Real-Time Peer-to-Peer Multicast and Gaming

Research era: 2001–2005 (Brown period) · Originally published on Blogger: 2012-10-31 · Archive recovered October 2026

> **2026 editorial context (not part of the original post).** Full thesis narrative, Arrow Multicast diagrams, latency and packet-order experiments, and documented Peer-to-Peer Quake prototype.
>
> 2026 context: an early embodiment of distributed decision-making under time constraints, relevant as engineering foundations for later physical-world AI.

Original Blogger address: http://adiinsf.blogspot.com/p/multicast-for-real-time-data-brown-u.html (live availability not established). Old report links below are carried forward but not guaranteed to work.

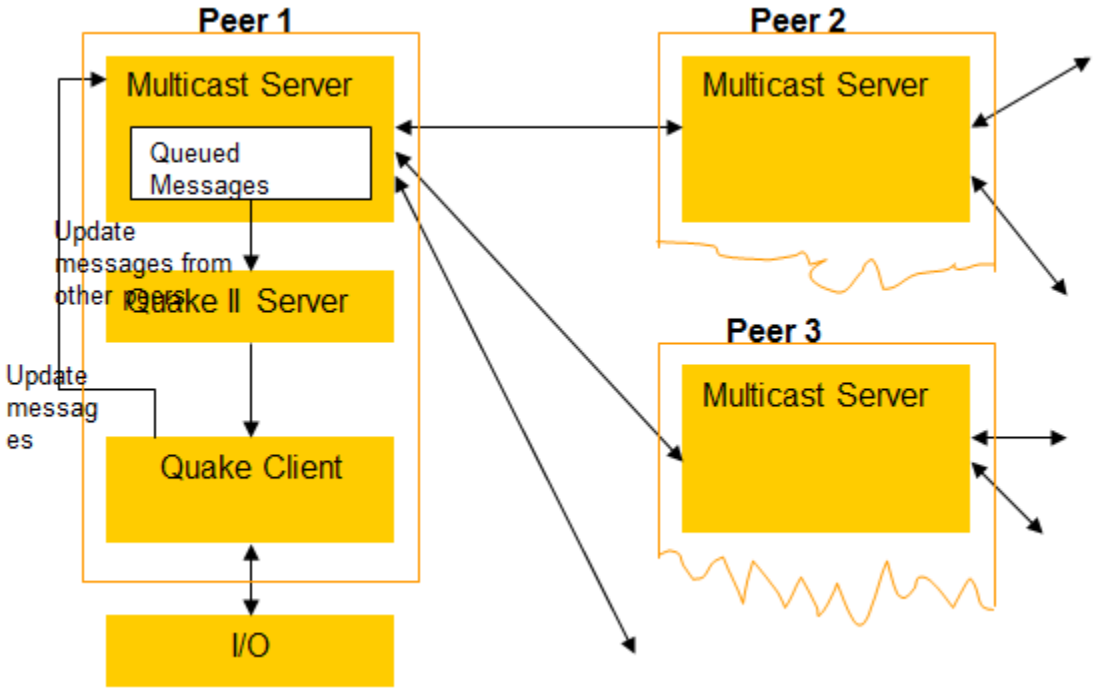
Original 2012 archive text

Publications

Maurice Herlihy, Aditya Mohan, "Peer-to-Peer Multiplayer Gaming using Arrow Multicast: Peer-to-Peer Quake ", The 23rd International Conference on Distributed Computing Systems (ICDCS-2003) Providence, Rhode Island USA, May 19-22, 2003. [www, www_poster]

Box.net folder: Summary Slides Thesis report

Graduate School Thesis @ Brown University: Peer-to-peer Multicast for real-time information



A major challenge in the current distributed network environment is to make multicast scalable and fault-tolerance. When working in a peer-to-peer domain, there are numerous applications that require ordering of the multicast packets. Moreover, latency is a concern, especially for interactive applications such as Multiplayer Gaming. We evaluated the effectiveness of a proposed ordered multicast algorithm called Arrow Multicast as a possible solution. The multicast was aimed to be asynchronous, ordered but best effort, where packets can be dropped if the required quality of service (QoS) requirement was not met. QoS in was quantified by the freshness of the packets received and the amount of packets dropped/delivered unordered. The outcome of the thesis was a Multiplayer Peer-to-Peer Quake prototype, which was used as a test bed for our proposed peer-to-peer multicast. The research findings were published as a poster titled "Peer-to-Peer Multiplayer Gaming using Arrow Multicast: Peer-to-Peer Quake ", in The 23rd International Conference on Distributed Computing Systems (ICDCS-2003) Providence, Rhode Island USA, May 19-22, 2003

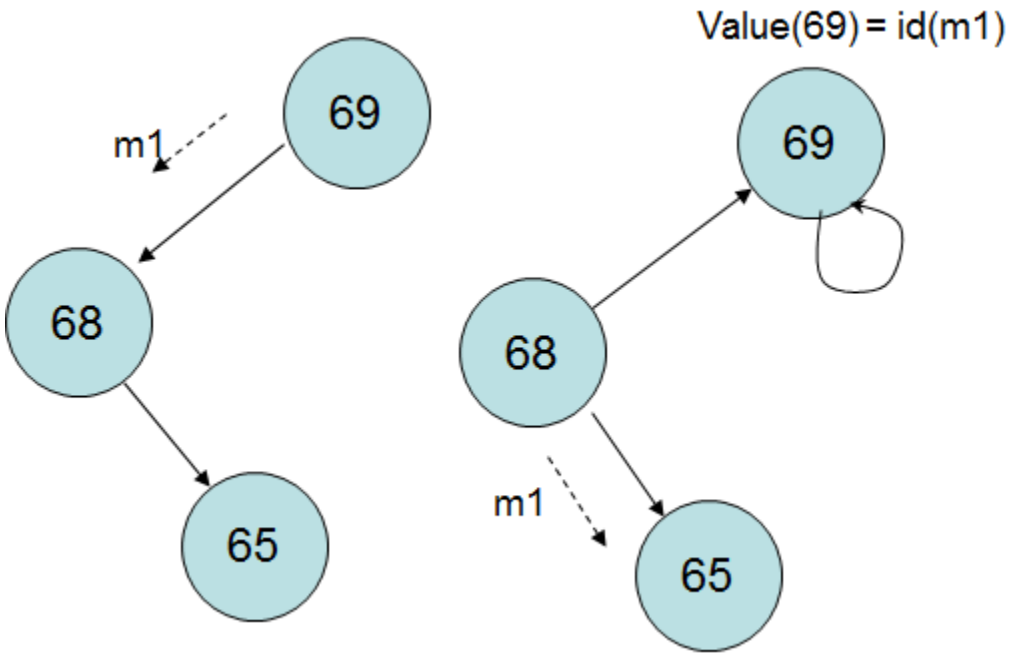
Characteristics of our approach

- No centralized server Each peer has the full game engine
- Asynchronous – lock-steps not required
- Ordering of Game frames to prevent divergence
- Best effort delivery of frames

We used Arrow Protocol used in directory services

- Message only knows it predecessor
- Path Reversal

- Light Weight
- Minimal state information
- FIND/FLOOD message
- FIND hunts down the predecessor
- FLOOD for unordered multicast



1. Message finds its predecessor
2. Unordered multicast of message with predecessor info
3. Receivers Order with help of predecessor info

Introduction

There are numerous constraints that one can put on a multicast environment. Applications may need the multicast to be atomic, where we either deliver a packet to all or we don't deliver to anyone. It may need to be reliable and guarantee delivery of packets while other application may just need it to be "best effort". Timely delivery of the packets is another constraint. It may need to be ordered, where each packet should arrive at the destination in the same order that they were generated to maintain consistency among the different receivers. Some may need *total ordering* [5, 26] where packets from all the sources arrive independently without any form of coordination between each other. These should be ordered based on the time they were sent; while the other may require *partial ordering* [5, 26] - ordering for packets from a single source. These requirements of ordering, reliability, timely delivery, and atomicity can be met in synchronous fashion, where we

proceed in lock steps. This solves most of the problem but is not suitable in an environment like the Internet where the delays can be variable and nodes have varying processing power. Another approach that has a wide acceptance among the multicast arena is that Virtual Synchrony, where we preserve the illusion that events occur instantaneously, but enforces weaker delivery ordering [3]. The requirement in this case is that of group membership - every node knows the exact group membership. Any approach that requires the maintenance of groups is bound to suffer from scalability problem. Hence if scalability is a requirement, node anonymity is a must. We are interested in an *asynchronous best-effort totally ordered multicast while maintaining the node anonymity as oppose to group membership*. We also have the constraint that we need to do the multicast in a truly decentralized peer-to-peer fashion. *Through out the paper, we will use the convention that ordering refers to total ordering.*

Total Ordering has been considered to be expensive. There has been a wide spread belief that total ordering using a asynchronous protocol with node anonymity (as opposed to group membership) is not a realizable task. This comes directly from the believers of the Virtual Synchrony [1].

There are numerous applications where a best-effort ordered multicast can be useful. We describe few of them here.

Peer-to-Peer multimedia video/audio streaming is a good candidate for best-effort ordered multicast. A network of machines on the Internet, each peer sending live video/audio stream from his/her web cam to everyone else. A certain packet loss rate is tolerable, but ordering from a single source needs to be maintained (partial ordering). Total ordering is required to make sure that, events (video/audio events) happening among the peers are perceived correctly.

Active Sensor Networks, where each sensor is capable of receiving, transmitting and processing information. Consider the data centers of Google Inc with 10,000 Pentium II/III servers running simultaneously at the same time with a density of 80 servers in a space 7 feet by 2 feet by 2.5 feet. The problem is to maintaining the data center's temperature to constant, while not keeping the air-conditioning on all the time. Moreover a processor may need to shut down and go to sleep mode if the air-conditioner is working at the maximum limit and the room temperature is still not optimal. Each processor is connected to an active thermal sensor, which can wirelessly communicate, with every other thermal senor embedded in the each of the 10,000 machines. Since the temperature of the machines keep on changing based on the load on the processors, the room temperature changes accordingly. Periodically every thermal sensor gets temperature reading of its local processor and multicast it to every other processor. Each thermal sensor can itself send commands to other sensors based on its present state. When a thermal sensor, based on the temperature reading from other sensors finds that its local processor needs to go to sleep, it sends a message to others telling them about the local processor 's future state. This is required since other local sensor may be converging to the same conclusion too. The same is done by the others. Some messages can be lost due to improper wireless link or due to the microcontrollers in the sensors not being able to process the information fast enough. But the messages need to be delivered in order.

The application that we are using to implement our protocol is Multiplayer Gaming. We use the Multiplayer version of Quake. The motive is to remove the central server that is a bottleneck and makes the system restricted to few players at a time in a game. Intent is to makes sure that in the Peer-to-Peer Quake, we are able to multicast the movement packets from each peer to every other peer and keep the state of the global environment same at each peer. We use Arrow Multicast for ordering packets in the network.

Our research contribution is to build and analyze an asynchronous multicast infrastructure based on Arrow Multicast protocol and evaluate its performance as an alternative to Virtual Synchrony. Since we are using a user driven application for our implementation (Multiplayer Quake), it is fairly easy to determine if our multicast infrastructure works or not.

System Model

Before we begin to describe our system, there are few assumptions that we make. We assume that we have fixed network of nodes. Ideally these should form a Spanning Tree. This is required to make sure that the message traversal for various Arrow Messages (described later in the paper) is optimal and we don't go into loops. [2] has proposed methods to make and maintain Spanning Trees and is not the focus of our research.

We don't consider fault tolerance for now. I.e. a node in the network cannot fail or join abruptly. This requirement implies that we start the multicast only after the Arrow Network is ready with all the TCP connections between different peers in place.

Our system can be considered as building an ordering layer on top of the existing multicast infrastructure. Since we are interested in a best effort approach, we don't care about the reliability of the message delivery. For the underlying multicast layer, we use the already existing spanning tree. When a node needs to do an unordered multicast (Flooding), it starts delivering the packets to its connected children in the tree. This is done for all nodes except for the one where the packet initially came from. This is done by all the nodes till the whole tree is covered. This could be replaced by other multicast options like IP Multicast.

System Architecture

The basic idea is to have a Quake Server (Local Server) running in tandem with a local client in each machine. The Local Server and the Local Client communicates via the Multicast Server in most cases. Multicast Servers of different peers form the Arrow network.

A. Peer-to-Peer Quake

Peer-to-Peer Quake uses a decentralized architecture in which there is no central server, but instead each client itself has the game engine. Each player in our Peer-to-Peer Quake has a logical server running inside her machine. To make sure that the game is playable and the user does not perceive too much delay because of some game frames, the delivery is best effort – the servers may discard some of the frames that are not able to come on time or may deliver the frames unordered. We choose to discard the unordered frames instead of delivering them. This happens whenever the Receive Queue is not able to take more packets. This decision was based on the fact that for applications like Multiplayer Gaming and Multimedia streaming, a packet loss is tolerable since this may only result in a short jitter. On the other hand, an unordered packet will result in a totally unexpected experience. Imagine a situation where we are hit by a missile, without our opponent actually firing it.

The Game Engine is modified to form a logical three-tier architecture consisting of the Multicast server, Quake Server and the Quake Client. The local client generates the movement messages using the keyboard input from the user.

Performance Metrics

A. Order Error

If we never delete any packet from the Queue structure or we don't deliver any packet/packet sequence that has a missing predecessor, then we are bound to have a 100 percent guarantee for packet ordering since we are not loosing any packet anywhere and we are not delivering unordered. But this may not always be in a condition to deliver all packets ordered.

Although, we are assuming a reliable transport, packets may get delayed in the network due to varying network latencies and congestion. This will result in the packets coming out of order. We may not want to wait for a full ordering before we deliver the packets because of the real time nature of the application under consideration.

We define the ORDER_ERROR as the cumulative ratio of packets that are sent unordered to the total number of packets sent to the local server. Further the total number of packet that are sent unordered are defined as the number of packets that gets added to the Queue and then have to be delivered forcefully (e.g. based on the Freshness constraints) to the local server. Hence any packet that got delivered due to a flush operation on the queue, are considered to be adding to the ORDER_ERROR.

B. Latecy – Freshness

If perfect ordering guarantees are not required, we can go to the best effort case. We can tolerate some degree of ORDER_ERROR but we want freshness guarantees. The Ordering condition here is that the packets should be delivered in the right order. Freshness condition is that the packets should be delivered to the Local Server at the earliest possible time keeping in the view that we deliver maximum possible packets in order.
We define freshness as the total time taken by a packet from the moment it was generated by a certain Local Client and the time it was finally delivered to a Local Server. Hence Freshness can be considered to be the end-to-end delay of a packet. Less is the end-to-end delay, fresher is the packet.

QoS Guarantees

A. Ordering Guarantees

If we never delete any packet from the Queue structure or we don’t deliver any packet/packet sequence that has a missing predecessor, then we are bound to have a cent percent guarantee for packet ordering since we are not loosing any packet anywhere and we are not delivering unordered.

B. Freshness/Latency Guarantees

If perfect ordering guarantees are not required, we can go to the best effort case. We can tolerate some degree of Staleness (imprecision) but we want freshness guarantees. The Staleness condition here is that the packets should be delivered in the right order. Freshness condition is that the packets should be delivered to the Local Server at the earliest possible time keeping in the view the Staleness condition.

An easy solution would have been to check the timestamp of packets that the Receive Queue is receiving and before putting the packet in the Queue, attach the time of arrival of that packet in the queue. We deliver all the packets that have a time difference of more than the freshness threshold. This ensures that the packet latency is kept to a minimum.

But we can't take use the time stamp from the packet since, each peer has an independent clock. Hence because of time skew between various clocks, we can't use the timestamp from a received packet as such.

The solution is to add another field in each movement packet. Each time a peer sends a movement packet, it also adds the time difference between the sending of the current and the last packet sent. This time difference gives an approximate estimation when the Receive Queue can expect the next message from the same peer. The receive Queue should ideally be able to deliver the current message to the Local Server before the next message from the same peer arrives – maintain the Freshness

When we receive a packet, we attach a time stamp (local time of arrival) to that packet and we add the packet to the Queue. Before adding, we get the inter-arrival send time from the packet. If this is less than the current Freshness_threshold, then it becomes the new value for the threshold. We could have maintained a Freshness_threshold for peer participating in the multicast. But that would have resulted in scalability problems since then we need to maintain a threshold for each peer. Hence to be on the safer side, we take the minimum of these thresholds as our Freshness_threshold.

We now need to check if there is any sequence (a sequence corresponds to a list of packets) of packets that has a packet that has been there in the queue for more than Freshnes_threshold amount of time. If yes, then we deliver the whole queue irrespective of the ordering constraint (staleness). If multiple sequences match this criterion, we deliver the sequences based on the decreasing order of the time that the head of each sequence has spent in the queue. (i.e. a sequence that has a head which spend more time in the queue is delivered first.)

On delivery the LAST_PACKET_SENT is reset to the TAIL of the sequence sent.

C. Bound on Freshness

Total end-to-end delay (Dt) encountered by a packet from the time it was generated by a *Local Client i* and received by a *Peer c*'s local server (LS_c) is given by:

$$Dt_c = Dt_{ai} + Dt_{ni} + Dt_q$$

It consist up of thee components, namely Dt_{ai}, Dt_{ni}, and Dt_q. We describe each one of them below.

Dt_{ai} : Time taken by a *Local Client i*(LC_i) to send the packet to the Local *Multicast Server i* (LMS_i) after being generate locally. This can be nonzero in the case when we are trying to aggregate packets based on network congestion as explained later in the document. Currently, we assume that this is zero.
Hence,
$Dt_{ai} = 0$

Dt_{ni} : Time taken for a packet that was sent by *Peer i*'s multicast server to reach LMS_c. This includes the time that the packet took to find its predecessor in the Arrow Network via the FIND Arrow message and once it found the predecessor

info to finally reach the LMS_c via the FLOOD message. If T_{max} is the time taken by an Arrow packet to move between the farthest two nodes in the Arrow Spanning tree, then:

$$Dt_{ni} = Dt_{FIND\ PHASE(i)} + Dt_{FLOOD\ PHASE(i)}$$
$$\leq 2T_{max}$$

Dt_q: This is the time that the packet will be spending in the Queue once it has been received by the LMS_c before its delivery to the LS_c. The maximum limit on Dt_q is given by *MAX_STORAGE_TIME*, which come out to be *FLUSH_TIMEOUT* + δ

δ comes into picture since it takes finite time for the packet scheduler to come back to the queue and check for the packets that are getting stale. But as was seen from the experiments, there were only few cases where δ was significant compared to the FLUSH_TIMEOUT. We ignore those cases here for simplicity of our analysis. Hence a packet after being added to the queue can be *stored* in the Queue for a time varying between 0 and *FLUSH_TIMEOUT* seconds.
i.e.,
$Dt_q \leq$ FLUSH_TIMEOUT
Hence,

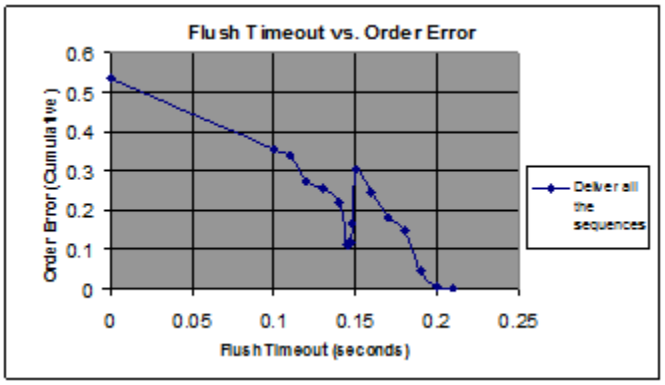
$$Dt_c = Dt_{ai} + Dt_{ni} + Dt_q$$
$$\leq 0 + 2T_{max} + \text{ FLUSH_TIMEOUT}$$

As already mentioned, Dt_c is defined as the *Freshness* of the packet. T_{max} is a constant and can be found at run time. FLUSH_TIMEOUT is user specific parameter that can be set. From the current 3 node Arrow network, T_{max} came out to be about 100 msec. I.e.

$Dt_c \leq 0 + 0.200 +$ FLUSH_TIMEOUT seconds

Hence by setting FLUSH_TIMEOUT to a preset value, we can guarantee that any packet generated in the network will be delivered in less than 0.2 + FLUSH_TIMEOUT seconds.

D. Freshness Consideration With Ordering Guarantees



With the current measurement of ordering via the ORDER_ERROR, is it possible to maintain certain Freshness while maintaining certain minimal ordering? I.e can we guarantee a given maximum ORDER_ERROR while maintaining a certain amount of freshness? It is quite evident that we will need a new instantaneous version of ORDER_ERROR. We need this to keep track of the ORDER_ERROR at a particular time t so that we can keep a check on the number of packets that needs to be delivered to the local server unordered. The basic idea is that since there will be only a small number of packets in the queue that needs to be delivered to the server to fulfill the minimum freshness condition, we don't need to flush the whole queue. Instead we deliver only that number of packets unordered that maintains the instantaneous order error to the guaranteed order error rate. If we can't satisfy both the ordering and freshness guarantees, we may need to make a compromise on one depending upon the application in hand.

In the current implementation setup for ORDER_ERROR calculation, it is not feasible to introduce instantaneous order error rate, since order error is only available when we need to flush the queue based on a criteria (freshness guarantee). At other times, packets may be added to the queue but we can't assume that they will constitute the unordered packets until we encounter a packet that is stale enough to trigger a flush operation.

E. Redefining Order Error

From the days of Ludwig Boltzmann, disorder in a system has been associated with Entropy. J. Willard Gibbs, the nineteenth century American theoretical physicist, called it "mixedupness." The American Heritage Dictionary gives as the second definition of entropy, "a measure of disorder or randomness in a closed system." Austrian physicist Ludwig Boltzmann was responsible to give Entropy its seconds meaning of disorder, which we will use to measure disorder in our system. In our context, the disorder signifies the total number of packets that are out of sequence. Hence a packet that is added to a new list in the queue will account for more disorder (It will increase Entropy) compared to a packet that gets added directly inside an existing list.
Effectively we define Entropy as the amount of work that the Delivery mechanism (constituting the queue, control logic and the packet scheduler) needs to do to order the packets in the right sequence and deliver them to the local server. Hence anything that makes the task of the Delivery mechanism hard add to the increase in Entropy. Since Entropy is based on the current state of the queue at any time, it can be calculated when needed without any wait. This makes it possible to control the order error based on entropy magnitude by only flushing that many packets that are required to maintain the guaranteed level of ordering (which itself will be specified as an Entropy value).

Predecessor window for a packet is the number of predecessor that packet has information about. Hence for Arrow multicast, since each packet knows only about one packet that should be delivered before, the predecessor window will be 1.

Implementation

We used 10 PCs running Windows XP Professional with 1GHz processor each. The machines were connected through a high speed LAN connection (100Mbps). Initial Phase of the experiment was done using 3 machines – a PC and two laptops.

A. Starting the Arrow Network

We store the Arrow network topology locally in each node. Currently, we use a static network of nodes. Ideally for best performance, the network should form a spanning tree. This ensures that each FIND and later FLOOD message cover the minimal distance to reach their destination. Each entry in the network configuration file consists up of the network connections that each nod has with other nodes in the network, direction of arrow for each link and the initial value. We arbitrarily choose a node that will have the initial value.

B. Bluffing the Local Server

The local server needs to know what all are in the games (and hence participating in the Arrow Multicast) during the start of the system. We do this by asking each peer to register itself with every other peer's Local Server. We use the Quake's inbuilt *Connect* network procedure call to do that. Each Client explicitly sends a *Connect* (*GET_CHALLENGE*) request to every other peer in participating in the Arrow Network. The local server from each of the peers responds with a *CHALLENGE*. The local client then accepts this *CHALLENGE* and sends it back in a *CCONNECT* message. On receiving the CCONNECT, the Server registers the peer as being connected and returns a S2C_CONNECTION as a final acknowledgement.
Now the client instead of proceeding with sending messages to that server drops the connection *without notifying* the server. This at the local client sides makes way for proceeding with *buff connection* with other Remote Servers. Since the connection procedure uses *Reliable UDP*, there is no state information maintained at the local client about the *bluff connections*. Once the local client has sent a *Connect* request to all remote local clients, it can connect in a normal fashion to its local server. See *Figure 6*.

C. Node to Node connection

Each node is connected to at most three more peers using TCP sockets. We restrict the number of simultaneous connection to three to make sure that we don't overload the machines.

D. Multicast Tree Maintanence

Although multicast tree maintenance in an interesting subject in itself, we currently use a fixed set of nodes forming a spanning tree. We later intend to use a centralized server (T-Server) to maintain the spanning tree structure. The spanning tree can be updated periodically by the T-Server. We can have multiple T-Servers (One for each LAN), so that the spanning tree formed does take into picture the 'true' distance between nodes. The next phase will be to distribute the functionality of the T-servers and merge it with each peer.

E. Peer-to-Peer Quake Prototype

The prototype broadly consists up of two main threads – the Listener Thread and the Main Thread. The Main Thread contains most of the Quake Engine functionality while the Listener thread contains the multicast primitives. The Main Thread gets movements commands from the user, encapsulates them into an Arrow Packet and forwards it to the Multicast Server that sends it to the Arrow network (if required). The Listener thread reads the Arrow packets via the TCP sockets using a blocking *select* call. The packets, which can be either a FIND or a FLOOD is either consumed by the peer or forwarded via the Multicast Server in the direction of the arrow (in case of a FIND) to all live connections except the one that we receive the packet from (in case of a FLOOD). If a FLOOD is received, it is consumed locally before forwarding.

F. Issues

1) LAN verses WAN

Although running our experiments on a Wide Area Network (WAN) would have been ideal, we decided to focus on a Local Area Network (LAN) environment for these preliminary results. LAN is reliable and the connection latency is minimal. On the other hand WAN tend to be unreliable and slow. Reliability due to message loss outside network is not an issue here since we are using TCP/IP for the connections between the peers. Hence it will not matter if we are on a LAN or a WAN. Network speed and bandwidth can vary a lot on a WAN. We still need to run our experiment on the WAN to see what actually happen.

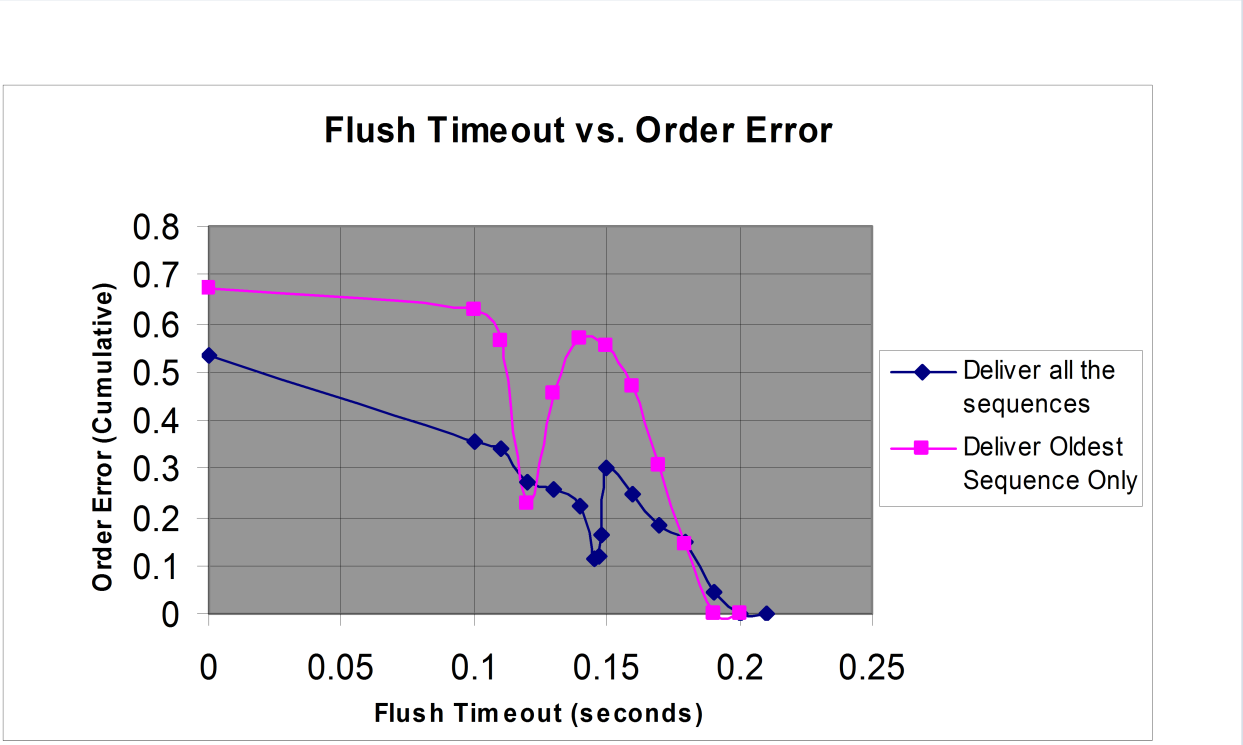
2) Message Data Rate and TCP Buffering

One of the interesting thing that was noticeable was the concatenation of packets due to the high incoming message receive rate. The Listener thread was not able to keep up with the incoming messages, hence the FIND/FLOOD messages coming from other peers used to concatenate in TCP buffers. This was due to the Main Send thread taking control of the processor to send packets out. Once the Listener thread got hold of the processor again, it got chunk of packets from the TCP buffer and process it sequentially. We found that for a network of 6 active nodes, the incoming TCP packet chunk was of size 8230 bytes. This is the TCP payload after removing the network related headers. With each Arrow packet consuming about 85 bytes, the total number of Arrow packets that the Listener thread received was about 97 messages

(8230/85). This meant that we received about 97 messages (which can be either FIND or FLOOD) and didn't send response for anyone of them at the time of receiving. Figure 8 shows such a scenario consisting up of 3 peers - P156, P137 and P16. Initially arrow between P16 and P137 points towards P137. The arrow between P137 and P156 is towards P156. P137 sends a FIND message towards the arrow to P156. P156 receive it and since the Listener thread was busy, it is buffered. Hence the arrow between P137 and P156 remains pointed towards P156. This lures P137 to send a FIND message to P156, which due to the first FIND should not have the value. But our protocol behaved well even under this condition and didn't go into an unstable state. When the first FIND is eventually processed, the arrow is made to point towards P137. Hence the second FIND will be eventually be forwarded back to P137 once it is processed by P156.

System evaluation

A. Order Error and Freshness



Variation of order error

Freshness can be varied by varying the FLUSH_TIMEOUT. We did multiple trace runs, each time using a different value of FLUSH_TIMEOUT. At the end of each run, we recorded the cumulative order error. Figure 10 gives shows the variation in order error as a function of flush timeout. X-axis corresponds to the FLUSH_TIMEOUT chosen for a particular trace run. Y-axis gives the cumulative order error obtained at the end of the run. As can be seen, as we increase the FLUSH_TIMEOUT, the order error decreases. After a FLUSH_TIMEOUT of 0.2 seconds, the ORDER_ERROR is zero. This shows that a time interval of 0.2 seconds is large enough to fully order the packets without the need for sending any packet unordered. The interesting thing to note is that there is sharp fall in order error at a FLUSH_TIMEOUT of

0.147 seconds. After this point the order error again jumps up and continues to decrease at a normal pace. It looks like that this is related to the rate at which packets are being received by the packet scheduler. If the packet receive rate becomes equal to the average rate at which the queue/packet scheduler mechanism is delivering packets to the Local server, then we reach a point where the order error is very low. i.e. at this point on an average most of the packets that come in are being periodically being delivered leaving the queue empty. Since now the chances of a stale packet are being left in the queue and then later on interfering with the LAST_PACKET_SENT becomes very low, the order error tends to be minimal. When we go left or right of this point there is mismatch between, the two rates and hence, order error maintain its gradual downward slope. When we decrease the FLUSH_TIMEOUT, we effectively reduce the chances for a packet to find its predecessor. This will be true since we are delivering the packet without waiting for new packets to arrive (here the instantaneous incoming packet rate tend to be higher that the outgoing delivery rate). Hence there are more unordered packets resulting in the increase in order error. On the other side, if we increase the FLUSH_TIMEOUT, the rate of incoming packets is more that the rate of outgoing packets. This results in the packets getting accumulated in the Queue. Hence our order error rises initially and then comes back to track of decreasing with a gradual slope.

B.

[Image missing from Blogger export: original authoring-file reference.]

Subjective User Experience and Order Error / Freshness Variation

We did multiple trace runs, where in each case we varied the FLUSH_TIMEOUT as before. In each trace run, we logged the subjective user experience. Goodness of game play cannot be measured quantitatively, but we can always get an expressive view of a game. We can let a user play the game and log the subjective user experience. This is what we did and is shown in Figure 11. It is clearly evident from the table that when the FLUSH_TIMEOUT was low (i.e. the order error was high), the user saw lots of false moves in the game. On the other hand when the FLUSH_TIIMMEOUT was relatively high (ORDER_ERROR was low, but freshness was high), the user experienced jitter in the game. Hence we should avoid these two extremes and try to have a compromise between the two. The point where we got a sharp decline in order error earlier seems like a good point to operate the game.

C. Queue Delivery

We experimented with two methods for how to deliver the Queue contents when there is a FLUSH_TIMEOUT. In the first set of experiments, we delivered all the sequences in the Queue. In the second set, when a FLUSH_TIMEOUT occur, we only deliver the sequence that contains the packet that was responsible for the FLUSH_TIMEOUT. The rest of the sequences in the queue remain stored. This is show in figure 12. Intuitively, one may think that the case when we delivered only one sequence unordered, the order error should be lower. But that was not what we saw in the experiments. We saw that the case in which we deliver all the sequences in the queue performed better that the case where we were delivering only one sequence. The graph characteristics were the same in both the cases and there was a sharp drop in order error at FLUSH_TIMEOUT = 0.12 seconds for the case when we delivered only the stale sequence.

It is interesting to see the reason behind the unprecedented results. When we were delivering just one of the sequences, it was quite possible that we were leaving some packets in the queue that were old but arrived late hence they were not yet stale. Now when they eventually got stale, they were forced out of the queue to be delivered to the local server. When this

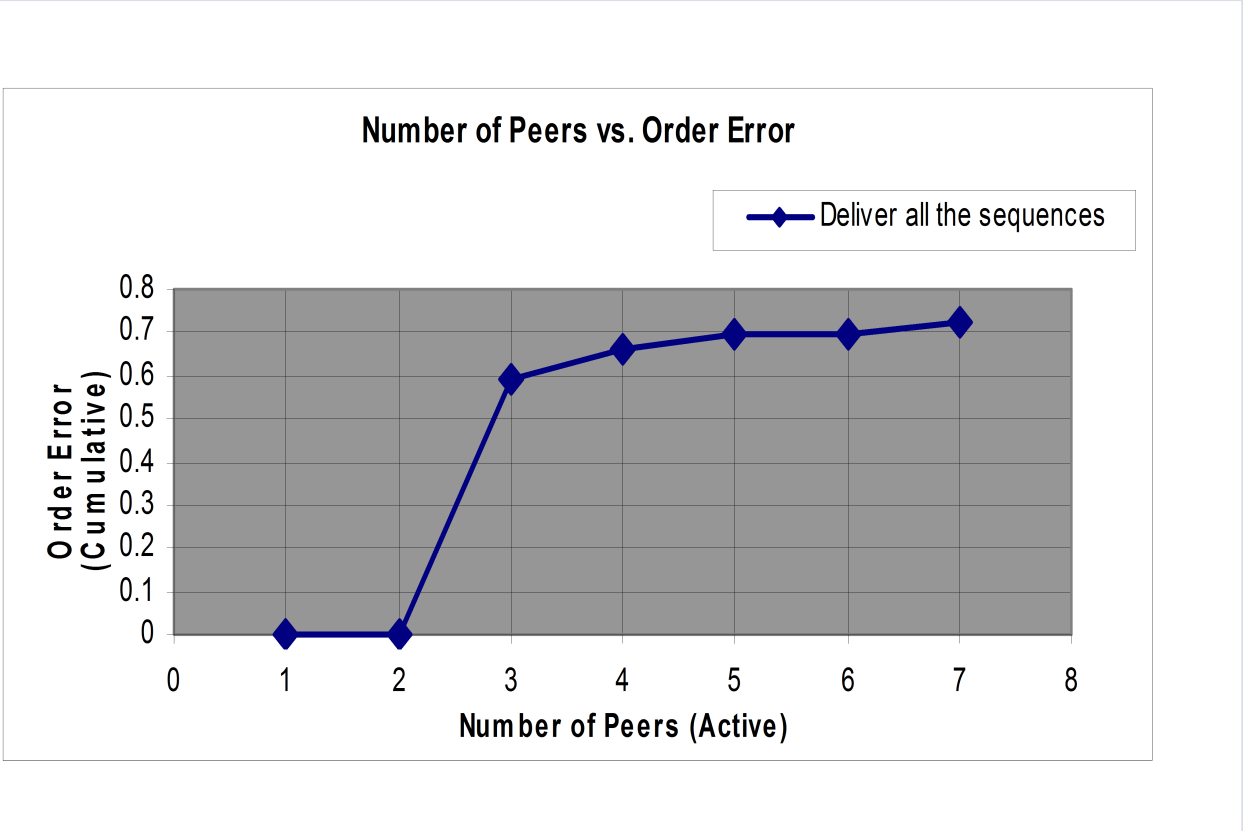
happened, the LAST_PACKET_SENT was set to a value corresponding to one of these old packets. This resulted in the new packets that came later to be added to the queue instead of being sent directly out to the local server. Hence the order error was increased.

Another interesting fact that can be seen from figure 12 is that the local minima in the case where we delivered just one sequence was attained earlier (at FLUSH_TIMEOUT = 0.12 sec) that the case when we delivered all the packets in the queue (FLUSH_TIMEOUT = 0.147 secs). This is related to the fact that when we deliver only one queue, on an average we are delivery less number of packets. Hence the instantaneous rate of packet deliver is less that when we deliver all the sequences in the queue. Hence to match with the incoming packet rate, we need to increase the frequency of packet delivery. I.e. we choose a lower value of FLUSH_TIMEOUT.

D.

[Image missing from Blogger export: original authoring-file reference.]

Varying The Number of Active Peers



Variation of order error by varying the number of active peers

To see if our system was scalable or not, we varied the number of active peer from 1 through 7. In each trace run, we took different number of peers while keeping the FLUSH_TIMOUT to a constant value of 0.15 secs. As shown in figure 13, the order error was zero for the case when the number of active peers was 1 and 2. From 3 onwards, the order error was relatively constant. It showed a variation of only about 0.1.

It is easy to image the theoretical reasoning behind the graph characteristics. FLUSH_TIMEOUT of 0.15 seconds was enough for the all packets to get ordered when the number of peers (active) was 1 and 2. For 3 onwards, the unordered packets increased since the time interval of 0.15 seconds was not sufficient for perfect ordering. But the number of unordered packets remained about the same. If we decrease the FLUSH_TIMEOUT to a lower value, it is quite possible that even at 2 active peers, the order error is non-zero. Hence the system order error is not dependent on the number of peers, but a system parameter (FLUSH_TIMEOUT). Hence our system should be able to scale.

E. State Divergence

This plays an important role in the over all design of the system. We can classify divergence into two main categories:

- Trivial Divergence: In this case the peers may have game state that varies to some extent, but still the game state passes the consistency checks by local server. Hence the game can itself correct and converge to the correct state.
- Non-Trivial Divergence: Here the game state between peers varies to the extent that the logical servers in peers reject subsequent game frames. In this case the server will disconnect the local/remote client.

Quake is robust enough to take care of trivial divergence. If few of the packets are lost, Quake game engine is able to correct itself. This happens transparently. This makes it difficult for us to detect trivial divergence. Non-trivial divergence can be easily detected by checking if the server disconnects or not. If any of the servers (in the peers) disconnects anyone of the clients that are registered with them, we can conclude that we are encountering a non-trivial divergence. Hence to evaluate the performance of our multicast architecture, one can:

- Check if the Local Servers at each Peer starts discarding frames since they are coming un-ordered and hence disconnects the Local/remote Client. Related directly to the timeout parameter.
- Check the consistency of the Environment – check if the scores at each Peer matches, position of each user entity is same on each machine.
- Check when the user tries to moves in the environment and interacts with the objects, he does not encounter unexpected results. E.g. he tried to go forward and the Quake Game Engine moves him backwards crashing him into the lava pit. This to some extent gives us an indication of non-trivial divergence.

For the first check, we saw that with a timeout of about 100msec, the Local Server never disconnected any of the clients (local as well as remote). This timeout value was derived directly from the End-to-End Message Delay that was obtained from the trace runs. We were not able to perform the second check since we were having some problems with the OpenGL graphic rendering engine of Quake which some how managed to just show the local user and not the other player in the game irrespective of the fact that the Local Server was executing commands for other remote peers too (confirmed by trace dump from the Local Server). The Last check was also successful and the user was easily able to move around in the Quake Environment without facing significant jitter in the movement.