The design of a reliable multipeer protocol for DVEs

Gunther Stuer
Jan Broeckhove and
Frans Arickx

The authors

Gunther Stuer is a Research Assistant and **Jan Broeckhove** and **Frans Arickx** are Professors, all at the Department of Mathematics and Computer Sciences, University of Antwerp, Antwerp, Belgium.

Keywords

Computer languages, Computer simulation, Error cause identification, Computer programming, Networks

Abstract

We present the design and implementation of a reliable multipeer protocol (RMPP). This protocol is suitable for applications in the area of distributed virtual environments. RMPP is written in Java because this language has many interesting features for DVE-builders. Motivation, protocol classification, design goals and the error recovery algorithm are discussed. Furthermore, some implementation issues are listed. This paper concludes by presenting two possible applications of the RMPP.

Electronic access

The Emerald Research Register for this journal is available at **http://www.emeraldinsight.com/researchregister**

The current issue and full text archive of this journal is available at **http://www.emeraldinsight.com/1066-2243.htm**

Internet Research: Electronic Networking Applications and Policy
Volume 13 · Number 2 · 2003 · pp. 80-85
© MCB UP Limited · ISSN 1066-2243
DOI 10.1108/10662240310469024

Introduction

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 (Zyda, 1996). With the introduction of multicasting this problem was partly solved, but traditional multicast protocols are all based on best effort approaches, i.e. message delivery is not guaranteed. In order to achieve this guarantee, reliable multicast protocols were introduced (Hall, 1994). Although there are already many such protocols, none is optimised for distributed virtual environments (DVE) (Birman, 1999). The most important problem is that in a typical DVE many nodes are simultaneously sending and receiving each other's information (Sato *et al.*, 1999) and classic reliable multicast protocols are not designed to handle this situation. For this reason, the development of a reliable multipeer protocol (RMPP) is an important contribution to the development of large DVEs. The RMPP is a member of the multipeer protocol family. This family generalizes the traditional multicast (one sender), many receivers to the situation of many senders as well as many receivers (Wittman and Zitterbart, 2000).

This paper describes the RMPP, in particular the algorithms involved in the protocol and a number of aspects of its design and development. In addition, some applications of the RMPP are discussed.

Protocol classification

In the classification of reliable multicast protocols (Obraczka, 1998), the one presented here is most closely-related to the Transport Protocol for Reliable Multicast (TPRM) (Sabata *et al.*, 1998). RMPP is a message based protocol which means that there is no stream between sender and receivers, but rather that a number of independent messages are transmitted. Every message consists of one or more packets, each one transmitted as a UDP datagram. The most important difference with TRPM is that RMPP is a multipeer protocol.

If one classifies on error correction schemes, RMPP is a receiver initiated protocol (Levine and

Internet Research: Electronic Networking Applications and Policy
Volume 13 · Number 2 · 2003 · 80-85

Garcia Luna Aceves, 1998), i.e. possible errors are detected by the receiver which informs the sender hereof through a negative acknowledgement (NACK). On reception the sender will retransmit the erroneous or missing packets.

This type of protocol has two important drawbacks. The first one is the danger of a NACK implosion which can happen when multiple recipients detect that a packet is missing. They will all send a NACK for the same packet with a serious regression in network performance as a result. This problem can be solved by making all receivers wait a random amount of time before actually sending the NACK. If, during this delay, they receive a NACK sent by another receiver, they drop their own request. A consequence of this solution is that all NACK requests and NACK responses have to be multicasted to warn all interested recipients.

The second problem is that in theory these protocols need an infinite amount of memory because you never can be sure whether all receivers correctly received a certain datagram. This problem is solved heuristically by assuming that in a virtual environment old messages have lost their importance and can be dropped.

The opposite of receiver-initiated protocols are sender initiated protocols where the sender is responsible for handling errors or missing packets. To do this, the sender maintains a list of all clients and after each transmission every client has to send an acknowledgement (ACK). If one is missing, an error has occurred and unacknowledged datagrams are retransmitted. This type of protocol does not exhibit the two problems stated above, but is not very scalable due to the large amount of ACKs.

The protocol

When one wants to recover from an erroneous or missing packet, it is very important to have a way to uniquely identify this packet. In the RMPP this is done in three steps.

The first one is at the level of the VR participants. Each one is identified by a unique ID, which is basically a random 32-bit number generated during construction. Every datagram transmitted will contain this ID. In this way, the receiver can determine which VR participant sent the datagram. Every datagram also contains a message sequence number (MSN) which uniquely identifies every message sent by a given participant. So, the combination (nodeID, MSN) uniquely identifies every message in the system. The third level of identification is "packetNr". This one uniquely identifies every packet within a message. As such, the 3-tuple (nodeID, MSN, packetNr) uniquely identifies every datagram in the system.

Whenever a gap is detected between messages received from the same node, or between two packets from the same message, a NACK-request is sent for every message involved. When a whole message is missing, the NACK-request contains its MSN, and the requested packetNr is set to one, since there is always at least one packet in every message. When one or more packets from a certain message are missing, the NACK-request contains this messages MSN and a list of all missing packetNrs. The sender will re-transmit all packets it receives a NACK-request for. These are known as NACK-response packets.

When the first packet of a given message arrives, a new empty message is created at the receiver side and its timer is set to receiveTimeout. Whenever another packet for this message arrives, it is added to the message and the timer is reset. When the timer reaches zero, the RMPP assumes all unaccounted packets lost and sends a NACK-request for them. If however all packets were received, the message is considered to be complete.

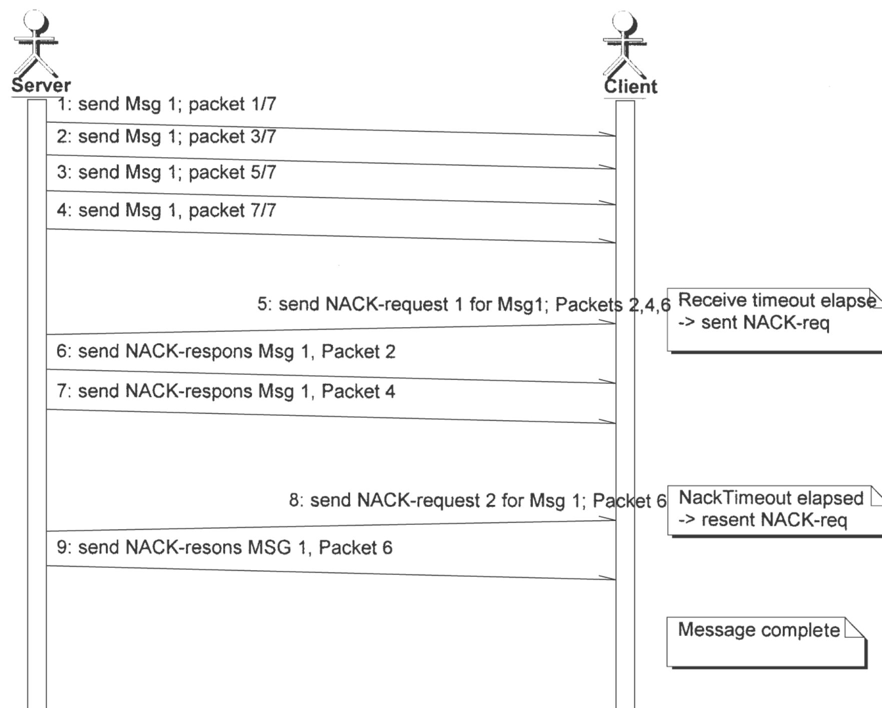
When a NACK-request is sent, the timer of the error producing message is set to nackTimeout. If no NACK-response is received before the timer ends, another NACK-request is sent. This goes on until maxRequests NACK-requests have been transmitted after which the message is considered lost and removed from the system. If, on the other hand, a NACK-response is received, the timer is reset to recvTimeout and the algorithm starts all over again.

The sender keeps every message in memory for sendTimeout seconds. This timer is reset with every incoming NACK-request. When it reaches zero, the sender assumes that all receivers did receive the message correctly and removes it from memory. Figure 1 describes the protocol.

Each of the timers RecvTimeout, NackTimeout and SendTimeout is responsive to the frequency with which timeouts occur. If timeouts are

Internet Research: Electronic Networking Applications and Policy
Volume 13 · Number 2 · 2003 · 80-85

Figure 1 The protocol



frequent the timeout interval is lengthened. If, on the other hand, the timer is frequently reset while there is a significant amount of spare time left, the timeout interval is shortened. This approach matches the timeout interval to current operating conditions and optimized responsiveness. This is also the technique that ensures that fluctuations in the responsiveness of Java due to garbage collection and datastructure optimizations do not destabilize the protocol.

Design goals

The primary goal of this research project was the creation of a reliable multipeer system optimized for distributed virtual environments written in Java. As a reference platform we used VEplatform (Demuynck, 2000), a 100 percent distributed VR framework developed in C++ at the university of Antwerp. Note that the 100 percent distributability rules out the use of tree based reliable multicast protocols like TRAM developed by Sun Labs (Chiu *et al.*, 1998) which is a part of Sun's JRMS project (Hanna *et al.*, 1998). The secondary goal was to emphasise good design. We considered the architectural aspect to be more important than top-notch performance. To achieve this, we extensively used object oriented techniques, UML (Booch *et al.,* 1999) and design patterns (Grand, 1998).

The fact that this protocol is tuned for virtual environments has three interesting consequences:

(1) The frequency with which VR nodes send updates has a maximum of 30 messages a second (Brutzman *et al.*, 1995) because this implies one message for every screen update. When one uses dead reckoning algorithms it is possible to reduce the update frequency to an average of one message a second (Demuynck *et al.*, 1998). This maximum and average value allows for the optimization of the data structures.
(2) The size of a typical VR message is usually less than 1 KB because only the position, orientation and some state information is transmitted. This made it possible to optimize buffer sizes.
(3) After a certain amount of time, an update message is of no importance anymore since the object that sent it probably already altered one or more of the transmitted parameters. This is why old messages may be dropped.

As an implementation language we had to choose between C++, the language used to implement

Internet Research: Electronic Networking Applications and Policy
Volume 13 · Number 2 · 2003 · 80-85

VEplatform, and Java. After carefully considering the benefits and drawbacks of both, we chose Java. Its benefits are:

- Java is a very expressive language with many standard class libraries. This allows for the construction of complex applications with much less code than C++.
- Javas syntax and semantics are much more clear than those of C++ which enhances the quality of the produced code.
- Java has built in features such as threading and dynamic class loading that are extensively used in our design.

The drawbacks of using Java are:

- Java applications are not the most performant. Although serious advances have been made, C++ is still faster. It is, however, our belief that this obstacle will slowly resolve as better compilers and interpreters become available.
- Java was never intended for real-time applications. This can be observed when monitoring the behaviour of the garbage collector and the collections framework. The RMPP depends heavily on timers to decide when to take action. Java, however, gives no guarantees whatsoever about their correct behaviour. The specifications state, e.g. that the method sleep(x) will cause the thread to sleep at least x milliseconds. In situations where one needs an immediate response to some action, such as handling a NACK, the garbage collector and collections framework overhead can be very problematic. However, a solution to this problem has been found and will be discussed in the next section

Implementation problems and solutions

In this section, the different pitfalls one can encounter when constructing an RMPP will be described. In four subsequent revisions, design and implementation solutions to problems encountered were accounted for. The current version seems to perform as expected.

Initial design

The very first design was conceptually the most pure one. Every message had its own thread (Kleiman *et al.*, 1996) which took care of this objects' timer. The strongest points of this design were its pureness and transparency. But unfortunately, some problems were encountered that led to erratic behaviour.

The first one is that the creation of a Java thread takes about 1.5 ms (Lewis and Berg, 2000). Let st be the sendTimeout, msi the maxSending value of node i, n the amount of nodes and rt the receiveTimeout, then the following formula tells us how many threads there are, at any given moment, active in the system.

A very plausible situation is that there will be ten nodes, each sending 30 messages a second. Typical values for sendTimeout and recvTimeout are respectively 30 seconds and one second. One can see that, in this case, the system contains 1,200 threads, of which each second 330 die and 330 new ones are created. If it takes 1.5 ms to create a single thread, the RMPP spends half of its time constructing threads.

The second problem is that there are many differences between threading libraries on different platforms. The most important differences are in the area of thread scheduling. Since the protocol depends on quick responses when e.g. a NACK-request is sent, this can have a devastating effect on protocol stability.

This is why this initial design attempt can best be seen as a proof of concept implementation (Stuer *et al.,* 1999).

The thread pool

To solve the thread creation problem mentioned above, a thread pool was used. When the application starts, the size of the thread pool is determined and it is populated with sleeping threads. Whenever a thread is needed, one is removed from the pool and reinserted when no longer needed. When additional threads are requested, the pool is expanded.

With these changes, the RMPP became much faster (Stuer *et al.*, 2001a), but stability remained an issue. Whenever the protocol was tuned for one OS, its behaviour was problematic on other platforms.

A new design

To solve the stability problem, a whole new design was necessary (Stuer *et al.*, 2001b). Instead of assigning one thread to each message, the RMPP now has a Timer Thread.

All messages wanting to be notified after some period of time can register themselves with this timer. When the requested period is over, they are informed of this using a call back mechanism. However, this timer turned out to be a bottleneck because the data structures used to construct it were not fast enough to handle the large amount of requests needed.

With this new design, the protocol was stable under all tested operating systems, but due to the bottleneck not as performant as it could be. The drawback is that part of the original design elegance is gone.

A final optimization

By taking into account the fact that most VR messages are small and fit in a single datagram, a final and important optimization could be achieved. On reception of such a message, the recvTimeout notification can be omitted since we already have the complete message. By doing this, the last remaining bottle neck is solved because the load on the Timer Thread is now significantly reduced.

Applications

In this section we will discuss two possible applications of the RMPP. Both are components to be used in a highly dynamical distributed virtual environment.

Probe classes

The best technique to make a virtual environment scalable is to make sure that each participant only receives the information it is interested in. One way to do this is to divide the world in regions where each region has its own multicast group (Morse, 1996). All objects in the same region transmit their updates on the same multicast group. Another, still experimental, but better method is described in what follows.

Consider a virtual world that is not divided in fixed regions, but where objects cluster dynamically dependent on some criteria. This can be implemented using a fuzzy clustering algorithm (Looney, 1999). Each cluster then corresponds to one multicast group. Participants decide periodically which set of objects they are interested in, and thus to which multicast groups they should listen.

The technique that makes this approach possible is that of probe classes. The protocol is described in Figure 2. An object (A) multicasts a probe class (1) to all participating objects (B). The probe class gives B information about A so that B can decide whether it wants to listen to A (3) or not. After this, the probe class receives information from B so it can decide whether A would be interested in B or not. If so, the multicast address of B is transmitted to A (5) and A starts listening (6). This technique allows for asymmetric interests, e.g. it is possible for A to listen to B, but not the other way around.

Every time a parameter used in the decision making process changes out of a preset range, an updated probe class is fired to notify all objects of the change in state. As an example one can assume a three-dimensional system. Two objects are interested in each other when their distance is less than 100 units. An object launches a new probe whenever it moved ten units from its previous launch position. An example of an asymmetric interest would be when one participant is looking through a binocular while the other is not.

Object mirroring

Object mirroring is a possible strategy to minimize communications in a distributed virtual environment. One can use the RMPP to multicast the VR objects to all interested nodes. With the use of Java class loading it is possible to create a local object from this stream of bytes. This local object serves as a proxy for the original one. Figure 3 illustrates this principle where object A gets replicated to objects A at the receiver sides.

All communication between the original object and its proxies also relies on the RMPP system. An XML message is transmitted which consists

Figure 2 Probe classes

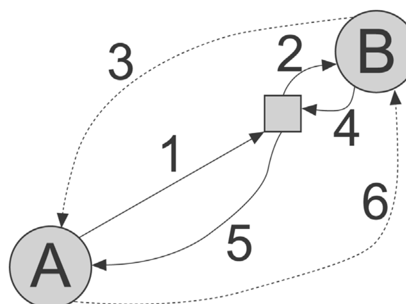
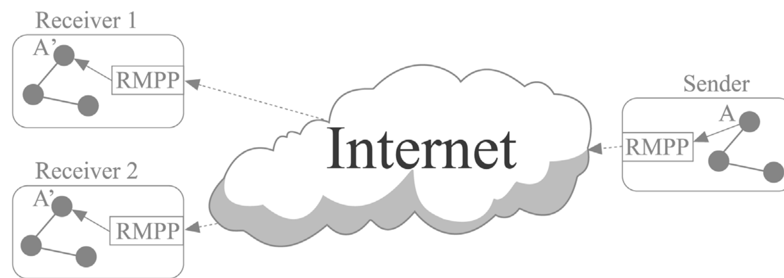


Figure 3 Object mirroring



of two parts. The first one identifies the target object and the second part contains the actual message. Using this technique, the virtual world is extremely expandable because each type of object (e.g. table, avatar, room, ...) can define its own message scheme. This way one does not need to specify the complete protocol when designing the virtual environment.

Conclusions

Our contribution in this paper has been to develop a reliable multipeer protocol (RMPP). This protocol is geared towards applications in the area of multi user DVEs. To achieve our design goals, particularly in scalability and 100 percent distributability, a receiver initiated protocol was used. We have found that the advantages of implementing the RMPP in Java outweigh the disadvantages. The performance penalty was well within the targeted limits.

References

Birman, K.P. (1999), "A review of experiences with reliable multicast", *Software Practice and Experience*, Vol. 29 No. 9, pp. 741-74.

Booch, G., Rumbaugh, J. and Jacobson, I. (1999), *The Unified Modeling Language User Guide*, Addison Wesley, Reading, MA.

Brutzman, D.P., Macedonia, M.C. and Zyda, M.J. (1995), "Internetwork infrastructure requirements for virtual environments", *Proceedings of the 1995 Symposium on Virtual Reality Modeling Languages*.

Chiu, D.M., Hurst, S., Kadansky, M. and Wesley, J. (1998), *TRAM: A tree based Reliable Multicast Protocol*, Sun Research Technical Report TR 98 66.

Demuynck, K. (2000), "The VEplatform for distributed virtual reality", PhD thesis, University of Antwerp, Antwerp.

Demuynck, K., Arickx, F. and Broeckhove, J. (1998), "The VEplatform system: a system for distributed virtual reality", *Future Generation Computer Systems*, No. 14, pp. 193-8.

Grand, M. (1998), *Patterns in Java*, Vol. 1, John Wiley & Sons, New York, NY.

Hall, K.A. (1994), "The implementation and evaluation of reliable IP Multicast", Master of Science thesis, University of Tennessee, Knoxville, TN.

Hanna, S., Kadansky, M. and Rosenzweig, P. (1998), *The Java Reliable Multicast Service: A Reliable Multicast Library*, Sun Research Technical Report TR 98 68.

Kleiman, S., Shah, D. and Smaalders, B. (1996), *Programming with Threads*, SunSoft Press.

Levine, B.N. and Garcia Luna Aceves, J.J. (1998), "A comparison of reliable multicast protocols", *Multimedia Systems*, Vol. 6, pp. 334-48.

Lewis, B. and Berg, D.J. (2000), *Multithreaded Programming with Java Technology*, Sun Microsystems Press.

Looney, C. (1999), "Fuzzy clustering: a new algorithm", *Proceedings of INCOSE 99*.

Morse, K.L. (1996), *Interest Management in Large Scale Distributed Simulations*, Technical Report ICS TR 96 27, University of California, Irvine, CA.

Obraczka, K. (1998), "Multicast transport protocols: a survey and taxonomy", *IEEE Communications Magazine*, pp. 94-102.

Sabata, B., Brown, M.J. and Denny, B.A. (1998), "Transport protocol for reliable multicast: TRM", *Proceedings of IASTED International Conference on Networks*, pp. 143-5.

Sato, F., Minamihata, K., Fukuoka, H., Mizuno, T. (1999), "A reliable multicast framework for distributed virtual reality environments", *Proceedings of the 1999 International Workshop on Parallel Processing*.

Stuer, G., Broeckhove, J. and Arickx, F. (1999), "A message oriented reliable multicast protocol for a distributed virtual environment", *Proceedings of Incose 99*.

Stuer, G., Broeckhove, J. and Arickx, F. (2001a), "Design and Implementation of a reliable multicast protocol for distributed virtual environments written in Java", *Proceedings of EuroMedia2001*.

Stuer, G., Broeckhove, J. and Arickx, F. (2001b), "Performance analysis of a reliable multicast protocol for virtual environments in Java", *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*.

Wittmann, R. and Zitterbart, M. (2000), *Multicast Communications*, Ch. 2., Academic Press, New York, NY.

Zyda, M.J. (1996), "Networking large scale virtual environments", *Proceedings of Computer Animation 96*.