<u>QoS Issues for Multiplayer Gaming</u>

By Alex Spurling 7/12/04

Introduction

Multiplayer games are becoming large part of today's digital entertainment. As more game players gain access to high-speed internet connections multiplayer games can grow becoming more ambitious than ever before. Multiplayer games can provide a much richer experience than single player games. Human opponents can be much more challenging than artificial opponents. They learn and adapt and can be much more rewarding to defeat. Collaborative gameplay can be equally compelling with humans when using teamwork to achieve a common goal.

Early multiplayer games used to be restricted to having two players on the same machine. This could be done with either sharing the same input device such as a keyboard or each player might have their own joystick or joypad. The game would accept input from both players and display the game state on a single screen. Later local area network games emerged which involved two or more computers communicating across a local network. Each player had their own screen and input device. Finally with the internet a player could play with or against anyone anywhere in the world.

In order for this to be possible there are many issues to deal with in order to achieve a level of quality that is satisfactory to the player. In the past with single-machine games either each player would share the same view of the game or the screen would be split in two. Split-screen games had to deal with rendering two displays at the same time which can be cumbersome and this is rarely seen in modern games. With multiplayer games today the key issue to tackle is synchronisation of the individual game states so that each player will see the same things as the others. We shall see the various methods of dealing with this and other problems.

Different Types of Game

There are two different types of multiplayer games to consider. One is *real-time* games and the other is *turn-based* games.

Real-time games are the most challenging when it comes to maintaining a decent quality of service for the players. The players will often have a lot more information to transmit between each other than in a turn-based game. For example in a multiplayer game of chess the only data that needs to be transmitted could be the current player's turn and the square the current player has moved to. Also this information does not need to be periodically updated – data would only be transmitted after each player's turn. Real-time games however can produce very large amounts of data all of which would be needed to keep the game states in synch. For example in a car racing game, the position of each car would have to be updated frequently. Also in order to give the cars smooth motion you might need to transmit their speed, and the current forces acting on them etc. The main problem with real-time games that makes

them difficult to implement compared with turn-based games is that the data needs to be transmitted repeatedly at very short intervals. This can cause problems when bandwidth is limited.

Multiplayer Game Models

There are two main types of multiplayer game models. The first is the *client-server* model. This consists of a single central server to which all the players (clients) connect. This model is usually simple to implement but can cause problems as there is a single point of failure; if the server crashes all the clients will be disconnected. Also this creates a bottleneck of bandwidth at the server end as all data passes in and out of this single point. The game state is stored at the server which periodically sends update messages to each client.

The second model is the *distributed* game model also known as *Peer-to-peer* or *P2P*. This model does not require a centralised server. Each client communicates with all the other clients and each stores a copy of the game state. Messages are sent between clients to notify them of changes to the game state and the bottleneck of a server is eliminated. This method is more complicated to implement and unless synchronisation is kept between the clients, the game states will diverge over time due to network delays and other factors.

The most popular game architecture by far is the client-server model. There are a number of reasons why this method is so popular. First the networking code is often a lot simpler to write and it can be separated more easily from the main game code. There are no elaborate protocols required as there are in P2P. Often a single player version of a game can be quickly adapted for client-server play with only minor changes. Secondly having a centralized server gives the game publisher more administrative control. Having control over game servers lets publishers perform authentication, copy protection, accounting and billing, and easy update of client code. [1]

There are other hybrid models such as the *mirrored-server* architecture proposed in [2] or the *network server* architecture shown in [3] and [4] which consists of two or more servers connected by a local network to which clients connect to as they would in the client/server model. This allows servers to share information with each other very quickly and opens the possibility of having an arbitrary number of players in a single game world. This type of game is called a Massively Multiplayer Online Game or MMOG and is growing to be hugely popular. In games such as *Everquest* or *Ultima Online* players can jump in and out of a massive persistent world at any time, continually developing their character and interacting with other players in the game. This creates perhaps the greatest challenges to QoS in this type of game. [3, 5]

We will concentrate on the client-server model and the basic QoS issues to begin with.

Synchronisation

The biggest concern when making multiplayer games is game state synchronisation. This is the problem of maintaining the same game state information on each of the

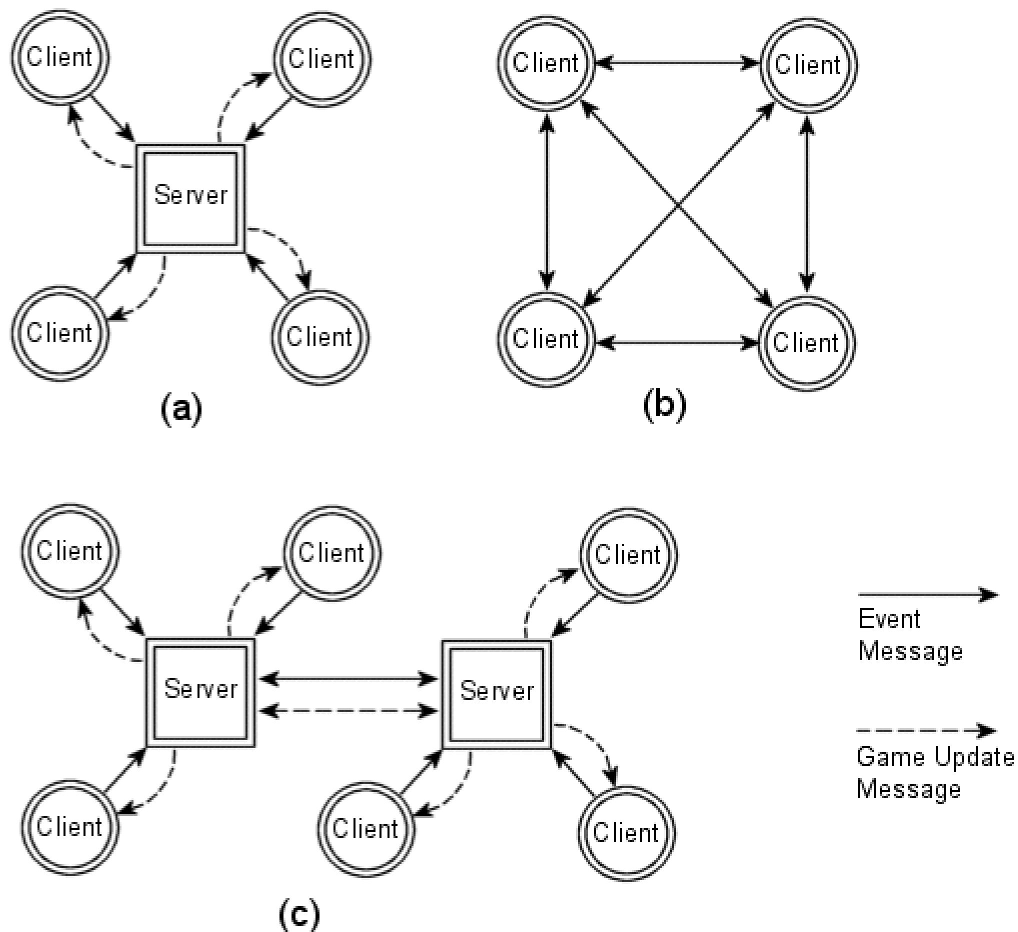

Figure 1. Multiplayer Game Models: (a) The client-server model, (b) The peer-to-peer model, (c) The network server model.

players' instance of the game and generating the effect of each player belonging to the same game instance. This problem is very important to the QoS of a multiplayer game. For example in a racing game the exact positions of all the players' cars must be represented on the players' screens at the same time. If there was a delay in the updating of the position of the cars then two players might think they were in first position at the same time, which of course will cause problems when they cross the finish line.

Another aspect of synchronisation to consider is state of random values in the game. For example in a street racing game the players might be racing with traffic on the streets. The position of the cars would be chosen randomly but the traffic must still appear in the same place for each player. In the client-server model this decision can be made by the server and then updated for each player but in a peer-to-peer design this kind of problem takes more consideration.

There are two main methods for maintaining synchronisation: *state synchronisation* and *input synchronisation*. [6]

State Synchronisation

State synchronisation involves each player sending the state of their instance of the game to all other players. For example in a multiplayer *Asteroids* game the player would send the current speed, position and orientation of their ship and of all the asteroids. This method is fairly robust as there is no possibility of information loss and the game state will not diverge to become totally different on different instances of the game. However there can be problems when dealing with larger games such as an online role playing game. In this case you may need to send much more information about the game world and this information would be constantly changing.

Input Synchronisation

Input synchronisation is where each player sends all their game events to the other players. For example in a racing game, events might be sent when the player accelerates, turns and brakes. This approach does not involve sending large quantities of data but it is prone to game state divergence. For example if the event to start turning left was given for a particular player to other players a delay for this event might mean the car started turning a few hundred milliseconds after the actual time. This could mean the difference between narrowly missing and hitting an obstacle.

A Hybrid Solution

A more practical approach to the synchronisation problem might be to use a combination of state and input synchronisation. For example you could use input synchronisation for aspects that are not time critical such as initiating a conversation between two players and you could use state synchronisation for random aspects such as the position of traffic on the road. The whole state of the game does not need to be transmitted but only aspects that have changed or those that are of interest to a particular player. For example in a massively multiplayer role playing game a player does not need to know about the position of monsters that another player hundreds of metres away is fighting. This is called *Interest Management* and is described in detail in [10].

There is no real general solution when designing a multiplayer game. Different techniques need to be combined for use in different games. Next we will look at some technical problems that can affect quality of service in multiplayer games.

Latency

Network *latency* or *lag* refers to the time taken for a packet sent to a to travel from its source to its final destination. It is never impossible to eliminate lag completely, for example a signal travelling along a fibre-optic cable is still limited by the speed of light which represents about 15ms to traverse the Atlantic. In fact for 2004 the global network communications company *MCI* specify an average roundtrip time of 79.30ms for a packet from any nodes on its network in London to any node in New York. [7]

There are a number of factors that will increase the delay of a packet such as the time taken to route and queue packets as well as processing packets for transit across

varying types of network. For example IPX packets may need to be encapsulated into TCP/IP packets. The acceptable length of delay of transmission depends a great deal on the type of game. Some games such as First-Person Shooters (FPS) are very delay sensitive and others such as turn-based games will have a very high tolerance to latency.

There have been many studies on the acceptable amount of lag for various types of game. Pantel and Wolf [8] demonstrate that a delay of more than 100ms in a car racing game begins to affect the gameplay. Armitage G.J. [9] shows how players of *Quake 3* prefer to select a server with a lag below 150-180ms. They also showed how players who played with a lag of 45ms on average achieved 1 point per minute more than players with a lag of 200ms. The acceptable level of lag tends to be quite subjective. Some people can learn to anticipate the delays caused by a high lag and adjust their aiming accordingly. Others find delays more than 100ms unacceptable.

Jitter

Network *jitter* refers to the variation in lag of a given connection. This factor is almost as important to QoS of a multiplayer game as lag. It can be caused by the way a particular router queues packets or high network traffic. Different packets from the same connection might take different routes if traffic is high and therefore the transit times of packets from source to destination will differ. A method of attempting to remove jitter that can be employed by routers is *fixed playout delay*. A timestamp is placed on all packets from the source. When they reach a router they might be in any particular order and arrive at different times. The router will then wait for a specific fixed time and then forward the packets in the correct sequence based on their timestamp and also with the correct delay between the packets. This method is a trade-off between delay and loss of packets. Another method is *adaptive playout delay*. This is where the jitter of the packets is estimated and the delay between receiving a packet and forwarding it is adjusted to reflect the jitter.

TCP vs. UDP

The internet is based on two main protocols *TCP* and *UDP*. An issue that often arises when designing a multiplayer game is which protocol to use as a transport method across the network. Each has their advantages and disadvantages.

TCP is a reliable protocol which means that if a packet sent with TCP is lost (for example if it was misrouted or if a particular router was too busy) then that packet will be retransmitted. Each packet from the sending device to the receiving device has to be accompanied by an acknowledgement packet from the receiving device to the sending device. This means that TCP will use much more bandwidth than is necessary. UDP does not have any form of reliability checks. Lost packets are not acknowledged or retransmitted. However it uses much less bandwidth than TCP. UDP is connectionless and involves small overhead whereas TCP is connection oriented and uses more overhead. The header sizes of UDP and TCP are 28 and 40 bytes respectively [10].

Most real-time multiplayer games today will use UDP over TCP. UDP allows the transmission of large amounts of synchronisation data and this data is usually time-

sensitive so the extra delays of TCP would degrade quality of service. Reliability of the data is not crucial in multiplayer games as the information will often be re-sent after a short time. It would not be worth re-requesting lost information only for it to arrive out-of-date. It is better to simply wait for the next arrival of data. There are methods of dealing with lost information which we will discuss later. Some games might use TCP for certain transmissions such as player-to-player chat and UDP for time-sensitive data.

Packet Compression and Aggregation

One method of reducing bandwidth use is to compress the data being transmitted. There are two methods of doing this. *Internal compression* is where each individual packet is compressed without reference to other previously sent packets. *External compression* takes place before the data is split into packets. The former method is more suited to UDP where packets might be lost or arrive in the wrong order. The latter method is better when used with a TCP connection and allows better compression as it can observe redundancy over a larger section of data.

Aggregation is the process of merging the data of one or more packets into a single packet. This reduces the overhead of packet headers.

Both of these techniques decrease bandwidth requirements but they increase delay so these factors must be balanced.

Dead Reckoning

Dead reckoning is a process of approximating the value of missing data. The data may be missing due to lost UDP packets or if the bandwidth of the particular internet connection is not large enough. Information is approximated based on the value of previously received information. The process consists of two parts the *prediction technique* and the *convergence technique*. [10]

An example of a use of dead reckoning might be in a car racing game. If a packet was lost containing the position of one of the player's cars, the game could guess the current position by assuming that the speed of the car has not changed. In the next update of the car's speed and position the values might be slightly different from the predicted values. In this case, rather than instantly re-position the car in the correct place the game could interpolate the car's position between the predicted and the correct position. This convergence technique attempts to avoid jerky movement of the cars.

Multicast and MiMaze

Multicast is a feature of networks which allows the same packet to be distributed to more than one recipient on the network at the same time. The recipients form a *multicast group* and any multicast packets will be received by all the people in this group. This technique avoids having to send the same packet several times to different clients and so reduces bandwidth requirements considerably. This of course can be of great benefit to games where the same information is sent to each of the players. However it is rarely used in games today. One reason is that often ISPs do not support

multicasting across the internet. Also programmers tend to believe that it is difficult to implement. [11]

MiMaze is a project developed by L. Gautier [12] to demonstrate how to create a completely distributed multicast based multiplayer game. It was the first game of this type and used a synchronisation method called bucket synchronisation which guarantees consistency regardless of network delay and it uses a dead-reckoning system to recover packet loss. MiMaze demonstrates the feasibility of a distributed architecture and a reliable method of maintaining synchronisation.

Conclusion

There are many aspects to consider when designing a multiplayer game that requires high QoS. First there are the three main network architectures client-server, P2P and network-server to choose from. Then there are several programming techniques needed to maintain synchronisation of the game state between the players such as event or state synchronisation and dead reckoning. Also there are technical issues such as which protocol to use and methods of compression and aggregation of packets. The best choice of course depends on the type of game and its sensitivity to QoS. There are more details which this paper does not go into but it should help give an overview of the issues that need to be dealt with when designing a multiplayer game.

References

1. Cronin E. et al. 2003. *An Efficient Synchronization Mechanism for Mirrored Game Architectures.*
2. Cronin E. et al. 2001. *A Distributed Multiplayer Game Server System.*
3. Caltagirone S. et al. 2002. *Architecture For A Massively Multiplayer Online Role Playing Game Engine.*
4. Smed J. et al. 2001. *Aspects of Networking in Multiplayer Computer Games.*
5. Fine. R. 2004. *MMOG Considerations.* http://www.gamedev.net/reference/articles/article2100.asp [Accessed 27/11/04]
6. SoftLookup. *Internet Game Programming in Java. Ch17.* http://www.softlookup.com/tutorial/games/index.asp [Accessed 23/11/04]
7. MCI / UUNET. *Latency Statistics*. http://global.mci.com/uunet/be/about/network/latency/ [Accessed 29/11/04]
8. L. Pantel and L. C. Wolf. 2002. *On the impact of delay on real-time multiplayer games.*
9. Armitage G.J. 2003. *An Experimental Estimation of Latency Sensitivity In Multiplayer Quake 3.*
10. Smed J. et. et. 2002. *A review on Networking and Multiplayer Computer Games.*
11. Lukianov. D. 2001. *Advanced Winsock Multiplayer Game Programming: Multicasting* http://www.gamedev.net/reference/articles/article1587.asp [Acessed 4/12/04]
12. Gautier. L. and Diot. C. 1998. *Design and Evaluation of MiMaze, a Multi-Player Game on the Internet.*