Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Student Thesis SA-2006-04
Winter Term 2005/2006

Multiplayer Gaming in Mobile Adhoc Networks

Christian Meyer, Peter Baenziger
cmeyer@ee.ethz.ch, baepeter@ee.ethz.ch

Tutor:	Károly Farkas	farkas@tik.ee.ethz.ch
Supervisor:	Prof. Bernhard Plattner	plattner@tik.ee.ethz.ch

24.3.2006

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Contents

1 Introduction

1.1 Background

Mobile ad-hoc networks (MANET) are self configuring wireless networks which are independent of any centralized infrastructure, may comprise heterogenous devices (mobile phones, PDAs, Laptops, etc.) and span over several hops. With the increasing number of mobile devices, providing the computing power and connectivity to run applications like multiplayer games or collaborative work tools, MANETs are getting more and more important as they meet the requirements of today's users to group and interact spontaneously.

Because of a lack of centralized infrastructure, service provisioning (specification, lookup, deployment and management of services) is a major challenge in MANETs. Typical client-server applications with high QoS requirements have a hard time because of the freedom of the nodes to join and leave the network whenever they want to. In order to prevent the network from choosing as server a node, which is about to leave, information about the future behavior of each of the participating nodes is crucial for a stable server selection algorithm (mobility prediction).

SIRAMON (Service provIsioning fRAMework for self-Organized Networks) [1] is a framework, currently being developed at the Computer Engineering and Networks Laboratory [3] of ETH Zurich, with the goal of coping with the challenges of service provisioning in MANETs and providing useful basic services for multiuser distributed applications.

1.2 Scope of the Project

The overall aim of this thesis is to provide a solid test application for SIRAMON, in the form of a game which is fun to play. The test application will be referred to as 'the game' from now on. The main concern will be to integrate the services provided by the SIRAMON framework into the game focusing on the game architecture and game state synchronisation. The test game application will be implemented for Windows/Linux based platforms. In order to meet these goals, the thesis is split into the following tasks:

1. *Literature Exploration (SIRAMON)*
   The aim of this task is to be familiar with the current state of SIRAMON, what it can and cannot do, and how to best integrate its services into the game.

2. *Literature Exploration (Network/Synchronization)*
   In order to gain an overview of the problems which are faced when synchronizing time critical applications like real-time multiplayer games, extensive literature research has to be performed.

3. *Game Design*
   This task deals with the design of the game in detail.

4. *Network Layer*
   Based on the previous tasks, the network layer of the game will be implemented in this phase.

5. *Game Programming*
   The game design will be implemented in this phase.

6. *Testing/Debugging/Polishing*
   Minor design flaws in the game and its network layer will be fixed, bugs will be tracked down and the network stability will be tested, and - where possible - improved.

1.3 General Regulations

The project will be guided by Károly Farkas. At the end of the project, a written thesis report describing the work and outcome as well as the documentation of the implemented code have to be delivered. The students understand and accept the terms and regulations of ETH in regard to the developed code which will be published as open source under the terms of the GNU General Public License (GPL) [4]. In the course of the work an intermediate and a final presentation have to be given. An accepted thesis report and successfully accomplished presentations are prerequisites of getting the final grade of the semester thesis work.

2 Clowns - Game Concept

2.1 Introduction

Clowns are, besides bad television shows, the only indestructible thing in the universe. So if they got differences with each other, instead of being able to take the approach most major governments are quite fond of and just shoot each other, they have to settle their arguments in some other, but nevertheless very entertaining way. They beat each other silly and they do it with style.

2.2 Overview

Clowns is a side scrolling jump-and-run game. It does not contain any kind of single player game content and focuses solely on the multiplayer experience. The objective of a game of Clowns is to slap the clowns of other players as spectacular as possible. Every time you send another clown slamming into a wall, bouncing off the ceiling or flying over a pond, the jury will rate your performance and award points. If you manage to chain up such events you get more points and special certificates. The player who gains the most points wins. For these deeds a pair of clowns stands at your disposal. You can actively control one of them at any given time and every one of them has a Pogo Stick at its disposal.

2.3 Playing the Game

2.3.1 The Happily Hopping Duo

Every player controls two clowns. Each of them carries a pogo stick which gives them a special set of abilities.

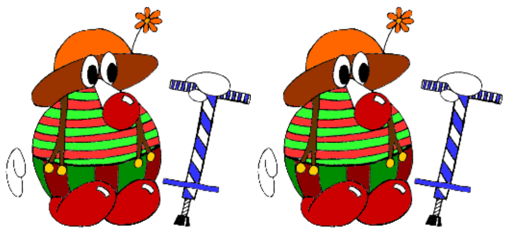

Figure 1: The Happily Hopping Duo

If you see a clown wearing a melon like hat, it's most certainly Pogo or his twin brother Pago. A photo of Pogo and Pago is shown in Figure 1. They are equipped with a pogostick which enables them to jump very high. As a surplus they can use their tool to smack clowns around, sending them flying across the playfield.

Pogojump Their pogostick enables the twins to propel themselves to incredible heights, enabling them to cross waste vertical distances at great speeds. However this ability has a few extra up and downsides. For one, you don't want to smash at top speed head first into a ceiling, and neither do the twins (doctor's orders). If the player surrenders the active control over one of the brothers while the other is hopping around, the passive one will continue to hop at his current location.

Pogobash The twins can use their tool to slap other clowns like a baseball and send them flying across the playgrounds. The more time they spend taking aim, the farther the target will fly.

2.3.2 Starting the Performance

The game starts with the server choosing a map and a color scheme for his clowns. Every player that joins has to specify his own unique color scheme and will join the running game on the fly. The game area is defined by the selected map. The border around that map is rock solid, fallig off the world is not possible. When a player joins the game his clowns are placed at predefined places in the map.

2.3.3 Running the Show

Controlling the Clowns Every player can control only one of his clowns at a time (referred to as the active clown), the other remains passive. Passive clowns do not move but perform their passive action if it is triggered by an appropriate event. All clowns can perform the basic actions which are running and jumping, and they can use the abilities granted by their tool to invoke carnage. All of them have to obey to cartoonish physics. A user can switch to another clown at any given time they see fit by pressing the switch button. This makes the next clown in line become the active one and the current clown goes into passive mode.
To make a clown act, push one of the keys which have been bound to commands in the configuration file. The default configuration is listed in Figure 2. The movement commands will make the active clown start walking into the chosen direction. If the clown should find itself falling around without being spun totally out of control, the movement commands will also allow some fine tuning of the fall while in the air. If the jump button is used, the clown will take a leap into the air. Jumping only works while the clown is standing with both feet on the ground, you cannot jump while already in the air. Using fire triggers the active clown's special abilities. Depending on the current situation this can have different effects. If the clown is currently not standing on solid groud, it will take out the pogo stick and start hopping around wildly. The pogo stick can be stoved away by triggering fire again. If the clown finds itself standing on the ground when fire is pushed, he starts charging a bash. As long as fire remains pressed, the magnitude of the resulting bash will increase to a maximal value. When fire is released, the clown will unleash the charged power upon any clown standing right in front of him, sending them flying across the playground.

Default Key	Command	Action
Up Arrow	Up	Jump
Down Arrow	Down	Not used
Left Arrow	Left	Move left
Right Arrow	Right	Move right
Left Control	Fire	Use your gimmick
Tab	Switch	Switch to the next clown

Figure 2: Basic actions performable by clowns

Cartoonish Physics Clowns don't obey to normal physical laws, they prefer a more humorous approach to inertia and energy conservation. They bounce off obstacles like rubber balls, survive falls of tremendous highs without a scar and generally behave like whole blooded toon characters.

Harvesting Points Every time you manage to hit a clown you get awarded points. If the clown was already flying around at that time, extra points are awarded.

2.3.4 The End

A session lasts for a defined amount of time or until the set amount of points to win is reached.

3 Game Design

3.1 Background

The main goal of this thesis was to implement a live game to be used to test the zone server selection algorithms embedded in Siramon. This posed a number of problems which had to be solved.
These guidelines provided the set of features which needed to be put into the application. First, the game has to be playable over a wireless network. Solutions had to be found to ensure synchronization over unreliable communication channels and the game itself had to feature interaction between multiple players and manage all of their actions.
Instead of a conventional centralized synchronization architecture used in most of corrent game applications, zone servers had to be introduced. The zone server architecture refrains from putting a single node as the central management entity and employs a cluster of zone servers which each maintain a reliable copy of the current game state. Clients then are able to connect to the zone server which offers the best performance for that particular client. Zone server architectures are much more robust against failing connections than centralized setups, yet the protocol required to synchronize zone servers is more complicated.
The game needed to be deployable by Siramon [1] and it needed to make use of the provided zone server selection algorithms. It turned out these tasks caused a lot more complications than initally anticipated since Siramon was still missing features to deploy services not written in Java.
Last but not least what has to be created was a game. Games usually consist of a large part of code and an even larger part of multimedia resources which have to be created by the designers. Needless to say there were a lot of images which had to be created for this game.

3.2 Gamestructure

The game itself consists of the following parts, most of which are explained in more detail in the following pages. Additional information can be found in Appendix B - Programmer's Guide, and in the source code itself.

Main Menu The main menu is an independant entity which is only connected to the actual game during the initialization, relaying the settings of the game to the core which processes them further.

Core The core of the game contains the main loop and manages almost all other parts of the game. It periodically calls the players (see Player) to update themselves, calls all functions which draw to draw themselves on the screen in the right order, initiates polling and handles the initialization of the game.

Player The players contain all units of a player and update themselves when called to do so. In order to affect objects (units of any player) players use the object collision (see Object Collision) which allows them to modify whatever objects they collide with.

Object Collision The object collision stores all objects and periodically re-sorts them to allow for quick checks to determine which objects are at a specific coordinate. All objects register themselves to object collision.

Gamemap The Gamemap manages the drawing of the map as well as all collision checks with solid obstacles, as well as loading the map and anything else which has to do with the map. It is called by the core, to draw itself, and by the players for collision checks.

Synchronisation Interface The synchronisation interface is directly linked to the core and the network framework. The synchronisation interface manages what messages are sent, and how they are sent, using input-events, deltastates, and many more (see Synchronisation). It also determines if a node is a server or not, and for this purpose it is connected to Siramon via a local TCP connection.

Network Framework The network framework provides the link between game data and the network sockets. It consists of a chain of classes handling the individual steps of preparing a deltastate or an event to be sent over the network and reading the resulting data from incoming data streams.

3.3 Implementation

There were several options on how to write the game. Java was a good candidate because SIRAMON was already written in Java, and because of the excellent portability of Java. The main alternatives were C and C++, and while there were other alternatives, these three were the ones we were the most familiar with. Java would likely have proven to be too slow because of performance issues and C++, being an object-oriented programming language had the advantage of generating more reuseable code than C. Therefore the game is written in C++ using the SDL [5], to allow for excellent performance while maintaining a high portability. The game now runs on both Linux and Windows.

3.4 Networking

In order to synchronize data between single nodes in a network, a communication protocol had to be designed and implemented. Main goals of the implementation were simplicity of usage, modularity and expandability. On top of those points the functionality provided by the implementation should be as generic as possible. The reason for this being the wish to be able to reuse the work spent on the creation of the communication framework for other projects. With these points in mind the design was based on multiple layers of functionality similar to the OSI model layers (see figure 3. The application should only have to provide a few generic interface functions to specify synchronisation data and the set of recipients to the transfer system which then would take care of the rest.

APPLICATION	Data management, Session detection, Latency measurement
PRESENTATION	State encoding, Message encoding, Data serialization and packeting
SESSION	Reliability services, Packet fragmentation
TRANSPORT	UDP socket administration
NETWORK	
DATA LINK	
PHYSICAL	

Figure 3: Data transfer implementation mapped onto the OSI model layers

The UDP protocol was used as the base interface to sending data. In opposition to TCP which operates on state based connections, UDP only sends raw packets which may or may not reach the target at all. The statelessness and the unrealiablity of UDP has its advantages when the underlying physical layer is of a volatile nature. If TCP is used under such conditions, a major amount of latency is introduced when a packet is lost among the way and a single delayed packet will delay the whole data stream. With UDP the loss of one packet will not affect the delivery of another. Moreover, packets which contain data being outdated a few moments later can just be dropped instead of delaying the whole sending process which would be the case using TCP.

This clearly defined the scope which had to be covered by the management entity responsible for handling inter-node communication. The following problems were isolated and solved:

1. *Data management*
   All data relevant to remote synchronisation has to be gathered from the game state and stored in structures fit for further processing. Also data recieved from the network needs to be relayed to the correct parts of the application.

2. *State encoding*
   Data representing game states is usually very bulky and contains vast redundancies. Encoding state data to compress the information is needed to reduce actual bandwidth usage when synchronizing by state updates. This of course includes the proper decoding of the data on the recieving node.

3. *Message encoding*
   All events and commands generated by the user or the application itself are considered messages. Messages do not share the same properties as states and therefore need their own separate system to be stored into a format which can be transferred over the network.

4. *Data serialization and packeting*
   All data which has been encoded needs to be properly serialized into streams of single bytes as

required by the Transport layer. Also a header has to be added to the stream to hold the necessary information for recovering the data from a serialized stream.

5. *Reliability services*
   For some data, like the one exchanged between zone servers, one needs to assure that it reaches its destination. The UDP protocol does not provide such functionality therefore a custom implementation is necessary.

6. *UDP socket administration*
   To maintain modularity a class wrapping the functionality of UDP sockets in a generalized interface is needed. This makes it possible to change the bottom level communication protocol at a later time without having to alter the rest of the implementation.

7. *Packet fragmentation*
   Packet sizes are limited when using UDP, chunks of data which exceed tolerable packet sizes need to be cut down into smaller parts and have to be sent piece by piece. Once arrived at the target they then have to be reassembled correctly.

8. *Latency measurement*
   To be able to deploy counter measures when experiencing high latency, the latency has to be measurable.

9. *Session detection*
   For testing and distribution purposes, the session detection algorithms have to be implemented in the application too to enable session detection if the Siramon framework is not available to the service.

3.4.1 Data Management

Every object which is relevant to synchronization needs to be able to turn the data it contains into a structure which can be transmitted over a serial byte stream. Also every object needs to be able to reconstruct itself completely from the generic structure it creates. Our approach to solving this problem makes full use of the chosen object oriented environment. All objects which are subject to network synchronization are derived from the same base class which contains the basic functionality to identify the object from within a generic structure, as well as interface functions which access such generic structures. The generic structure holding all the data and information about a single object will be named container in the following text. All the objects override two of these interface functions in their implementation. One function handles the conversion of the internal object state into a container, the other fills the internal state of the object from the values provided by a given container. By requesting a container from all the objects in a simulation snapshot and storing them in an appropriate structure, a full copy of that simulation state is created. Since each container has attained all necessary information about its associated object it is possible to reconstruct the entire simulation state even after the original state has been deleted or, more importantly, recreate the simulation state on another node in the network. The advantages of using this container representation include ease of adding new classes to the system without having to recode state aquisition routines and the possibility to compute the differences between two separate states by simply comparing the containers in both states.

3.4.2 State Encoding

There are two straightforward methods to replicate a state of simulation on a remote node: Complete transmission of the current state or transmission of the difference between the current state on the sender and the outdated state on the recipient. As there are very few state variables that change frequently in a game application, the latter is much more bandwidth efficient than the first. To make use of this advantage a structure named deltastate was created. Deltastates can be computed by essentially comparing the containers of a past and a present state and noting the difference between them. If done correcty adding the information stored in the deltastate to the past state will reconstruct the present state. On a side note it sould be mentioned that a deltastate can actually hold a complete state if the present state was compared with an empty state, thus implicitely enabling the less efficient way of replicating a complete state. Once a deltastate has been assembled it is packeted and sent to the recipient it was created for.

3.4.3 Message Encoding

This was done in a rather straighforward manner. When an event is generated all information about it is packed together in a multi purpose event structure. This structure is then packeted and sent to the proper recipients.

3.4.4 Data Serialization and Packeting

All types of data, be they messages or deltastates, need to be serialized into tansferable byte streams. Therefore a class was created which could convert the contents of a deltastate or a message into an array of bytes and vice-versa. This class can be handed to the basic socket wrapper and sent to another node. Also this class is what is assembled by the socket wrapper when a new packet is received.

3.4.5 Reliability Services

In order to be able to monitor whether a specific packet has reached its destination or got lost in the progress a reliability service had to be implemented. When calling the function to send a packet, one can specify a parameter to tell the communiction system to transfer the packet reliably. Packets which are requested to be sent reliably are marked as such, assigned a unique ID and stored to a local buffer. If such a packet is recieved by the network handlers they immediately answer with an acknowledgement message to confirm that the packet was indeed correctly transfered. If the acknowledgement does not arrive at the sender before the specified timeout runs out, the packet is sent anew. This goes on until the maximum number of resend attempts is reached, after which the packet is discarded as unsendable, which usually means that severe connection problems are occurring. In case of high latencies, where it is possible that the timeout runs out before the acknowledgement reaches the sender, it is possible to receive duplicate packets. These packets however can be filtered out because every packet is assigned a unique ID, if an ID is received multiple times, the packet is discarded yet is still acknowledged in case it was the acknowledgement message which was victim of packet loss.

3.4.6 UDP Socket Administration

The class encapsulating the UDP socket functionality was kept as simple as possible. The sockets can be created with or without port association, where the latter will bind the socket to the first unused port number detected. They have a straightforward function which sends a packet to a specified destination IP and port and a receive funtion which has to be polled regularly to check for pending incoming data. When the socket sends a packet it prepends a header which contains the unique service identifier (specified when the socket is created) and fragmentation information. When a socket receives data it will discard all packets which do not start with the service identifier they have been given upon creation. This serves to filter unrelated broadcast traffic and other stray packets which may be received by the socket. To centralize the processing of all data which arrives at the application from external sources (keystrokes, system messages, network packets) calling the receive funtion does not return the received data. If there is any packet received by the socket, it is stored into the event queue which is featured by the Simple Directmedia Layer. This is useful so network events can be processed the same way as keyboard events are.

3.4.7 Packet Fragmentation

If an array of data scheduled for sending is too large to be sent in a single packet, the socket functionality cuts it down into separate fragments which are numbered and sent individually. When a socket receives a fragmented packet it will store its parts locally. Once all individual fragments have been recieved, the original packet is assembled and then stored onto the event queue.

3.4.8 Latency Measurement

To measure latency the network handler class maintains a list of addresses and their time to answer found by the last 16 measurements. The implementation is simple. A message containing a ping request is sent to the target address and the time until a pong from that address received is taken down as latency. The current latency is then calculated from the median of the last 16 values measured.

3.4.9 Session Detection

To be able to detect running zone servers in a network when there is no Siramon framework available, session detection had to be implemented in the application. Session detection is very straightforward. A node which is looking for available sessions sends out a broadcast packet containing a simple discovery request to the designated server port. If a server receives such a request it will answer with a packet containing information about itself. By periodically sending out requests and listening for answers the detecting node will have an accurate list of currently available servers at any given time. It has to be mentioned that this kind of server detection only works in local networks since broadcast packets are usually filtered in routed networks.

3.5 Synchronisation

3.5.1 Overview

The game uses a zone server architecture, meaning there are several servers which are connected to each other, and clients are always connected to one of these servers. All communication is either between a server and a single client, a server and all of its clients, or between one server and all other servers. An example for communication between a server and a single client is when a client requests its server to make it a server as well. This server will then tell all other servers to properly update the new server, and when it is ready the server will give the client to go ahead. Then it will tell all of its remaining clients about the new server which they may join.
The synchronisation of the game uses four different kinds of messages. Input-events, system-events, requests and commands, and gamestates. Input-events are used to relay keyboard inputs and need to be known by all servers. System-events like the notification of a client leaving the game are generally distributed to all servers and relayed by the servers to their clients. Commands and requests, like the request to become a server, are sent from a client to a server or a server to a client. Gamestates are used to keep clients updated and to resynchronize servers.
At the end of this chapter there are several examples of what the game actually does, to deepen your understanding.

3.5.2 Framenumbers

Time on all servers and clients is sliced in timeframes to allow for better synchronistion. A timeframe lasts for 30 ms, and the numbering of the timeframes is synchronized throughout the game. Everything that happens in the duration of a frame is treated as if it happened at the exact same time - at the end of the timeslice. In effect, discrete time is used with 30ms timeslices.
The synchronization of frame numbers is done by constantly sending the current frame number to all clients which then adjust their frame number if theirs is lower needed and otherwise use their own system clock. This works well in environments with only a few nodes like the one the game was programmed for, but is something which will have to be observed carefully if using a larger amount of participating nodes (more than 32).
Most transmissions, meaning all input-events, all gamestates and the majority of system-events, are marked with a frame number, so all servers know what happened when. In the game there is no actual time, there are only frames.

3.5.3 Input Events

Input events (Figure 4) are the key inputs which a player uses to control the game. All key inputs are sent to a server which then stores and relays those inputs to all other servers in the network. They are not relayed to the clients, since clients are synchronised by deltastates (see Gamestates and Deltastates).
All key inputs are marked with a frame number, and are used by servers to recalculate what happened. On every new frame a server rewinds a few frames and recalculates all of them, using the events he currently knows about. So even if an event arrives delayed it will not be lost unless it is more than a few seconds late. If an event arrives too late, the server knows immediately it is probably desynchronised and requests a resynchronisation. Server resynchronisation is done by the oldest server, which takes the role of a master server, if needed.
Input events are always sent reliably, meaning if they are not acknowledged, they are resent. Sending of such events, if necessary, takes place immediately after they occur.

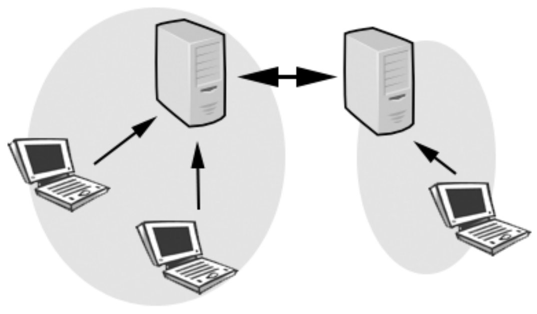

Figure 4: Propagation of input events

3.5.4 System Events

Joining, leaving, becoming a server, becoming a client, changing a client's server and requesting resynchronisation are all handled by system events, altough they are usually not initiated by them.
System events (Figure 5) are relayed from the first server that initiates them to all other servers and then relayed to all clients. System events can only be started by servers, although clients start requests (see Commands and Requests) which when granted will make a server start a system event.
System events only have a frame number if they need to, that is if it desynchronises the game if it doesn't happen at the same time on all nodes. It is not crucial to know the time when a server became a client, but it is very important to know at what time a new player entered a game, in order to spawn the player simultaneously on all nodes. Example: If a client requests to switch to a new server, that server will start a system event telling all other servers that the client changed its server.
System events are always sent reliably, meaning if they are not acknowledged, they are resent. Sending of such events, if necessary, takes place immediately after they occur.

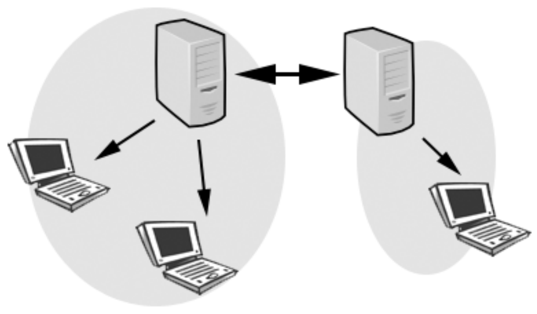

Figure 5: Propagation of system events

3.5.5 Requests and Commands

Sometimes a server has to speak with a specific client, or a client needs something special from its server. In such a case they issue a request or a command (Figure 6), referred to as R/C for the span of this paragraph. These never involve game synchronization directly, but they are used to change the zone server structure running in the background, to allow new players to join the game, and to allow players to leave the two which have been mentioned last actually do effect synchronisation.
An R/C will almost always be from a client to its server or from a server to its clients. The first exception to this rules is a new client joining the session - because the client doesn't exactly have a server before joining. The second exception is a client wanting to change its server. In that case the client contacts the server it wants to join directly - the server then initiates a system-event, making the client one of its own.
The reaction on an R/C will usually be either another R/C, a system event, or both. Very often it is accompanied by a full gamestate update (see gamestates and deltastates).
Example 1: An R/C to become a server, sent from a client to its server will be answered with a system event of the client becoming a server, keyed to a frame number. As soon as it receives a system event, the client knows it is a server. The other servers send old input events to the new server as soon as they get the system

event. And the original server sends a full gamestate update to its former client.
Example 2: An R/C to join the game will result in a full gamestate update of the new client, a system event informing every node about the new client, and a R/C telling the player which ID he needs to use.

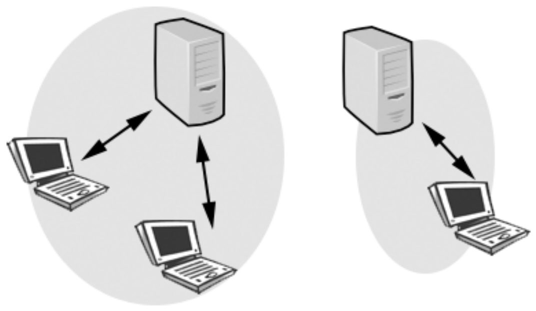

Figure 6: Propagation of commands and requests

3.5.6 Gamestates and Deltastates

Gamestates (Figure 7) usually contain all current information about the game, as well as the information about how the server thinks the game will look in the next three frames. Gamestates can be sent either as full gamestate updates, or as deltastates. Full gamestate updates are always sent reliably and are composed of all information contained within a gamestate. They are used whenever desynchronisation was detected, or when there is a chance of desynchronisation occuring (example: server change), usually initiated by a request or command (see 3.5.5 Requests and Commands).
Deltastates on the other hand are gamestates which are reduced to the changes between the current gamestate and the gamestate which was up to date three frames ago. Clients are regularly updated with deltastates, which are unlike full gamestate updates - not sent reliably. Deltastates are only used for the periodic update of clients. This happens every frame which means once every 30 milliseconds.

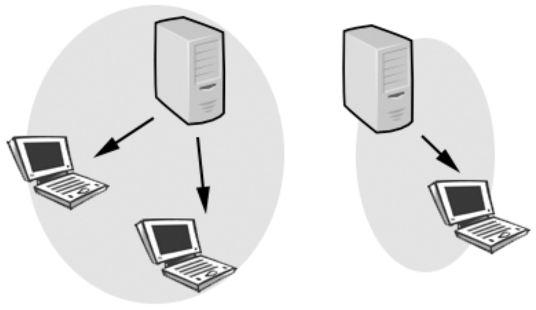

Figure 7: Propagation of deltastates

3.5.7 Hiding Lag

In order to hide lag, all keyboard input is treated as if it happened five frames later than it really happened. This delay is added on the local machine where the original input occurred.
The reasoning behind this delay is that a typical player has no trouble getting used to a constant lag, as long as it is not too high and does not fluctuate too much. That way any lag below 150ms will have no observable effect at all.
Additionally all servers send their clients information about future gamestates, how they expect them to be. In case of a temporary connection loss (still in the milliseconds range), the clients draw from these first, before starting to draw their own guess on how the game will behave - which is how the game would look if no other player would press any key. If a server isn't responding for longer than a very few seconds a client independently switches its server. This is done by having the clients periodically scan for better servers than their own, and then switching to the better server, if at all possible.

3.6 Integration into Siramon

At the time this project was realized, the implementation of Siramon still missed a generalized interface towards services which are deployed by Siramon. Also the decision to implement the game in C++ to ensure performance and portability left the problem of starting the game from the context of the Java runtime environment.

3.6.1 External Service Deployment

The current implementation of Siramon does not feature ways to create service descriptions which describe services to be started outside of the java runtime environment. Specifically, in the case of this project, the ability to deploy binary executables was missing. To cope with this problem a small class[1] was implemented in Java. This class acts as a mediator between the deployed foreign service and the Siramon framework. It is capable of starting the service when demanded to do so, it monitors the state of the local service and it manages the communication between the deployed service and Siramon.

3.6.2 External Service Communication

By default Siramon will automatically start a new instance of the zone server selection algorithm when a service using it is deployed. This is acceptable for services which do not require the usage of multiple sessions or for services whose session management is administrated by Siramon itself. The game, however, performs session management and deployment in it's own code, making it necessary to circumvent the automatic creation of a server selection instance when deploying the game.
To prevent Siramon from automatically creating a zone server selection instance is easily done by altering the service description document. Omitting all the parameters related to the server selection takes care of unnecessary algorithm instances being created. To actually being able to start a zone server selection computation from an outside service at will a communication channel was implemented. The communication channel is based on a simple TCP/IP connection which uses the localhost adress to connect Siramon with the foreign service. A simple, human readable protocol is then used to exchange information relevant to the zone server selection system.

Protocol Description Before actual data can be exchanged a quick handshake protocol is employed. As soon as the client connects to Siramon, Siramon will inform the client it is connected to a Siramon framework by sending out `SIRAMON_CONNECT`. The client then has to reply with `SERVICE_`*`<identifier>`* to make himself known as the project application. If the server accepts the service identification reply it will finish the handshake by sending `SIRAMON_ACCEPT`. In the case that Siramon does not accept the client for any reason, it will reply with `SIRAMON_DECLINE` and close the connection. A graphical interpretation of the handshake is shown in figure 8.

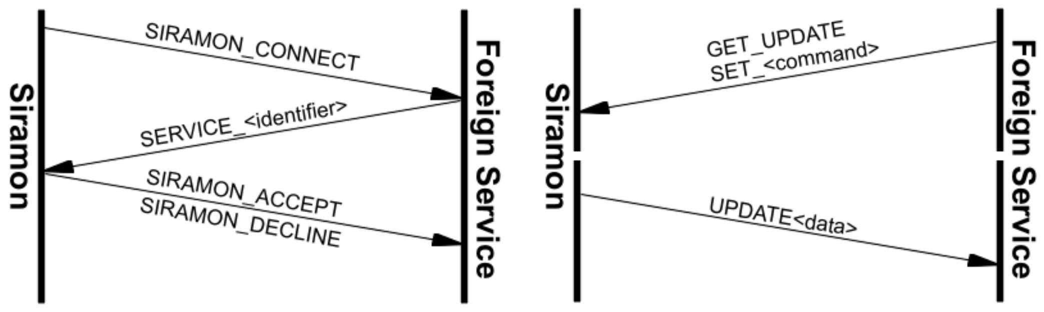

Figure 8: Handshake protocol and control message communication schematic

As soon as the connection is established, Siramon will use it keep the service updated about the state of the zone server selection instances requested by the service. The service on the other hand can issue commands to the Siramon framework to start and stop the zone server selection or request state updates manually. Figure 9 contains all commands which can be used by the service.

The only other messages transferred to the service contain information about the current state of the zone server selection instance on the local machine. These messages are automatically generated by the

[1]Services.Clowns.Game

Command	Description
`SET_NEW`<*session*>	Create a new zone server selection session for this service
`SET_JOIN`<*session*>	Join the specified zone server selection session
`SET_EXIT`	Remove the service from it's current zone server selection session
`GET_UPDATE`	Request a message containing information about the zone server selection session

Figure 9: Valid command messages accepted by Siramon

zone server selection implementation when the list of neighboring zone servers or the role of the local node changes. If the need arises, the service can manually request an update message by sending `GET_UPDATE` to Siramon. Update messages contain the current role of the local node assigned by the zone server selection, the identifier of the session and a list of IPs. The listed IPs correspond to all neighboring nodes which are currently selected as zone servers. The service is obliged to switch its behaviour in the session it is participating according to the update messages it receives from Siramon. Update messages are sent to the service every time the zone server selection detects a change in the network layout and reevaluates the current selection scheme. These updates are usually very rare, thus the need for a method to manually request state updates.

4 Related Work

This section covers work related to the task of this thesis.

4.1 Multiplayer Gaming

Since the childhood days of computer gaming, when someone let a bright dot bounce between two equally bright movable bars and then called it 'Pong' alot has happened. Computing power as well as the availability of high performance input and output devices has dramatically increased. Virtual reality which once was associated with a couple of monochromic triangles roughly forming the shape of a landscpe now heads with giant's steps towards photorealism. Old fashined single channel audio was replaced by digital directional audio systems and storage devices can hold more information per area than ever. Realistic simulation of real behaviour of objects has become a commodity, simulation of whole armies instead of a few humble heroes has become a standard. What began with 'Pong' where you plugged two joystick into the same computer slowly developed with the innovation of inter-computer communication. Modems made it possible to play 'Pong' with a player on the other end of a phone line. The Internet started providing digital communication link to any destination. Networking became affordable for home usage. Geeks started bunkering themselves in their garage, linking their computers together, eagerly consuming the early products of networked multiplayer games. And slowly the number of players which could participate in a single game increased. From a game one played with his friend to today's massive multiplayer online games which entertain tens of thousands of players at the same time, requiring state of the art server infrastructures to handle the computational load and the network traffic, multiplayer gaming has experienced a veritable explosion in its possibilities.
The following subsections contain a small overview over the basic concepts of the different types of multiplayer games available today.

4.1.1 Shooters

The basic concept of the Shooter is putting all the players into a common environment, equip them with weapons and declare the last player standing as winner. This basic concept has been refined in multitudes of directions. What started with simple 8 player deathmatch has evolved into tactical team combat simulations with player numbers ranging anywhere from 4 to 256 players at the same time. Games of this type are percieved as very fast paced, very demanding on the eye/hand coordination and the reflexes of the players competing. From a technical point of view, these types of games are very sensitive to network latency. Examples of shooters are Unreal Tournament 2004 [6], Battlefield 2 [8] and Doom 3 [7].

4.1.2 Real Time Strategy Games

Real time strategy games put the players into the role of a general commanding an army or a major administrating a city. These games focus on a player's ability to make quick decisions, manage resources and think strategically. Due to issues with overview and computation limits, games of this type usually do not support more than 8 players simultaneously. There have been experiments with 16 and more players, however, game playability suffers considerably with large player numbers in these kind of games. Examples of real time strategy games are StarCraft [9], Act of War [10] and Age of Empires 3 [11].

4.1.3 Role Playing Games

Role playing games were spawned from the pen and paper role playing systems. Though the actual role playing part in these games is rather minuscule. The main feature of this game is the characters which are developed by the player. By achieving goals the game character can be improved, making him stronger and enabling him to master more difficult task. The actual representation of this improvement in the game is what makes or breaks this kind of games. As of today, these games can support any given amount of players, from 2 to 8 players in local networks or tens of thousands of players experiencing the same environment at the same time in a massive multiplayer role playing game this type of game has seen tremendous success in providing a base for a game suitable for a large audience. Examples of role playing games are World of Warcraft [12], Everquest 2 [13] and Final Fantasy XI [14].

4.2 Game Architectures

There are two types of architectures which are used by most games which feature multiplayer capabilities. One focuses on a single authoritative server which administrates the game simulation and relays data to all connected clients. In the second case what is usually referred by Peer-to-Peer, every client manages his own state of the game and communicates with all other clients in the session to maintain synchronization. However, relevant to this thesis is a rather new, hybrid approach which introduces zone servers [2]. A zone server only receives state changes from its local set of clients and propagates these changes to remote clients by contacting the other zone servers in the session. Typically the zone servers are distributed topologically across the network and the clients connect to the zone server which is closest.
Using zone server yields a few advantages over the commonly used architectures. Zone server architectures scale much better than centralized server structures or peer-to-peer systems, since they can distribute computation load and bandwith stress if necessary. Also zone server architectures are more resistant to errors than a centralized approach which crashes totally when the server fails. Unfortunately these advantages come at the price of increased latency because information has to travel multiple hops in a zone server environment.

4.2.1 Zone Server Selection

A crucial task when using a zone server architecture is actually electing a node to perform the tasks of a zone server. This decision has to be made from many different parameters like position, bandwidth, computational power and so on. The resulting problem can be interpreted as finding a dominating set of nodes on a given weighted graph. There are numerous algorithms to find a dominating set of nodes. Maurer [2] introduces the Priority Based Algorithm (PBA), which is the algorithm currently used by Siramon to provide zone server selection functionalities. PBA computes a dominating set of nodes in a fully distributed fashion using node weight and neighborhood information.

4.3 Service Provisioning

4.3.1 Siramon

Siramon (Service provIsioning fRAMework for self-Organized Networks) [1] is a service provisioning framework based on a decentralized and modular design. Siramon has to integrate the functions required to deal with the full lifecycle of services, such as service specification, lookup, deployment and management of not only trivial but also complex services (e.g., mobile ad hoc group game applications). Every device runs an instance of Siramon which handles the control and synchronization among the devices, as well.

4.4 QoS Requirements of Multiplayer Games

This chapter shall give an overview over some of the various techniques and solutions to problems that arise in the context of network synchronisation. Sources include various publications from game producing studios and institutional research. The information is divided into a few classes of fundamental problems and the techniques that are proposed to solve that particular problem.

4.4.1 Bandwitdh

Bandwidth is a highly critical resource when it comes to networking. It limits the amount of data that may be exchanged between hosts per time unit. Violations of this limit causes extra latency and packet loss beyond normal operation parameters. Network architetures need to be designed in consideration of the amount of available bandwidth.

Delta State Encoding This has established itself as a standard when synchronising a game by distributing the server's current game state at regular intervals. The method is quite simple. When a client joins a session, the server sends a complete snapshot of the current gamestate to the client and retains a copy of that snapshot for himself. Once the client acknowledged the full snapshot the server will regularly send messages containing only the changes between the gamestate last acknowledged by the client and the current gamestate on the server. This method alone suffices to create a basic server-client synchronisation of decent quality over an unrealiable network. Quake 3 employed this technique and according to the Book of Hook [17] it worked well.

Reliable Event Synchronisation Theoretically one can synchronize a game progress by ensuring the simulation on all of the hosts involved processes in exactly the same way. For this to work, all the hosts have to start their simulation with an identical initial state and then all the user interactions have to be inserted at exactly the same time slot into the simulation progress on all hosts. If implemented perfectly the sending of gamestate synchronisation messages becomes unnecessary and user interactions can usually be encoded very efficiently into single timestamped messages which take up only a small amount of bandwidth.

Localized Replication Localized replication works based on the assumption that a client only needs to know about a small part of the global gamestate to be able to select his next action. Unreal's architecture [16] classifies the set of objects which are important for the client as the 'relevant set of actors'. Keeping this assumption in mind network traffic can be greatly reduced by only keeping the client updated on objects that are relevant for him and not sending information about the rest of the gamestate.

4.4.2 Latency

Latency is a major hazard when trying to synchronize real time critical programs. Ideally all the interactions performed by a user should be immediately relayed to the server, processed and results should be sent back to the client. In reality all of these tasks take up a certain amount of time which results in more or less significant skews in interaction timing. Users expect their actions to have consequences based on their current perception of the game state. But when latency is introduced the user's actions usually reach the server only after a few additional steps of simulation have been computed which may or may not result in drastically different results than originally intended by the user. On top of this problem the user may experience significant delays between pressing a button and the actual reaction of his protagonist in the game, which may render a game quite frustratig if it is based on timing one's actions exactly.

Fixed Event Delay Real time strategy games usually make use of this technique [18], instead of relaying interaction events on a best effort basis. A fixed delay is introduced so all commands given to a unit are always processed after a defined amount of time in the future. This reduces latency fluctuation between low and high delays which is perceived by most users as more annoying than a constant mediocre delay.

Gamestate Rollback To reduce the reaction time to user input, this technique can be employed to ensure client interaction is always computed at that time in the simulation when the user wanted it to occur. For this purpose, the server maintains a cache of past gamestate snapshots. When a user interaction is received, the server rolls back the simulation to the exact time where that interaction occurred and recalculates the current gamestate beginning from that moment. Detailed information how this was used in Half Life can be found in the Valve Developer Wiki [15].

Localized Simulation To give a user the impression of his actions being processed instantly, the client extrapolates a possible gamestate from the user's interactions until it gets reliable game state information back from the server. This is usually combined with state intepolation to smooth over minor discrepancies between the reliable game state and the simulated one on the client. Both Unreal [16] and Half Life [15] make use of this technique to reduce perceived latency.

4.4.3 Packet Loss

Unreliable packet transmission is a major problem when communicating between hosts. Even though there are protocols which guarantee ordered and full delivery of packages, a connection can still break and data can be lost. Choosing between a reliable protocol (TCP) which can increase latency and bandwidth issues when something goes wrong or an unreliable packet based protocol (UDP) which will lose packets or mix their order of arrival at the worst possible times is highly dependent on the application's needs. Based on this, most applications assume unreliable connections and optimize their solutions for the average amount of chaos.

Brute Force Replication Easiest and most error-proof approach is synchronizing data in a brute force manner, which involves broadcasting the current gamestate every step of the simulation. Implementations usually refine this approach a little to take some strain off bandwidth. The most popular example using this kind of synchronization is Quake 3 Arena [17]

Reliability Contracts Some implementations provide mechanisms to distinguish between data which has to be reliably transmitted and data which is good to have but doesn't have to be reliably synchronized. Those applications usually monitor the current state of the network and prioritize synchronisation of reliable data over all other data which reduces latency on critical parts of the gamestate when the network is experiencing problems. The framework used for Unreal [16] implements a sophisticated reliablity management system.

5 Conclusions and Outlook

This section shall give an overview what was achieved during the course of the project. An overview over the final state of the application is given as well as pointers as to what could be improved by further work on the subject.

5.1 Application State

The application in it's final state contains all the functionality to play a game with a few friends over a local network. All the functions necessary to employ video and audio data are in place and working. An experimental implementation of a network protocol with synchronization for zone server based architectures is running. Similarly, there was an expermiental interface implemented between Siramon and Clowns. The game itself had to be kept very simple. Only one type of clown could be implemented, the score tracking system is kept very simple and the overall handling of the game has a couple of rough edges. Yet it is a good estimate as to what the requirements of a complete game application would be.

5.2 Obstacle Summary

During the development of the projects a few problems were encountered which deserve to be mentioned as further work might have to find solutions for them. Also there seem to be a couple of pitfalls when using Siramon as a provisioning framework.

5.2.1 Application Design

The zone server selection algorithm was designed to start and stop applications where client and server are distinct executables. Our approach was designed to have a single executable which would change its behaviour according to the specifications dictated by Siramon. Also the ability to run multiple sessions of Clowns within the same network was a design goal for this project. Experience shows that the current implementation does not support either without some major workarounds being employed.

When designing an application to be used with Siramon, server and client functionality should be separated to ease the administration required to run the service correctly, a topic for further improvement of this project might be separating client and server processes from each other.

A major flaw in the current implementation of Siramon is its inability to deploy services of the same kind into individual sessions. Currently the framework assumes that all applications of a distinct kind deployed will connect automaticaly to any other running instance of that application in the network. This problem also involves defining an interface which communicates the selected session to the deployed service and monitoring currently active sessions by Siramon itself. For this project, session management had to be implemented into the application. Designing and implementing an interface to manage sessions within Siramon could be the topic of further work. This is actually an enhancement of the current advertisement functionality.

5.2.2 Non-Java Services

When designing a service outside of the Java runtime environment, there is no standardized functionality to deploy it. The observations during this project also showed that the scheduling of the runtime environment and the external application can have a major impact on the performance of either. Especially the neighborhood detection used by the zone server selection is very prone to lag introduced by the lack of computational power.

A standardized process how external services are deployed should be introduced into the service provisioning framework. Along with this process a method to relay session information to the launched service needs to be implemented.

When designing a service to be run while using the Siramon framework, it is crucial this service behaves nicely towards other applications present in the system. This is especially problematic for applications which need to be run in real time such as games. Games tend to use up all the available computational resources in a system to deliver a best effort gaming experience. All further applications designed for the use with Siramon should keep in mind that the provisioning framework needs computational power while the service is running to maintain its functionality.

5.2.3 Synchronization

The implemented network code works quite nicely between clients and their respective zone servers. However, there are some occasional dissynchronizations between the zone servers to be observed. Tracking the sources of these glitches has proven rather difficult since the code was implemented in an experimental fashion. Creating an universal data synchronization system for a zone server based architecture might be a topic for further work. However given the amount of diversity of applications and their requirements posed to their synchronization protocols, this task might be impossible to accomplish.

5.2.4 Testbed Hardware

The testbed which was available for this thesis was terribly outdated. Though this might have been a good thing considering it helped discover some of the bugs which were overshadowed by excessive computational power. However, considering the application designed in this project was a game using today's presentation standards, a slightly more accurate representation of today's hardware used for gaming might have been helpful.

5.2.5 iPAQ and Clowns

There was the initial discussion about porting the application to the iPAQ nodes available to the institute if time permits. Even though theoretically possible, there is a port of SDL for iPAQ linux availabe, the application would need major overhauls before it could be run on a platform with such limited resources. Porting Clowns to the iPAQ itself would yield enough material for a semester's thesis all by itself.

A User's Manual

A.1 Installation

A.1.1 Linux

On the CD provided for this thesis, you can find all necessary software to get a running copy of the environment required to successfully deploy Clowns with Siramon. First find the directory 'installation/linux' where you should find four files (named 'install', 'uninstall', 'install.tar.gz' and 'j2re-1_4_2_11-linux-i586-rpm.bin'). Copy all of them into a directory of your liking. As an example, the test bed used for the thesis put them directly in the home directory of a user. To actually perform the installation procedure, root permissions are required. If you already have installed a working Java Runtime Environment (JRE), you may skip the next step. If there is no JRE installed, you can unpack the provided self-extracting archive ('j2re-1_4_2_11-linux-i586-rpm.bin') by executing it. This yields a rpm package which can be installed on your local machine using the command 'rpm -i j2re-1_4_2_11-linux-i586.rpm' if your distribution contains a port of the redhat packet manager. After the JRE was installed, you can run the provided script file 'install' which will automatically extract the three components required to run Clowns (the appliation, Eclipse and Siramon). After the installer completes successfully, you need to configure two script files according to your local settings. You find both in the 'Siramon' folder which was created by the intaller. In the script 'RunSiramon' you need to set the javaPath variable to the directory where the binaries for the JRE can be found as well as the IP address of the local node. The other file you might need to adjust is 'InitWireless'. It contains the basic commands to configure a wireless network interface. If your ethernet adapter is already configured correctly, you do not need to use it or change it. Once the installation is complete you should be able to execute 'RunSiramon' to start the environment.

A.1.2 Windows

To install Clowns on a windows machine, just extract the contents of the archive 'clowns.rar' found in 'installation/win32' to a folder of your liking. To run Siramon on a Windows platform you need an installation of Eclipse. A distribution thereof can be found in the 'installation/win32/software' directory. After installing Eclipse, set up the provided Siramon source code in a directory of your liking and create a workspace environment within Eclipse on that directory. If done correctly you should now be able to configure the libraries required for Siramon execution in the Eclipse project management. But before you can actually compile and run the source code, you need to change the path described in Services/Clowns/Game.java on line 85 to the precise location of the run batch file within the directory you installed Clowns. Once all this is done, Siramon can be run from the Eclipse environment.

A.2 Configuration

A.2.1 clowns.ini

The file `clowns.ini` which can be found in the basic installation folder contains some settings which can be manipulated if desired. The file itself consists of seven sections and parameters associated with that section. It can be manipulated by any text editor.

[Video] This section contains three parameters, `resolution` sets the resolution which should be used when the game was started. This does not affect the resolution used while in the main menu. The resolution has to be specified by a triplet of values separated by a `'x'`, e.g. `<width>x<height>x<depth>`. The other parameter is `fullscreen`. It can be either set to `0` (disabled) or `1` (enabled). If enabled, the game will use full screen mode to display the scene when playing. A third parameter `parallax_bg` which is representing a boolean value enables or disabled parallax backgrouns scrolling. If not enough computational power is available, this can be disabled at the cost of visual quality.

[Audio] Though there are five parameters in this section, only `volume_effects` and `volume_music` should be manipulated manually. Those two parameters regulate the volume at which background music and sound effects should be played back. Both can be set to anly float value from `0.0` to `1.0`, `1.0` means maximum volue. `channels`, `frequency`, `chunksize` should only be manipulated by advanced users who know what they are doing. The default settings should run with almost any common sound card available, so there should be no need to manipulate them.

[Input] This section is unused and does not do anything. Theoretically `mouse_sensitivity` would control the sensitivity level of the mouse cursor. Since Clowns is not using the mouse whatsoever, the parameter is superfluous.

[Network] Settings relevant to the network system are manipulated here. However, only `use_threads` is currently available. If set to `1`, sockets will use threads to read incoming data. Unfortunately, due to a flaw in the design, this is not working and provokes an application crash when enabled.

[Keybindings] By manipulating entries in this section the key bindings of the clown actions can be remapped to the user's taste. Keep in mind that the names of all special keys have to be written in capitals whereas the plain alphanumeric keys have to be typed in lower case.

[Player] These settings can be manipulated in the main menu too, but for convenience issues a default player setup can be defined by using the parameters in this section. `name` defines a default name for the local player, which may not be longer than 8 letters and only alphabetic upper case letters will yield valid results. Entries `primary_color` and `secondary_color` set the color indexes used for the two base colors used to dye the clown's clothing. They can both be set to any value from `0` to `29`.

[Maplist] to be able to select a map from the maplist in the main menu, it has to be added as one of the `map<nr>` entries in this section. Also the `mapdir` entry has to be properly set to the directory where the map files are located.

A.3 Starting Clowns with Siramon

If the installation is working properly, just run Siramon. Once the GUI available select Clowns from the local service list and click deploy local.
If a node running Siramon already advertised Clowns to the framework, alternatively Clowns can be started by selecting it from the remote service list and clicking on deploy remote.

A.4 Starting Clowns without Siramon

To start Clowns without Siramon support, go to the installation folder of the game and execute the script file called 'runlocal'.

A.5 The Main Menu

Once the game is started you enter the main menu. In the main menu you can chose the colors of your clowns, your playername, if you want to start or join a game session, which session you want to join, or with which map you want to start a new session.
The individual menu points are chosen by moving the blue marker with the up and down keys next to them and pressing Return.

A.5.1 Color Selection

The color selection is represented by two colored balls, one for the primary and one for the secondary color of your clowns. You change their colors with the up and down keys and change the active ball with the left and right keys. You can leave the color selection by hitting Return.

A.5.2 Playername

Simply type your playername name in here.

A.5.3 Session and Map Selection

Pressing the up and down keys will switch through all available maps. Left and right allow you to scroll through the currently active sessions. Hitting Return does leave the selection but not immediately start the game.

A.5.4 Start

This will attempt to start or join a game, depending on the settings in the Session and Map Selection. On a failure, the game will return to the main menu after 10 seconds.

A.5.5 Exit

Quits the game.

A.6 The Running Game

A.6.1 Interface

The interface of the game consists of two parts. In the top left corner all participationg nodes are listed, along with their game internal ID, their role ('server' or 'client of server x'), and the lag delay to each other node in milliseconds. Each node prints the line with info about itself in a brighter yellow.
In the top right corner the players are listed along with their current score, sorted by their current score, highest score on top, lowest score at the bottom. On the center screen you see your clowns and those of your enemies - the clown with the red arrow above its head is the one currently under your control.

A.6.2 Controls

Playing the Game Playing the game requires five different keys, by default these are the following (additionally, there is the ESC key, which returns to the main menu without further warning)

- Cursor Left: Moving the currently active clown to the left.

- Cursor Right: Moving the currently active clown to the right.

- Cursor Up: Jump.

- Left Ctrl: This key is trickier. If the active clown is not standing on solid ground, for example because he just jumped, this toggles the use of the pogo stick. If the active clown is standing on solid ground while you press this button, he will bash away whatever clown stands in front of him. The longer you hold the key down, the harder your clown will bash.

Server-Client Controls You can manually change the role of a node with the following keys, although SIRAMON might undo your changes:

- 's': This will try to promote the node to become a server itself. It has no effect if used on a server node.

- 'c': This will degrade a server without clients to become a client itself. If used on a server to which a client is connected, it will tell all clients of the server to get themselves another sever. Repeatedly pressing this button will therefore result in the server becoming a client if there is any other server out there. It has no effect if used on a client node.

- '0-5': These keys will make a client try to connect to a specific other server ID. This has no effect if used on a server.

B Programmer's Guide

This appendix is designed to serve as a guide to understand the implementationary details of the Clowns source code. It also contains some tips as how to extend the existing code.

B.1 Compiling Clowns

Compiling is fairly easy given the development environment is setup correctly. Following prerequisites are required:

1. *Clowns source code*
2. *C++ compiler*
3. *SDL developer libraries*
4. *SDL_image developer library*
5. *SDL_mixer developer library*
6. *SDL_ttf developer library*

Installing the developer libraries (they can be downloaded from the SDL homepage [5]) with default settings should suffice to get everything to work. When the source code is unpacked running `make windows` (or `make linux` respecively) in the trunk folder will compile all the code files and link them into the clowns executable in the game folder.

B.2 Adding New Classes

A new class should always be added inside a fresh code file with a new header file containing the class declaration. First thing to do in the header file is to envelop it in a #ifndef SOMEHEADER_H_ #define SOMEHEADER_H_ ... #endif structure to avoid multiple inclusions. If the need arises to #include other header files the place to do it should be in the code file preferably. An exception of this rule are base classes of the newly created class. In the code file the first two things to add is #include "frontend.h" and the inclusion of the header file for this class. Note that frontend.h may not be included *anywhere* else than as the first thing in a code file. *Never* include frontend.h in a header file.

B.3 Main Menu

files `mainmenu.cc, mainmenu.h, gameparameters.cc, gameparameters.h`
The main menu serves as an interface to find running game sessions and let the player select the one he likes to join. Also it serves to let the player costumize his name and the appearance of his clowns.
This class is designed as a means to interactively assemble the parameters needed to launch the game engine. This purpose is realized by the run() function. Calling this function will activate the main menu and set the visual system to the main menu's needs. The function will then enter a loop which handles all the necessary server detection, user input processing and drawing onto the interface. As soon as the user exits the game entirely or selects to start a game, the run() function assembles the parameters entered and returns them. These are then used to start the game engine.
The menu knows one special functionality, if started with Siramon framework support, the menu can perform an autostart if it is ordered to work in dedicated mode by Siramon. When autostart is engaged, all user interactions are disabled and the menu will return a usable set of game parameters as soon as it found the servers specified in the update message.

B.4 Event Queue

files `eventhandler.cc, eventhandler.h, baseevent.cc, baseevent.h, eventmanager.cc, eventmanager.h, eventinterface.h, eventinterface.cc`
The Simple Directmedia Layer library collects all data gathered by input devices such as keyboard, mouse and joysticks in a queue structure as simple structures called events. This queue can be polled to get all the incoming events for internal processing. The event handler does exactly that, it polls the event queue and

processes the found events into a format which can be used by the rest of the implementation. All the entities in Clowns which have to parse a certain type of event (such as key presses or packets that arrive from the network) inherit the event interface protocol and register themselves to the event handler by calling the register_to_handler() function provided by the event interface class. When the event handler parses an event for which an event interface has been registered, that event is handed down to that interface by calling the handle_event() interface which must have been implemented in every class inheriting the event interface. Events for which no interface is registered are discarded.
Aside from distribution, the event manager is responsible to translate basic key events to actions actually interpretable by the game. This is usually refered to as key binding and is used to be able to customize the position of the available action commands on the keyboard according to the tastes of the player.

B.5 Resource Management

files `datatank.cc, datatank.h, loadfuncs.cc, loadfuncs.h`
To avoid problems with loading static resources such as images, sound effects, animations and so on, the data tank takes care of static resource management. Instead of loading and storing resource files inside the classes which make use of them, the data tank takes care of loading and storing and just passes pointers to the resources out. This has the advantage of having a defined storage place for any resource which has to be loaded at runtime, therefore relieving the programmer of having to keep track of memory allocation issues. Also when multiple classes use the same resource, it will be only loaded once and then used by all of those classes.
Accessing the data tank functionality is very simple, first a database of functions needs to be specified, once that is done any resource file can be loaded by simply calling get(”resourcefile.type”). The functions implement the functionality of actually loading a resource from a requested file. To add such functions to the data tank, a call to add_filetype() is required. This function associates a file postfix (3 alphanumeric letters plus the prepending dot) with two function pointers (one function to load a resource file, the other to free the memory taken by a loaded resource) and the file name of the default file which should be returned if the file requested by get() could not be loaded. When the filetype associations are in place calling get(”resourcefile.type”) will first make the data tank search his database whether the specified file has already been loaded, if so no actual loading is performed and the stored pointer is returned. Otherwise it will make the data tank check whether it knows the postfix of the file and then selects the appropriate loading function. It will then attempt to load the resource and store it in its resource database while returning the pointer to the newly loaded resource.

B.6 Network Management

files `networkmanager.cc, networkmanager.h, basesocket.cc, basesocket.h, udpsocket.cc, udpsocket.h`
The network manager proides functionality related to data transfer, packeting, latency measurement and reliability services. When created, the network manager automatically sets up the basic communication socket used to send out packets and initializes the data structures used to measure latency and provide reliable packet transfer.
The main functionality of the network manager is sending and receiving packets. To actually perform the latter, the pump() method of the network manager has to be called at regular intervals, all packets currently pending will then be read from the input buffer and put into the main event queue. To send packages to another node, an overloaded function is available, it takes a pointer to the class which should be packaged and sent, the address of the destination (which may be NULL to broadcast) and a timeout and resend counter which are used for reliability services. If the timeout is set to zero, the packet will be sent non-reliably. The two classes currently accepted for transmission are Baseevent and Deltastate. Discovery and acknowledgment messages are usually generated from inside the network manager and therefore do not need a special function to make sending of these available to outside objects. When a packet has been sent in reliable mode get_ack_state() can be used to monitor the state of that packet by using the identifier number which was previously returned by send(). States a packet can be in are ACK_UNKNOWN (the packet id was not found in the resend buffer), ACK_PENDING (the packet is still waiting to be acknowledged), ACK_TIMEOUT (all attempts to send the packet have failed and the resend counter has run out), ACK_RECEIVED (the packet was transmitted successfully). If a packet should have been delayed excessively and it is deemed unfit to continue retransmission, remove_ack() can be used to remove packets from the resend buffer. Should a connection to an address get lost, the packets associated with that destination can be removed by using

remove_ack_address().
Latency measurement to a certain destination can be performed by calling get_latency(). The function will instantly return the current state of measurement and send out a ping message requesting an answer. The times until a matching pong message arrives after a ping are stored in a circular array, the median of all the values currently stored in that array is taken as reference for the overall connection latency.
To make the application actually visible to the network a socket on a commonly known port has to be opened. This socket can be enabled and disabled by calling enable_listensocket(). The port used for Clowns is 20632. To make a server available for detection, this socket has to be enabled.

B.7 Gamestate Encoding

files `gamestate.cc, gamestate.h, deltastate.cc, deltastate.h`
Correctly handling the data contained in a gamestate can be confusing. Therefore a detailed rundown on how to use the game state class properly shall be given.
When a game state is constructed it is empty. So the first thing to do is to fill it with data. This can either be done by using the set_delta() function given a deltastate which was built from an empty game state and the outdated game state which should be brought up to date are available. This is usually done on a client which recieves its initial state update. On a server however, the game state has to be assembled with the data in the running simulation.
To initiate a data aquisition pass, the game state to be updated has to be locked for input by calling input_lock() on it. Once locked, input_process() has to be called for *all* objects which are scheduled to be synchronized with another node. Once the whole simulation state has been crammed into the game state, it is necessary to end input mode by calling input_unlock(). This action purges all data containers in the game state which have not been updated previously and are therefore considered to have become obsolete. After updating set_timestamp() should be used to set the game state's timestamp to the current time.
Once the updated game state has been computed, a deltastate can be calculated by calling get_delta() on two game states to get the differences beween them. This deltastate can then be sent through the network.
When reconstructing a gamestate, either when wanting to update the simulation state after having recieved an update in a deltastate or when reassembling the simulation state completely from scratch, a procedure similar to the one used when aquisitioning data from the simulation has to be performed. Assuming a game state containing an updated simulation snapshot is available, first output_lock() has to be called on it. This method also takes a boolean parameter specifying whether a full update should be performed or if only data containers that were changed should be considered. Once locked, the output_process() was to be called repeatedly until it returns NULL. When it does not return NULL, a pointer to a data container with updated contents is returned. By using get_id() and get_type() on that container the simulation should be able to identify the object which this container corresponds to and update it accordingly. If the object does not yet exist, the simulation should create an appropriate object and update the new object. Data containers also contain a boolean which can be parsed by was_deleted(). If true, the simulation should remove that object from simulation if it is still part of it. When output_process() returns NULL, all updated containers have been processed and output_unlock() has to be called on the game state.

B.8 Data Aquisition

files `datacontainer.cc, datacontainer.h, object.cc, object.h`
To get the state variables out of an object and into a data container a common interface had to be defined. The abstract object class provides such an interface which has to be implemented in any new class which has to be synchronized. Objects can do two things with a data container, they can write their own current state into the container, or they can read the container and update themselves with it's contents. The two functions used for these actions are write_data() and read_data(). Reading from or writing to data containers are done by using the set() and get_type() functions the containers provide. These functions just store values to and read values from an array which is automatically adjusted to the classes needs. The only thing that needs to be kept track of is the indexes at which the variables are stored into the container so they may be retrieved properly afterwards.

B.9 Data Serialization

files `packet.cc, packet.h`
The packet class can be created from any discovery, baseevent, deltastate or acknowledgement class. It

serializes the contents of those classes into a serial byte array fit for network transmission and can convert such a byte array back into the original classes. A detailed rundown of the serialized array layouts is given below. Field sizes are measured in bytes.

B.9.1 Event Packet

`[Head][EventID][EventType][Timestamp][RequestAck][EventInformation]`		
Field	**size**	
Head	1	An event packet is identified by a leading 'E'
EventID	2	A unique number assigned by the instigator of the event
EventType	1	Identifies the event type
Timestamp	4	Identifies the exact simulation time slot when this event occurred
RequestAck	1	If set request acknowledgment from recipient
Eventinformation	X	Data specific to the event type

B.9.2 Gamestate Packet

`[Head][StateID][Timestamp][RequestAck][Payload length][Gamestate payload]`		
Field	**size**	
Head	1	An event packet is identified by a leading 'G'
StateID	2	A unique number assigned to the state packet
Timestamp	4	Identifies the event type
RequestAck	1	If set request acknowledgment from recipient
Payload length	2	Contains the size of the gamestate data payload attached
Gamestate payload	X	Deltastate encoded information, see below

B.9.3 Gamestate Payload

`[ID][Type][Load][Flag1][Data1][Flag2][Data2]...` `[ID][Type][Load][FlagA][DataA1][DataA2]...[FlagB][DataB1]...`		
Field	**size**	
ID	2	Contains the unique object identifier to which the data is addressed
Type	2	Contains the unique class type identifier associated with this object
Load	1	Amount of following flag/data pairs, 0 and 255 reserved, see below
Flag	1	Contains data association information, see below
Data	1 to 4	Contains state information

When a deltastate is serialized, all the individual container differences are stored in one of the two formats depicted below as a single array of values. The reason for having two distinct ways of storing data is bandwidth efficiency. When a changed state varibale is stored to the serialized array it is assigned a flag which identifies the variable within the object it belongs to. This is efficient if only a few variables in the same object change simultaneously. However if the object is sent as a whole this would generate significant unnecessary overhead. Therefore a second data representation is introduced and used solely when an object is updated in its entirety. In the second format, only one flag is assigned to an array of following values which are then considered to be ordered in ascending fashion from lowest to highest index. Usually the load field contains the number of variables which have been changed in the object associated. Two possibilities have been introduced to cope with special cases of the updating process. If the load is set to 0, the object was deleted between the old and the new state. If the load is set to 255, the following data must be interpreted in full update mode and not in single flag/data pairs. The flag bytes which prepend the data fields are split into bitfields, the 2 most significant bytes are used as an identifier for the type of variable which follows (char, short, long, float), and the 6 other bytes denumerate the index of that variable within the data container it belongs to. If the flag is used in full update mode, the 6 least significant bytes are interpreted as the amount of values of the selected type which follow the flag.

B.9.4 Acknowledgment Packets

`[Head][Type][PacketID][PacketTimestamp]`		
Field	**size**	
Head	1	Acknowledgment packets are identified by a leading 'A'
Type	1	Stores the type of the acknowledged message
PacketID	2	Id of the packet that is acknowledged
PacketTimestamp	4	Timestamp of the acknowledged package

B.9.5 Discovery Packets

`[Head][Type][Data Length][Data]`		
Field	**size**	
Head	1	Discovery packets are identified by a leading 'D'
Type	1	Type of the discovery package, see below
Data Length	2	The size of the following data field
Data	X	Data payload, see below

There are three types of discovery packets: FIND tells the recipient to send information to the sender; PLAYER contains information about an unoccupied player; SESSION contains info about a joinable game session. According to the type, the data payload holds the necessary information. In a FIND packet, the payload is empty. In a PLAYER packet name and color scheme are included. SESSION packets hold server addresses, player names and colors, the session identifier and the name of the used map file. The PLAYER event is never used in the game.

B.10 Main Loop

files `game.cc, game.h`
The main loop which consists of calculating the current state, drawing the game on the screen, polling and handling events, sending packets, and handling incoming network messages, is handled in this class, as soon as the game is running.

B.11 Creation and Handling of Network Commands

files `game.cc, game.h`
All network messages related to the running game eventually end up here, in the game class. There is one event for every type of keypress, as well as serveral events which manage server switches, servers becoming clients, clients becoming servers and more. Much more specific information can be found in the source code.

B.12 Storing Key Inputs

files `inputbuffer.cc, inputnuffer.h, game.cc, game.h`
Key inputs are stored and if they arrive too late, the game recalculates what would have happened if they had arrived in time. The inputbuffer class stores all inputs belonging to a single player, and constructs and registers itself to the game whenever a new player object is created. The class uses a cyclic buffer to store all the entries, the current size is 512 entries, which lasts for 15 seconds worth of data - anything older than that is ignored.

B.13 Maps and Collisions

files `gamemap.cc, gamemap.h, collision.cc, collision.h`
Gamemap handles everything related to the map, reading it from the ini file, creating a fast-check collision map, drawing the map and background, and checking for collisions with the solid parts of the map. It uses a fancy approach to significantly speed up collision detection which is further explained in the code itself. Collision is responsible for checking on collisions between different clowns.

B.14 Clowns

files `player.cc, player.h, clowns.cc, clowns.h, pogo.cc, pogo.h`
These functions are the core of the non-network part of the game. The player class manages all commands given to the individual clowns like drawing themselves, or key inputs. It also is the interface which connects the clowns to the datacontainers, allowing the gamestate synchronization. The clown class handles movement and basic functions of the class while the pogo class is a child class of the clown class which contains more specific functions. Prototypes of different clowns exist in the unused classes trampo (trampo.cc, trampo.h) and jojo (jojo.cc, jojo.h). More documentation can be found in the code.

B.15 Siramon extensions

files `Services/Clowns/Game.java, Services/Clowns/ServerApp.java`
These two files contain all the necessary code to handle the communication between Siramon and Clowns when it has been deployed.
Once deployed, the java code opens a socket and a thread which will be used to communicate with the service on port 22881. If the socket was successfully created Clowns is deployed by starting the run script. This script sets all the necessary environment variables and executes the binary with the '-s' option. This option makes Clowns establish a connection to Siramon on startup. Once the connection has been established, the Siramon side waits for commands from the service. A few callbacks are hooked into the Siramon zone server algorithms when something in the network structure changes. This way the service can be immediately notified about relevant changes.
When the zone server selection turns a node into a server for a given service, it automatically starts a separate server application for the service. In Clowns, server and client software are packed into the same executable therefore starting the service anew when it is already running would be a waste of resources. The ServerApp class implemented circumvents this by turning the event of redeploying the service if it is already running into a simple update message which tells the service to switch to server mode. If there is no such service currently running on the node when it is elected as server, the executable is started and, once the connection is established, ordered to go into dedicated mode.

B.16 Audio

files `soundmanager.cc, soundmanager.h`
The basic needs of playing back audio data are covered by the sound manager. To play back a previously loaded chunk of audio data, call play_sound(). To start or end the music running in the background, use start_music() or end_music() respectively.
Caution is advised when using play_sound() inside a loop which is executed multiple times per frame. This will only lead to loud repetitive noise since all the mixing channels will be flooded with the same sound samples over and over.

B.17 Ini Parser

files `ini_parser.cc, ini_parser.h`
The ini parser is used to read data from a file which represents configuration data in a human readable format. These files store data in plain text as `<variable name>=<value>` pairs which can be divided into sections defined by `"[<secion name>]"` tags. The clowns.ini file is a good example of such a file. The function which provides access to such a file when it has been loaded with an ini parser is get_var(). This returns a pointer to the value of a specified variable in the provided section. If that variable is not found a value can be defined to be returned by default.

B.18 Palettization

files `palettizer.cc, palettizer.h`
To be able to color the clothes of the clowns in the game according to the requests of the user, the palettizer implements the necessary functions to create a linear gradient of a specified color. These gradients are then mapped into the ranges of the palette arrays which are reserved for that purpose (color indexes 1 through 14 for the primary color and indexes 15 trough 28 for the secondary color). For this to work the image files processed in this way have to be created according to these restrictions. Modifying the animation files of the clowns without caution will result in ugly graphical artifacts.

B.19 Animation

files `animation.cc, animation.h`
Animations are stored as multiple image files containing the frames of an animation lined up in a sraight line. A script file written by using the ini parser format then associates these files with animation states and directions which can easily be accessed by the game when it needs to draw an animated image. To get the necessary information to draw an animation, the get_frame() function is called. It returns a pointer to the image containing the requested animation and stores the coordinates where the frame of the requested state, time and direction is located inside that picture into the given buffer.

B.20 Fonts

files `fonts.h, fonts.cc`
These classes wrap the functions of a small library which serves the need of simple typesetting when drawing text information onto the game screen. The only function it provides is blit() which draws a string using the specified font to a given image. Information about the parameters can be found in the header file of the wrapped library (externals/ttfutil/ttfutil.h).

B.21 Simple Directmedia Layer

All basic interfaces to input/output devices except the network sockets are handled and administrated by SDL. For reference consult the Simple Directmedia Layer documentation [5].

C File Listings

C.1 Launch script without Siramon

```
Contents of file 'runlocal'
--- start of file ---
#!/bin/sh

#DISPLAY=:0 ; export DISPLAY
export LD_LIBRARY_PATH=/usr/local/lib
cd /home/rogrueni/clowns/game
./clowns
--- end of file ---
```

C.2 Launch script with Siramon

```
Contents of file 'run'
--- start of file ---
#!/bin/sh

#DISPLAY=:0 ; export DISPLAY
export LD_LIBRARY_PATH=/usr/local/lib
cd /home/rogrueni/clowns/game
./clowns -s
--- end of file ---
```

C.3 Service Description

```
Contents of file 'Clowns.xml'
--- start of file ---
<?xml version="1.0" encoding="UTF-8"?>

<SERVICE
uri = "rosamon://Service/Entertainment/Games/Multiplayer/RealTime/Clowns"
url = "rosamonTransport://192.168.0.10:4440/Siramon/Services/Clowns/Descriptions"
location = "Services.Clowns.Game"
completeness = "true"
comment = "Realtime multiplayer games featuring clowns."
>
<GENERAL
name = "Clowns"
version = "0.1"
producer = "Christian Meyer and Peter Baenziger"
producerEmail = "none@nowhere.na"
>
</GENERAL>

<IMPLEMENTATION
engagement = "0"
stateless = "false"
replaceable = "false"
remoteable = "false"
>
<CODE
url = "rosamonTransport://192.168.0.10:4440/Siramon/Services/Code"
>
<ENVIRONMENT>
<EE
name = "SDL"
```

```
version = "1.2"
>
</EE>
<DEMANDS
programSize = "5000000 byte"
memorySize = "1000000 byte"
graphicOutput = "800x600;color"
netProtocol = "datagram"
netBandwidth = "10000 byte/s"
zss = "false"
zss_server = "Services.Clowns.ServerApp"
>
</DEMANDS>
</ENVIRONMENT>
</CODE>
</IMPLEMENTATION>

<SESSIONS
uri = "rosamon://Service/Entertainment/Games/Multiplayer/RealTime/Clowns"
>
<ROLES>
<MANDATORY
uri = "rosamon://Service/Entertainment/Games/Multiplayer/RealTime/Clowns"
numberOf = "2"
auxiliary = "false"
>
</MANDATORY>
</ROLES>
</SESSIONS>
</SERVICE>
--- end of file ---
```

C.4 Configuration File

```
Contents of file 'clowns.ini'
--- start of file ---
[Video]
resolution=640x480x16
fullscreen=0
parallax_bg=0

[Audio]
volume_effects=1.0
volume_music=0.75
channels=2
frequency=44100
chunksize=2048

[Input]
mouse_sensitivity=2.5

[Network]
use_threads=0

[Keybindings]
Up=UP
Down=DOWN
Left=LEFT
```

```
Right=RIGHT
Fire=LCTRL
Switch=TAB
Leave=ESCAPE

[Player]
name=LINURI
primary_color=5
secondary_color=8

[Maplist]
mapdir=data/
map0=mini.map
map1=lions.map
--- end of file ---
```

References

[1] K. Farkas, *“SIRAMON - Service provIsioning fRAMework for self-Organized Networks”*, http://www.csg.ethz.ch/research/projects/siramon/, ETH Zurich, January 2005

[2] F. Maurer, *“Service Management Procedures Supporting Distributed Services in Mobile Ad Hoc Networks”*, ETH Zurich, August 2005

[3] *“Computer Engineering and Networks Laboratory”*, http://www.tik.ee.ethz.ch

[4] GNU.org, *“The GNU General Public License”*, http://www.gnu.org/licenses/licenses.html

[5] Sam Lantinga, *“Simple Directmedia Layer Documentation”*, http://www.libsdl.org/docs.php

[6] Epic Games & Digital Extremes, *“Unreal Tournament 2004”*, http://www.unrealtournament.com/ut2004/

[7] id Software, *“Doom 3”*, http://www.doom3.com/

[8] Electronic Arts, *“Battlefield 2”*, http://www.ea.com/official/battlefield/battlefield2/us/home.jsp

[9] Blizzard Entertainment, *“StarCraft”*, http://www.blizzard.com/starcraft/

[10] Eugen Systems, *“Act of War: Direct Action”*, http://www.atari.com/actofwar/uk/flash_high.htm

[11] Ensemble Studios, *“Age of Empires III”*, http://www.ageofempires3.com/

[12] Blizzard Entertainment, *“World of Warcraft”* http://www.worldofwarcraft.com/

[13] Sony Online Entertainment, *“Everquest 2”*, http://everquest2.station.sony.com/

[14] Square Enix, *“Final Fantasy XI”*, http://www.playonline.com/ff11us/index.shtml

[15] *“Valve Developer Community Wiki”*, http://developer.valvesoftware.com/wiki/

[16] *“Unreal Networking Architecture”*, http://unreal.epicgames.com/Network.htm

[17] *“Book of Hook - The Quake3 Networking Model”*, http://www.bookofhook.com/

[18] Paul Bettner & Mark Terrano, *“1500 Archers on a 28.8”*, http://zoo.cs.yale.edu/classes/cs538/readings/papers/terrano_1500arch.pdf