Small group multiplayer games

Markus Sabadello, April/May 2001
Vienna University of Technology, Austria

ABSTRACT

Multiplayer capabilites are an important part of almost all computer games developed today. No matter what kind of game, playing against real human opponents is simply more challenging and motivating.
However, the realization of such games requires communication between players, and programmers have to deal with some limitations caused by the underlying network. Depending on the kind of game, different measures can be taken to address this problem. This paper will give an overview of such approaches and of communication architectures in small group multiplayer games.

1 INTRODUCTION

Computer gaming has been evolving for a long time now, almost since software development itself began. Today, games are popular among a broad range of users and offer more features and possibilities than ever before, like amazing 3D graphics and advanced artificial intelligence. Also, every current computer game worth mentionning offers some kind of multiplayer mode, giving players the possibility to compete against each other in so-called virtual environments. The goal of such environments is to give players the impression of interacting in a common virtual world, this can range from playing a simple chess game to fighting high-speed battles in science-fiction like spaceships.
In contrast to massive multiplayer games, which can handle several hundred or more players, small group multiplayer games are limited to about 16-32 players and usually require all players to be present before a game can start (examples include strategy games, 3D action games, driving simulations and more). When designing small group multiplayer games, several networking issues have to be taken into account. Section 2 of this paper will introduce some communication architectures, including their advantages and disadvantages. In section 3, the purpose and role of matchmaking services is described. Section 4 focuses on the most important problem of networked virtual environments, the management of dynamic shared state. Finally in section 5, conclusions are drawn.

2 COMMUNICATION ARCHITECTURES

This section deals with how networked players actually exchange information with each other, explaining what logical connections can exist between them and what advantages different architectures can have.

Client-Server Systems

In client-server systems, players do not exchange data directly with each other, but send and receive all required information via a server. Systems of this kind are sometimes also referred to as dedicated server games. Such servers can fulfill various purposes. In a very simply scenario, they blindly pass all messages they receive to all other players participating in the game. The advantage of this architecture is that each player has to maintain only a single connection to the server, no matter how many players are actually participating. This can typically reduce network traffic, since players have to send their messages only once. Also, some pieces of information may only be required by a few players and therefore need not be distributed to every single client. Servers can also compress data before distributing it to further minimize traffic.
But Client-Server based systems can provide even more advanced services than reducing message flow. In some scenarios, servers take an active role in managing gameplay. This can range from coordinating some aspects of the game (e.g. weather conditions, game statistics or other aspects of the virtual environment that are not directly influenced by the players) to complete control over what happens in the game. For example, in Blizzard Entertainment's role-playing game Diablo II™, the whole virtual environment is managed by servers. Here, clients only tell the server what the player tries to do, and the server calculates the results of that action, e.g. if an arrow hits the monster or not and how many damage it causes. The purpose in this case is to effectively prevent cheating and hacking, since players can not manipulate gameplay calculations when this is all done on a server they do not have access to.
Another advantage of servers involved in gameplay management is that the game can be changed or extended by the developer without the need to update client software on the users' machines. This can include simple bug fixing, changing minor

gameplay aspects or even adding new levels or maps to the game.
However, despite of all those advantages client-server systems offer, they also have some major disadvantages. First, all messages sent by clients have to travel along two communication pathways (client → server → other clients) before they reach their final recipients, which can be a big problem if network latency (the amount of time a packet needs to travel from sender to recipient) is high.
Another problem of client-server systems is that servers are likely to become bottlenecks if the number of clients is getting to high. There are some approaches to this problem, like operating multiple servers and balancing load between them, but all servers still have limits, either in computing power or network bandwidth, and are always only be able to handle a certain amount of players.

Peer-to-Peer Systems

This communication architecture addresses both problems of client-server systems. Peer-to-peer means that all messages travel directly from the sending player to the receiving player, or set of receiving players.
So there is no need for servers that pass along messages or even actively influence gameplay. All communication and calculations are done by the clients themselves, therefore removing the bottlenecks servers can be. Also, since messages only have to travel along one communication pathway, they are usually delivered to the recipient twice as fast as in client-server systems. This is often the key argument for network designers to choose a peer-to-peer solutions, because the speed of message delivery can directly result in the realism of a virtual environment (see section 4). Today most developers use a peer-to-peer architecture in conjunction with the UDP protocol to ensure the fastest possible packet travel times. Examples can be found in the popular strategy game Starcraft and in most 3D shooter games.
Of course this communication architecture also has some disadvantages. Peer-to-peer games are often vulnerable to cheating and hacking, because players can quite easily manipulate their client software or messages on the network. There are approaches to this problem, like some kind of controlling mechanisms where the participating peers try to validate each other's behaviour, but it is hard to find a perfect solution here.
Another disadvantage of the peer-to-peer architecture is higher network traffic than in client-server systems. For example if 8 players are competing in a game, each player is likely to communicate with each other, which can in the worst case result in 7 times higher network traffic at each peer than if a client-server architecture was used. One approach to this problem is the multicasting technology, which allows to specify multiple recipients for network messages, but is not yet implemented everywhere on the Internet and is therefore no real alternative at this time.

3 MATCHMAKING SERVICES

Before peer-to-peer games can actually begin, there has to be some way for players to come together and find each other. On local area networks or old-style modem or serial cable connections this is trivial, because communication with other players can begin instantly using broadcast or serial line transmission, but on large-scale networks like the Internet this is different: No matter if you want to play against a friend or look out for some random opponent, you have to find them somehow so that your game ‚knows' who to communicate with in the game.
This can be realized with so-called matchmaking services, which run on central servers the players log on to. Such servers are usually either operated by the company that developed the game (e.g. the battle.net™ platform by Blizzard Entertainment [1]), or in some cases also by the fan community (e.g. the FSGS™ network by Net-Games [2]). The main purpose of matchmaking services is to provide a place where players can meet and exchange the information they need for further communication in the game. This includes the possibility to ‚open' and advertise new games and to ‚join' games opened by other players. Sometimes (depending on the server software) also more advanced features are provided by matchmaking services, like user account management (password protection), lobby functionality (chatrooms where players can discuss games or agree on game settings), player profiles (personal information and win/loss records), player rankings and more.

4 MANAGING DYNAMIC SHARED STATES

As mentionned before, one of the biggest problems in networked virtual environments is network latency. For example, if it takes a network message 200 ms to travel from sender to receiver, all actions you set in the game will only be ‚seen' by other players after that amount of time, and there is little that can be done. Furthermore, network latency is never constant and can only be estimated, not exactly predicted. These conditions make it practically impossible to reach the goal of a networked virtual environment, which is to provide users with the illusion that they are always seeing the same things and perfectly interacting with each other. In some very slow games this might not be a problem (like chess or card games), but in most it is. In 3D action games for example, you want to know the positions of your opponents as accurately as possible to be able to hit them with your gun. All such changing information that multiple machines must maintain in a virtual environment is called dynamic shared state. Depending on the kind of

game, this can include general player information (status, health, etc.), positions, behaviours or properties of the world like weather effects.
So how is dynamic shared state realized? Basically, every piece of shared information is produced at one host and must be transmitted to others. Of course, by the time the information arrived at other hosts, it may have changed again already. Especially if dynamic state information like player position data needs to be updated frequently (which is usually the case), it is impossible to achieve that all participating players see identical versions of that information (this problem is often referred to as the Consistency-Throughput Tradeoff). Put differently, if absolute consistency in a virtual environment is desired, players must always wait for messages to arrive at their recipients, before new events can be generated. Therefore updates can only be done at a limited rate, which is not an option in most games. An alternative is dead reckoning.

Dead Reckoning

The idea behind this technique is that shared state information needs to be changed at any time and any frequency. Players do not wait until everyone has been informed of a change, but simply go on with their actions. Also, network messages are not necessarily transmitted everytime a change occurs, but at a maximum (or fixed) rate. The goal is not to achieve absolute consistency among players, but rather accept potential discrepancies and approximate the truth. To achieve this, additional information is transmitted with the network messages. For example, instead of simply sending player positions, also the current orientation and speed is included. This makes it possible for other players to estimate following movements and likely position, until the next message arrives for updated information. Depending on these prediction techniques and on the actual network latency, dead reckoning can be a powerful way to realize virtual environments. Many games use it today, including simulations, strategy games and 3D shooters.

Delayed Reaction

This is a technique that is not widely used, but can still be an interesting approach to dynamic shared state management. The idea behind it is to estimate network latency first (e.g. at the beginning of the game), and then delay all *local* state changes by that time. For example, if the network latency is measured to be 200 ms, all actions a player sets occur on his local machine 200 ms later. Therefore state changes occur on all participating machines at almost the same time, resulting in a virtual environment that does not only allow state changes at a high rate like dead reckoning, but is also very consistent.
Of course this is not suitable for all games. In 3D shooters for instance you probably do not want your gun to fire 200 ms after you hit the button, but in some games delayed reaction works well. The strategy game Starcraft for example uses a combination of dead reckoning and delayed reaction and usually provides a comparable good virtual environment even if network latency is high.

5 CONCLUSIONS

The goal of a networked virtual environment is to provide participants with the illusion of a common and consistent world. However due to network limitations this goal is impossible to reach, and designers have to face several decisions and accept trade-offs.
The communication architectures server-client and peer-to-peer have been introduced, which define how messages travel from sender to receiver. Both methods have their advantages and disadvantages and which one to choose depends on the kind of game and its requirements. The same is true for dynamic shared state strategies, which need to cope with the fact that all generated information in a network need a certain amount of time to be transmitted to all participants. In high-speed environments, dead reckoning is usually the way to go, because it allows shared state updates at a high rate and still makes it possible to at least approximate consistency among players.

REFERENCES

[1] Blizzard Entertainment's matchmaking service battle.net (http://www.battle.net)

[2] Net-Games' matchmaking service FSGS (http://www.net-games.com)

[3] Sandeep Singhai, Michael Zyda: Networked Virtual Environments. Addison-Wesley.

[4] George Coulouris, Jean Dollimore, and Tim Kindberg. Distributed Systems: Concepts and Design. 2nd or 3rd edition. Addison-Wesley.

[5] DirectPlay Overview, Microsoft Visual Studio Documentation (http://www.msdn.com)