Crossroads
The ACM Student Magazine

Game-State Fidelity Across Distributed Interactive Games

by ***Aaron McCoy***, ***Declan Delaney***, and ***Tomas Ward***

Introduction

Distributed interactive games offer players a three dimensional virtual world experience. Within this virtual world, players interact with each other and with their environment in real-time. They experience the same events, but from different viewpoints.

As interactive games have evolved, they have driven the technologies underlying them. The distinguishing feature of any distributed interactive game is the network: the medium by which information is exchanged and shared between participants. The network impacts not only the design and development of distributed interactive games, but also their potential entertainment value.

In this article, four of the fundamental networking issues and their effect on the design and operation of distributed interactive games will be discussed. In addition, a description of the different communication architectures used in distributed interactive games will be provided. Finally, as an illustrative example, these issues will be related

to *Unreal Tournament*, a popular distributed interactive game.

The Dawn Of A New Age

In the not too distant past, distributed real-time virtual environments were science fiction; novels envisioned a world without rules or boundaries, a so-called 'cyberspace', a place where the only limitations were that of the human mind [**6**]. A significant milestone in the history of distributed virtual environments was the Multi User Dungeon (MUD), completed by Rob Trubshaw and Richard Bartle at Essex University, in 1978 [**4**]. MUD allowed people all over the world to interact with each other, share environments, and implement their own environments. More recently, technological advances in processing power and graphics capabilities, coupled with the widespread availability of the Internet, has led to the diffusion of distributed virtual environments within the public domain.

Networked, multi-player games drew mainstream attention with the release of Doom in 1993 [**8**][**10**]. Doom enabled players to compete with each other in both four-player local area network (LAN) games and direct head-to-head modem games. In 1996, the developers of Doom released Quake [**8**][**10**], another milestone in the history of networked multi-player games. Quake was a pioneer of the client-server architecture widely used in modern online games; it introduced players to a larger type of interactive environment than previously available. The popularity networked multi-player games were receiving prompted the development of more complex distributed games [**5**][**9**]. There are now numerous online, dedicated gaming services using high-bandwidth connections that provide server hosting for the most popular games [**5**].

In this article, we will investigate the networking features that must be considered by distributed games developers. We will start by briefly examining the potential network architectures and then investigate four issues that affect the maintenance of game-state fidelity across all participants in a distributed application. We will then investigate these issues as they apply to one of the most popular distributed games, Unreal Tournament (UT) [**3**].

Speak *Friend* And Enter

Game-state fidelity is a measure of the consistency of the game-state among all participants within the game. In this context, the game-state refers to a description of the shared environment and all dynamic objects within this environment at an instant

of time. Distributed interactive games require as much information as possible to be exchanged and processed in real-time in order to maintain a reasonably consistent game-state. One of the issues for game developers to consider is how they want this information to be communicated between participants in the game. There are two main communication architectures in use today: peer-to-peer architectures (**Figure 1**) and client-server architectures (**Figure 2**).

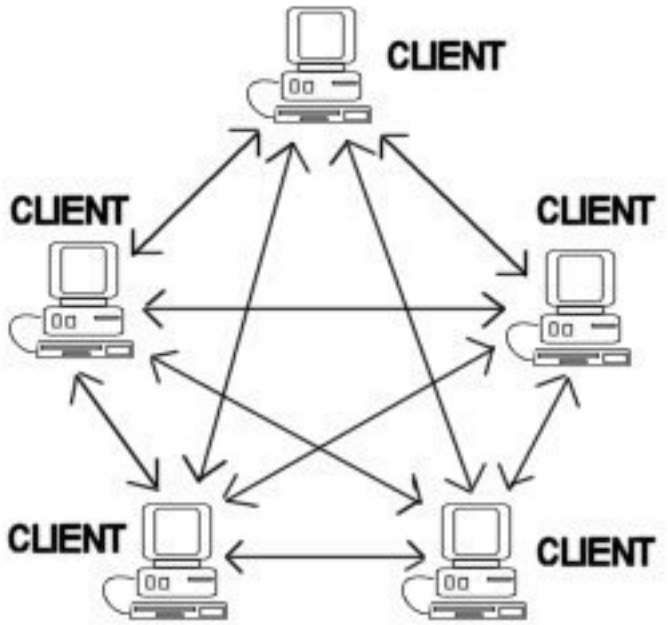

Figure 1: Peer-to-peer Networking Architecture.

With peer-to-peer architectures, communication is sent directly between participants in the game. Players wishing to update other players about their activity, must send a message to all of the other players so that they can then update their game-state information, and hence maintain global game-state fidelity.

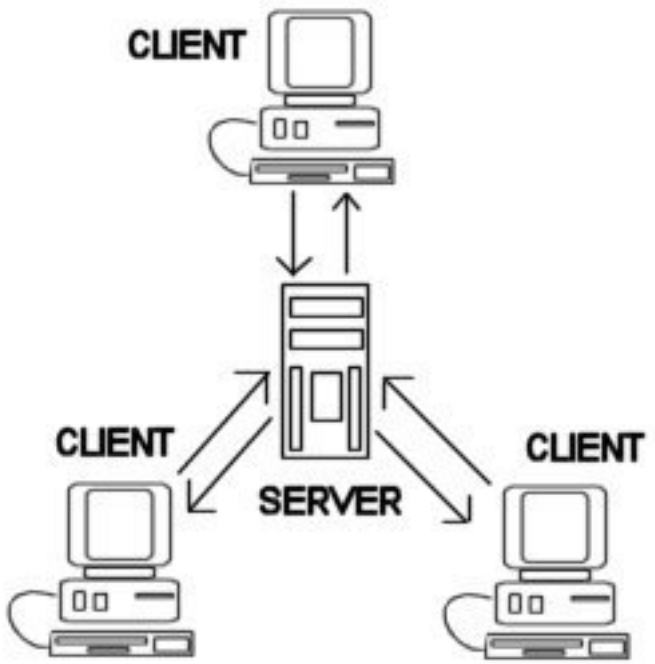

Figure 2: Client-Server Networking Architecture.

With client-server architectures, one machine is designated as the server and is responsible for maintaining the overall game-state and making the game-play decisions. In this respect, the server determines the game-state. The rest of the machines are designated as clients, and they are responsible for rendering a view of the shared virtual world to players and for updating the server with details of any actions that the players are performing. If a player wishes to update other players about his or her activity, the player sends a message to the server advising of the changes, and then the server distributes the information to all of the other clients. Communication is never sent directly between clients, it always passes through the server.

Peer-to-peer architectures offer theoretically better scalability [**12**], but in general, overall game-state fidelity and player interactions are harder to keep consistent because no one player has complete authority over the game-state. In contrast, client-server architectures offer less scalability, but they implicitly provide the ability to maintain accurate game-state fidelity and manage player interactions.

Regardless of the architecture chosen, the network itself raises a number of technical issues that are critical to distributed applications. We will now focus our interest on these.

The Four Horsemen Of The Apocalypse

The Internet poses a wealth of challenges for games developers. In particular, four issues affect data transfer over any network, regardless of the network architecture [**15**].

Network Latency

Network latency is a measure of the time it takes for a packet of information to travel from one computer to another across a network.

Network latency arises for a number of reasons. First, a lower limit is imposed by the finite speed of light, which results in data traveling at about two thirds the speed of light in a vacuum through a fiber optic cable [**2**]. Second, the endpoint computers introduce delays when they process the data [**1**]. Finally, the network introduces delays as the data propagates through network routers before reaching its destination.

Network latency represents one of the greatest challenges to the development of distributed interactive games. As the network latency increases, maintaining game-state fidelity between the game participants becomes more difficult, with each player's view of the shared virtual world becoming increasingly different depending on how up-to-date their information is. Network latency means that game developers must assume that all information received by participants is already out-of-date when it arrives.

Network Bandwidth

Network bandwidth is the rate at which a network can deliver data from a source point to a destination point.

The type of channel used to transport data determines the available bandwidth, and it is also limited by the hardware used to transmit the data [**16**].

The available bandwidth limits the amount of information that can be shared and exchanged between participants per unit time. If a player is connected through a low bandwidth line, which is often the case for a home connection, they will not be able to receive complete information relating to every other participant in the virtual world. As a result, it is up to the game developers to allocate the available bandwidth to networked players. Given these bandwidth limitations, it is the main goal of a distributed interactive game to enable the communication of sufficient game-state

information to enable players to determine events within a reasonable level of accuracy.

Network Reliability

Network reliability is a measure of how much data is lost by the network during the journey from source to destination host.

Network data loss occurs for two main reasons [**15**]. First, data can be lost as it travels along the network transmission channels. This is the most obvious but also least frequent cause of data loss. The most frequent cause of data loss is due to the network routers that transfer the data. If a network router receives too much data for it to handle, it will discard all incoming packets that arrive while the queue is full. This policy is known as drop-tail (DT) [**13**].

In distributed games development, network reliability does not pose as much of a problem as one might first assume, provided that the rate of data loss throughout the game remains low. The reason for this is because dynamically changing information within the game is usually being updated at a very fast rate among participants, so any data that is lost is usually replaced quite quickly.

Network Protocol

A **network protocol** describes the set of 'rules' that two applications use to communicate with each other.

A network protocol consists of three components. First, the **packet format description** allows the endpoints of the communication channel to identify the various individual parts of data that are contained within the information stream. Second, the **packet semantics description** allows the communication endpoints to understand the various individual parts of data. Finally, the **error-handling description** governs how the communication endpoints should respond to various error scenarios that may occur during data transmission.

The basic protocol for Internet transmission is the Internet Protocol (IP) [**7**]. However, applications almost never use IP directly. Instead, they use one of the higher-layer transport protocols that are built on top of IP. The two most common transport protocols used in distributed interactive games are the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) [**16**].

TCP offers reliable, connection-oriented, stream-based transport of data. It guarantees the delivery of all data in the correct order by using a system of positive acknowledgement with re-transmission. It can only send data between two directly communicating hosts.

In contrast, UDP offers unreliable, connectionless, packet-based transport of data. Data can be sent from a single host to any number of different hosts without having to set up individual connections, by using either IP broadcasting or multicasting.

Distributed interactive games are real-time systems, and it is this need for real-time information exchange and processing that usually influences the choice of communication protocol. The reliability and ordering guarantees provided by TCP introduce extra overhead. These guarantees are not necessarily required as it is often more important for data to be delivered quickly than it is for it to be delivered reliably. With UDP, distributed interactive games can send out data as soon as it is generated without having to wait to make sure the data is ordered and without having to subsequently wait for an acknowledgement to ensure that the data was delivered. In addition, the ability to broadcast and/or multicast UDP data packets to multiple sources greatly aids the ability for information to be distributed quickly and efficiently.

Ultimately, the choice of protocol depends on the specific requirements of the distributed interactive game. Recently, developers have been using multiple protocols together to provide different levels of service for data transport, with the choice of protocol being determined by the measure of how crucial the data is in maintaining overall game-state fidelity.

So, Just Who Is *In Charge*?

Distributed interactive games target real-time interactivity between participants and between participants and their environment. This interactivity highlights an interesting problem: who determines what events happen, how the events occur, and what the final outcomes of those events are? This is directly related to the problem of maintaining game-state fidelity between participants. Due to out-of-date or incorrect game-state information, it is entirely possible that one or more players will incorrectly conclude that an interaction of some sort took place, when in fact it may not have. This can lead to participants disagreeing about whether an interaction actually occurred. Furthermore, even if all the participants agree that an interaction did take place, they

may not all agree about the specific details of the interaction.

The distributed interactive game must manage these interactions and provide accurate detection and resolution of collisions among participants. Examples of such collisions may include direct player-to-player contact, player-to-environment contact, or perhaps weapon fire/damage. Usually a client-server architecture is adopted, so that the server is *in charge*, and it alone determines the true game state. In classical peer-to-peer architectures, no one client can be considered *in charge* as each client communicates with all other clients, so game-states are determined individually.

Let us now turn our attention to Unreal Tournament (UT) and discover how they have dealt with the four networking issues we have described previously.

Totally *Unreal*

Unreal Tournament's design architecture essentially consists of two parts [**18**]. First, there is the underlying game engine itself. This engine provides most of the game mechanics and the graphics capabilities. It also provides the generalized network code. Secondly, in Unreal Tournament "the 'game state' is self-described by an extensible, object-oriented scripting language which fully decouples the game logic from the network code. The network code is generalized in such a way that it can coordinate any game which can be described by the language" [**18**]. This scripting language is known as UnrealScript, and it provides developers with a built-in, fully object-oriented language with which to program events into the game. UnrealScript is based on a C++/Java variant and contains similarities to both languages. The power and usefulness of UnrealScript lies in its implicit support for game-specific paradigms, such as concepts of time and state within the game environment. Ninety percent of the code that governs game-state dynamics in UT was written in UnrealScript [**14**].

UT's Communication Architecture

UT uses a permissible-client-server architecture to maintain game-state fidelity. This is a standard client-server architecture with one notable exception: whenever a player wishes to perform an action, he/she must first ask permission from the server. Clients cannot perform an action, such as firing a weapon, without getting permission from the server to do so (the one exception to this is movement, which is predicted by each client - this is detailed further below). This ensures that no disagreements can arise between clients as to whether an interaction takes place or not. All events are,

ultimately, determined by the server.

The server is completely authoritative over the flow of play, and in addition gameplay logic (code that evolves the game-state) should only be carried out on the server. This means that the server's game-state can always be regarded as the only true game-state, and the game-states that exist on client machines are approximations to the server's state.

The main advantage of UT's permissible client-server architecture is that it provides a means of minimizing the effect of game-state inconsistency among clients. This is because the server is kept aware of all activities being performed.

The main disadvantage of this architecture is that it introduces extra lag, or delay, in the response time of the game to various client events. For instance, if a player fires a weapon, there will be a noticeable delay between when the player presses the fire button and when the weapon actually fires onscreen. This is because the client machine is waiting for permission from the server before it can fire the weapon. This lag is compounded by the fact that if the player is playing over a connection that has a high latency time the noticeable delay between firing will be increased.

UT's Network Protocols

UT employs a network driver that is layered on top of the UDP protocol. This network driver provides some of the services of TCP by handling point-to-point connections and positive acknowledgements. Hence it replicates at the application layer TCP's mechanisms when reliability is requested, but always sends UDP packets over the network. By utilizing both types of transport protocol, UT gets the best of both worlds. It can send and receive unreliable, packet-based data as well as being able to set up TCP connections and send reliable, stream based data. Data that is critical to maintaining game-state fidelity between participants in the game is sent reliably, to ensure that it is delivered to the destination. Data that is not so critical, and hence can reasonably afford to be lost, is sent unreliably, with no guarantees of receipt at the intended destination.

UT and Network Latency

If UT used the permissible client-server architecture for every action that the player can take, then player movement would be very slow and sluggish. For instance, if the

player was playing on a network connection that had a 200ms round trip time between itself and the server (100ms each way), then after the user pressed the movement key they would not see themselves move onscreen until 200ms later. This would be extremely frustrating and would reduce the enjoyment of the experience for the participant.

To eliminate the above client-movement lag caused by network latency, UT uses a form of client prediction or dead reckoning [**17**] that can best be described as a "lock-step predictor/corrector algorithm " [**18**] (this high-level feature is actually implemented in the UnrealScript language rather then the engine's generalized network code). When a player performs a movement and requests permission from the server to do so, the client machine actually predicts where the player will move to while it is waiting for permission from the server to move. As a result, both the client and the server execute the same move for the player. However, the server has the last say, so that when it is finished executing the requested movement, it sends the result back to the client. In the meantime the player is viewing the client prediction on screen. If this position differs from the server's player position, the client must correct the player's position. It does so by using convergence algorithms to smoothly converge to the true game-state position.

At any point in time, the UT client is predicting ahead of what the server has told it, by an amount of time equal to half it's round-trip latency value. As a result the local player movement doesn't appear to lag. The client movement prediction usually mirrors the client movement determined by the server. Only in rare cases (such as when a player is getting hit by a weapon or bumping into another player) does the player's location need to be corrected by the client.

UT and Network Bandwidth

Bandwidth limitations impose restrictions on the amount of information that can be transmitted between the server and the client. In order for UT to allocate bandwidth resources efficiently and effectively, it utilizes a load-balancing technique that prioritizes actors. Actors are objects capable of interacting with any other objects in the game. Players, software opponents (a.k.a. BOTs), and movable environment objects are all actors. Each actor is assigned a network priority value, indicating how important it is for maintaining game-state fidelity between participants. The available bandwidth is then allocated based on the ratio of network priorities. If, for example, an actor has a priority value that is twice the priority of another actor, it will get updated twice as

often. With this system, the most important information, such as player movements and weapons fire, will be updated more frequently and given higher bandwidth preference than information that is not very important, such as world objects that have little or no effect on game-play.

In order to complement the above-mentioned load-balancing technique, UT also utilizes a technique known as relevant sets. At any one time, a player will only ever be interacting with a small subset of all actors within a game environment. In addition to prioritizing each individual game actor based on how important they are to game-play, UT also prioritizes actors based on how important they are to individual players. The server calculates the set of actors that it deems are relevant to or capable of affecting each client and stores them in a relevant set. Using this system, clients will not receive redundant information from the server about actors that are of no consequence or relevance to it. This is a form of relevance filtering [**15**].

UT and Network Reliability

As already stated above, UT uses both the UDP transport protocol and a form of the TCP transport protocol provided for by it's own network driver. The decision as to which game-state information should be sent reliably and which information should be sent unreliably is left to the sole discretion of the game developers through the high-level UnrealScript scripting language. By providing both types of protocol, UT ensures that the influence of network reliability on the game-state fidelity is kept to a minimum. Critical information is sent reliably, ensuring that if the information is lost it will be re-transmitted.

Conclusion

This article has explored some of the more important networking issues that relate to the development of distributed interactive games, and it has provided descriptions of how they may affect the operation of such games. For the interested reader, an additional case study is available detailing the development of a game called X-Wing vs. Tie-fighter, and how the designers of that game coped with the networking issues described above [**11**]. Developing distributed real-time games for use over the Internet is difficult. Bandwidth, latency and reliability vary tremendously in such a heterogeneous network. Game developers have absolutely no direct control over the limitations imposed by the Internet. All they can do is react to these limitations and work to provide the best software solution possible. It is a testament to their talent, creativity, and ingenuity that games such as Unreal Tournament exist.

References

1

Blow, J. A look at latency in networked games. In *Game Developer Magazine.* July 1998.

2

Cheshire, S. *It's the Latency, Stupid.* <**http://rescomp.stanford.edu/~cheshire/rants/Latency.html**>.

3

Unreal Tournament is copyright (c) 1999 - 2002 Epic Games Inc, <**http://www.epicgames.com**>.

4

Friedl, M. *Online Game Interactivity Theory.* Charles River Media Inc, 2003.

5

Gamespy Arcade. <**http://www.gamespyarcade.com**>.

6

Gibson, W. *Neuromancer.* Ace Books, New York, 1985.

7

Gulati, S. *The Internet Protocol Part One: The Foundations.* <**http://www.acm.org/crossroads/columns/connector/july2000.html**>.

8

Kushner, D. The Wizardry of id. *IEEE Spectrum* Vol.39 No.8, August 2002.

9

Huebner, D. Industry Watch - The Buzz about the Game Biz. In *Game Developer Magazine.* June 2001.

10

Doom and Quake are copyright (c) id Software, Inc. <**http://www.idsoftware.com**>.

11

X-Wing vs. Tie-fighter is copyright (c) LucasArts Entertainment Company LLC. <**http://www.gamasutra.com/features/19990903/lincroft_01.htm**>.

12

Macedonia, M. R., Zyda, M. J., Pratt, D. R., Brutzmann, D. P., and Barham, P. T. Exploiting reality with multicast groups. In *IEEE Computer Graphics and Applications* Vol.15 No.5, September 1995, pp. 38-45.

13

Pentikousis, K. *Active Queue Management.* <**http://www.acm.org/crossroads/columns/connector/july2001.html**>.

14

Reinhart, B. Epic Games Unreal Tournament. In *Game Developer Magazine.* May 2000.

15

Singhal, S., and Zyda, M. J. *Networked Virtual Environments: Design and Implementation.* ISBN 0-201-32557-8, Addison-Wesley and ACM Press, 1999.

16

Stallings, W. *Data and Computer Communications.* Fifth Edition, Prentice Hall, Upper Saddle River N.J., 1996.

17

Standard for Distributed Interactive Simulation - Application Protocols. *IEEE* 1278.1, 1995.

18

The Unreal Technology Webpage. <**http://unreal.epicgames.com**>.

Biographies

Aaron McCoy (**amccoy@eeng.may.ie**) received his B.Sc. degree from the National University of Ireland, Maynooth in 2002, specializing in computer science and theoretical physics. His main areas of interest are distributed systems, networked virtual environments and the use of artificial intelligence in interactive games.

Declan Delaney (**decland@cs.may.ie**) received his B.E. and M.Eng.Sc. degrees from University College Dublin, Ireland in 1991 and 1998 respectively. After working in industry for eight years he is currently pursuing a doctorate in network latency masking techniques.

Tomas Ward (**tomas.ward@eeng.may.ie**) received his B.E., M.Eng.Sc. and Ph.D degrees from University College Dublin, Ireland in 1994, 1996 and 1999 respectively. From 1999-2000 he taught at the Department of Computer Science at the National University of Ireland, Maynooth. Currently he teaches in the Department of Electronic Engineering at NUI Maynooth. His research interests lie in the areas of human-computer interaction and distributed interactive systems.