DESIGN & IMPLEMENTATION OF A RELIABLE MULTICAST PROTOCOL FOR DISTRIBUTED VIRTUAL ENVIRONMENTS WRITTEN IN JAVA

Gunther Stuer
Frans Arickx
Jan Broeckhove
University of Antwerp
Department of Mathematics and Computer Sciences
Groenenborgerlaan 171, 2020 Antwerp, Belgium
gstuer@ruca.ua.ac.be

KEYWORDS

Reliable Multicast, Java, Distributed Virtual Reality

ABSTRACT

The goal of this paper is to present the full design and implementation scheme regarding the construction of a reliable multicast protocol suitable for distributed virtual environments written in JAVA. Several techniques that are used to reduce system resource usage and to improve performance will be highlighted.

INTRODUCTION

First, let us put this paper in a broader perspective by describing its relevance in the development of a highly dynamical distributed virtual environment.

One of the main bottlenecks in virtual environments has always been the availability of sufficient network bandwidth to allow the participating objects to communicate with each other [1]. With the introduction of multicasting this problem was partly solved, but traditional multicast protocols have two drawbacks. The first problem is that they are based on *best effort approaches*, i.e. message delivery is not guaranteed. In order to achieve this guarantee, *reliable multicast* protocols were introduced [15]. Although there are already many such protocols, none is optimised for distributed virtual environments [2]. Also, to our knowledge, an implementation in Java has not yet been attempted. The second problem is that multicast groups are statically allocated [16]. With virtual environments one usually considers spatial criteria to divide the world in partitions, where each partition transmits its data on one multicast group. However, with dynamic environments this is not sufficient anymore. Participants have a tendency to flock together and this leads to situations where some groups are very heavily used, while others are completely idle. Allocating multicast groups in a dynamical way can solve this problem [3]. Techniques that can be used for this include probing [4] and fuzzy clustering [5]. With these methods we can determine at runtime which participants should be put together in the same multicast groups.

DESIGN GOALS

We have built our reliable multicast protocol taking into account a number of design goals:

1. The protocol will be used in distributed virtual reality systems. From previous work [6] we know that this has some interesting implications. (a) The typical message size used in virtual reality applications is rather small (< 1kB) because once the viewers know what an object looks like and where it is positioned, one only needs to transmit the changes with respect to that information. (b) Because a frame rate of 30 Hz is considered acceptable, there is no point in sending more than 30 update messages per second. (c) When dead reckoning algorithms – i.e. determination of the current position on the basis of previous positions – are applied, an update rate of once per second will often suffice. (d) When a message does not arrive during the first few seconds after is has been sent, it has completely lost its relevance to the virtual world.

Based on the average message size and the maximum number of message sent per second, we can make a realistic prediction about buffer sizes and timeout windows that are key parameters in implementing the reliability in the protocols. It can also improve performance because we do not need to resize our buffers while in full action. Because we know the average and the maximum throughput, we can apply the Usage Parameter Control (UPC) algorithm a.k.a. leaky bucket algorithm. This algorithm can be used to control bandwidth usage. An example of this is its use in ATM networks [18]. And most importantly, we can relax the reliability criteria. Having the sender buffer messages, for possible retransmissions, only for a *certain amount of time* and then discarding them is appropriate for our problem. The amount of time may vary depending on the type of message. This way we can assign each message an *importance factor*. Important messages should be kept longer in the buffer.

2. The reliable multicast protocol has to be implemented in JAVA. The main motivation is that the virtual reality system that is being designed will be implemented in Java. We chose Java because it has features that we want

to use, such as multithreading, loading classes across the network and the write-once-run-everywhere strategy.

3.Since we do not need high throughput, we decided to let good design prevail over performance. This way, the code is more maintainable and can also be used for educational purposes. One example of this is that we kept the interface as simple as possible.

4.The primary design goal for our VR system is that it has to be distributed. The termination of a node should only have minimal impact on the whole. As a consequence of this, the reliable multicast architecture used also has to be completely distributed. This means that every participating node should be independent of all others to make the multicast reliable.

BACKGROUND INFORMATION

In the classification of reliable multicast protocols [7,8] the approach that we use is most closely related to the Transport Protocol for Reliable Multicast. When one classifies on the basis of data buffering mechanisms [9], it is a receiver-initiated approach. This means that no acknowledgements (ACKs) of receipt are used. Instead, the receiver transmits a negative acknowledgement (NACK) if retransmission is needed, because of an error in the message, because a skip in sequence numbers indicated a missing message or because a timeout has elapsed.

With this approach, two problems can arise: (1) a NACK implosion due to the detection of a missing packet by many receivers, and (2) buffer size limitations at the sender side. Indeed, in principle, the sender needs to keep all messages available for retransmission because a NACK may arrive at any time. One never known whether a message has been successfully received by all interested parties. This leads to the fact that buffers should in theory be infinite.

Waiting a pseudo-random time interval before sending a NACK solves the first problem. When a client is waiting to send a NACK and in the mean time it receives a NACK-request from another client for the same missing packets, it can drop his own request. The second problem is solved heuristically by assuming that messages are of no further interest after a configurable amount of time as indicated above.

IMPLEMENTATION

In this paper we consider the implementation in Java of our reliable multicast protocol, suitable for distributed virtual reality applications. Details of some of the algorithms and a proof of concept implementation were discussed in [10]. The full implementation, considered here, is characterized by improvements in the overall design, by the introduction of design patterns [12], by increased stability and by significant performance enhancements. In our prototype implementation the use of multithreading was pushed to the limit. Each message was handled in its own thread, both at the sender and the receiver side. Tests indicated that thread creation was a bottleneck. Hence the design was modified to include a thread pool and the recycling of used threads to handle messages. This resulted in a speedup of approximately 200%. But because thread scheduling in Java depends upon the multithreading library of the underlying operating system, important differences, which lead to instabilities, were seen between e.g. Windows and Solaris based hosts. This is why we abandoned the thread pool principle and wrote our own scheduler. Now, each message can register itself with this scheduler and request to be awoken at a certain time.

ALGORITHMS

In this section we will describe the most important algorithms at work in our multicast library. Each algorithm is characterized by one or more parameters. The initial values are set on the basis of empirical data. To discuss the algorithms we describe each parameter and indicate its effect on them.

packetSize: This parameter determines the size of one packet. Packets are the atomic units sent across the Internet. A message is split up into a sequence of packets. Too small values will create an overhead on the algorithms, and an unrealistic header/data ratio. Too large values on the other hand will lead to redundant bytes in final message packets. This parameter defaults to 1024. We chose this value for two reasons. We know from experience that the size of a typical VR message is less than one kilobyte. Secondly, because multicast uses UDP, we must take care not to exceed its limits [17].

sendTimeOut: After a message has been sent, it has to be kept available for a sufficient long time to handle incoming NACK requests. This time interval is configured using the *sendTimeOut* parameter. By default it is set to thirty seconds because after this, it is of no further interest to the virtual environment and the message can be discarded. This value can be overridden for individual messages. The default is only an initial value. Our multicast system uses an adaptive algorithm to determine the optimal value for the timeout. In figure 1 we show its effect on the sending algorithm.

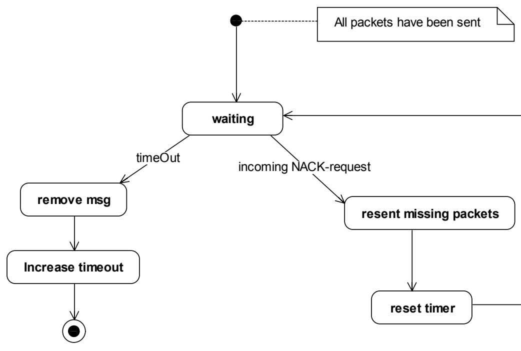

Figure 1: timeout effect when sending

maxSending: To prevent flooding, we introduced the *maxSending* parameter. It indicates the maximal amount of messages that can be sent per second. By default it is set to thirty because of the characteristics of VR applications.

maxNacks: The amount of NACKs sent consecutively for a particular message is limited by this parameter. Whenever a response to one of the NACKs is received, the counter is reset to zero. This limitation is important to account for senders disconnected in the midst of sending a message. By default it is set to ten retries.

recvTimeOut: Indicates the time interval the receiving side waits between two incoming packets of one message before sending a NACK containing the list of missing packets for that particular message. If set too large the system will idle for too long on defective messages and responsiveness will drop. If set too small unnecessary NACKs for well-sent and not yet received packets will be transmitted, leading to redundant duplicate packets. The value defaults to 150 milliseconds. This parameter is also updated using an adaptive algorithm. Whenever a NACK is transmitted, the connection is understood less reliable than expected, and the timeout is increased by 40%. Whenever a message is received without having to issue a NACK the connection is expected to be better than presumed, and the timeout is decreased by 10%. This way one can cope with changes in the network and will always find an optimal setting. This is done on a per sender base. Its context is shown in fig. 2.

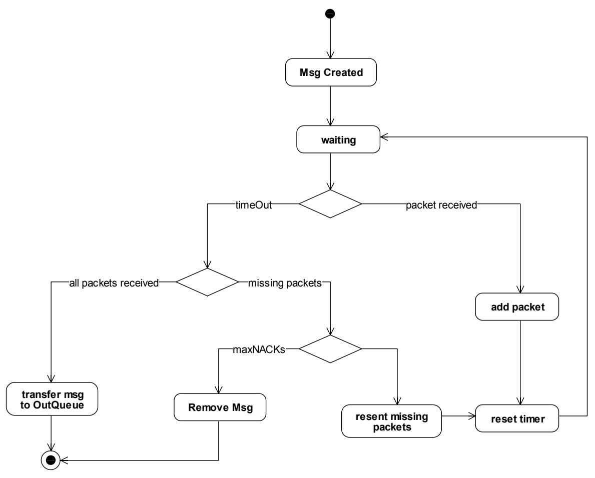

Figure 2: Receiving a message

nackTimeOut: This is the time interval between sends of two consecutive NACKs. If no response on a NACK after this time interval was received, one can assume it was lost, and a new NACK has to be sent. If the value is too large, too much idle time will be spent on defective messages and responsiveness will drop; if too small unnecessary NACKs will be sent. The same adaptive algorithm is applied as for recvTimeOut.
socketNr: The socket number the protocol will use. By default it is 6789.

THE DESIGN

We present an overview of the design of our reliable multicast protocol using the diagrams of the UML notation [11]. When appropriate, the applied design pattern [12] will be indicated.

User-View
This part describes the users and what they can do. The reliable multicast protocol is implemented as a library with two *users*, being the network and an application with the need to reliably multicast a message (see fig. 3).

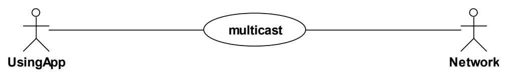

Figure 3: Context Diagram

User *network* can only transmit and receive individual packets. The library is designed as a singleton pattern [12]. This means that there can only be one instance active. Once the *using application* has obtained the singleton-instance it can perform the following tasks (see fig. 4):

When the application want to receive messages coming from group X, it can join this group using the *joinGroup (InetAddress X)* method call.

When the application is no longer interested in messages coming from group X, it can instruct the library to stop listening using the *leaveGroup (InetAddress X)* method call.

Sending a message is performed in two steps. First one needs to acquire an output stream to write the data to. A call to *getPacketOutputStream ()* allow for this. Once an output stream is obtained and filled up, a *sendMessage (PacketOutputStream, InetAddress)* is issued to send the data to the supplied multicast group. Note that it is possible to send the same data to different groups.

All incoming messages are queued until the user picks them up through the *recvMsg ()* method. If the queue is empty, this method will block. To prevent this, a call to *recvMsgAvailable ()* will return the amount of received messages available in the queue.

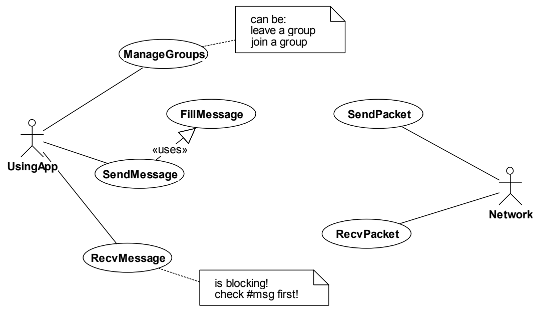

Figure 4: Use-Cases

Data Flow
Secondly, the dataflow through the multicast system is considered. This describes the type of buffers an outgoing or incoming message must pass through and discusses the choice of data structures [13]. There are two major data flows in the library. When sending a message there is a flow from the application to the network. When receiving a message there is a flow from the network to the application.

Let us start by describing the dataflow when sending a message (fig. 5). When *sendMessage()* is called, the message is appended to a queue, which is used as an input buffer temporarily storing messages to be sent. The draining rate of the buffer is governed by *maxSending*. After this, the message is transferred to a hash map [13], where it is split up in a sequence of packets, which are added to a priority queue from where they are individually sent across the network. The message is discarded from the hash map whenever a *sendTimeOut* timeout occurs. If a NACK is received, the concerned message should be localized as quickly as possible. As each message has a unique id, and the NACK contains it, a hash map is the best choice for optimal localization.

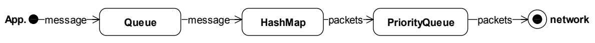

Figure 5: Data Flow - Send

Next, we describe the dataflow that occurs upon reception of a message (see fig. 6). Messages typically consist of multiple packets. An incoming packet from the network is inserted in a queue with the sole purpose to decouple the network from the library. When packets arrive too quickly to be handled in real-time this is where they are temporarily stored. From here they are transferred one by one to the hash map of (id, message) pairs. The packet is then appended to the corresponding message. When the message is complete it is transferred to another queue where it waits for the *user*.

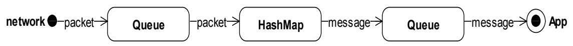

Figure 6: Data Flow - Receive

Components
Finally the individual components are presented in some more detail to describe their action and implementation.

SettingsParser
This component reads all parameters, described above, from an XML [14] file. It is implemented as a singleton [12]. The file is parsed and stored in a hash map. The user can call the *getValueAsString (path)* or *getValueAsInt (path)* methods. They will search for the specified path and return the corresponding value if found. A *path* has the following syntax: "[tagName/]*[tagName]".

Timer
The timer component notifies interested parties that a certain amount of time has elapsed. It consists of several classes (*Timer, TimerEvent and LessTimerEvent*) and one interface (*ITimer*) (see fig. 7).

New notifications have to be scheduled. A *TimerEvent* object that has to be created with two values, the object to notify and a timeout in milliseconds implements the notifications. A timeout can be set in three distinct ways: as an absolute value in milliseconds, a relative value w.r.t. the current time in milliseconds, or a *Calendar* object.

All objects that want to be notified must implement the *Itimer* interface. It contains one method, *wakeup (TimerEvent)*, which is called by the scheduler when the specified amount of time has elapsed. Because an object can wait for more than one event, the *TimerEvent* that has generated the notification is returned.

The *Timer* class implements the scheduling algorithm. It has an *OrderedSet* [13] of *TimerEvent*s. *LessTimerEvent*, a binary predicate, determines the order. The *TimerEvent* that comes first will also be the first in the list, and so on. The timer-mechanism is implemented as a scheduler to allow for a single thread that we experienced to be faster and more stable than allocating individual threads to each timer event. We chose an *OrderedSet* as a data structure because (1) it is ordered and (2) it is fast in finding a *TimerEvent*.

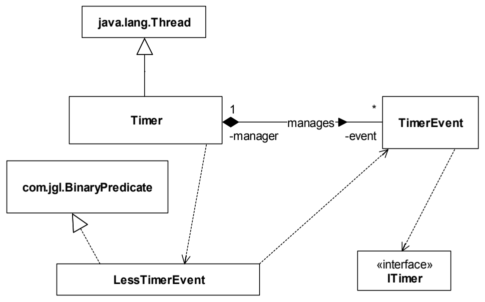

Figure 7: Timer - class diagram

Multicast
The *Multicast* class is the only one a user of the library needs to know about. It is an implementation of the façade [12] design pattern. It collects all the methods the user needs to access. (See user-view).

Input/Output
All data enters or leaves the library in a stream. When the user wants to send a message, a *PacketOutputStream* is returned. As far as the user is concerned this is just an ordinary output stream with all the operations one can expect. Internally however the data are organized in a singly linked list of *Packets*. Each packet contains a parameterised amount of data. Packets are added as needed. When the message is to be sent across the network, this can be done with minimal overhead because it is already chopped up in a sequence of packets.

For incoming packets, we use the same reasoning. The user only accesses an input stream with the usual operations. Internally, the data is organized as a linked-list of packets. As above this ensures a minimal overhead.

Packet
In the Packet component three classes appear, the abstract super class *Packet* and two derived classes *MsgPacket* and *NackPacket*. Every packet contains a configurable amount of bytes. The first eight bytes always contain the common header information; the rest contains specific header information and user data depending on the packet type. Each packet shares its buffer with one *DatagramPacket*. This eliminates the need to copy the packet buffer to the datagram buffer when it has to be transmitted. *MsgPacket*s are used, as the name suggests, for sending and receiving messages. Bytes 9 through 19 are used for storing the specific header information. The streams mentioned earlier start at byte 20. A *NackPacket* is used to transmit and receive NACKs. Every *NackMsg* consists of exactly one *NackPacket*.

SerialNrManager.
Because every message needs to be uniquely identified in the multicast system we introduce a component to specifically manage this. The *SerialNrManager* contains a hash map containing the last known serial message ids received from a certain host via a certain multicast group.

When a packet is received, its host id and the group it came from are extracted. Using this information the hash map is checked for the last serial message id received. In this way gaps in serial message ids can be detected.

When a new message is to be sent to a certain group a new message id must be provided. It is obtained from the *SerialNrManager* that returns the last active message id, on this group for the local host, incremented by one.

PacketFactory
The *PacketFactory*, based on the factory pattern [12], has two responsibilities. The first is receiving packets from the network. All incoming packets are inserted into a queue to decouple the network from the multicast library. The second one is to keep track of all multicast groups one should listen to. It is important to note that all NACK replies will be transmitted on the same multicast group that the original message was received from. This implies that the group on which the message is sent has to be listened to until the message has timed out and left the system.

Messages.
This component represents the most important concept of our multicast system. It collects all the messages to be transmitted across the network. The component contains an abstract super class *Msg* and three derived classes *NackMsg, SendMsg* and *RecvMsg* (see fig. 8). When constructed, every message registers itself on the scheduler so that it will be notified when its timeout (resp. *SendTimeOut,* or *RecvTimeOut*) period has elapsed. When a new message is to be sent, a *SendMsg* object is created. This class of messages is responsible for constructing the message header for every packet it consists of, to handle incoming NACK requests for this message, and to pass the individual packets to the *PacketDispatcher* where they can be transmitted. When a new message is received, a *RecvMsg* object is created. It contains the logic to add incoming packets to its data structure and to check whether all packets have arrived. If not, and the *maxNackRequest* has not been reached, a NACK is issued. A *NackMsg* is constructed when a NACK has to be sent. It contains the id of the targeted message and a list of all missing packets. It is transmitted on the same group the original message was received from.

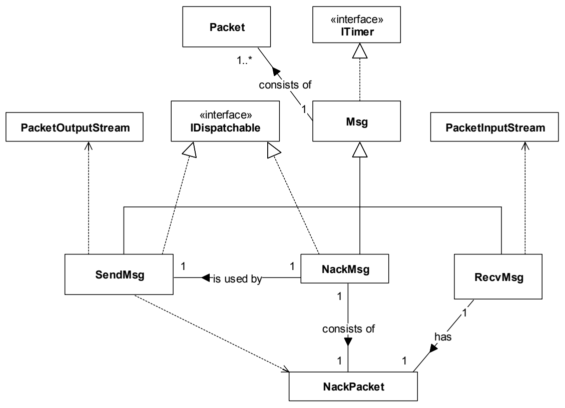

Figure 8: Msg class diagram

MsgContainer
If one component should be identified as the core component of the reliable multicast system this would be it. It contains all incoming and outgoing messages for which the timeout has not yet expired. It is implemented as a hash map to enable very fast location of individual messages upon reception of NACK requests as explained extensively above. Fig. 8 shows the processing of incoming packets in the *MsgContainer*.

Dispatcher
The dispatcher is the component responsible for the transmission of packets. The *PacketDispatcher* is implemented as a priority queue of *DispatchObjects*. Each one holds a message to dispatch and a priority. We differentiate between three priorities. From high to low we have: NACK-requests, NACK-responses and messages. The dispatching algorithm is non pre-emptive. When a message is transmitted, no matter what type, all packets will be transmitted in sequence without interruption. This is important to make sure that no *recvTimeOut* would be triggered at the receiver.

Messages have the lowest priority because no entity is waiting on a message not yet sent. NACKs on the other hand have to be handled as fast as possible because of the strict timeout values.

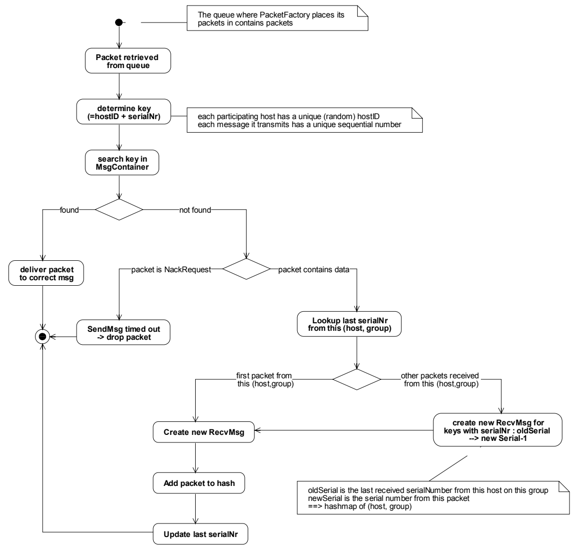
The queue where PacketFactory places its packets in contains packets
Packet retrieved from queue
determine key (=hostID + serialNr)
each participating host has a unique (random) hostID each message it transmits has a unique sequential number
search key in MsgContainer
found
not found
deliver packet to correct msg
packet is NackRequest
packet contains data
SendMsg timed out -> drop packet
Lookup last serialNr from this (host, group)
first packet from this (host,group)
other packets received from this (host,group)
Create new RecvMsg
create new RecvMsg for keys with serialNr : oldSerial --> new Serial-1
Add packet to hash
oldSerial is the last received serialNumber from this host on this group newSerial is the serial number from this packet ==> hashmap of (host, group)
Update last serialNr

Figure 8: Processing incoming packets

FUTURE WORK

This reliable multicast protocol is an important element in a much bigger project: the creation of a highly dynamical distributed virtual environment.

The next step will be the design and implementation of the probe classes [4], which will strongly decrease the total amount of messages sent when new objects enter the virtual world. This technique is based upon the idea of sending chunks of code to the participants instead of data. This code will be able to negotiate whether the two objects are interested in each other, or not. If they are, the multicast groups on which they transmit their data will be exchanged.

As a next step, we will implement the fuzzy clustering algorithm [5] to dynamically allocate a fixed set of multicast groups to all participating objects. The most challenging task will be to define the criteria, which will determine which objects should be grouped together at a certain moment in time. To make this mechanism as flexible as possible, the criteria will be described in XML [14].

CONCLUSIONS

We think that we can safely conclude that the current version of the reliable multicast protocol for distributed virtual environment written in Java meets its design goals. Performance analyses, which will be presented in two forthcoming papers, prove that the performance is more than adequate. The only limiting factor is the amount of messages a node can handle per second, which is currently about 350.

REFERENCES

1. Michael J. Zyda, “Networking Large-Scale Virtual Environments”, Naval Postgraduate School, Monterey, California, USA.
2. Kenneth P. Birman, “A Review of experiences with reliable multicast”, Software – Practice and Experience 29(9), 741-774 (1999)
3. Chris Greenhalgh, “Spatial Scope and Multicast in Large Virtual Environments”, Technical Report NOTTCS-TR-96-7 University of Nottingham, UK, 1996
4. Gunther Stuer, Jan Broeckhove, Frans Arickx, “A message oriented reliable multicast protocol for a distributed virtual environment”, ICSE’99 (CS-163)
5. C. Looney, “Fuzzy Clustering: A new algorithm”, ICSE’99 (CS-115)
6. Kris Demuynck, Jan Broeckhove, Frans Arickx, “The VEplatform system: a system for distributed virtual reality”, Future Generation Computer Systems 14 (1998), pp. 193-198.
7. Katia Obraczka, “Multicast Transport Protocols: A survey and taxonomy”, IEEE Communications Magazine, January 1998, pp. 94-102.
8. B. Sabata, M.J. Brown, B.A. Denny, “Transport Protocol for Reliable Multicast: TRM”, Proc. IASTED International conference Networks, January 1996, pp. 143-145.
9. Brian Neil Levine, J.J. Garcia-Luna-Aceves, “A comparison of reliable multicast protocols”, Multimedia Systems 6 (1998), pp. 334-348
10. Gunther Stuer, Frans Arickx, Jan Broeckhove, “A message oriented reliable multicast protocol for J.I.V.E.”, Parco99, Parallel Computing – Fundamentals & Applications, pp. 681-688.
11. Grady Booch, James Rumbaugh, Ivar Jacobson, ”The Unified Modeling Language User Guide”, Addison-Wesley.
12. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns”, Addison-Wesley.
13. Mark Allen Weiss, “Data Structures & Problem Solving using Java”, Addison-Wesley.
14. St. Laurent, “Building XML Applications”, Osborn McGraw-Hill.
15. Kara Ann Hall, “The implementation and evaluation of reliable IP multicast”, University of Tennessee, Knoxville, USA, 1994.
16. Chris Greenhalgh, “Dynamic, embodied multicast groups in MASSIVE-2”, Technical Report NOTTCS-TR-96-8, University of Nottingham, UK, 1996.
17. W. Richard Stevens, “TCP/IP Illustrated, Vol 1: The protocols”, ISBN 0201633469
18. William Stallings, “Data & Computer Communications, 6th edition”, ISBN 0130843709, pp 405