RZ 3528 (# 99542) 01/19/04
Computer Science 15 pages

Research Report

A Federated Peer-to-Peer Network Game Architecture

Sean Rooney, Daniel Bauer, Rudy Deydier

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
{sro,dnb}@zurich.ibm.com

Research
Almaden · Austin · Beijing · Delhi · Haifa · T.J. Watson · Tokyo · Zurich

A Federated Peer-to-Peer Network Game Architecture

Sean Rooney, Daniel Bauer, Rudy Deydier [⋆]
Zurich, Switzerland. E-mail: {sro,dnb}@zurich.ibm.com

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

Abstract. A federated peer-to-peer game is one in which many small areas of interest within a game each supported using a peer-to-peer model are 'knitted' together to form a game capable of supporting a very large number of players. Our work has involved determining whether such an architecture is a feasible alternative to the more common central server one for supporting large multiplayer network games.

1 Introduction

Gaming is a relatively neglected topic for communication's research — in part this may be ascribed to the closed and proprietary nature of commercial gaming systems — while at the same time gaming presents distinct challenges to network designers, particularly in respect to its sensitivity to latency and loss.

A Massive Multiplayer On-line Game is a networked game in which many thousands of players participate simultaneously in the same game. 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 — involving unnecessary expense — or too small — meaning lost revenue and/or unhappy players.

One solution to this problem is to pass from a model in which each game providers own their own infrastructure to one in which they rent cycles, storage and bandwidth from some third party which gains economies of scale by hosting multiple games simultaneously. The third party may then dynamically reallocate resources to a game in response to demand. The problem of sharing infrastructure between games is much harder than that for web applications due to both the lack of standards and the demanding nature of gaming applications in regard to latency. While a slow responsiveness from a web site is irritating, slow responsiveness from a games server renders the game unplayable. Consequently crosstalk between gaming applications running on the same shared infrastructure can have serious consequences.

Our work has considered a novel architecture for large games that we term a *federated peer-to-peer* architecture 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. Each of these areas of interest is supported by a peer-to-peer game.

[⋆] Rudy Deydier contributed to this work while visiting the IBM Zurich Research Laboratory.

While some previous work in the literature has proposed broadly similar architectures, for example [10], details are scant about how the systems actually work in practice making it difficult to evaluate their feasibility. To the best of our knowledge no actual commercial large game uses the federated peer-to-peer model.

In previous work, we extended a model for gaming proposed in [5] to formulate a cost model that allows us to assess game architectures through a set of cost functions [3]. An evaluation of the server components of the client-server and federated peer-to-peer architectures showed that while from the game provider's point of view both grow quadratically in the number of players, the predominant cost in the client-server architecture is the processing of game state while in the federated peer-to-peer architecture it is communication. This means that the projected faster-than-Moore's-law increase in bandwidth will make the federated peer-to-peer model increasingly attractive.

Here, we consider whether such an architecture is really feasible in practice by describing our implementation of various important components. Our intention is to give enough detail about our implementation in order to allow an evaluation of the strengths and weaknesses of such an approach. This work synthesizes and extends previously published work [3, 4, 12].

2 Overview

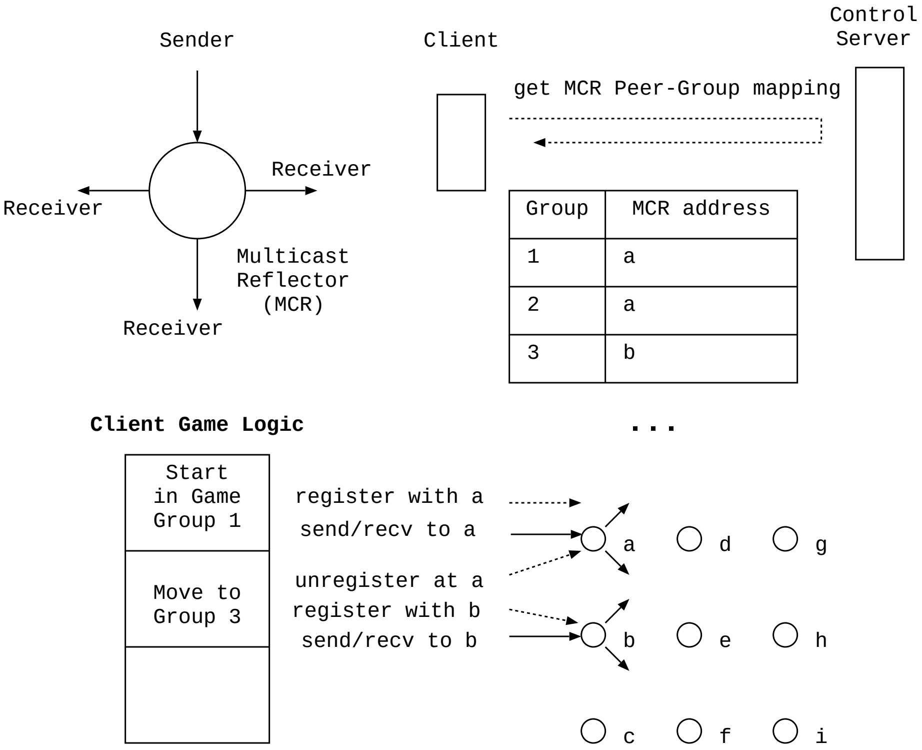
We use an interest management system that aims to deliver packets only to the subset of clients for which the packets are relevant. The main challenge is to make that system simple enough such that the highly latency-sensitive game traffic is not adversely effected by the mechanisms which support the interest management system.

The game is divided into areas of interest. Clients subscribe to one or more areas of interest and receive information sent by all other clients within that area. The game logic is executed only at the client in the standard peer-to-peer fashion.

The areas of interest are supported by multicast reflectors capable of maintaining lists of current subscriber and disseminating packets among the players. The multicast reflectors are unaware of game logic which, as we show, allows them to be simple and efficient. The association of multicast reflectors to areas of interest is performed by control servers. The client, after receiving this information, from the control servers determines to which multicast reflectors it should be subscribed using game logic instrumented at the client. The multicast reflectors themselves are only required to ensure the efficient forwarding of packets to the subscribed clients. Control servers may change the association of areas of interest to multicast reflectors due to changing circumstances, e.g. failure, load-balancing, and clients are alerted as to this change.

The action of the control servers is to a large extent independent of the nature of the game, they partition the game into areas of interest, as well as performing bookkeeping tasks such as authentication, character storage, etc. Most of the logic specific to a given game is executed at the client. This permits most of the complex processing to be offloaded by the game provider to the users reducing the cost of the game to the provider. Figure 1 shows the structure of our generic federated peer-to-peer architecture.

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



Group	MCR address
1	a
2	a
3	b

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

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 [6], 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. Selective retransmission mechanisms have been proposed for multimedia streams [11]; the Selective Retransmission Protocol (SRP) is a client/server protocol that allows a client to trade packet loss ratio against average packet latency in a continuous media stream.

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 our *Shaker* transport protocol — detailed in Section — unreliable packets are just sent using UDP. 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.

3 Implementation

In this section we explain in more detail the implementation and performance of some of the key components in our federated peer-to-peer architecture.

3.1 Game Architecture Control Layer

The multicast reflector is an IP addressable entity capable of maintaining a peer group of end host addresses between which received packets should be broadcast. Each peer group in effect constitutes a virtual broadcast domain. A given multicast reflector may handle many such peer groups. In the current implementation different peer groups at the same multicast reflector are distinguished using the destination port number.

Clients obtain the 'multicast reflector'/ 'area of interest' mapping from the control server at login and thereafter are informed when it changes. An area of interest may be supported by a single peer group or, in order to achieve scalability, it may also be supported by multiple peer groups on different multicast reflectors. How the bridging between the peer group is achieved is explained later.

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. 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, whereby some party publishes information within some area of interest and parties who have subscribed to that area of information receive it.

As well as the efficient distribution of packets the multicast reflectors assists in supporting the *Shaker* protocol. It 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 efficiently implemented in software or programmable hardware. Its implementation is explained in Section 3.3. There is one *Shaker* session for every peer-group that a client participates in.

The control servers have some sets of variables each of which corresponds to some parameter upon which the game can be divided. For example, the virtual locations within the game. Each of these values is represented by a single positive integer and the mapping of this value from the actual game value, for example a three dimensional coordinate, is performed by the client For the purposes of explanation we assume just a single variable, that we call the location variable.

If belonging were binary then a player would either belong to a group or not, and consequently get all information about that group or nothing. This would not be desirable as it would mean that if a player needed some information in a group they would get all of it and then have to do the filtering themselves. Instead we introduce the notion of an affinity value [1]. The affinity value is a measure of the level of participation that the client wishes to have in a given peer group. For example, a player may subscribe with a high affinity value to the location at which they are currently resident and also subscribe to all neighboring locations with a much lower degree of affinity. The affinity values acts like a filter in a publish/subscribe mechanism, only delivering information to a player if they have subscribed at the appropriate affinity value. However, it is much more restrictive than a general purpose filter language; the matching is done by the clients themselves within the control layer not in the data path.

At the control server the location variable is associated with a set of records each corresponding to the destination multicast reflector IP address and port that should be used for a given affinity value. Each IP address, port number pair is distinct for each affinity value. Although there may be multiple different pairs for a given value. The higher the degree of affinity the greater the level of belonging and consequently the more information the client will receive.

A client send a packet to a peer group with a degree of affinity that is equal or less than that with which it subscribed. The packet is delivered to all clients subscribed with a degree of affinity equal or greater. This ensures that a client always receives its own packet; intuitively this makes sense: a client cannot send information that it itself is not interested in. The higher the degree of affinity that a client sends with, the more restricted the interest group. Note that the ingress multicast reflector to which the player sends the packet need not be one at which it is subscribed, i.e. it is one corresponding to an affinity value less than the client's own, but players always receive the packets from multicast reflector to which they are subscribed.

When different multicast reflectors handle different affinity values of the same peer group then the rule is that multicast reflectors with higher degrees of affinity always subscribe to that of the next lowest. The multicast reflector subscribes to a peer much as a normal client does except that packets received from another multicast reflector are never transmitted back. The multicast reflector modifies the sequence number and buffers the packet just as if it came from a host. Retransmission of packets is achieved by communication with the egress multicast reflector. Note that we assume that there is little or no loss between multicast reflectors: multicast reflectors never ask for retransmissions from each other. If loss occurs between two multicast reflectors, then the clients on the receiving side have no way of detecting that a packet has been lost as the receiving multicast reflector never saw it and therefore never assigned it a sequence number. We consider that requiring the multicast reflector themselves to implement the transport protocol would make them more complex and therefore less efficient. Consequently, the game provider must provision sufficient capacity between multicast reflectors such that loss is unlikely.

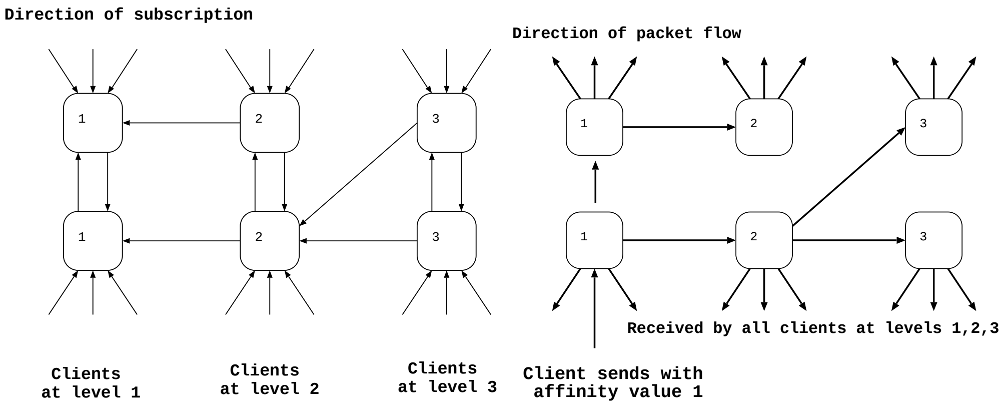
For reasons of scalability and fault tolerance multiple multicast reflectors may manage the same affinity level within a peer group. All multicast reflectors within the same peer group affinity level subscribe to each other to form a full mesh. A multicast reflector at a higher affinity subscribes to *exactly one* at the next lowest level. The multicast reflector only

[1] The work described in this section is covered by Pending IBM Patent CH8-2001-0088

forwards packets to other multicast reflectors if they came directly from a host, otherwise a packet could be received multiple times by a multicast reflector. A multicast reflector can determine if a packet came from a host or another multicast reflector by examining whether the sequence number is set or not, i.e. if its value is zero. When a packet is received by a multicast reflector the algorithm is then:

```
Algorithm 3.1: Multicast Reflector Packet Forwarding Algorithm()

comment: Action to perform at a MCF on receiving a packet
for each $e$ in the peer group
    if pktReceived is from host
        then send pktReceived to $e$
        else if not $e$.affinity == my affinity
            then send pktReceived to $e$
```


Direction of subscription
1
2
3
1
2
3
Clients at level 1
Clients at level 2
Clients at level 3
Direction of packet flow
1
2
3
1
2
3
Received by all clients at levels 1,2,3
Client sends with affinity value 1

(a) Subscription

(b) Packet Flow

Fig. 2. Example of interaction between multicast reflectors

Figure 2(a) shows the direction of subscription between multicast reflectors with different affinity values *1*, *2* and *3* while Figure 2(b) shows the direction of packet flow. In effect the algorithm allows the passage of a packet through the multicast reflector to follow a tree extracted from the graph of subscriptions.

A softstate model is used for maintaining subscription in a peer group, subscribers must periodical renew their subscription. If one multicast fails in its attempt to register or renew its subscription with another it informs the control server. When the control servers changes the division of the game it distributes the mapping to the multicast reflectors which reorganize themselves in accordance with the rules described above. The *update alert* flag is set in the *Shaker* transport protocol header — see Figure 3 — on the packet returned to the sender for some duration. Clients are expected then to obtain the new mapping from the multicast reflector.

3.2 Shaker Transport Protocol

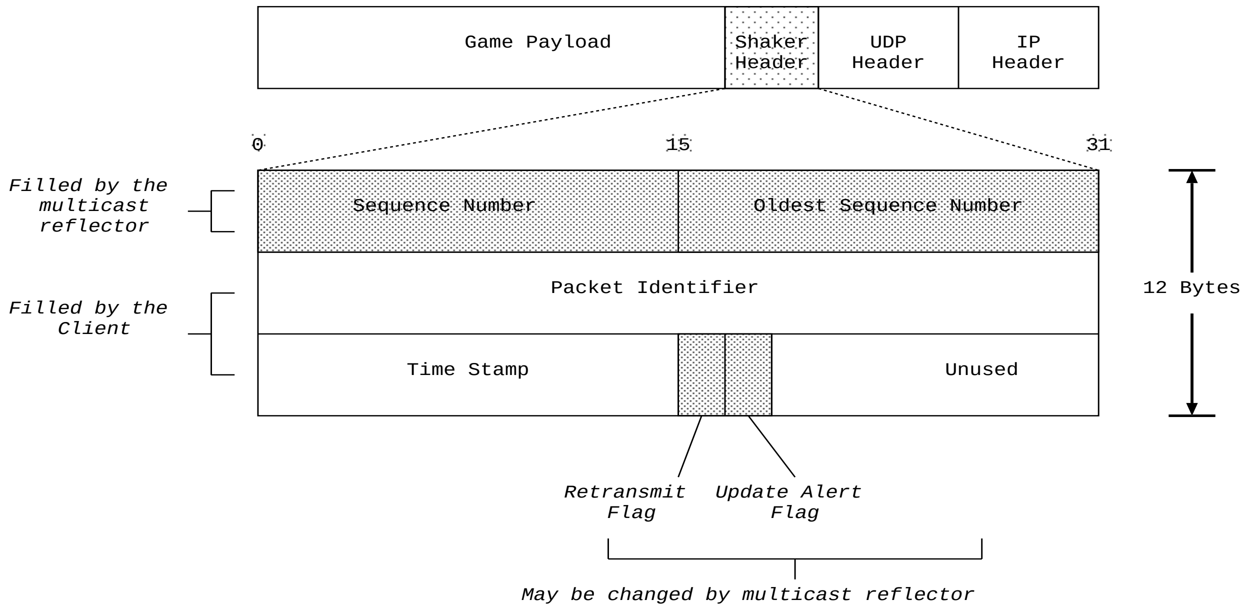
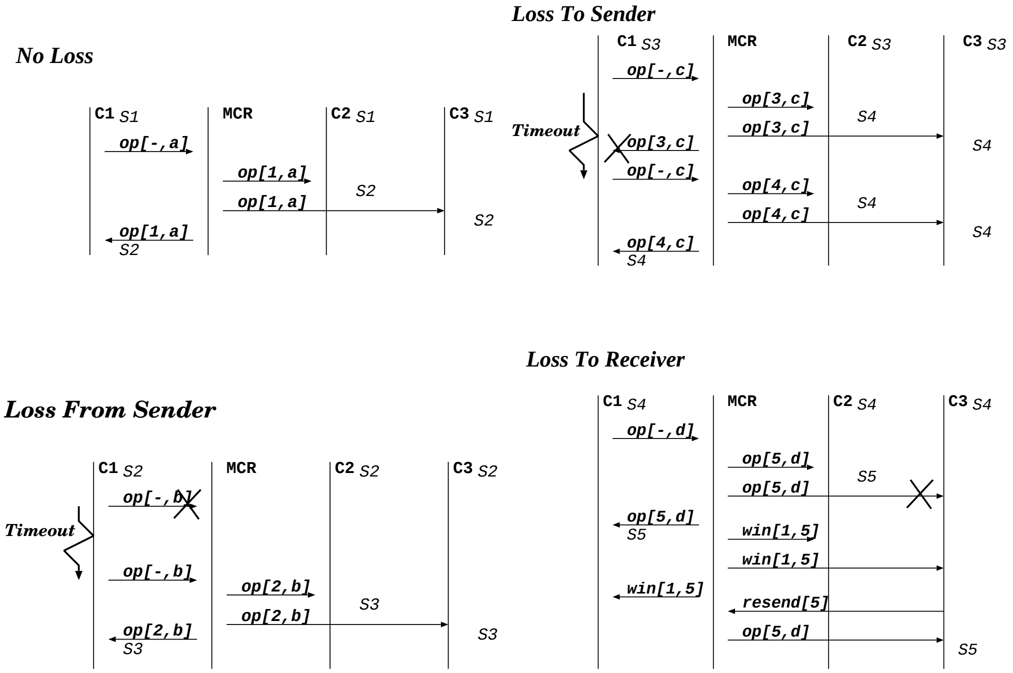


Fig. 3. Shaker Header

The *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.

The *Shaker* adds its own header — shown in Figure 3 — 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.

Figure 4 shows the communication patterns under four conditions: no packet loss, loss from sender, loss to sender and loss to receiver. As the sequence 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. 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



(a) (b)

Fig. 4. Interactions within the Shaker protocol

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.

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) [8] as shown in Algorithm 3.2; 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 3.2: 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 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 [8].

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 occur, 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 3.3: 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 3.3 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.

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 incur 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 3.4: RRTO()

comment: Compute the RRTO based on $RelevancyTime$ and $D$
if $RelevancyTime < 1.5 * RTO$
  then { $DiscardRequest$
         $NotifyApplication : LostPacket$
  else $RRTO = Min(RelevancyTime - 1.5 * EstimatedRTT, 2 * D)$
```

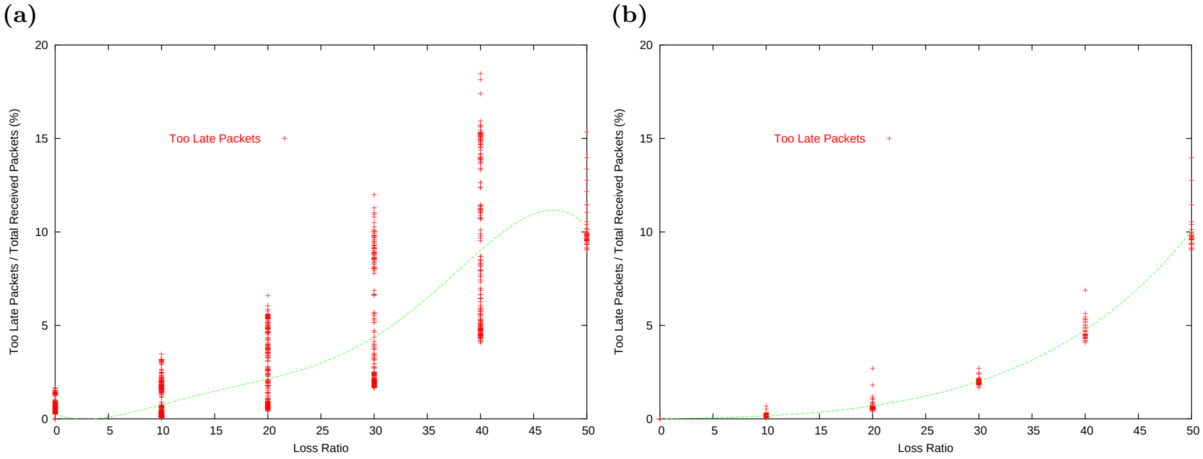
Efficiency of the Shaker Protocol We developed an simple game which allowed us to observe the performance of our protocol in terms of synchronization, delay and loss. We used bucket synchronization between clients, similar to that used in [7], in which time is divided into fixed length periods called 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 Linux 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 comparison 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.

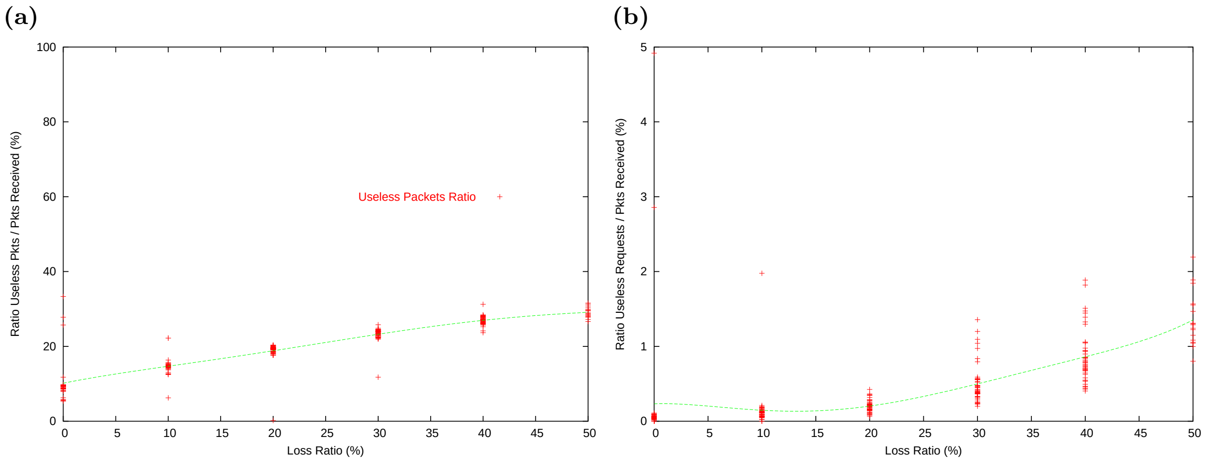
For relatively low loss conditions (less than 20%) we conclude that our protocol's two timeout algorithms are adequate to fulfill the requirements of both loss and latency, even when the variance of the RTT is very high.

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.

Our protocol trades reactivity against unnecessary 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.

Figure 6(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 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;



Total Useless Packets/Received Packets Ratio

Useless Requests/Received Packets Ratio

Fig. 6. Ratio of Late Packets in the Shaker

- 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 6(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 6(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%.

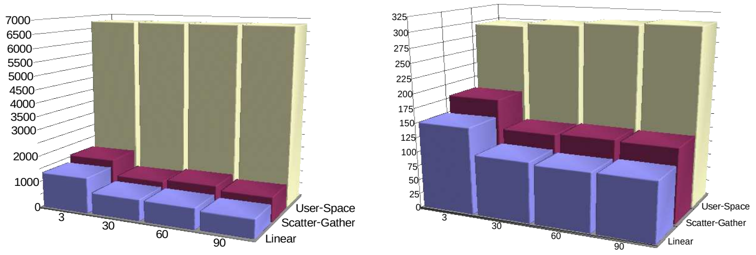
3.3 Software Multicast-Reflector Performance

A key feature of the multicast reflector is the ability to send UDP datagrams to a set of receivers very efficiently. Efficiency is measured by both a high throughput and a low CPU load.

A user-space implementation of the multicast reflector has the advantage of being simple and portable. However, efficiency is not optimal. For each registered entity, the payload data is copied from the user space to the kernel space even though the payload itself does not change. The other problem, which is more severe, is the fact that UDP datagrams may be dropped when the kernel's device queue overflows. The multicast reflector then has no other possibility than to resend the dropped packet.

A more efficient implementation can be achieved by extending the UDP layer in the kernel. The principle is to maintain a list of address/port tuples per socket in the kernel. A datagram sent on the socket is then sent to each entry in the list. In order to achieve a performance benefit, the payload is copied and checksummed only once per multicast operation instead of once per destination. The problem of device queue overflows can be handled more elegantly in the kernel than in user space. If a packet was dropped owing to queue overflow, then the process is suspended (put to sleep) until the queue is almost empty. If the queue is drained below a low-water mark, then all processes that have been put to sleep are woken up and continue queuing packets. This is implemented using the kernel's wait queues.

Measurements of kernel and user-space implementations have shown that the throughput achieved by all implementation is in the same range, whereas the CPU load is much higher in the user-space implementation [4]. The kernel-space implementations consume considerable less CPU resources in packet forwarding thus leaving more resources available for control tasks. Figure 7 shows an example of the different CPU utilization for a user-space implementation and two versions of kernel-space implementations over a 100 Mb/s Intel E100 Ethernet adapter. One of the kernel-space implementation uses the scatter-gather feature of the network device, the other uses a linear buffer to store packet header and payload. In the experiment, packets of 100 bytes and 1000 bytes have been sent to 3, 30, 60, 90 clients. The ordinate gives the CPU load in millions of clock ticks.



(a) 100-byte packets

(b) 1000-byte packets

Fig. 7. CPU load using Intel E100

Throughput figures, of course, are heavily dependent on the used network device and the packet size. For example, when using a 100 Mb/s Intel E100 network device, for 100-, 500- and 1000-byte packets, the net throughput (payload only) are 60, 88 and 94 Mb/s, respectively for all implementations. For Gigabit Ethernet, the net throughput ranges from 90 to 140 Mb/s for small packets of 50 bytes up to 700 Mb/s for 1000 byte packets.

Assuming a client of a typical game produces a stream of 8 Kb/s using an average packet size of 100 bytes [1], then our software multicast reflectors are able to support 7'500 clients with fast Ethernet and between 20'000 and 30'000 for Gigabit Ethernet. This means that even for very large games with up to one million participants, less than 100 multicast reflectors are required.

4 Conclusion

The federated peer-to-peer game architecture's key advantage is that the amount of resources available to a game rises with the number of players. By suitable design of the multicast reflector and transport protocols the game provider need only perform relatively infrequent control operations thereby reducing their investment in the infrastructure. The multicast reflector themselves can be geographically distributed spreading the network load across multiple sites and enabling better resilience. The disadvantage of the architecture is that as the clients maintain their own state that desynchronization, whether intentionally or unintentionally, between the actual and perceived state of a player becomes a major problem. As all clients within a group receive all events it maybe possible for clients to checkpoint each others states allowing a rollback in the case of inadvertent desynchronization or alarms in the case of cheating.

References

1. A. Abdelkhalek, A. Bilas, and A. Moshovos. Behavior and performance of interactive multi-player game servers. In *Proceedings of the International IEEE Symposium on the Performance Analysis of Systems and Software (ISPASS-2001)*, Nov. 2001.
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, I. Iliadis, S. Rooney, and P. Scotton. Communication Architectures for Massive Multi-Player Games. Technical Report RZ3500, IBM Research, June 2003. `http://www.research.ibm.com/resources/paper\_search.shtml`.
4. 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.
5. N. E. Baughman and B. N. Levine. Cheat-proof Playout for Centralized and Distributed Online Games. In *Proc. IEEE Infocom*, pages 104–113, 2001.
6. Y. Bernier. Latency Compensation Methods in Client/Server Game Protocol Design and Optimization. In *Proceedings of GDC 2001*.
7. 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.
8. V. Jacobson. Congestion Avoidance and Control. *ACM Computer Communications Review*, 18(4):314–329, Aug. 1988.
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. M. R. Macedonia, M. J. Zyda, D. R. Pratt, D. P. Brutzman, and P. T. Barham. Exploiting Reality With Multicast Groups: A Network Architecture for Larce-scale Virtual Environments. *IEEE Computer Graphics and Applications*, 15(5):38–45, Sept. 1995.
11. 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.
12. S. Rooney, D. Bauer, and P. Scotton. Efficient Programmable Middleboxes for Scaling Large Distributed Applications. In *6th International Conference on Open Architectures and Network Programming (OPENARCH)*. IEEE, Apr. 2003.