when several threads at one client send to the same server is measured and compared to the results of simultaneous connections above. Since all threads in one oz engine use the same communication channel, this will test effects on the performance due to messages being sent in one or more chunks.

III. CONCLUSION

It was possible to implement a new network layer with more complex features, and still pertain or even enhance the performance of the existing and working network layer of Mozart. It was also likely to achieve a higher scalability than the network layers of Java and Mozart with effortless means. The maximum amount of memory needed for message passing can be known in advance. There is much more to be done to refine this model and further improve its performance and make it dynamically changeable.

REFERENCES

[1] Seif Haridi, Peter Van Roy, Per Brand, Christian Schulte. Programming Languages for Distributed Applications, 1998
[2] Java Remote Method Invocation Specification, JDK 1.2. Sun Microsystems, 1998.
[3] Andrew S. Tanenbaum. Distributed Operating Systems, ISBN 0-13-143934-0, 1995
[4] Luca Cardelli. A Language with Distributed Scope. Digital Equipment Corporation, Systems Research Center, 1995.
[5] Ralf Scheidhauer. Design, Implementierung und Evaluiering einer virtuellen Maschine für Oz. Dissertation, Universität des Saarlandes, 1998
[6] Andrew Birrell, Greg Nelson, Susan Owicki, Edward Wobber. Network Objects. Research Report 115, Digital Equipment Corporation, Systems Research Center, 1995.
[7] Connection procedure, being designed and described by Konstantin Popov and Erik Klintskog
[8] Frequently Asked Questions - RMI and Object Serialization. Available at: http://java.sun.com/products/jdk/1.2/docs/guide/rmi/faq.html, January 2000
[9] Java Remote Method Invocation. Available at: http://java.sun.com/marketing/collateral/rmi_ds.html, 1999
[10] Andrew S. Tanenbaum. Computer Networks, ISBN 0-13-394248-1, 1996
[11] Jan Tångring. Mozart: koncis och snabb, Datateknik 3.0, No 3 1999