An Analysis of the Reliability Overhead Generated by the JRM-Protocol

Gunther Stuer, Jan Broeckhove, and Frans Arickx

University of Antwerp,
Department of Mathematics and Computer Science,
Groenenborgerlaan 171, 2020 Antwerp, Belgium.
gunther.stuer@ua.ac.be

Abstract. An important aspect of any reliable communications protocol is its robustness against adverse network conditions. This paper presents a stochastic model for predicting the overhead introduced by the error handling algorithms of the JRM-protocol for various levels of network degeneration. To validate the model, a series of experiments were conducted. This paper compares those experiments with the predictions of the model.

1 Introduction

The effective use of network bandwidth has always been an issue in distributed virtual environments (DVE) [1]. Multicasting addresses this issue, but traditional multicast protocols do not guarantee message delivery. For this one needs reliable multicast protocols [2]. Although there are already many such protocols, none is suitable for distributed virtual environments [3]. The most important problem is that those protocols are typically designed for a single sender - many receivers situation. In DVEs many nodes are simultaneously sending and receiving messages, i.e. one has a many-to-many interaction pattern [4]. The Java Reliable Multicast (JRM) [6] protocol is a member of the more general multipeer [5] protocol family and handles the many senders - many receivers situation, with most nodes exercising both functions.

A feature of the reliability protocol is the overhead it entails as a function of worsening network conditions. We present a stochastic model to determine this overhead. The predictions of the model are verified against experimental data.

2 The Protocol

In this description of the JRM-protocol we will highlight the error-handling components because they are the subject of discussion in this paper. Details of the general background can be found in [6]

JRM is a message based protocol. This means that there is no connection and datastream between sender and receiver. Instead, messages, each unrelated to the previous, are transported from sender to receiver. Every message consists

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2657, pp. 474–481, 2003.

of one or more packets, each one transmitted as a UDP datagram. Typical values for DVEs are a send frequency of 30 messages per second, a message size of 1 packet and a packet size of 1 KB.

JRM is a receiver-initiated protocol [7]. The receiver has the responsibility of detecting errors and missing packets. If one occurs, a negative acknowledgment (NACK-request) is send to the sender requesting retransmission of the particular packet. This approach is much more scalable than the sender-initiated protocols [7].

To recover from an erroneous or missing packet, it is essential to be able to identify all packets uniquely. In JRM this is realized by the 4-tuple (VRN, MCG, MSN, PSN). The first number is the unique number assigned to the sending DVE-object (VRN) when it enters the virtual world. The second number is the multicast group (MCG) on which this message is transmitted. The third number is the message sequence number (MSN). It uniquely identifies every message sent by a participant on a given multicast group. The fourth number is the packet sequence number (PSN). This one uniquely identifies every packet within a message. As such, the 4-tuple (VRN, MCG, MSN, PSN) uniquely identifies every datagram in the system.

The identifiers discussed above are embedded in every transmitted datagram. They make it straightforward to reconstruct the message and identify the sender. Whenever a gap is detected between MSNs received from a particular sender, one or more messages are missing. Then, for each missing message, the receiver transmits a NACK-request containing the MSN in question and PSN set to one. The packets with a particular value for (VRN, MCG, MSN) constitute a message. Observing the sequence of PSNs in such packets, the receiver can detect missing packets in a message and transmit a NACK-request identifying the message and listing the missing packets. In both cases, the sender will then retransmit all missing packets. These are known as NACK-response packets.

In order for the protocol to function sensibly in a DVE context, certain time-related constraints are required. When the first packet of a message arrives, a new message-holder containing that packet is created by the receiver. The header of each packet contains the total length of the entire message. This information is used to determine the number of packets necessary to completely receive the message. A count-down timer is associated with each message-holder and on construction it is set to *receiveTimeout*. Whenever another packet for this message arrives it is inserted into the correct message-holder and the corresponding timer is reset. When the timer reaches zero, the message is inspected for missing packets. If there are none, the message is complete. Otherwise a NACK-request is generated and the timer is set to *nackTimeout*. If no NACK-response is received before the timer runs out, *nackTimeout* is increased and another NACK-request is sent. This cycle repeats a preset maximum number of times. After that, the message is considered lost and removed from the system. If on the other hand, a NACK-response is received, the timer is set to *recvTimeout* and the algorithm starts all over again.

The sender keeps every packet of every outgoing message in a buffer. Again a timer is associated with each message and it is initially set to *sendTimeout* seconds. This timer is reset with every incoming NACK-request for any of the packets of that message. When the timer runs out, the sender assumes that all receivers did receive the message correctly and removes it from the buffer.

Each of the timers *recvTimeout*, *nackTimeout* and *sendTimeout* is responsive to the frequency with which timeouts occur. If timeouts are frequent the timeout interval is lengthened. If on the other hand the timer is frequently reset while there is still a significant amount of time left, the timeout interval is shortened. This approach matches the timeout interval to current operating conditions and optimizes responsiveness.

Two key concepts in our discussion are the "effective throughput" and the "total throughput". The effective throughput refers to the messages that have been received in their entirety. The total throughput refers the datagrams arriving at the receivers end. Both are expressed in kilobytes per second (KB/s). The discrepancy between both is a consequence of retransmission of missing packets i.e. of the activity of the reliability mechanism.

3 The Reliability Overhead Prediction Model

Let us assume that we can characterize the operating conditions of the communication channels between each of the nodes by an *errorRate* e. It expresses the probability that a packet will not arrive at its destination due to network conditions, buffer overruns, and so on. The probability for a message consisting of n packets to arrive directly, i.e. without retransmissions of missing packets, is then $(1-e)^n$. This number decreases significantly with increasing n: at *errorRate* 0.2 (20%) a message with three packets has only a fifty percent chance of making it directly. At *errorRate* 0.3 a message with one single packet has 70% chance and a message of 10 packets only 2.8%!

Obviously, the number of retransmissions needed to send the message completely increases with e and n. Let us consider this in more detail. Initially each of the n packets are transmitted. Of these n packets, on average, a number of $n \times e$ will be missing and need to be retransmitted in response to a NACK-request. Of these $n \times e$, a number of $(n \times e) \times e$ or $n \times e^2$ will again fail to arrive and have to be retransmitted. Thus, on average, the number of transmissions required by a message of length n packets is given by (1).

$$[N_{packets}] = n \times \sum_{i=0}^{\infty} e^i = \frac{n}{1-e} \tag{1}$$

However, in addition we need to take into account the intervening NACK-requests that are sent by the receiver to signal missing packets. Above we deduced that after j NACK-requests and retransmissions, an average of $n \times e^j$ packets are sent. The probability that all of them arrive and no further NACK-requests will be needed is $(1-e)^{n \times e^j}$. Otherwise we have (2) for the probability that

more NACK-requests will have to be sent. One has to be careful because $n \times e^j$ is only an average and for this reason (2) will only be an approximation of the real probability.

$$P\{\#Nacks > j\} = 1 - (1 - e)^{n \times e^j} \tag{2}$$

The probability that exactly j NACK-requests are required to complete the message transmission is given by (3)

$$\begin{aligned} P\{\#Nacks = j\} &= P\{\#Nacks > j - 1\} - P\{\#Nacks > j\} \\ &= (1 - e)^{n \times e^j} - (1 - e)^{n \times e^{j-1}} \end{aligned} \tag{3}$$

So, on average and in the assumption that none of the NACK-requests are lost, (4) gives the number of NACK-requests that need to be transmitted to receive one complete message.

$$[N^*_{Nacks}] = \sum_{j=0}^{\infty} j \times P\{\#Nacks = j\} \tag{4}$$

After substituting (3) into (4) and canceling terms one obtains (5).

$$[N^*_{Nacks}] = \sum_{j=0}^{\infty} (1 - (1 - e)^{n \times e^j}) \tag{5}$$

Of course, the above is derived on the basis of averages and constitutes an approximation. In an exact approach one must consider all delivery scenarios separately and sum them with their probabilities. That approach however quickly becomes intractable for increasing n.

As with the data packets, the NACK-requests are subject to the effects of the *errorRate* n. Thus by the same reasoning as before we arrive at (6) for the final (approximate) number of NACK-requests.

$$[N_{Nacks}] = \frac{\sum_{j=0}^{\infty} (1 - (1 - e)^{n \times e^j})}{1 - e} \tag{6}$$

Taken together with (1) we find (7) for the total number of packets, data and NACKs, required to complete the message transfer.

$$[N_{Transmissions}] = \frac{\left[n + \sum_{j=0}^{\infty} (1 - (1 - e)^{n \times e^j})\right]}{1 - e} \tag{7}$$

When one uses the same example as above, (7) predicts that at *errorRate* 0.3, a message with length 1 will, on average, need 2.07 transmissions. With a volume of 30 messages per second, one has a total of 61.99 transmission per second, or an overhead of 31.99 transmissions per second (52%). For the second example where a message of length 10 is considered, on average, 17.19 transmission per message are needed. With a volume of 30 messages per second, one has a total of 515.67 transmission per second, or an overhead of 215.67 transmissions per second (42%).

4 The Experiment

A series of experiments has been conducted to investigate the effective and total throughput of the JRM. The objective is to estimate the overhead due to the reliability algorithms in the protocol. To simulate the effect of dropped or colliding packets caused by a adverse network, an artificial *errorRate* is introduced at the sender's side. This is done by means of a uniform random number generator. Just before the Java-send() method is called, a random number is generated and if the result is below a configurable threshold, the packet is discarded.

The experiments are performed on a local area network (LAN) consisting of only two computers. One of them acts as the sender while the other one is the receiver. This way one can be sure that almost all packets on the network are JRM-related. The first experiment compares the predicted number of transmissions to the actual amount. As such, one can state that the first experiment is used as a validation for the derived stochastic model. The second experiment measures the influence of the JRM error handling protocol on the effective throughput. Consequently, one can say that this experiment shows the effect of the *errorRate* from the JRM-user's point of view. Both experiments are repeated for an *errorRate* of 0, 5, 10, 15, 20, 25 and 30% and a message size of 1 and 10 packets, with each packet 1 KB in size. Other aspects of the JRM-protocol have already been tested and are described in [8].

All experiments were repeated using different operating systems (Windows 98, Windows 2000, SuSe Linux 8.0 and Solaris 8). The computers used for these experiments are equipped with a Pentium-III 733MHz processor, 64 MB memory and a 100Mb network interface card (NIC). For Solaris 8, a Sun 450-Enterprise server was used. The Java runtime system is version 1.4.0. Because the results of these experiments are exactly the same on all of the operating systems, only one result is shown on the graphs.

The goal of the first experiment is to measure the influence of the *errorRate* on the overhead caused by the error handling algorithms. This is done by correlating the *errorRate* to the number of transmissions necessary to completely send one message. In this experiment, the sending node transmits a total of 1000 messages at a rate of 30 messages per second. The sending node registers the amount of transmitted Msg-packets and NACK-responses. The receiver registers the amount of transmitted NACK-requests. The average number of packets $[N_{packets}]$ necessary to transmit one message is then given by (8)

$$[N_{packet}] = \frac{MsgPackets + NACKrequests + NACKresponses}{1000} \tag{8}$$

Figures 1 and 2 show the result of the experiment for respectively 1 and 10 packets per message. The predicted values are very close to the measured ones when a message size of 1 packet is used. When one uses a message size of 10 packets, both values remain very close to each other until an *errorRate* of 20%. After this, the model consistently underestimates the real value. The most important reason for this phenomenon is that when there is a high volume of packets (in this experiment 300 per second) combined with a high *errorRate*, the

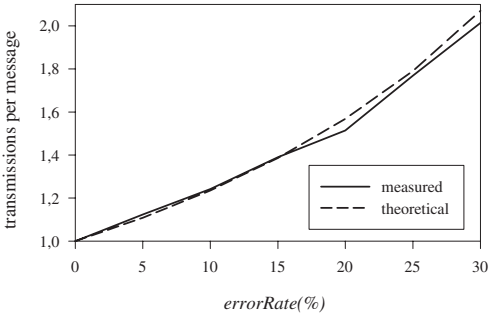

Fig. 1. Influence of the *errorRate* on the number of transmissions per second with a message size of 1 packet.

total number of network datagrams per second is so high that the NIC starts missing datagrams and that network collisions start occurring. Furthermore, the model is only an approximation because the dynamic aspects of the protocol, the adapting timeouts, are unaccounted for. From both figures, one can conclude that in the domain typically applicable to DVEs, the formula (7) yields an accurate estimate of the reliability overhead.

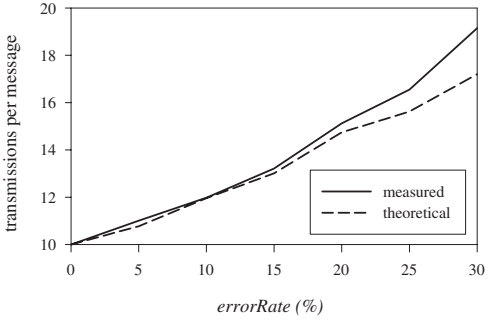

Fig. 2. Influence of the *errorRate* on the number of transmissions per second with a message size of 10 packets.

The goal of the second experiment is to measure the influence of the *errorRate* on the effective throughput. For this experiment, the sending node generates 30 new messages per second. It does so for the entire duration of the experiment. The number of messages reaching the receiver in a 60 second window will be measured. To ensure a steady-state condition of both sender and receiver, this

60 second test period will only start after at least 10 seconds of activity. This way, one can be sure that all objects and data structures are fully initialized.

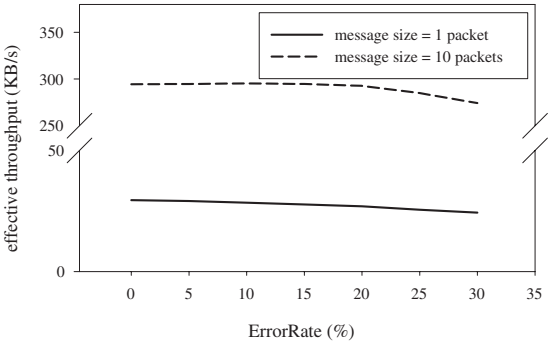

Fig. 3. Influence of the *errorRate* on the effective throughput.

Figure 3 shows the results of this experiment. One can see that the effective throughput remains very good, even under a very high *errorRate*. This is because the design decision was made that the *leaky bucket* algorithm does not take the error handling messages into account [6]. And as such, 30 new messages will be transmitted each second, independently of the current network condition. A disadvantage of this approach is that when the network degenerates, the load on the sending nodes increases, which could eventually lead to instability. The strength of our approach is that, as long as the network problems are moderate in nature and duration, the effective throughput remains optimal.

When one compares the results of both experiments, one can see that the increase in the number of transmissions per message coincides with the decrease in effective throughput. It is very important to note that the decrease in effective throughput is much less than the increase in transmissions per message. The remaining decrease originates from the fact than NACK-packets have precedence over Msg-packets and the more NACK-packets there are, the longer the Msg-packets have to wait before they can be transmitted.

5 Conclusion

From this paper, two important conclusions can be drawn. The first one is that the stochastic model proposed in this paper adequately predicts the overhead introduced by the reliability algorithm of JRM when one restricts to the domain typically applicable to DVEs.

The second important conclusion is that the JRM-protocol is very resilient against adverse network conditions. Only under extremely high *errorRates* of 25% and more is a degradation of the effective throughput observable.

References

1. M. J. Zyda: Networking Large-Scale Virtual Environments. *Proceedings of Computer Animation'96*, Geneva, Switzerland, 1996.
2. K. A. Hall: The Implementation and Evaluation of Reliable IP Multicast. *Master of Science Thesis*, University of Tennessee, Knowville, USA, 1994.
3. K. P. Birman: A Review of Experiences with Reliable Multicast *Software - Practice and Experience*, Vol. 29, No. 9, pages 741–774, 1999.
4. F. Sato, K Minamihata, H. Fukuoka, T. Mizuno: A reliable multicast framework for distributed virtual reality environments *Proceedings of the 1999 International Workshop on Parallel Processing*, Wakamatzu, Japan, 1999.
5. R. Wittmann, M. Zitterbart: Multicast Communications *Academic Press*, chapter 2, 2000.
6. G. Stuer, F. Arickx, J. Broeckhove: The Construction of a Reliable Multipeer Communication Protocol for Distributed Virtual Environments. *Proceedings of the 2002 International Conference on Computational Science (ICCS2002)*, volume 2330 of *Lecture Notes in Computer Science*, pages 679–686, 2002.
7. B. N. Levine, J. J. Garcia-Luna-Aceves: A comparison of reliable multicast protocols *Multimedia Systems*, Vol. 6, pages 334–348, 1998.
8. G. Stuer, J Broeckhove, F Arickx: Performance and Stability Analysis of a Message Oriented Reliable Multicast for Distributed Virtual Environments in Java. *Proceedings of the 2001 International Conference on Computational Science (ICCS2001)*, volume 2073 of *Lecture Notes in Computer Science*, pages 423–432, 2001.