RZ 3521 (# 99537) 12/01/2003
Computer Science 13 pages

Research Report

Transport Layer Protocol Support for Large Federated Peer-to-Peer Games

Sean Rooney, Rudy Deydier* and Daniel Bauer

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

{sro,dnb}@zurich.ibm.com

*Rudy Deydier contributed to this work while visiting the IBM Zurich Research Laboratory.

Transport Layer Protocol Support for Large Federated Peer-to-Peer Games

Sean Rooney, Rudy Deydier *, Daniel Bauer

IBM Research, Zurich Research Laboratory
Säumerstrasse 4
8803 Rüschlikon, Switzerland

Abstract. A federated peer-to-peer game is constructed as a set of peer groups within which players communicate with all other members and between which players can move. Federated peer-to-peer games potentially allow a more scalable and fault tolerant gaming architecture than that achievable using a central server. While the literature contains many proposals for such gaming infrastructure, little work has been done describing the network protocols required to support such a model. Gaming presents distinct challenges to network protocol designers, particularly in respect to its sensitivity to latency and loss. We describe the algorithms and communication patterns of gaming specific network protocols and present the results of our experiments. We show that a dedicated game transport protocol achieves good end-to-end delay across very lossy, jittery networks.

1 Introduction

A game is a system in which some of the system states are considered by the participants to be more desirable than others and whose unique purpose is the placement of the system by the participants in these states. They achieve this through a set of well-defined operators. Different participants may have different desired states.

A networked game is one in which state changes are performed across the network. In practice two ways of achieving this are possible: each participant keeps a copy of the relevant state locally, and all participants communicate the operations to be performed to all other participants who use this information to update their local state; one reference copy of the state is kept and all operations are transmitted to an actor which updates the state and distributes it to all participants. The first system is commonly called peer-to-peer while the second is a central server system. The second method is more efficient in resource usage as the operations are transmitted only once and their execution is performed only once. However, a large burden is placed on the actor who maintains the reference state.

A Massive Multiplayer On-line Game is a networked game in which many thousands of players participate simultaneously in the same game, for example [13]. In the rest of this paper we shall call such games *large games* or simply *games*. Currently commercial large games use a central server approach in which the majority of the game logic is executed on a server under the control of the game provider. A server farm has to be provisioned such that it is capable of supporting a given number of players. The game provider cannot know *a priori* how popular a game will be leading to a possibility that the server farm is too large or too small. Moreover, a game's popularity will inevitably change over times in ways which are difficult to predict.

* Rudy Deydier contributed to this work while visiting the IBM Zurich Research Laboratory.

It is desirable that the amount of resources available to a game dynamically increases with the number of players. Peer-to-peer architectures have this property but as the number of messages increases as a square of the number of players such architectures are not usable for large games. Work in the literature [5,8,11] has proposed using what we term a *federated peer-to-peer model* in which a large game or simulation is broken up into smaller islands corresponding to different areas of interest, for example virtual location, and within which players may move. To the best of our knowledge no actual commercial large game uses the federated peer-to-peer model.

A detailed description of the required underlying network support is lacking in the literature, making it difficult to evaluate their feasibility. Our work has attempted to address this deficiency. Our investigation has involved the design and implementation of a generic game transport protocol — *Shaker* —which allows the quasi reliable delivery of packets while allowing both sender and receiver to make trade-off decisions about latency and loss.

The architecture rests on a multicast reflector capable of receiving packets from players and distributing them to all other players present in the same group. We have given a description of the implementation and performance of an efficient multicast reflector elsewhere [3] and this description is not repeated here.

2 Structure of a Federated Peer-to-Peer Game Architecture

Each peer group within the federated peer-to-peer game is handled by a multicast reflector. The multicast reflector is an IP addressable entity capable of maintaining a list of end host addresses to which game packets should be distributed.

The clients send unicast IP packets to the multicast reflector, which in turn generates many unicast IP packets to send to registered members of the peer group. A given multicast reflector may handle many different peer groups, different peer groups are distinguished in our current implementation by the destination port. The multicast reflector does not use IP multicast. IP multicast is inappropriate for our purpose as we require many, small multicast groups to which members join and leave quickly. For a description of the limitations of IP Multicast in this context see [9]. The multicast reflector is a means by which a client can register an interest and receive the packets from a given peer-group, in this regard it is similar to — but much simpler than — a publish/subscribe mechanism, e.g. Gryphon event broker [1], whereby some party publishes information within some area of interest and parties who have subscribed to that area of information receive it. Senders and receivers are decoupled allowing the logic at both to be simplified; all packets are sent to the multicast reflector and all packets are received from it[1].

As well as the efficient distribution of packets the multicast reflectors serializes packets (using a sequence number) and buffers packets up to some window size to allow packet retransmission. This contrasts with existing publish/subscribe systems in which sophisticated QoS parameters, delivery polices and filter languages are performed by the notification server. The basic data forwarding functions of the multicast reflector is simple permitting it to be

[1] Note that when using the multicast reflector clients receive many more packets than they send; this is consistent with the asymmetric properties of access technologies such as ADSL.

efficiently implemented in software or programmable hardware. Game specific logic instrumented at the clients makes use of these basic functions to support their own game specific transport protocols.

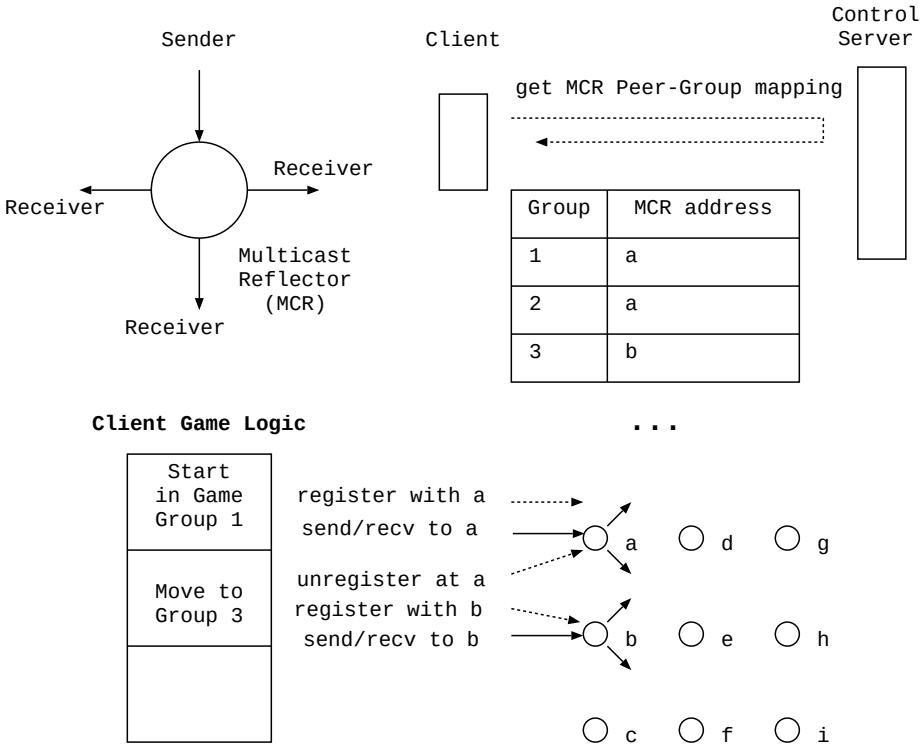


Fig. 1. General Schema of a Generic Federated Peer-to-Peer Game

The allocation of multicast reflectors to peer-groups is performed by one or more control servers. Clients obtain the association from the control servers and then use game specific knowledge to determine to which set of peer groups they should join. Clients are responsible for determining when and how they should change groups. The association of peer-groups to multicast reflectors can be changed dynamically allowing the number of reflectors allocated to a given group to be altered as a function of popularity. The clients alter their peer group membership using a control protocol — the *Mover* — which we do not explain any further here due to reasons of space. Figure 1 shows the structure of our generic federated peer-to-peer architecture.

The set of generic mechanisms for handling loss and retransmission we have instrumented as a simple protocol running over UDP that we call the *Shaker*. The rest of this paper describes the *Shaker's* design and reports on its performance under different network conditions.

3 The Shaker Transport Protocol

Game traffic is typically highly dependent on latency, i.e. a packet which arrives at an application late may be useless. As such, at-least-once transport protocols such as TCP are

not appropriate. TCP also suffers from a head of the line blocking problem by which out of order packets are not delivered to an application until the missing packets arrive resulting in increased latency. In consequence nearly all large games of which we are aware use UDP.

Games typically allow for late or lost packets within their logic using techniques, for example dead reckoning [4], to mask the effect to the user. However not all information transmitted to players is susceptible to such techniques; some information must be delivered reliably in order to assure the coherence of the game. A selective retransmission mechanism in which *certain* packets are retransmitted if there is a good chance that the retransmitted packet will arrive in time is desirable. The commercial game framework defined in [10] describes a method of application-layer transmission between a central game server and client. The application-layer protocol allows senders to distinguish between packets that should be reliably and unreliably sent, keeping a copy of the reliable ones in memory in case of the need for retransmission. Selective retransmission mechanisms have also been proposed for multimedia streams [12]; the Selective Retranmission Protocol (SRP) is a Client/Server protocol that allows a client to trade packet loss ratio against average packet latency in a continous media stream.

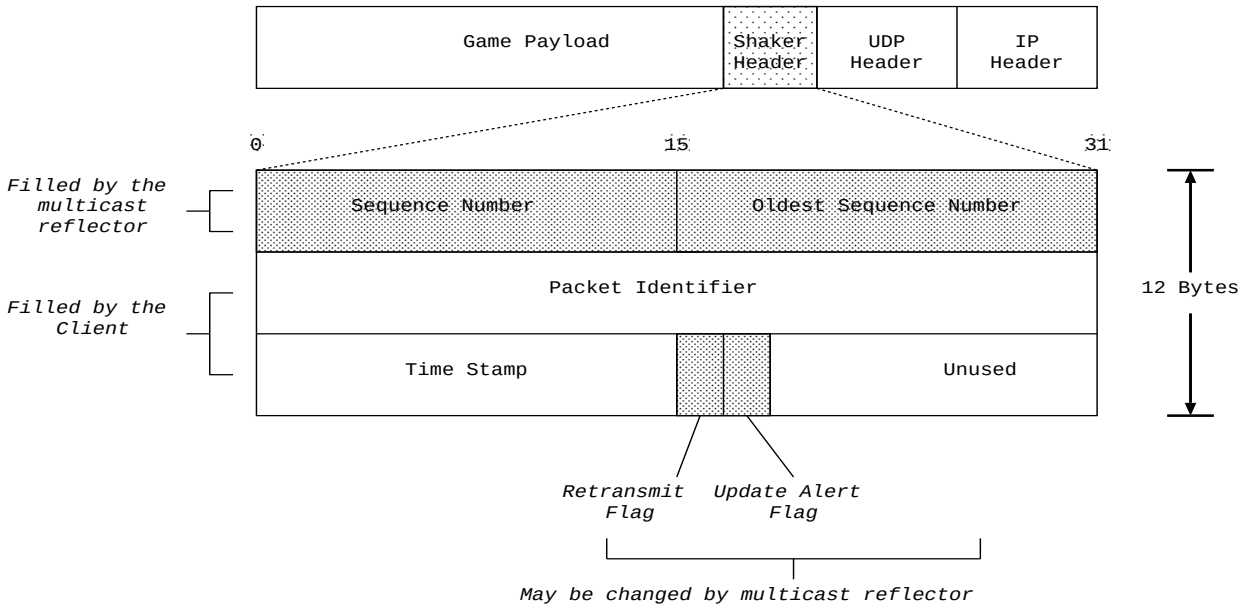


Fig. 2. Shaker Header

Within a federated peer-to-peer system, requiring clients to retransmit packets places the burden on them to maintain packets in memory and to respond to retransmit requests. While a sender may indicate that a packet should be delivered reliably a receiver need not agree, for example it might decide that the minimum of one Round Trip Time (RTT) needed to retransmit a packet is such that the packet would no longer be relevant, or it might consider that its desired level of participation in the peer group from which the packet is emitted is so low, that it would not be worthwhile. A packet that a sender decides should be sent reliably is sent using the *Shaker* protocol; unreliable packets are just sent using UDP, and therefore

are not buffered at the multicast reflector. In summary, the *Shaker* is a transport protocol that allows packets to be retransmitted between sender and receiver if both agree that it is worthwhile, but unlike TCP does not offer totally reliable transport.

Our *Shaker* protocol uses the multicast reflector to decouple the senders belief that a packet ought to be reliably delivered from a receiver's decision as to whether it has to be. The *Shaker* uses UDP as the basic transport mechanism between clients and the multicast reflector.

There is one *Shaker* session for every peer-group that a client participates in. The *Shaker* adds its own header — shown in Figure 2 — after the UDP header.

When a *Shaker* packet arrives at the multicast reflector the forwarding mechanism adds a packet sequence number and the packet is stored in a buffer at the multicast reflector before it is sent to all participants of the corresponding peer group. Note that all participants in the peer group including the sender itself receive the packet; the transmission of the *Shaker* packet back to the sender acts as an ACK allowing the sender to know that the multicast reflector has received the packet and enabling the RTT between itself and the multicast reflector to be continually recalculated.

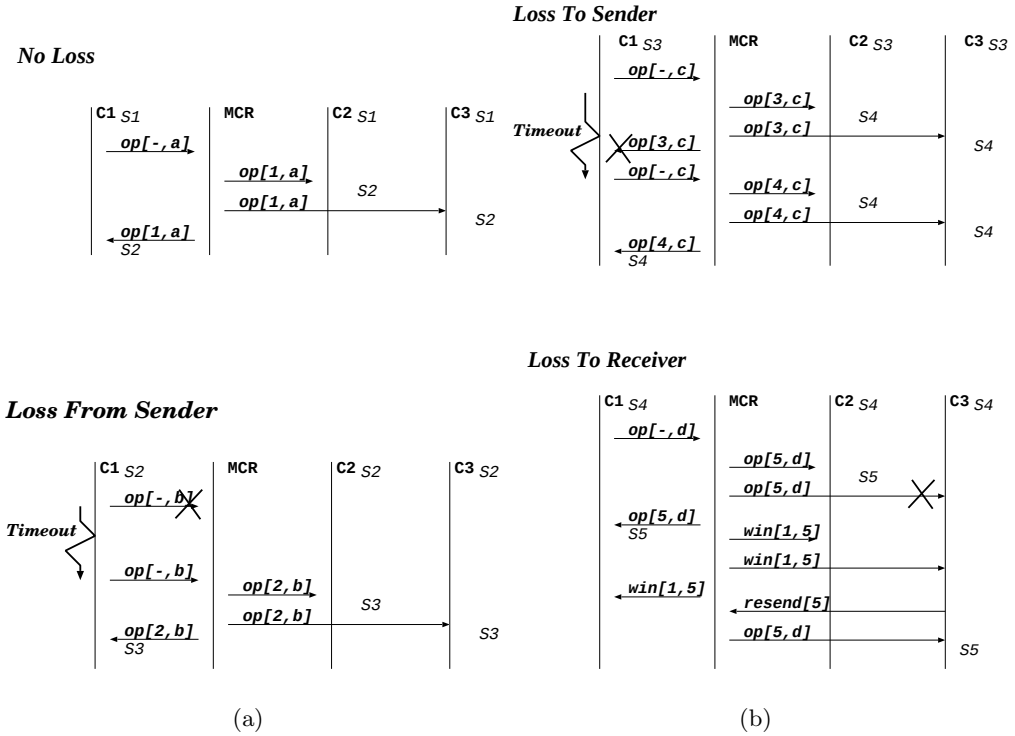


Fig. 3. Interactions within the Shaker protocol

Figure 3 shows the communication patterns under four conditions: no packet loss, loss from sender, loss to sender and loss to receiver. The algorithm that senders use to calculate the time they should wait for an ACK before sending is explained in Section 3.1. As the se-

quence numbers increase monotonically, a receiver can determine whether it has not received a reliable packet by examining whether any sequence numbers are missing in the stream of packets it receives. Noncontiguous arrival may be caused by out-of-order delivery as well as loss, so the receiver waits some time before requesting the retransmission of the packet, the algorithm for doing this is given in Section 3.2. If a receiver considers it worthwhile having a missing packet retransmitted it sends a retransmit request to the multicast reflector. A retransmitted packet is identified by the receiver as such by the corresponding flag in the header; a retransmitted packet is already late and should be sent to the application with minimum delay and in preference to non retransmitted packets in front of it in the queue.

The multicast reflector can only keep a finite number of already transmitted packets in memory; packet retransmission is not possible for packets older than a certain threshold. The multicast reflector writes the oldest retransmittable sequence number in the header of each packet sent to the client.

Senders set a thirty two bit identifier in the *Shaker* header. The top sixteen bits identify the sender while the bottom sixteen bits are a monotonically increasing series. Two packets with the same identifier may arrive with different sequence numbers at the receiver. This can occur in two cases: a sender times out too soon and resends a packet to the multicast reflector that was already correctly received by it; the ACK was lost going back to the sender. Receivers keep a window of recently received packet identifiers allowing them to detect and discard duplicates.

3.1 Sender Retransmission Time Out

As the sender also receives its own packet, the RTT between the multicast reflector and the sender can be calculated by placing the senders transmit time in the time stamp field of the packet header and simply subtracting that from the time at reception.

A sender will retransmit a packet if some timeout has expired. Our first attempt used the TCP Retransmission Time Out (RTO) [7] as shown in Algorithm 3.1; with the recommended values of gain g for the $EstimatedRTT$ is 0.125 and the gain h for the the mean variance D is 0.25.

Algorithm 3.1: JACOBSON'S RTO ALGORITHM()

$Error \leftarrow MeasuredRTT - EstimatedRTT$
$EstimatedRTT \leftarrow EstimatedRTT + g * Error$
$D \leftarrow D + h * (\|Error\| - D)$
$RTO \leftarrow EstimatedRTT + 4 * D$

$RtxTimeouttriggered : EstimatedRTT \leftarrow 2 * EstimatedRTT$

We found that in practice over a real WAN — measured between France and Switzerland — that the RTO was often twice the actually RTT, and the Root Mean Square Error (RMS) of this estimator as high as 90% of the actual mean RTT.

There are two reasons for this: the first is the TCP backoff algorithm, which doubles the value of the *EstimatedRTT* after each retransmission timeout, the second is that even when the error between the *EstimatedRTT* and the actual RTT reaches zero, the mean variance D converges slowly to zero — during this time the RTO is higher than the RTT. This is quite in keeping with TCP's conservative use of network resources, however, it adds a large penalty for game traffic when the RTT is varying and some loss is experienced. In order to reduce this lagging affect of the mean variance, we reduce the weight given to the mean variance in the calculation of the RTO from 4 to 2 — this incidentally was the original weight that Jacobson proposed [7].

Games because of their logic are implicitly rate-based, i.e. the packet sending rate will not grow in an unbound way, but in general is limited by the speed of human interaction, as such we do not see the need for any flow control in the basic data sending rate — although just as for UDP an application could instrument such an mechanism on top of the infrastructure if thought necessary.

To reduce the large penalty paid for the backoff algorithm, we do not double the *EstimatedRTT* when a timeout occurs. We increase the *EstimatedRTT* by a factor k — in our tests set to 0.1 — when three consecutive retransmission timeouts occurs, if the RTO has not been updated meanwhile. If a packet arrives after it is timed out we still use the RTT to update the RTO. Note that Karn's algorithm — which avoids updating the RTT when a packet is retransmitted — does not apply in our case as the sender can distinguish retransmitted packet from the original ones by their timestamps.

Our algorithm reacts to losses only if they are sustained as the objective is not that of slowing down the sending rate but to allow the RTO to track the RTT more closely by assuming the timeout is due to a slight increase in RTT.

Algorithm 3.2: SHAKER ALGORITHM IN CASE OF TIMEOUT()

comment: Modify the EstimatedRTT in case of consecutive RTX

$counter \leftarrow 0$
while **true**
do
- **if** pktReceived=ownPacket
  - **then** $counter \leftarrow 0$
- **if** RTX Timeout
  - **then**
    - $counter \leftarrow counter + 1$
    - **if** counter=3
      - **then**
        - $EstimatedRTT \leftarrow (k+1) \times EstimatedRTT$
        - $counter \leftarrow 0$

Algorithm 3.2 shows the means the *Shaker* uses for calculating the RTO. By trying to keep the RTO as close to the RTT as possible the *Shaker* will often timeout too soon and sends unnecessary additional packets if the RTT is varying. We examine these effects in Section 3.4.

Figure 4 shows the measured Root Mean Square (RMS) error of the RTT estimator of the TCP retransmission timeout algorithm, and the *Shaker's* own timeout algorithm, for

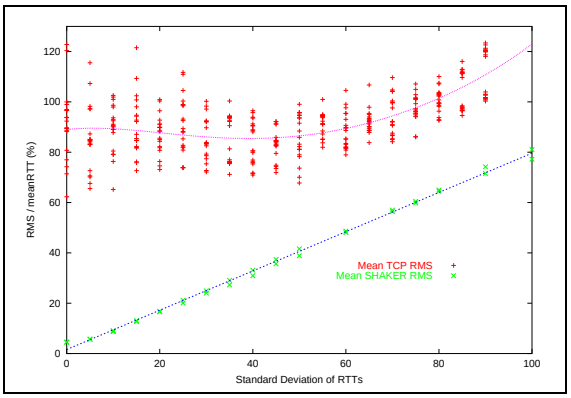


Fig. 4. RMS error of the TCP and Shaker RTT estimators

a mean RTT of 100 ms over a range of variances. The TCP estimator was applied to our *Shaker* protocol with the unmodified backoff algorithm, and the measurements were made using a single client sending packets to the multicast reflector, which sent them back using the mechanisms discussed previously. The wide fluctuations of the RMS error of the TCP RTT estimator for a fixed variance ratio is due to the backoff algorithm introducing errors into the estimated RTT, and consequently creating slightly higher estimations of the RTO.

3.2 Receiver Retransmission Time Out

A receiver can identify that a packet *may* have been lost by a missing sequence number. It cannot know if the packet has actually been lost or simply will arrive out of order. Waiting will encur a latency penality while an immediate request for retransmission may cause a large number of unnecessarily retransmitted packets. In addition, a receiver should not request the retransmission of a lost packet if it will arrive too late to be useful. The application initialises a given *Shaker* session with a game specific $RelevancyTime$, which is the maximum time that a packet is useful after it was emitted. For instance, for a stratgey game this value might be 500ms, while much lower for a FPS (First Person Shooter). The receiver does not know when a packet actually was transmitted, but it knows that it will take at least one RTT between itself and the multicast reflector to retransmit it. Since it takes at least 0.5 RTT to recognize that a packet is lost after being forwarded by the multicast reflector, a request for the packet is not considered worthwhile if $RTO > RelevancyTime - RTO/2$, that is if the $RelevancyTime$ is less than $1.5 * RTT$. In this case, the protocol layer notifies the application layer that a packet is presumed lost.

Out-of-order delivery maybe a transient phenomena due to a change in route, or long lived, for example due to load balancing between routes. At the detection of a missing packet, the receiver waits a certain time before requesting its retransmision. This time is a function of both $RelevancyTime$ and the mean variance D of the $EstimatedRTT$. In times of low variance in RTT, $D \approx 0$, the receiver assumes that the packet is lost and immediatly asks for its retransmision. Otherwise it waits either twice the variance if it less than the the maximum amount of time we can wait before the packet becomes irrelevant, or the maximum amount of time if it is greater. The reasoning behind using a function of the mean variance of the RTT

is that out-of-order-delivery indicates that the packets are coming over multiple different paths having different RTTs. In practise we found two times D was adequate.

Our algorithm for the Receiver Retransmission Time Out (RRTO) is as follows:

Algorithm 3.3: RRTO()

comment: Compute the RRTO based on $RelevancyTime$ and D

if $RelevancyTime < 1.5 * RTO$

then $\begin{cases} DiscardRequest \\ \\ NotifyApplication : LostPacket \end{cases}$

else $RRTO = Min(RelevancyTime - 1.5 * EstimatedRTT, 2 * D)$

3.3 Probability of Delivery within Bounded Time at Levels of Loss

Assuming a constant error rate of L between multicast reflectors and all clients and no packet reordering, then the probability of a packet being sent from a sender to a receiver without loss is: $P_x = (1 - L)^2$. Assuming all clients have the same RTT with the multicast reflector and this is symmetrical, as it takes at least one RTT after a packet was emitted by the sender to recognize loss and at least one RTT to retransmit the packet, whether or not the packet is lost from the sender to the multicast reflector, or from the multicast reflector to the sender then, the probability of a packet arriving within n RTTs can be simply expressed as: $P(n) = 1 - (1 - P_x)^n$

So for example, for high loss probability, L=0.1 (i.e. 10%) then there is a 99.7% chance of getting the packet to the receiver within 4 RTT, using *Shaker*. We emphasize that this is the theoretical best-case under simplfying assumptions. In the next section we test the actual measured performance of the *Shaker* across a network under various conditions of loss and jitter. The measurements allow for the real effects of a network: variable RTT, out of order delivery, consecutive packet loss, retranmission request loss.

3.4 Efficiency of the Shaker Protocol

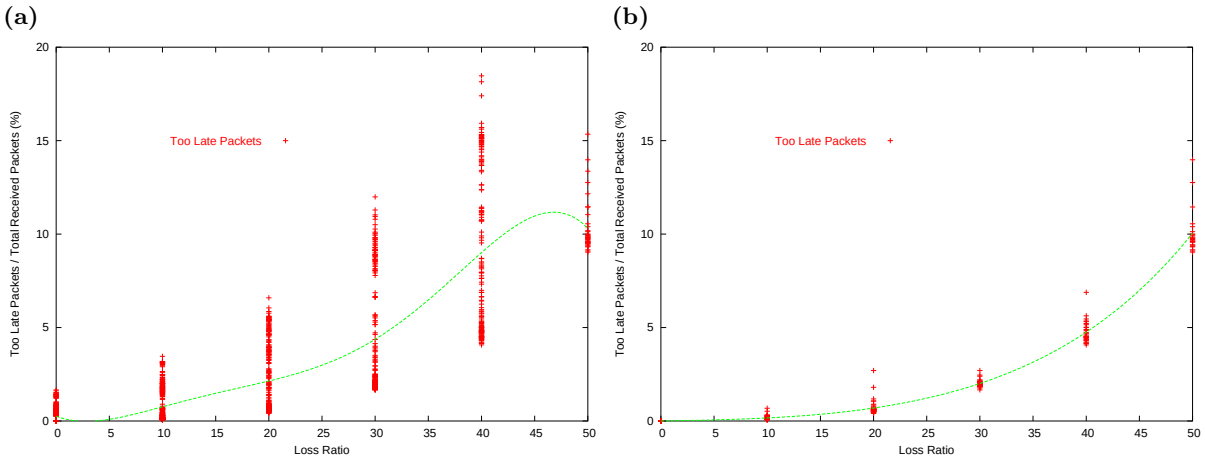
We developed an simple game which allowed us to observe the performance of our protocol in terms of synchronization, delay and loss. We used bucket synchronization between clients, similar to that used in [6], in which time is divided into fixed length periods called rounds. Each event is transmitted marked with the sender's current round. Clients consider packets received from other clients relevant if they were emitted in either the 3 previous or 3 following rounds as well as the current round. The relevancy time of the game is the three previous rounds plus the current one. A round is considered successful if a client receives all the packets sent by other clients in that round while they are considered still relevant. Packets are considered as having arrived *TooLate* if they are not received within the relevancy time. *TooLate* packets occur in the following cases:

– the transmission of the packet took more than the *RelevancyTime*.
– the packet was lost, and the client received the retransmitted packet after the *RelevancyTime*.
– the client received out of order packets, and was missing the packet. Then the request did not arrive on time.

We chose a *RelevancyTime* of 240 ms for our game, this corresponds to an approximate upper bound for FPS games [2]. The peer group size used was ten players, and players remained in the game from start to finish. We tested the performance of the game with losses from 0 to 50% and with RTT variance (jitter) from 0 to 100 % of the mean RTT value set to 75ms. We carried out the measurements across a LAN using the nistnet [**?**] tool at the multicast reflector to induce controllable loss and delay in the system. We used our Linux implementation of the *Shaker* and multicast reflector. When the RTT variance is at 100% the RTT of a given packet can be anything between 0 and 150 ms.

Figure 5(a) illustrates the percentage of *TooLate* packets plotted against the loss probability, for various levels of RTT variance. For a given loss probability, the higher the variance in the RTT, the greater the probability of a a packet being *TooLate*, i.e. for a given value of the x-axis, higher values on the y-axis correspond to higher jitter.

Our protocol, even faced with high levels of loss (50 %) and jitter (50 %) still manages to transmit on average 90% of the packets on time. For an FPS losing some round information can be dealt with by using *dead-reckoning* algorithms, ensuring that the game would be playable even at very high levels of loss. By way of comparision the behavior of UDP would be a line such that x=y, i.e. all lost packets are *TooLate*.



Ratio of Late Pkts, Loss and Jitter

Ratio of Late Pkts, Loss but no Jitter

Fig. 5. Ratio of Late Packets, specific network conditions

In Figure 5(b), we isolated the loss parameter to verify the effects of loss without jitter on the Shaker. We notice that only the combined effects of jitter and loss creates conditions that makes the Shaker wait too long on the receiver side before requesting a packet.

It is not possible to give a meaningful comparision between the Shaker and TCP in terms of reactivity to loss as under such high levels of loss TCP quickly reaches the upper bound of its timeout algorithms — 64 seconds; all that can be said is that TCP is not appropriate for games over lossy networks.

For relatively low loss conditions (less than 20%) we conclude that our protocol's two timeout algorithms are adequate to fulfill the requirements of both loss and latency, even when the variance of the RTT is very high.

We also noticed that the amount of *TooLate* packets is very low in the case of jitter without loss, since the retransmission of packets only provides redundant packets that are filtered by the protocol on the receivers' side. The useless retransmitted packets are not considered as *TooLate* as they are not transmitted to the application.

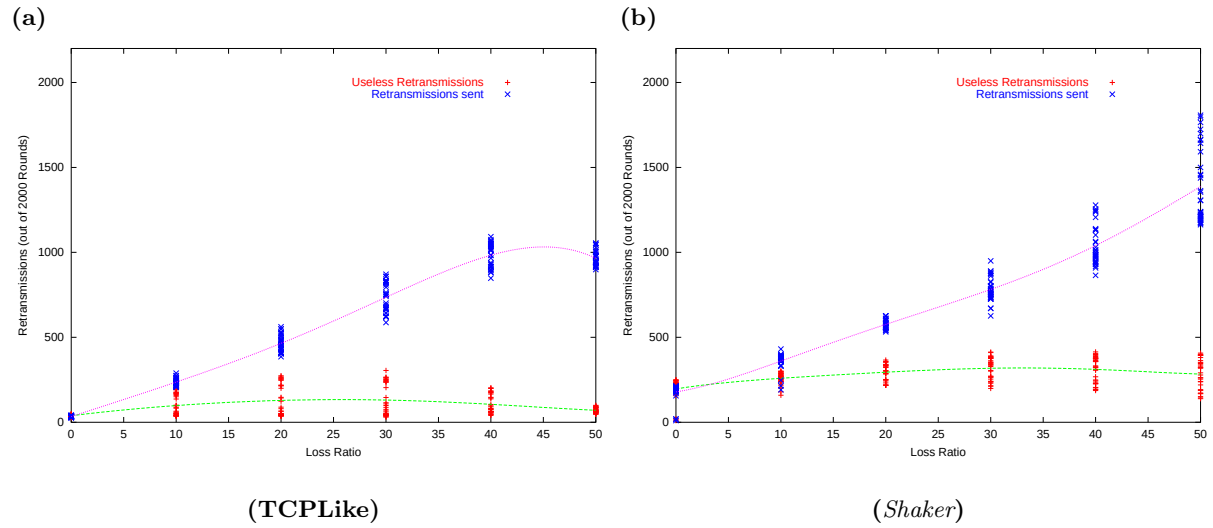
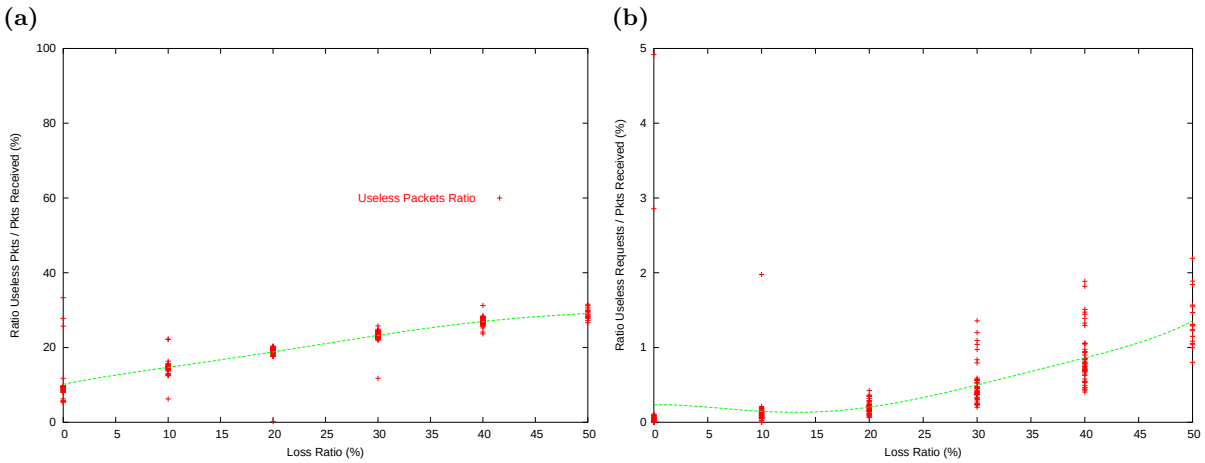


Fig. 6. TCPLike and Shaker Retransmission Ratios

Our protocol trades reactivity against unnecesary retransmissions. The RTO tracks the RTT much closer than for TCP, leading to timeout and retransmission of packets in the case that the RTT rises. In order to quantify how much overhead the Shaker's RTO algorithm has over that of TCP, Figure 6, plots both the total number of packets retransmissions and the number of unnessary ones against loss probability and RTT variance, for both the TCP and *Shaker* RTO algorithm. The *Shaker* retransmits larges number of unnecessary packets when variance is high and loss is low. For example, as Figure 6 shows, at 10% loss with 100% RTT variance, the TCP algorithm leads to about 250 retransmits of which about half are useless, while the *Shaker* retransmits about 450, of which more than half are useless. The *Shaker* mistakes more frequently an upperward change in the RTT with loss, due to the lower weight given to the variance in calculating the RTO, leading to a higher fraction in the number of retransmits which are useless. The *Shaker* retransmits more often as it adjusts its RTT more conservatively after loss.

Figure 7(a) shows the additionnal overhead that the two timeout algorithms place on the network and the receiver, i.e. the cost we pay for the systems better reactivity. It plots the



Total Useless Packets/Received Packets Ratio

Useless Requests/Received Packets Ratio

Fig. 7. Ratio of Late Packets in the Shaker

fraction of uselessly received packets to the total number of received packets. Packets can be uselessly received for four reasons:

- the sender retransmits a packet due to loss of the ACK, the receiver may then get this packet twice;
- the sender retransmits a packet due to too early timeout, this differs from the previous case as both sender and receiver get the useless packet;
- a receiver asks for a packet to be retransmitted due to out of order delivery;
- a receiver asks for retransmission of a packet it has already received. This can occur if the sender has sent the packet twice.

Figure 7(b) shows the percentage of the useless packets that are due to the RRTO algorithm mistaking out-of-order packets for loss. This is always less than 2%. Waiting twice the variance seems to keep unneccesary requests for retranmission minimal. The effect due to unnecessary receiver retransmission can be further decreased by the application by increasing the relevancy time parameter. However, as Figure 7(a) shows, the additional network load as measured on the receiver side varies between 10 % and 30% increasing with loss probability. The main reason for the increase is due to losses of ACK packets to senders, leading them to send the same packet two or more times. Note that the percentage increases in network load is independent of the number of clients since the decision to retransmit or request retranmission depends only on the conditions between the client and the multicast reflector. The additional load is less than 15% for loss probabilties inferior to 10%.

4 Conclusion

We have described the design and implemention of a quasi-reliable transport protocol — *Shaker* — for use in federated peer-to-peer games. This protocol has been designed to allow for the fact that games requires low latency and some selective reliability in packet delivery.

The protocol allows the game designer to trade-off reactivity across a lossy network against the transmission of redundant packets. As senders and receivers are disconnected the game designer might configure this per player, i.e. if the player's access bandwidth is limited only transmit using UDP and never request retransmissions, but use the *Shaker* and UDP if bandwidth is more abundant. It would be possible to alter the trade-off dynamically over time. We have quantified both the benefit and the cost of the use of the *Shaker* through measurement across a wide range of network conditions.

References

1. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content based subscription system. In *In Eighteenth ACM Symposium on Principles of Distributed Computing (PODC '99), Atlanta GA, USA, May 4–6 1999.*
2. G. Armitage. An Experimental Estimation of Latency Sensitivity In Multiplayer Quake 3. Technical report, Centre for Advanced Internet Architectures, 030405A, 2003.
3. D. Bauer and S. Rooney. The Performance of Software Multicast-Reflector Implementations for Multi-Player Online Games. In *Proc. of the Fifth International Workshop on Networked Group Communications (NGC'03)*, Munich, Germany, Sept. 2003.
4. Y. Bernier. Latency Compensation Methods in Client/Server Game Protocol Design and Optimization. In *Proceedings of GDC 2001.*
5. A. Bharambe, S. Rio, and S. Seshan. Mercury: A Scalable Publish-Subscribe System for Internet Games. In *NetGames 2002 – First Workshop on Network and System Support for Games*, Braunschweig, Germany, Apr. 2002.
6. C. Diot and L. Gautier. A Distributed Architecture for Multiplayer Interactive Applications on the Internet. *IEEE Networks magazine*, 13(4):6–15, July/August 1999.
7. V. Jacobson. Congestion Avoidance and Control. *ACM Computer Communications Review*, 18(4):314–329, Aug. 1988.
8. E. Léty and T. Turletti. Issues in Designing a Communications Architecture for Large-Scale Virtual Environments. In *Proceedings of the First International Workshop on Group Communications*, Pisa Italy, 1999.
9. B. N. Levine, J. Crowcroft, C. Diot, J. Garcia-Luna-Aceves, and J. F. Kurose. Consideration of Receiver Interest for IP Multicast Delivery. In *Proc. IEEE Infocom*, volume 2, pages 470–479, 2000.
10. D. Levine, B. Whitebook, and M. C. Wirt. *A Massively Multiplayer Manifesto.* Butterfly.net, Inc., 123 East German St. Shepherdstown WV 25554, May 2002. Version 1.1.
11. K. Morse. Interest management in large-scale distributed simulations. Tech report 96-27, Dept. of Information and Computer Science, University of California, Irvine, 1996.
12. M. Piecuch, K. French, G. Oprica, and M. Claypool. A Selective Retransmission Protocol for Multimedia on the Interner. In *Proceedings of the SPIE International Symposium on Multimedia Systems and Applications*, Boston MA, USA, 2000.
13. Sony Online Entertainment Inc. EverQuest. http://www.everquest.com, 2002.