Sixrm: Full Mesh Reliable Source Ordered Multicast

Pedro Ferreira, João Orvalho and Fernando Boavida
LCT - Laboratory of Communications and Telematics
Centre of Informatics and Systems of University of Coimbra (CISUC)
Polo II, Pinhal de Marrocos
3030 Coimbra, Portugal
+351 239 790 091, +351 239 790 035, +351 239 790 012
E-mail: pmferr@dei.uc.pt, orvalho@dei.uc.pt, boavida@dei.uc.pt

Abstract: **In this paper we describe Sixrm, a reliable multicast protocol that supports multiple sources that are at the same time multiple destinations and that works both on ipv4 and on ipv6. The protocol is completely configurable by profile settings and is implemented completely in Java Standard Edition 1.5.**
The protocol is included on the distributed server part of network middleware for large scale mobile and pervasive augmented reality that we also introduce in this paper, and supports the ordering of messages by source without duplicates and its reliable delivery to the application, without the support of network elements.

1. INTRODUCTION

A significant requirement of pervasive applications is fast service development and deployment [1], which implies the introduction of various service and application frameworks and platforms. For this, middleware is a common solution. The benefits of middleware utilization are the improved programming model, and the hiding of many implementation details, which make middleware based application development much faster. It is now becoming quite clear that entertainment, and more specifically mobile gaming, will be one of the killer applications of future wireless networks [2]. Augmented reality extends reality with virtual elements while keeping the computer in an assistive, unobtrusive role [3]. It is possible to create games that place the user in the physical world through geographically aware applications. Most of the latest mobile phones are equipped with cameras and some of the latest ones are coming with some form of 3D rendering technology [4] [5]. Bluetooth technology and increasing miniaturization will lead, in the near future, to low-cost, specialized pervasive equipment for augmented reality. In [6] we described the main objectives of our research concerning systems that satisfy the requirements of network middleware for large scale mobile and pervasive augmented reality games. In [7] we described a middleware system that is being developed for large scale mobile and pervasive augmented reality games that satisfies these objectives. The system targeted by the middleware is composed of 3 levels: the back-office central level, the large scale network level, and the personal area network level. This paper addresses one of the constituents of the large scale network level.

2. RELIABLE MULTICASTING

Reliable multicast of packets is a requirement of large scale distributed entertainment applications. However, it has been shown that current solutions for that effect have significant problems [8], examples of those being the nak or ack explosion problems. The reliable multicast transport working group of IETF [9] has been addressing the problem of reliable multicast transport, but only in the one-to-many approach, not in the many-to-many area. The experimental protocol PGM [10], the only many-to-many protocol that has reached RFC status in IETF, requires support by network elements. This is hard to get implemented in reality. We felt it was necessary to create a protocol capable of working in the many-to-many scenario, without the nak implosion problem but nak based, source ordered and that avoided duplicates. We also needed a protocol that could work in ipv6, and that does not require the support of network elements. So, we created Sixrm.

3. THE SIXRM PROTOCOL

The Sixrm protocol consists of six types of packets. These are: Data packets (TYPE_DATA), Open packets (TYPE_OPEN), Close packets (TYPE_CLOSE), nak packets (TYPE_NAK), information packets (TYPE_INFO), and error packets (TYPE_ERROR). It is a nak based protocol, in which the destinations, when they detect that they did not received a packet from certain source, transmit a nak for the group that only that source will listen to and repeat the transmission of the lost packet. If that is not possible, an error packet will be transmitted for the destination. A source first opens the channel for communication sending a packet for all destinations in the multicast group, an open packet (TYPE_OPEN), that all currently listening destinations will receive. This corresponds to the Open operation on the Sixrm node. The packet contains the source identity and source ip address. All currently open destinations (those that already have executed the Open operation), when receiving the open packet, perform initialization for that source, and respond to the open packet with an information packet containing the destination (and possible source) identity and ip address. The source (or destination) identities are Java long data types. When receiving the information packets, the source performs

the same initialization that the destinations did when they received the open packet.
For the close operation, the source transmits the close packet and removes itself from the group. All destinations, when they receive the close packet, perform cleanup operations for that source.
As for data transmission, data is transmitted sequentially. Each new byte buffer that is handled to the Sixrm protocol for transmission is buffered for transmission and subsequently transmitted, in the next available time slot. When the data is transmitted, it is given a monotonically increasing sequence number, packed in a packet of type data (TYPE_DATA) and send to the group for all destinations to listen.
It must be mentioned by now that every packet transmitted by Sixrm contains the source transmit memory buffer state (send window), which includes all messages since the last one memorized until the newest one that was sent (the biggest sequence number). The newest and oldest sequence numbers are transmitted with each message.
Each destination maintains a receiving window, for each source, that is smaller than the source transmit window and which has is greatest sequence number always equal to the source transmit window newest sequence number and its oldest sequence number adjusted to be that number minus some minimum value.
Each time a source transmits a value, it puts that value in its sent buffer (transmit window), and if necessary, if the buffer can not grow anymore, discards the oldest message that was in the buffer. The value is transmitted and stored in the buffer in a packet of type data (TYPE_DATA).
Each time a destination receives a packet, it verifies witch kind of packet it is. Depending on the kind of packet, the destination acts differently.

3.1 Packets of type nak

When a destination (which is also a source) receives a Nak packet, it means someone did not receive a packet. The Nak packet contains the source address, the source identity, the sequence number, the target address, and the target identity.
We can tell if the nak packet is for us by comparing the target address and target identity with our ip address and identity. If equal, it was our packet that was not received, if not equal, it was someone else's packet that was not received, but we still need to memorize the nak. Our answer will depend on the fact that the nak was direct to us or not.

3.1.1 Nak directed to us

If the Nak was directed to us, then we must try to resend the packet that was lost. Unfortunately, that may not always be possible. Given the sequence number of the nak, we access the sent memory (transmit window), and verify if we have the packet.
If we have the packet, then, we resend the packet to the group. We do not memorize the nak.
If we do not have the packet, then we can not recover from the error and must send an error packet to the destination informing the destination that we do not have the packet with that sequence number. We also send an error packet to the application telling the application that a data packet with a determined sequence number did not reach all destinations.

3.1.2 Nak directed to someone else

If the nak was directed to some other destination (and source) then we must memorize the nak to prevent us from, in the near future, transmitting a nak for the same packet. For this effect, we memorize the nak and a time value that is equal to the current time plus a random back off time value during which we cannot transmit a nak for this packet. We do not prevent the transmission forever because even naks can be lost. In this way, probably we will receive the packet related to the nak if it is missing and clear the nak before the need to transmit it again arises.

3.2 Packets of type error

When a destination (which is also a source) receives an error packet, it means a source could not retransmit a packet for a destination. The error packet contains the source address, the source identity, the sequence number of the packet that generated the error, and the target address and target identity.
We can tell if the error packet is for us by comparing the target address and target identity with our ip address and identity. If equal, it was our packet that wasn't available, if not equal, it was someone else's packet that wasn't available. Only if the error is for us do we deliver it to the application.

3.3 Packets of type data

When receiving a packet of type data, the destination first verifies if it did already initialize the memory structures for that source.
Then, if the packet sequence number is within the receiving window, it buffers the data packet. It also removes from memory any naks for that data packet that may exist.
A data packet contains the source address, the source identity, the sequence number, the data buffer, the transmit window newest sequence and the transmit window oldest sequence.
Taking for granted that the memory structures were already initialized, they are altered by the data packet, namely the receiving window. The newest element in the receiving window is always the transmit window newest sequence number.
After buffering the data packet, the destination verifies that the receiving window oldest sequence number is below the minimum difference with the newest and a packet with that sequence number exists. In that case, the packet is handed to

the application. The process repeats while possible until the receiving window oldest sequence number aligns with the minimum configured difference from the newest sequence number.
After this, the receiving buffer is verified for missing packets in between the receiving window. Naks are transmitted for each missing packet if is not forbidden to do so. It is forbidden to do so if the nak is memorized and its associated time value is greater than the current system time.

3.4 Packets of type open

Packets of type open means a source has executed the open operation and expects us to execute initialization of internal memory structures and respond with information packets with our information (source address and identity).
Open packets contain the source address and the source identity. When receiving an open packet, the destination (and source) will initialize its memory structures for that source, and send an information packet containing its information (that would be sent on an open packet) on an information packet.

3.5 Packets of type close

Packets of type close means a source has executed the close operation and will not be sending or receiving more packets with sequence numbers bigger than the latest.

3.6 Packets of type information

Packets of type information means a destination (and source) is responding to a request from a source open packet. It expects us to execute initialization of internal memory structures, if we did not that already for it.
Information packets contain the source address and identity. When receiving an information packet, the destination (and source) will initialize its memory structures for that source, if it did not already did so.

4. THE SIXRM API

The Sixrm API is a simple API with only 4 classes, all on the package pt.uc.dei.lcst.stf.sixrm .These classes are the following: SixrmEntity, SixrmFileKey, SixrmListener and SixrmPacket. We now describe in more detail each of these classes.

4.1 Class SixrmEntity

This class SixrmEntity takes care of communication for a Sixrm node (entity). A sixrm node acts simultaneously as a source and destination of Sixrm packets.
A sixrm entity is constructed from the identifier (long), the address (ip address – Java InetAddress), the group (ip address – Java InetAddress) ,the port (int), the ttl (time to live – int), the profile (Java Properties), and the listener (SixrmListener).
The profile contains configuration parameters summarized in Table 1.

Configuration	Meaning
maxOutBufferSize	Maximum size of an output packet
maxInBufferSize	Maximum size of an input packet
maxSentBufferSize	Maximum size of the sent buffer (transmit window)
maxQueuedBufferSize	Maximum size of the queue of messages to transmit
maxRecvBufferSize	Maximum size of the receive buffer
minRecvWindowSize	Minimum size of the receive window
maxRecvWindowSize	Maximum size of the receive window
minRandomBackoff	Minimum random back off time of nak packets
maxRandomBackoff	Maximum random back off time of nak packets
minIntervalTime	Minimum interval time between transmissions (adaptive throughput)
maxIntervalTime	Maximum interval time between transmissions (adaptive throughput)
intervalSteppingDown	Interval stepping down the interval time when there is no error or nak.
intervalSteppingUp	Interval stepping up the interval times when there is an error for us (2 xs) or a nak for us (1x).

Table 1 - Profile settings

When the SixrmEntity receives data packets or error packets, methods of SixrmListener are called.
Important methods of the SixrmEntity class are summarized on Table 2.

Method	Operation
void setListener(SixrmListener listen)	Sets the listener for received packets.
void sendMessage(byte[] b)	Sends a message (a data packet)
void Open()	Open operation
void Close()	Close operation

Table 2 - Methods of SixrmEntity

4.2 Class SixrmPacket

The class SixrmPacket represents a Sixrm packet. A Sixrm packet may contain a data buffer, a source ip address, a sequence number, a packet type, a source identity, a target identity, a target ip address, a source transmit window newest sequence number and a source transmit window oldest sequence number.
Basically, this class has constructors for the various types of packets that exist (TYPE_DATA, TYPE_OPEN, TYPE_INFO, TYPE_CLOSE, TYPE_NAK and TYPE_ERROR), and getter and setter methods for the data fields it contains.

4.3 Class SixrmListener

The class SixrmListener contains only one method, summarized in Table 3.

Method	Operation
Void receive(SixrmPacket pack)	Receives a packet (error or data).

Table 3 - Methods of SixrmListener

This is the interface that applications must implement.

4.4 Class SixrmFileKey

The Class SixrmFileKey is a helper class used to uniquely identify a source by its address and identity. A SixrmFileKey contains an ip address and an identity (long).
A SixrmFileKey instance identifies a source or destination of Sixrm packets (a Sixrm node).

5. TESTS

The Sixrm protocol API has it is implemented was subject to various functional and stress tests, with several profile settings. It was demonstrated that, according to the profile settings, one can have many instances of Sixrm running on the same computer (over 10 instances, if the settings are right). It was demonstrated also, that according to the settings, many more instances could be run on different computers. This is because they will not consume all CPU time, like we did on these stress tests, but only a minimum, and the sources will try to adapt to the slowest computer and weakest link (because of nak throughput adaptation, and error throughput adaptation, within the limits set forth in the profile).
We did tests on two computers connected by a 100 Mbit full duplex switch configured in a ipv6 network, and tests in one only computer configured to run in a ipv6 network.
Those were real tests, not simulated tests. The test results presented here are extracts from the tests with one computer, because they show the delay and jitter introduced solely by the responsibility of the Sixrm protocol.
We present here graphics for delay and jitter for one of the nodes (received at the first node) of a Sixrm network witch achieved twelve nodes in the same computer without errors. When the thirteen's node was added, there were briefly some errors that were handed to the test application by the Sixrm protocol, in a period when the computer's processor was at a peak load of 100% and availability was arriving at its physical limits, which we believed were the causes of the inability of the node to handle processing data.
The profile settings for these tests are described in Table 4.
The Figure 1 represents delay for the second node on the system as received on the first node. The Figure 2 represents jitter for the same situation.

Profile value	Setting
maxOutBufferSize	3000
maxInBufferSize	3000
maxSentBufferSize	1000
maxQueuedBufferSize	5
maxRecvBufferSize	1000
minRecvWindowSize	10
maxRecvWindowSize	250
minRandomBackoff	10
maxRandomBackoff	50
minIntervalTime	25
maxIntervalTime	250
intervalSteppingDown	1
intervalSteppingUp	16

Table 4 - Profile settings for the tests

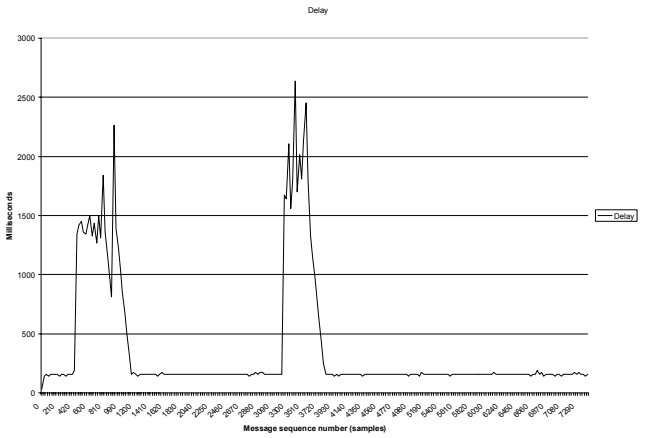

Figure 1 - Delay from second node on first node

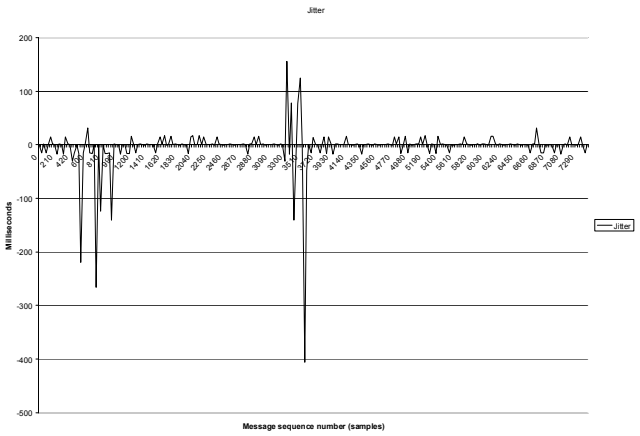

Figure 2 - Jitter from second node on first node

We do not show here results from more nodes both from lack of space and because the results are in all cases similar to these.
From these results we can conclude that the delay is normally always approximately 150 milliseconds and that the jitter normally varies between 0 and approximately 17 milliseconds. This happens consistently except some

exceptions, which occur exactly on the moments where we were adding nodes to the sixrm network and the network was adjusting itself. We can see that in these moments the delay and jitter briefly increase and then stabilise in acceptable values again. In fact, they stabilize in the same values as the number of nodes increases.
These numbers for delay and jitter (the stable ones) are adequate for most interactive delay sensitive applications.
As the server network will mostly be a stable one, as probably all nodes will be functioning since the first few minutes when integrated into the large scale network, the sixrm protocol is, we think, adequate for the purpose we create it for.

6. CONCLUSIONS

In this paper, we described the Sixrm reliable multicast protocol. We first described the protocol, then its implementation API, then the tests we have submitted the implementation.
We can conclude that the Sixrm protocol is adequate for most interactive delay sensitive applications that require a reliable multicast protocol that supports ipv6 with multiple sources and multiple destinations in which each source is a destination, delivering source ordered packets without duplicates, while not requiring support from network elements.
As for future work, we are going to integrate the Sixrm protocol in our large scale mobile and pervasive augmented reality network middleware, test it further too see how well it behaves, and do some optimizations.
For this, we will be integrating it with the ARMS – Augmented corba Reliable Multicast System – corba event service [11].

7. AKNOWLEDGMENTS

This work is being partially financed by the Portuguese Foundation for Science and Technology – FCT, and by the E-NEXT IST FP6 NoE.

8. REFERENCES

[1] Kimmo Raatikainen, Henrik Bærbak Christensen, Tatsuo Nakajima, "Application Requirements for Middleware for Mobile and Pervasive Systems", Mobile Computing and Communications Review, Volume 6, Number 4, October 2002, pp. 16 – 24 , ACM Press

[2] Keith Mitchell, Duncan McCaffery, George Metaxas, Joe Finney, Stefan Schmid and Andrew Scott, "Six in the City: Introducing Real Tournament – A Mobile IPv6 Based Context-Aware Multiplayer Game", Proceedings of NetGames'03, May 22-23, 2003, Redwood City, California, USA, pp. 91-100, ACM Press

[3] Hideyuki Tamura, Hiroyuki Yamamoto, and Akihiro Katayama, "Mixed Reality:Future Dreams Seen at the Border between Real and Virtual Worlds", Virtual Reality, November/December 2001, pp. 64 –70, IEEE

[4] Nokia – Developer resources (Forum Nokia), http://www.forum.nokia.com/, Accessed April 2004

[5] Sony Ericsson Developer World, http://developer.sonyericsson.com/, Accessed April 2004

[6] Pedro Ferreira, "Network Middleware for Large Scale Mobile and Pervasive Augmented Reality Games" in Proc. of the CoNext 2005 - ACM Conference on Emerging Network Experiment and Technology, pp. 242-243, CoNext 2005 - ACM Conference on Emerging Network Experiment and Technology, Toulouse, France, October-2005

[7] Pedro Ferreira, João Orvalho, Fernando Boavida, "Large Scale Mobile and Pervasive Augmented Reality Games", in Proc. of the EUROCON 2005 - The International Conference on "Computer as a Tool", pp. 1775-1778, Vol. 1, # 1, EUROCON 2005 - The International Conference on "Computer as a Tool", Belgrade, Serbia and Montenegro, November-2005

[8] M. Pullen, M. Myjack, C. Bouwens, "Limitations of Internet Protocol Suite for Distributed Simulation in the Large Multicast Environment", RFC 2502, IETF, February 1999

[9] Reliable Multicast Transport (IETF Working group), http://www.ietf.org/html.charters/rmt-charter.html, Acessed April 2006

[10] T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, L. Vicisano, "PGM Reliable Transport Protocol Specification", RFC 3208, IETF, December 2001

[11] João Gilberto de Matos Orvalho, "ARMS – Uma plataforma para aplicações multimédia distribuídas, com qualidade de serviço", Phd Thesis, December 2000, DEI-FCTUC