PARSEC: Building the networking architecture for a distributed virtual universe

Andreas Varga
sid@parsec.org
http://www.parsec.org/

Institute of Computer Graphics
University of Technology
Vienna / Austria

Abstract

Parsec (*http://www.parsec.org/*) is a real-time multiplayer 3D space-combat game giving the user the possibility to explore a whole virtual universe. The Parsec universe consists of a network of servers on the Internet, each representing a single galaxy containing multiple solar systems. The player can travel between solar systems and galaxies. The underlying client/server networking architecture transparently handles server transitions and tries to ensure best game-play even for low-bandwidth and high-latency connections. A special masterserver keeps track of all running gameservers, and can be used to easily find other players.

This paper first gives an overview of existing architectures for distributed virtual environments and multiplayer games and then describes the structure and special properties of Parsec's networking architecture, its advantages and drawbacks. Special attention is paid to performance and portability issues.

Keywords: Distributed virtual environment, multiplayer game, client/server design, latency, bandwidth, scalability, portability.

1. Introduction

Recently a lot of work has been done to improve distributed virtual environments. These are networked virtual reality systems, where multiple users can interact in a shared world, connected via a network of workstations. Each user sees his/her own view of the simulation, and all changes to the state of the environment are distributed among the other participants. These systems can be very useful for research, and have successfully been used for military training and medical research.

However, the most impressive application for distributed virtual environments, is found in the domain of online multiplayer games. These games are becoming increasingly popular lately, and many people spend their time by pursuing a second life in some alternate reality. Many game designers have the goal to improve the immersiveness of the simulated environments by enhancing the realism of graphics and sound effects. Special care has to be taken to efficiently use the available network resources in order to distribute the current state of the world among all players. Certain tradeoffs have to be made to ensure a minimum delay between the time of an action, and when this action is replicated at the remote players screen.

There are several important differences between generic distributed virtual environments used in research or professional simulations, and those found in multiplayer games:

- It has to be assumed that the clients are connected using standard analog modems, with a bandwidth of 28.8kbps. Although faster modems are already common, it is safe to assume a lower speed, to compensate for such things as line noise.

- The computational resources that are available to render images of the world on the client's screen are unknown and may vastly differ from the hardware the game was developed on. However we have to ensure that frame rates stay high enough to keep the game interactive.

- Realism of the simulation is not as important as smooth game-play, yet the quality of the graphics and sound effects has to be high enough to make the player think he is part of the simulated world.

- The game should be easy to understand, there should not be a need to learn a lot of rules. The learning curve should not be too steep. It should be easy to join the game, find other players and have fun.

All these aspects have to be taken into consideration if one wants to create a multiplayer game that others will enjoy. In this paper we will use the game Parsec, an online multiplayer game currently under development, to describe the details of design and implementation of such a game. Parsec is a three-dimensional space-fight game, that can be played via the Internet and will be distributed as freeware. The networking architecture of Parsec is based on a client/server protocol using multiple servers, each representing a single galaxy consisting of multiple solar systems. All the servers combined form a simulated universe, that players can travel in.

2. Related Work

In this chapter we will describe some existing architectures for distributed virtual environments and take a brief look at their special properties, such as scalability, and how these systems compare to Parsec.

2.1 Peer-to-peer unicasting systems

Designing a distributed virtual environment using peer-to-peer communication to distribute state updates between clients, is a rather naive approach. It is easily understandable that if N workstations want to take part in the simulation, every state update or event that happens on one workstation has to be distributed to N-1 other workstations. Therefore these systems usually do not scale very good for a large number of simultaneous users, because the required effort is $O(N^2)$. Nevertheless, this approach also has some benefits. Since no other entities (e.g. servers) except for the actual clients are required, it is very easy to get the simulation or game up and running.

A lot of early multiplayer games used peer-to-peer communication (e.g. DOOM, Descent,...), and also Parsec has its own peer-to-peer mode, which is described in section 3.1.

2.2 Broadcasting and multicasting systems

Since scalability of peer-to-peer communication is less then desirable, many other systems using improved technologies were designed. One of the most common techniques is the use of broadcast messages. Probably the most well known of these systems is called SIMNET [4], and was sponsored by the Defense Advanced Research Projects Agency. It was designed for military training simulators and is based on the IEEE DIS standard protocol.

A broadcasting architecture is basically still like a peer-to-peer architecture, but instead of sending unicast messages to each participating client, only a single broadcast message has to be sent for one update. This reduces the number of required message updates to O(N), however in a large-scale distributed simulation with several hundred participants, there is an obvious overhead, because every client has to receive and process each state update. Since it is unlikely that all clients are directly interacting with each other, there are a lot of useless messages transmitted. Players which are spatially separated (either by long distances in the virtual space, or by other barriers such as walls) are sending updates to each other, even though they cannot see each other, hence these updates are not necessary for rendering the next frame.

Therefore one of the main approaches to improve the scalability of a distributed virtual environment is to get rid of those unnecessary state updates. A natural extension for broadcasting systems which tries to remedy the scalability problem is the use of multicast communication. With multicasting each participating client has to be part of a certain multicast group, and will only receive network messages that are destined for that group. Only those clients that want to get updates from each other are in the same group. By logically partitioning the virtual space, the number of transmitted network messages can be greatly reduced. NPSNET [8, 9, 10], VERN [1] and [3] are examples for multicasting virtual environment systems, designed for military simulations.

Since multicasting is not commonly available in consumer network systems, there are no large-scale multiplayer games using this technique. However most older peer-to-peer games partially use broadcasting in certain situations (e.g. to find other clients on a LAN).

2.3 Client/server systems

Certainly client/server communication is the most versatile way of distributing state updates among a large number of users. The basic idea is that each client sends each state update only to the server, which then has the responsibility to forward the information to all other clients. However, since the server receives all state updates, it has a complete view of the world, and knows each players position in the environment. It can therefore use special rules to decide which state updates are forwarded to which clients, and this can be used to efficiently reduce the amount of network traffic required.

For example, a server can filter packets from client A to client B, if those two clients are in separate buildings, or otherwise separated from each other. In this case client B does not need to receive state updates from client A and vice versa.

Additionally, if the virtual environment is spatially partitioned, multiple servers can be used, each of which handles clients in one part of the world. This can improve the scalability of the system even more. Distributed virtual environments designed for high scalability include the RING system [6, 7] and NetEffect [5]. These systems are able to host simulations for several hundred or even thousands of simultaneous users. RING uses visibility algorithms to filter state updates between clients that can not possibly see each other.

Parsec uses a client/server protocol for playing with other people over the Internet.

3. The Parsec Networking Architecture

Parsec was solely designed to be a network game. Therefore its networking architecture is a key part of the game, and has undergone several major revisions. In this chapter we will give you an overview of how Parsec clients communicate with each other on the network, and how we tried to solve some of the common problems in distributed virtual environments. A more detailed description of Parsec's networking architecture can be found in [11].

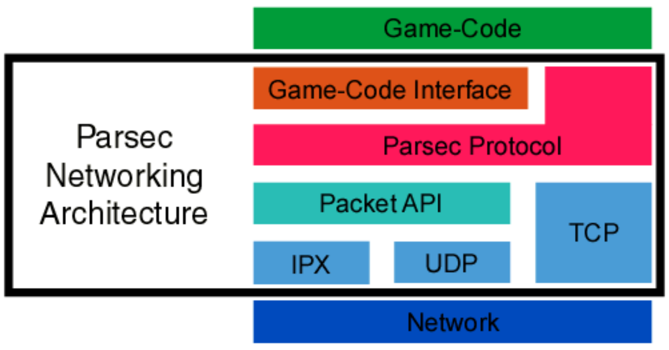

Figure 1: The layered structure of Parsec's networking architecture.

Figure 1 shows the layered structure of Parsec's networking architecture. The game logic uses the game-code interface to pass state information to the protocol layer, which defines the way clients communicate with each other (peer-to-peer or client/server-based). Parsec currently uses three distinct protocols:

- Peer-to-peer (see section 3.1)

- Slotserver
- Gameserver

The latter two protocols are client/server-based. The slotserver protocol is a hybrid between peer-to-peer mode and the gameserver mode. If this protocol is selected, Parsec clients will have to connect to a server, but the actual game state transmission is done like in peer-to-peer mode, i.e. the server is not involved in the game-play at all. This mode is useful if one wants to play peer-to-peer games with people that are not connected to the same Ethernet segment (see section 3.2).

The protocol layer itself uses any of the available transport protocols provided by the operating system via the packet API, to actually transport information over the network. Both the protocol and the packet API can be switched on the fly, to allow the user to choose his preferred mode for network play, e.g. peer-to-peer/IPX or gameserver/UDP.

Additionally TCP is used for initial connection establishment with the server, and some other communication tasks that are not time-critical.

3.1 Peer-to-peer roots

Since Parsec was planned and conceived in early 1996, the first choice for a networking protocol was peer-to-peer, because that was what all games at that time have been using successfully. Also, online multiplayer gaming on the Internet was not widely available, and did not catch on in the mainstream yet. Therefore the networking platform during development was an Ethernet-based LAN using Novell IPX as packet exchange protocol. Since then, the peer-to-peer mode of Parsec has been improved, stabilized and support for the UDP protocol has been added.

In peer-to-peer mode, each Parsec client performs a special initialization procedure when starting a network game. It sends a certain number of broadcast requests on the network, and waits for replies from other clients. The clients then distribute lists containing their node addresses among each other, to ensure that each client has knowledge of all the others. This startup phase is rather complicated, because all kinds of error conditions that might occur have to be handled, to avoid any inconsistencies. Once the startup phase has been completed, each client will have built a list of the node addresses of all other participating players. During the game it will continuously send state updates to all nodes in that list, and will also receive update packets from other clients.

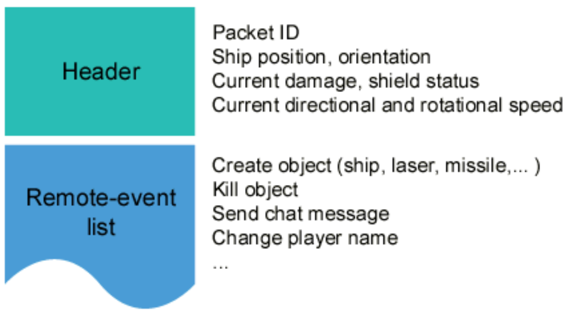
Header
Packet ID
Ship position, orientation
Current damage, shield status
Current directional and rotational speed
Remote-event list
Create object (ship, laser, missile,...)
Kill object
Send chat message
Change player name
...

Figure 2: Structure of Parsec's network packets

Parsec's network packets (see figure 2) consist of two distinct parts: A fixed-size header containing state information about the local ship, and a variable-size remote-event list. Remote-events are those events that happen in a game, and which have to be replicated at the remote-players workstation, e.g. the creation of an object, a missile shot, or the destruction of a ship. If client A receives an update packet from client B, it will:

- Update B's position and orientation on A's screen depending on the information in the packet header.
- Process the remote-event list in the packet, and re-create all the events that were initiated by B on A's machine, e.g. create a missile at B's position if B shot a missile.

Each packet's header also contains an id number, that allows the receiver to filter out old packets. If a packet is received whose id is smaller than that of an already received packet, the packet will be ignored, since it most likely has been received out-of-order. This can happen since an unreliable transport protocol is used (IPX or UDP). See section 3.4.2 for a closer look at how Parsec is affected by packet loss.

3.2 Client/server protocols

The startup-phase of peer-to-peer mode is relying on the ability to send broadcast packets, to find all participating clients for the game. However, it is usually only possible to receive broadcast requests if both the sender and the receiver are connected to the same Ethernet segment. It is not possible to broadcast UDP packets to the whole Internet, therefore peer-to-peer initialization will not work. This fact is a serious restriction for peer-to-peer mode, and to remedy this problem (and to provide a next logical step between normal peer-to-peer mode and a real client/server based game), the slotserver mode was introduced. A special slotserver is required which is responsible to mediate between single Parsec clients. A client sends a request for a ”slot“ to the server. Each server has a limited number of slots available, which directly correspond to the number of players it can handle. If the server finds a free slot for the client, it will acknowledge the slot request, and will transmit a list of node addresses of other players (those that have already been given a slot) to the client. If no free slot is available, the server will deny the client's request and tell the player that he should try to join at a later time.

This procedure is an equivalent replacement for the peer-to-peer startup phase, and once it has been completed, the client can commence the game, just like in peer-to-peer mode. The server will never receive any game-state updates, and has no impact on game-play at all.

But this protocol does not really allow for a client/server-based distributed virtual environment, as described earlier. This functionality is implemented in the gameserver protocol. The gameserver has a similar structure as the slotserver, since it also allows clients to connect and assigns them a slot for the game, but this time it is also responsible for the actual transmission of game-state updates. Instead of sending updates to all other clients, each client just sends a single update packet to the gameserver. The packet is received by the server, and is (in the most simple case) just forwarded to all other joined clients. However, since the packet containing the update has to travel from client A to the server and from the server to client B, it takes approximately twice as long for client B to receive the packet as if client A would have sent it directly (as is the case in peer-to-peer mode). Therefore if player A shoots a missile at B, there is a delay time until B will know about that remote-event, usually called the latency time. Our goal is to minimize this time as much as possible, but since it also depends on the underlying network connection, we can only partially influence it. See section 3.4.1 for a closer look at latency and how it affects game-play.

3.3 Client-side vs. server-side simulation approaches

In an ideal distributed virtual environment, the state of the simulation should be exactly the same at each client. This means that all the entities that are part of the world should be in the exact same state, at the exact same position with the same properties. However to achieve this goal, an enormous amount of data needs to be transferred between the clients, to fully describe the state.

In Parsec, there can be dozens of missiles, ships, extra objects and particle systems present at the same time. Since we can not transfer the complete state information for all these entities several times in a second, we only send remote-events that indicate that an entity has been created that the state of an entity has changed or that the entity has died and should be removed. It then is the responsibility of each client to continuously update the state of the local representation of each object. In many cases, these state changes are predictable, e.g. an extra object is rotating at a constant speed and a missile flies at a constant speed. So once a missile has been created, there is no need to distribute new positions of the missile over the network, since the position can be predicted on the client-side. Therefore we call this technique client-side simulation.

The problem with client-side simulation is that it is not guaranteed that each player sees the exact same picture of the environment on his screen. Small differences can be visible, that are caused by such factors as network latency (objects might appear at wrong relative positions), and packet loss (some objects or events might be missing at some clients).

To remedy this problem, it is possible to use server-side simulation. In this case the server has full control over the state of the world, and each state change is exclusively done by the server. This is necessary to ensure a consistent behavior of all entities in the simulation. So if a player shoots a missile, the client tells the server about this event, and then the server creates the missile, and notifies all connected clients about its creation. In a strict server-side simulation approach, even the position updates of the missile as it flies along its path are calculated by the server and sent to the clients.

The main problem of client/server distributed environments with server-side simulation is that each action that the player performs, is delayed by a certain amount of time, depending on the network latency. If this time exceeds a value of about 100ms, the human brain will notice a very annoying delay between the action and its visual feedback.

As described earlier in this section, Parsec uses a strict client-side simulation approach. Each entity in the game is controlled and owned by the clients, and the server does not keep track of the world state. However, we plan to move certain critical state control to the server, to remove some consistency problems that might occur.

3.4 Problems in an Internet-based multiplayer game

3.4.1 Latency

One of the main problems in all real-time networked games is the concept of latency. We have to assume that a large number of people playing Parsec are connected to the server with a standard analog modem, which increases the problem of latency even more, since those modems usually have internal buffers and use special compression and error correction schemes. These add to the total amount of time it takes for a UDP packet to arrive at its destination.

Games using server-side simulation are much more affected by latency, because the server is responsible for performing all state changes. This means that the player will feel the game is lagging behind his own actions. This is usually called perceptible lag, and can make the game unplayable.

Parsec does not suffer from perceptible lag, because of its client-side nature. A missile appears immediately after firing, at least on the screen of the local player. However it is possible that a missile misses its goal, although the player aimed very accurately. Since collision detection with the target ship is performed by the client of the target player, this can happen if the target ship has already moved on, before the packet (containing the missile shot remote-event) has reached the target client. In this case the state of the world at two different participating clients is not exactly the same, because of the latency time. This problem can be solved by implementing collision detection on the server.

Even though high latency times are bad, it is even worse if they fluctuate [2]. The variance of latency should be low, otherwise the player will have a hard time getting accustomed to the delay, since the human brain can adapt to a moderate delay between action and consequence only if that delay time is constant.

In order to reduce latency in a game or distributed virtual environment, we should try to avoid introducing additional delays in the software, since we have no direct influence on the latency of the network connection. Therefore we should not buffer state updates for later sending, but try to send them as soon as possible. For modem users it has proven to be a good choice to turn off compression and error correction.

3.4.2 Packet loss

Another major problem in modem-based multiplayer games is packet loss. Since modem connections are far from being ideal, packets can be lost at several stages of transmission. Parsec uses the UDP protocol for game-state updates, which is unreliable, therefore care has to be taken to resend critical information.

As described earlier, Parsec’s network packets consist of two distinct parts, the packet header containing important state information of the local ship (position and orientation) and the remote-event list which contains state updates for events that happen during the game. If a packet with an empty remote-event list is lost, the result is not dramatic. Since all Parsec clients perform linear interpolation for all ship orientations, all ships will keep flying at a constant speed in the direction they had before the packet loss. As soon as a new packet (which contains an up-to-date ship-state of the sender) arrives, the interpolated position and orientation of the ship might differ from the actual state of that ship, therefore it is not wise to just overwrite the interpolated state with the received one. Instead Parsec tries to smoothly move the ship from the incorrect interpolated position to the correct one, in a fixed amount of time (this can be adjusted). This ensures that no ”jumping“ of ships is visible, and to guarantee smooth trajectories.

In a strict server-side simulation approach, packet loss will cause the simulation to suspend until a new packet arrives, or if packet loss happens constantly at a fixed rate (e.g. 30% of all packets transmitted are lost), it will cause the game to stutter.

3.4.3 Bandwidth requirements

Since we have to assume players to be connected with a standard 28.8kbps modem, the available bandwidth for sending UDP packets to the server is usually very much limited. We therefore have to take care to ensure smooth game-play over these connections, because we can not expect the player to be connected via Ethernet or other high-speed networks.

In an ideal modem situation (no protocol overheads from the UDP and PPP layers), we should be able to transmit roughly 28,800/10 = 2,880 bytes (10 includes the start- and stop-bit used in serial communications). If we want to send state updates at a rate of 30Hz, we will only have 2,880/30 = 96 bytes to cram our state into. Currently, the size of Parsec’s packet header containing the state information is 112 bytes, which is already larger, and we didn’t even include a remote-event list yet.

To reduce the bandwidth requirements of the network communication, we can basically change two parameters: The size of the packet, and the rate at which packets are sent. The latter one is easier to do, because we definitely should not need to send a packet every time the state of a ship has changed.

In Parsec we use a very simple form of dead-reckoning [5, 9, 10]. A client only sends a packet if either a remote-event has happened, if the player has changed the ship orientation, rotation or speed, or if a specified amount of time has passed since the last packet was sent. In most cases if the player just flies with a constant speed in the same direction, no updates need to be sent, since all clients will be able to interpolate its position from the last known state data.

4. The design of a universe

The first approach in Parsec’s networking game-play, was to have one single large space, were all participating players meet and have fast-paced combat. In peer-to-peer mode this worked fine, since the number of players was always limited to 4. However, after the client/server architecture was introduced this did not mean that hundreds of clients can now play together. Although it was designed to be scalable from a technical point of view, it is often not useful to let a single server handle dozens of clients, because game-play will suffer if too many players are present.

We therefore decided to structure the virtual space, and came up with the idea of natural partitioning via solar systems. In this scenario, a gameserver is equivalent to one or more solar systems forming a galaxy. This has two advantages: We can move from a single server, to a system of multiple servers, and still offer a large playground for hundreds of players. We can therefore limit the number of players per server to a reasonable value (depending on the computing and network resources of the server), to improve game-play.

All solar systems are interconnected by stargates. If such a stargate leads into a solar system that is handled by a different server, the player flying through the gate will be automatically disconnected from the old server and reconnected to the new one. Except for a small delay, the player will not notice that he has moved to a new server.

4.1 The masterserver

Playing a networked game usually means finding other players who want to participate first. In the past, a game session had to be planned beforehand. This is easy if all players are geographically close to each other. In this case scheduling a game beginning time, and finding other players for the game is usually not an issue.

However, the most common situation is a single player that just wants to play the game. He might be anywhere on the globe, and needs to find other players that want to play too. One possible solution to this problem would be the introduction of chatrooms or a public forum, for example Internet Relay Chat (IRC). However this would require the player to have an IRC client installed, and the necessary knowledge to use it.

A different solution that is commonly used is the concept of masterservers. These are special servers, that normal gameservers connect to, to announce their availability. The masterserver can therefore maintain a list of currently running gameservers, and provide potential players with this list. The player can then choose a gameserver from the list, and connect there to play with the other players on that server. Whenever a gameserver starts up or shuts down it sends a message to the masterserver to inform it about its existence. It also tells the masterserver how many players are currently connected and how many players are allowed as a maximum. The masterserver maintains an internal list of all gameservers, and creates an HTML file of this list, which is dynamically updated. An http server should be running on the masterserver host, to enable users to download the list with any standard web browser.

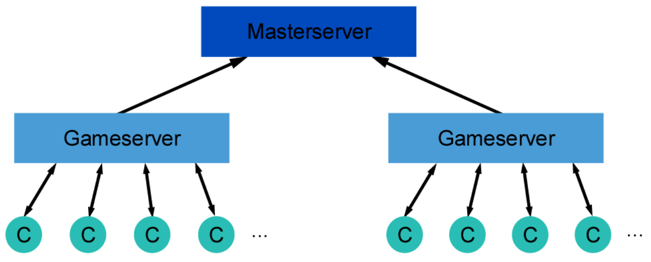

Figure 3: Hierarchical structure of the Parsec universe

Each stargate only knows its destination solar system but nothing about server addresses. Since these addresses can change, or servers can be down temporarily, the masterserver is responsible for connecting players to their desired destination. If the masterserver cannot find a suitable server, the jump will be refused. In this case, the corresponding galaxy is currently not reachable via stargates.

5. Implementation details and portability

Our current implementation of Parsec runs natively on Win32 (Windows 95/98/NT), MacOS, Linux and DOS. We support 3D hardware-accelerators using the Glide API by 3dfx, and the OpenGL API.

Networking support is a substantial part in Parsec. We support packet exchange via Novell’s IPX protocol on Win32, MacOS and DOS (for peer-to-peer games on a LAN), as well as UDP/IP (for both peer-to-peer and client/server games) on all platforms except DOS. Certain critical functions in Parsec’s client/server protocol, such as the connection establishment with a server, are done via TCP/IP to ensure reliability. In the following we will describe Parsec’s code structure in detail, and how we tried to achieve high portability, even though the target platforms are rather different.

5.1 Subsystem structure

All networking code is written in a C-like subset of C++. This means that we use things like C++ comments and inline variable declaration but we do not use classes, templates, and the like. In order to still be able to determine which subsystem a, say, function belongs to we encode this information in the function name itself.

We use a rather strict naming convention for module and function names. Every module name of the networking subsystem starts with the NET_ prefix (e.g., NET_RMEV.C, which contains all remote-event functions) if its implementation is the same for all target systems. If the module implements a specific functionality but the implementation differs from system to system its name starts with the Nx_ prefix where x denotes the target system. For example, UDP function implementations for Win32 are contained in NW_UDP.C, whereas the corresponding implementations for MacOS are contained in NM_UDP.C. These modules export the exact same interface, that is, the caller always uses the same function name, although the actual implementation will be quite different for each system.

Every interface function of the networking subsystem starts with either the NET_ or the NETs_ prefix. The implementation of NET_ functions is the same for every system, i.e., these functions are not system-dependent. Their implementation is portable code that is used for every target system. For this reason these functions are always bound statically. In C++ NET_ functions would be non-virtual member functions. For example, utility functions like NET_FetchPlayerName() belong to this category.

The implementation of NETs_ functions, however, differs from system to system and this fact is already announced by their function name. The caller doesn't know which implementation will actually be called at run-time. First, for each system the implementations will always be different, although the function name will be the same. Furthermore, if dynamic binding is enabled (via a compile-time switch) the implementation may be switched at run-time. For example, the abstract interface specifies a NETs_SendPacket() function whose implementation is different on, say, Win32 and Linux, and, moreover, whose implementation is different depending on whether a UDP or an IPX packet should be sent. Nevertheless, this is entirely transparent for the caller. Wherever a call to NETs_SendPacket() is specified this will work correctly on every system for whatever subsystem implementation may be currently active at run-time, say, Win32/UDP, Win32/IPX, or Linux/UDP. NETs_ functions are specifically named to announce this system-dependence to the caller. If dynamic binding is enabled NETs_ calls will be routed through a jump-table, although this may be disabled transparently at compile-time if on-the-fly subsystem switching is not desired and the overhead for the indirect function-calls is considered a performance issue. In C++ NETs_ functions would be virtual member-functions.

5.2 Portability

One of the major technical properties of Parsec is its high portability. Using the code naming convention described in the previous section it is possible to separate system-dependent code from fully portable code. For example, the modules containing the implementation of the packet API for IPX are called ND_IPX.C for DOS, NW_IPX.C for Win32 and NM_IPX.C for MacOS. Since these three operating systems use different APIs to provide access to IPX services, all three modules implement the same functions, but differently.

This approach worked fine for IPX, because IPX is only used for the peer-to-peer protocol, and the IPX module for DOS was finished before it was ported to Win32 and MacOS. However the other parts of Parsec's networking code that use the IP suite of protocols (TCP/IP and UDP/IP) where written for Win32, MacOS and Linux simultaneously. These three operating systems use different APIs to provide access to IP services (WinSock on Win32, OpenTransport on MacOS and BSD sockets on Linux). For testing purposes during development it was necessary to have different OS versions of the network code available (the server code was written under Linux, and the clients where tested with Win32 and MacOS).

Maintaining three different versions of the packet and protocol API code during development would have been a very demanding task. Therefore we decided to use a common API for all IP-based code. We used a compatibility library that implements the WinSock API on MacOS, and a library containing wrapper functions to work around differences between BSD sockets and WinSock.

For example, the BSD sockets API uses the close() system call to close a socket, however WinSock requires the programmer to use the function closesocket() to do the same thing. A wrapper-function called Close() (note the upper-case character) was written, that transparently calls the correct function to close a socket.

Using this method we were able to write code that immediately runs on all three operating systems, with only very minor differences. Although an additional layer (albeit a very thin one) was added to the networking code, no noticeable slowdowns have been encountered.

6. Conclusions and future work

We have presented Parsec, an online multiplayer game who aims to offer gamers around the planet the possibility to explore a distributed virtual universe, and giving them the possibility to create their own galaxies and solar systems.

We have described the technical properties of a distributed virtual environment and the obvious problems that appear if a large number of participants interacts in a shared simulation of a world. We tried to offer an insight in the natural differences between distributed virtual environments for research or military training and those found in state-of-the-art multiplayer games. Although the techniques are similar the details of implementation are vastly different, and special attention has to be paid to ensure satisfying game-play for all participants.

In the future we will continue to improve Parsec, its networking architecture and the visual quality of the game itself. We hope to release a playable version of Parsec later this year, and there is still a lot of work to do, until the universe will be online. We hope to incorporate discrete and progressive level of details for our spacecraft and other objects, to ensure high realism, while keeping the amount of polygons to render low. We are planning to implement a situation-dependent music system and support for 3D position-dependent sound effects. Furthermore we want to incorporate new ships, weapons and improve game-play in general by balancing the impact of weapons and give the player better control over his ship.

By releasing both the server (for a wide variety of platforms) and the game itself for free, we hope to attract a lot of people, either playing the game or maintaining a gameserver. As the gameserver will be configurable in a lot of ways, each server administrator has the chance to make his or her galaxy a unique experience for players from all over the world.

7. Acknowledgments

I would like to thank Markus Hadwiger for creating the initial peer-to-peer networking architecture of Parsec and for the section about the subsystem structure which he designed and implemented. I would also like to thank him and the whole rest of the Parsec team (namely Clemens Beer, Michael Wögerbauer, Alex Mastny and Stefan Poiss) for their great work, encouragement and support. Additional thanks to Gerd Hesina for proofreading this paper, and all the online gamers out there on the Internet for contributing great ideas and useful suggestions.

8. References

[1] Blau, Brian, et al. Networked Virtual Environments. *ACM SIGGRAPH Special Issue on 1992 Symposium on Interactive 3D Graphics, Cambridge MA,* pp: 157-164.

[2] Blow, Jonathan. A Look at Latency in Networked Games. *Game Developer issue 7/98,* pp: 28-40.

[3] Broll, Wolfgang. Distributed Virtual Reality for Everyone – a Framework for Networked VR on the Internet. *Proceedings of the IEEE Virtual Reality Annual International Symposium 1997,* pp: 121-128.

[4] Calvin, J., Dicken, A., et al. The SIMNET Virtual World Architecture. *Proceedings of the IEEE Virtual Reality Annual International Symposium 1993,* pp: 450-455.

[5] Das, Tapas K., Singh G., et al. NetEffect: A Network Architecture for Large-scale Multi-user Virtual Worlds. *Proceedings of ACM VRST 1997,* pp: 157-163.

[6] Funkhouser, Thomas A. RING: A Client-Server System for Multi-User Virtual Environments. *Symposium on Interactive 3D Graphics, April 1995, Monterey, CA USA,* pp: 85-92.

[7] Funkhouser, Thomas A. Network Topologies for Scalable Multi-User Virtual Environments. *Proceedings of the IEEE Virtual Reality Annual International Symposium 1996,* pp: 222-228.

[8] Macedonia, Michael R., Zyda, Michael J., et al. Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments. *Proceedings of the IEEE Virtual Reality Annual International Symposium 1995,* pp: 2-9.

[9] Macedonia, Michael R., Brutzman, Donald P., et al. NPSNET: A Multi-player 3D Virtual Environment over the Internet. *Symposium on Interactive 3D Graphics, April 1995, Monterey, CA USA,* pp: 93-94.

[10] Macedonia, Michael R., Zyda, Michael J., et al. NPSNET: A Network Software Architecture for Large-Scale Virtual Environments. *Presence 3(4) Fall 1994,* pp: 265-287.

[11] Varga Andreas, Hadwiger Markus. The Parsec Networking Architecture. *Available from http://www.parsec.org/netdocs/.*