**C:\Users\patrick.ravi-pinto\Desktop\BFH_TI_DE.gif**

**Projektbeschrieb**

**PacMan Multiplayerspiel**

Joos Patrick, Ravi-Pinto Patrick, Zahnd Stefan

**Bern, 24. Januar 2011**

# Inhaltsverzeichnis

[Inhaltsverzeichnis 2](#_Toc283661593)

[Einleitung 3](#_Toc283661594)

[Aufgabenstellung 4](#_Toc283661595)

[Funktionsumfang 4](#_Toc283661596)

[Voranalyse 5](#_Toc283661597)

[überlegungen 5](#_Toc283661598)

[Client / Server 5](#_Toc283661599)

[Peer-to-Peer 6](#_Toc283661600)

[Konsistenz & Synchronisation 6](#_Toc283661601)

[Übermittlungsfehler, Übertragungsgeschwindigkeit und Latenz 7](#_Toc283661602)

[Lösungsansätze 8](#_Toc283661603)

[TimeWarp-Synchronistation 9](#_Toc283661604)

[Clientside-Prediction 9](#_Toc283661605)

[Unsere Lösung 10](#_Toc283661606)

[Klassendiagramm 10](#_Toc283661607)

[Entscheide 12](#_Toc283661608)

[Technologien 12](#_Toc283661609)

[XNA-Framework 12](#_Toc283661610)

[Lidgren-Bibliothek 13](#_Toc283661611)

[Vorgehensweise 14](#_Toc283661612)

[IST-Zustand 21](#_Toc283661613)

[Ausblick 21](#_Toc283661614)

[Fazit 22](#_Toc283661615)

[Zusammengefasst 23](#_Toc283661616)

[ANhang 24](#_Toc283661617)

[Literaturverzeichnis 24](#_Toc283661618)

[Bilderverzeichnis 24](#_Toc283661619)

[Code Snippets 24](#_Toc283661620)

[Revisionsverlauf 25](#_Toc283661621)

# Einleitung

Im Rahmen des Moduls 7301q (Projekt 1), führen wir in kleinen Gruppen ein Projekt durch. Die Themen werden vor Beginn der Projektarbeit ausgeschrieben. An regelmässigen Sitzungen (i.d.R. wöchentlich) wird der Verlauf des Projektes mit dem verantwortlichen Betreuer besprochen und die weitere Planung erarbeitet.

Wir, Patrick Joos, Patrick Ravi-Pinto und Stefan Zahnd, haben eine Dreiergruppe gebildet und werden uns dem Multiplayer PacMan Projekt annehmen.

# Aufgabenstellung

Hier soll anhand eines Mehrbenutzerspiels (Pacman) eine verteilte (interaktive) Simulation realisiert werden. In einer verteilten Simulation müssen die auf unterschiedlichen Rechnern gestarteten Teilsimulationen (Spieler-Clients) miteinander gekoppelt und synchronisiert werden. Dabei kann sowohl eine Client-Server-Lösung als auch eine Peer-to-Peer-Lösung realisiert werden.

# Funktionsumfang

Nach einem äusserst fruchtbaren Brainstorming und einer ausgedehnten Diskussion wurden folgende Funktionen definiert, die in der untenstehenden Tabelle aufgeführt werden.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Beschreibung** | **Kriterientyp** | | **erledigt** |
| **Muss** | **Kann** |
| 1 | Multiplayer | X |  | √ |
| 2 | Netzwerkbasiert | X |  | √ |
| 3 | Fairness allen Mitstreiter gegenüber | X |  | √ |
| 4 | High Score | X |  | √ |
| 5 | Geister mit eigenen Namen versehen |  | X |  |
| 6 | High Score pro Spieler |  | X |  |
| 7 | Eigene Power-Up’s |  | X |  |
| 8 | Von uns eingebaute Cheats |  | X |  |
| 9 | Möglichkeit, eigene Levels zu kreieren |  | X | jein |

Tabelle 1 - Gewünschter Funktionsumfang

# Voranalyse

## überlegungen

Die Aufgabenstellung ist klar formuliert. Die PacMan Anwendung soll sowohl als standalone Applikation wie auch als Mehrspielerversion funktionieren. Mehrere Kontrahenten können aber schlecht an einem einzigen Computer gleichzeitig gegeneinander spielen.  
Das Spiel muss also so aufgebaut werden, dass die einzelnen Instanzen auf irgendeine Weise miteinander über das Netzwerk Daten austauschen. Will man ein netzwerkbasiertes Multiplayer-Spiel aufbauen, dann muss man sich zuerst überlegen, mit welcher Architektur diese Informationen untereinander ausgetauscht werden sollen. Die zwei gebräuchlichsten Methoden sind das Peer-to-Peer und das Client/Server-Prinzip (beide haben ihre Vor- und Nachtteile). In der weiteren Überlegung sind Aspekte wie Datenübertragungsgeschwindigkeiten, Übermittlungsfehler und Spielsynchronisation wichtige Themen eines solchen Projektes.

Dieses Kapitel soll einen kurzer Abriss bieten, welche Technologien zur Verwendung kommen könnten und welche in diesem Projekt angewendet wurden.

### Client / Server

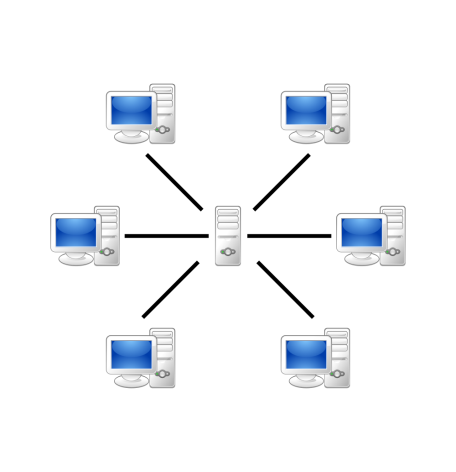
Die meisten Action-Spiele, die heutzutage gespielt werden, sind modifizierte Client/Server-Spiele. Games wie Half-Life (einschliesslich seiner Mods wie Counter-Strike und Team Fortress Classic) operieren auf solch einem System, wie es auch Spiele tun, die auf der Quake3 oder Unreal Tournament Spielengine basieren. In diesem System existiert ein einziger, autoritativer Server, der dafür verantwortlich ist, die zentrale Game-Logik auszuführen. Diesem sind ein oder mehrere „dumme“ Clients angeschlossen. Diese Clients sind vorerst nichts anderes als ein Mittel, um Spielereingaben aufzubereiten und diese dem Server zur Ausführung zu schicken. Der Server verarbeitet die Befehle, schiebt die Objekte an die richtige Stelle und schickt den Clients eine Liste der Objekte, die zu rendern sind.

Abbildung - Client / Server Architektur

In der Wirklichkeit besitzt diese Lösung einige Komponenten mehr. Dieser vereinfachte Abriss ist aber hilfreich, um Überlegungen bezüglich Verzögerungskompensation (lag compensation) und Vorhersage (prediction) anzustellen.

Eine typische Client / Server Spielengine-Architektur sieht in der Regel ungefähr so aus:

Client

Eingaben aufbereiten

Objekte zeichnen

Netzwerk  
Verbindung

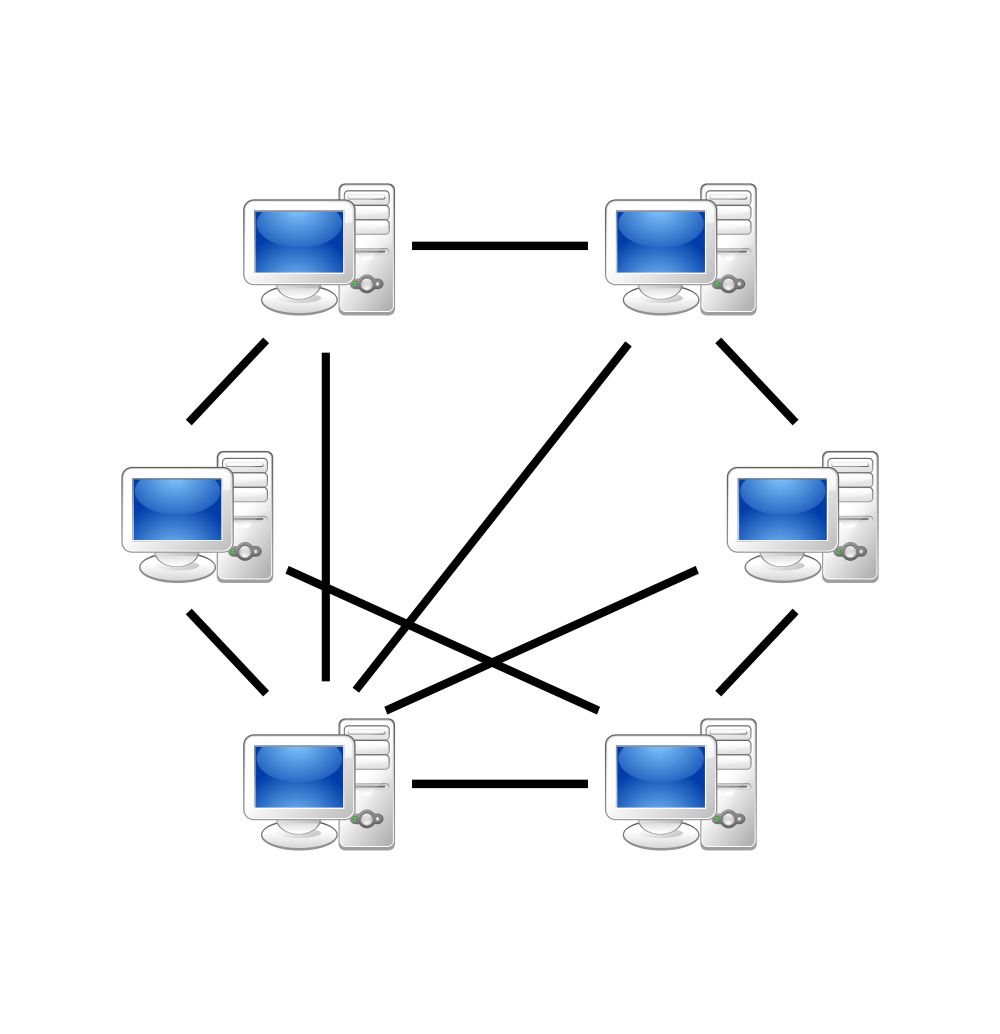
Server

Verarbeiten der Eingaben  
Objekte bewegen

Abbildung 2 - Client / Server Spielengine Architektur

Ein weiterer wichtiger Grund, weshalb diese Variante oft für Multiplayerspiele gewählt wird, ist, dass dem Mogeln durch die Clients (besser unter „cheating“ bekannt) entgegengewirkt wird, indem allein der Server für die Berechnung des Spielstatus ist zuständig ist.

### Peer-to-Peer

In einer Peer-to-Peer-Architektur kommunizieren alle Teilnehmer direkt und ohne Umwege miteinander. Ist die Entfernung dieser nicht allzu gross, fällt auch die Latenz gering aus. Da aber jeder Spieler alle anderen über seine Aktionen informieren muss, fällt der Kommunikationsbedarf weitaus höher aus als in der Client / Server Architektur. In einem reinen Peer-to-Peer-Netz sind alle Computer gleichberechtigt und können sowohl Dienste in Anspruch nehmen, als diese auch zur Verfügung stellen. In modernen P2P-Netzwerken werden die Netzwerkteilnehmer jedoch häufig abhängig von ihrer Qualifikation in verschiedene Gruppen eingeteilt, die spezifische Aufgaben übernehmen.

Anders als beim Client/Server-Prinzip, existiert hier bei Konflikten keine autoritäre Entität, die entscheidet, was richtig, oder falsch ist. Stattdessen müssen die Teilnehmer langwierige Verhandlungen führen (LAMPORT, 2004). Da der Ausfall eines Teilnehmers nicht so schwer wiegt wie der Ausfall eines zentralen Servers, ist die Peer-to-Peer Architektur sehr robust. Cheating ist das grösste Problem in dieser Variante, da ein Cheater (Mogler) die Spielwelt unfair manipulieren kann oder seine Aktionen solange verzögert, bis die Aktionen der Anderen bekannt sind.

Abbildung - Peer-to-Peer Architektur

### Konsistenz & Synchronisation

Wird das Spiel im Singleplayer-Modus gestartet, existieren alle die zu simulierenden Objekte auf dem gleichen Rechner. Obschon die Objekte teilweise auf mehrere Prozesse verteilt werden, sorgt das Betriebssystem dafür, dass das Spiel in einem atomaren Zustand gehalten wird. Inkonsistenzen sind keine möglich, denn es gibt nur eine Simulationsinstanz.

Startet man das Spiel hingegen im Multiplayer-Modus, werden mehrere Simulationsinstanzen gestartet, und zwar so viele, wie sich Spieler angemeldet haben, plus eine für den Server. In dieser verteilten Simulation werden die einzelnen Objekte, die einen Teil der Objektmenge modellieren, auf mehrere Prozesse gesplittet, die einen oder mehrere logische Prozesse beinhalten. Diese wiederum laufen teilweise auf unterschiedlichen Rechnern. Jeder logische Prozess arbeitet seine Objekte sequentiell ab, muss auf Reize von aussen reagieren können und selbst in anderen Prozessen Ereignisse auslösen, da normalerweise die verschiedenen logischen Prozesse nicht unabhängig voneinander sind. Die Kommunikation zwischen den einzelnen Prozessen erfolgt mittels Nachrichten.

Da diese Prozesse parallel laufen, müssen diese so synchronisiert werden, dass keine Kausalitätskonflikte / Inkonsistenzen auftreten können. Eine Inkonsistenz liegt immer dann vor, wenn das Simulationsmodell eine externe Nachricht empfängt, die ein Ereignis in der Vergangenheit des Teilprozesses beschreibt. Konkret bedeutet dies, dass die zwei Simulationsmodelle ihre Objekte unterschiedlich voneinander gerechnet haben, weil die externen Ereignisse nicht korrekt verarbeitet wurden.

### Übermittlungsfehler, Übertragungsgeschwindigkeit und Latenz

Ein weiteres Problem bei der Entwicklung von Multiplayer-Spielen ist die Paketübermittlungskonsistzenz. Die Nachrichten werden über das Netzwerk gesendet. Äussere Störfaktoren können Teile dieser Pakete inhaltlich so verändern, dass sie entweder nicht mehr lesbar sind bzw. keine brauchbaren Informationen enthalten oder ganz zum Verschwinden gebracht wurden. Die häufigsten Gründe für solche Verluste sind:

* Hardwarefehler
* Kollisionen im Netzwerk
* Überlastung

Erschwerend kommt hinzu, dass alle Spieler über unterschiedliche Übertragungsgeschwindigkeiten verfügen. Im vornherein kann nicht festgestellt werden, wer wie viel Bandbreite besitzt. Setzt man nun diesen herunter, damit ein für alle „faires“ Spiel gestaltet werden kann, sinkt damit auch automatisch die maximale Frequenz der Spielaktualisierungen (Nachrichten). Aus diesem Anlass ist hinsichtlich der Nachrichtenübermittlung Sparsamkeit angesagt, um nicht an die Kapazitätsgrenzen der Übertragungsrate zu stossen und somit Paketverluste hinnehmen zu müssen.

Was ausserdem nicht ausser Acht gelassen werden darf, ist die Latenz. Diese beschreibt die Verzögerung einer Nachricht vom Sender zum Empfänger. Der Hauptgrund für diese liegt in den unterschiedlichen Verarbeitungsgeschwindigkeiten der Nachrichten in den beteiligten Systemen (Nachrichten erstellen, Nachrichten empfangen, Nachrichten ausführen), sowie die Anzahl Netzwerke zwischen den Sendestationen (Subnetze). In Multiplayerspielen stellt die Latenz die wichtigste Grösse dar. Studien belegen, dass der Mensch Verzögerungen wahrnimmt, sobald sie jenseits der 100ms Grenze liegen(BAILEY, 1982) (CHEN, et al., 2006).

## Lösungsansätze

Wie vorhin erwähnt, ist der Hauptkern der Multiplayer Spiele die Erhaltung der Konsistenz. Das heisst, dass genau ein Zustand der Spielwelt existieren darf, auf den alle Teilnehmer blicken. Dafür gibt es zwei Ansätze:

Verteiltes Konsistenzmodell:  
Jeder Teilnehmer berechnet das Modell der Spielwelt selbst. Die Teilnehmer setzen jede Aktion sofort um. Damit das Modell bei allen Teilnehmern konsistent ist, darf keine Aktion verloren gehen. In diesem Fall muss der Teilnehmer eine verlorene Aktion erkennen und erneut anfordern, da er sonst Gefahr läuft, nicht mehr konsistent mit den anderen Teilnehmern zu sein. Dieses Model beruht darauf, dass alle Teilnehmer vertrauenswürdig sind, da jeder Teilnehmer das Modell zu seinem Vorteil ändern kann. Das US-Militär verwendet dieses Modell für Simulationen.

Zentrales Konsistenzmodell:  
Bei dieser Technik existiert genau ein Modell der Spielwelt: Kein Teilnehmer berechnet das Modell, stattdessen existiert eine autoritäre Entität, die das Modell für alle Teilnehmer verwaltet und verteilt. Ein Teilnehmer stellt das Modell lediglich graphisch dar und leitet seine Aktionen an die autoritäre Entität weiter. Die Teilnehmer gleichen somit Terminals ("thin client"). Kommerzielle Multiplayerspiele verwenden ausschließlich autoritäre Entitäten, die sicherstellen, dass jeder Teilnehmer das Modell nur innerhalb der zulässigen Regeln ändert  
(Society, 1996).

Beachten wir die im vorgehenden Kapitel erläuterten Strukturen und die Client/Server Architektur, so sieht die Simulation folgendermassen aus:

1. Als erstes erstellt und sendet der Client eine Benutzereingabe an den Server
2. Der Server führt den Befehl aus und sendet alle aktualisierten Positionen dem Client zurück
3. Der Client zeichnet die Szene mit den neuen Objektpositionen

Das Prinzip ist recht simpel, funktioniert aber nicht gut / performant unter realen Bedingungen, da hohe Latenzzeiten in den Internetverbindungen auftreten können. Der Hauptgrund dafür ist, dass der Client in der Tat „dumm“ ist und lediglich die Eingaben aufbereitet und dann auf die verarbeiteten Resultate des Servers wartet. Wenn in der Clientconnection eine Latenz von 500 Millisekunden steckt, dann wird es 500 Millisekunden dauern, bis jede einzelne Aktion des Clients vom Server bestätigt wird und bis die Resultate an den Spieler zurückgelangen. Während eine solche „Round-Trip“-Verzögerung in einem lokalen Netzwerk (LAN) akzeptabel ist, so ist sie es nicht für Internetverbindungen.

Um dieses Problem zu abzuschwächen, wurden eine Vielzahl von Algorithmen entwickelt. Die zwei, die fürFür dieses Projekt verwendeten wir die nachfolgend kurz erklärten.

### TimeWarp-Synchronistation

Das Time-Warp Synchronisationsverfahren (dt. „Zeit-Verzerrung“) stellt eine optimistische Synchronisationsmethode dar, wie sie z. B. zur Synchronisation von verschiedenen Prozessen in einem verteilten System genutzt wird. Time-Warp Synchronisation zeichnet sich – wie andere optimistische Synchronisationsmethoden – durch folgende Dinge aus:

* Alle Prozesse laufen, ohne auf Nachrichten von anderen Prozessen zu warten.
* Treffen Nachrichten aus der Vergangenheit von anderen Prozessen ein, führt dies zu einem Kausalitätsverlust.
* Ein Prozess ist dann in der Lage, einen sog. Rollback durchzuführen, also zum letzten korrekten Zustand zurückkehren und die Berechnungen unter Berücksichtigung der verspäteten Nachricht des anderen Prozesses fortzuführen.

Sobald ein Konflikt auftritt, führt das Verfahren einen Rollback aus. Um einen Konflikt zu erkennen, überträgt Time-Warp mit jeder Nachricht einen Zeitstempel. Das Verfahren prüft die kausale Ordnung aller Nachrichten: Trifft eine Straggler-Nachricht ein, liegt ein Konflikt vor. Time-Warp leitet daraufhin ein Rollback ein und korrigiert die zuletzt ausgeführten Zustandsänderungen durch „incremental state saving“ oder „copy state saving“. Zwar löst Time-Warp einen Konflikt auf, allerdings treten dabei graphische Anomalien auf, beispielsweise Teleportation eines Spielobjekts.

### Clientside-Prediction

Bei der clientseitigen Vorhersage rücken wir ein bisschen vom Konzept des „dummen“ Client oder thin Client ab. Das bedeutet nicht, dass der Client die volle Kontrolle über die Simulation besitzt wie in einem Peer-to-Peer Spiel ohne zentralen Server. Es existiert immer noch eine autoritäre Instanz, ein Server, der die Simulation ausführt. Wenn nun ein Teilnehmer eine Nachricht an die autoritäre Entität sendet, schätzt diese die Latenzzeit zum Teilnehmer ab und berechnet die Spielwelt zum Zeitpunkt zurück, in dem der Teilnehmer die Nachricht losgeschickt hat. Der Client lässt in der Zwischenzeit seine Spielinstanz weiterlaufen und speichert seinen Spielstand kontinuierlich ab.

Unterscheiden sich Server- und Clientsimulation, so korrigiert der Server die clientseitige Berechnung. Da aber grosse Latenzzeiten in der Internetverbindung auftreten können, kann es vorkommen, dass die Änderung erst nach einer ganzen Round-Trip-Zeiteinheit angewendet werden kann. Das führt dann zu deutlich wahrnehmbaren Verschiebungen der Positionen der Spielfiguren.

# Unsere Lösung

## Klassendiagramm

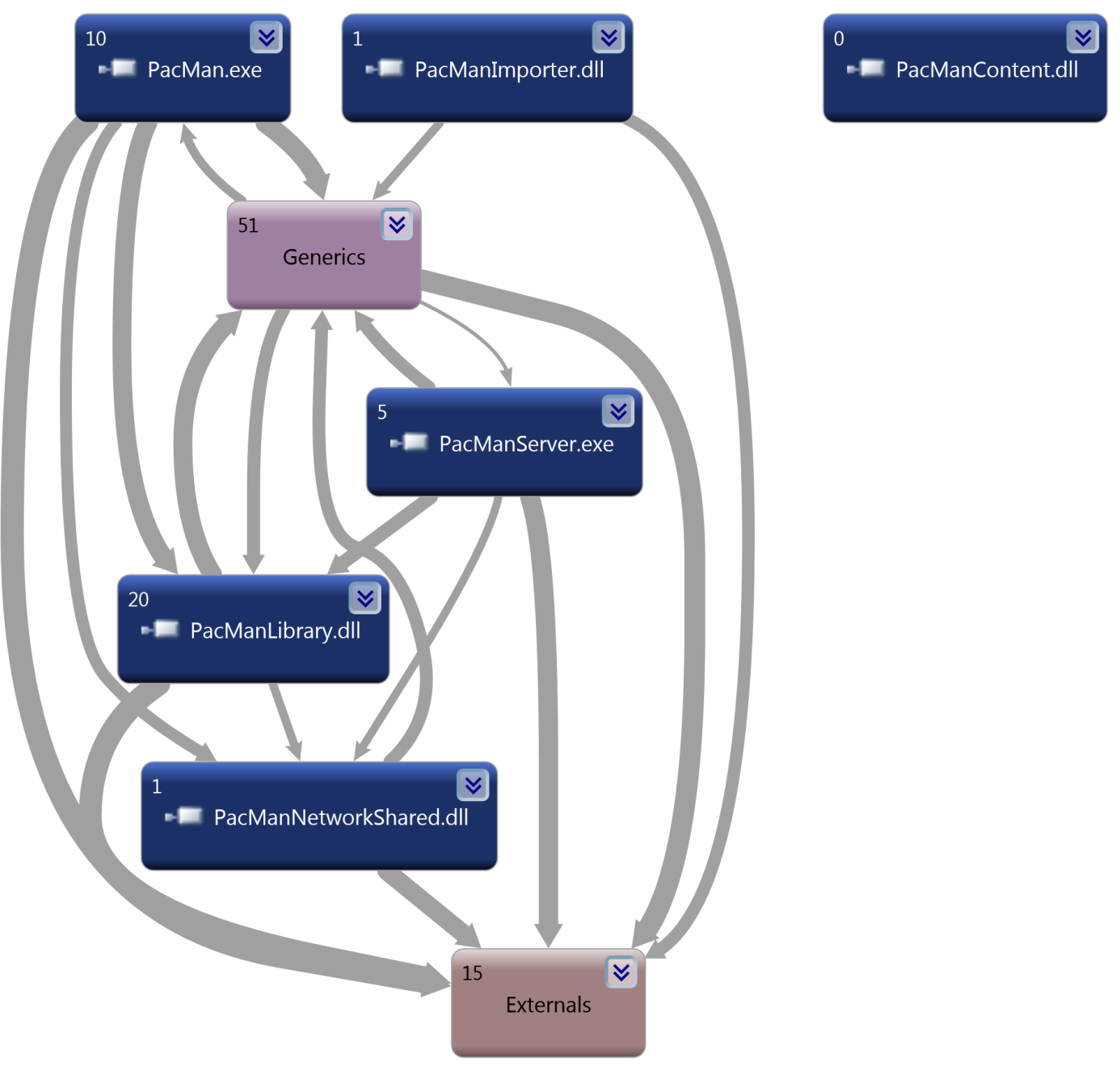


Abbildung 4 - Einfaches Klassendiagramm (nach Assembly zusammengefasst)

Das gesamte Klassendiagramm würde aus Platzgründen den Rahmen sprengen. Abbildung 4 zeigt den groben Zusammenbau der eigentlichen Klassen, die in Assemblies zusammengefasst sind und die einzelnen Abhängigkeiten. Die dunkelblauen sind die eigens programmierten Klassen, während die lila und erdfarben gehaltenen Quadrate lediglich Hilfskonstrukte sind, auf die das Programm zugreift und deren Funktionen es sich bedient. Die kleine weisse Zahl im oberen linken Quadranten weist die Anzahl Klassen aus, die die Assembly in sich einschliesst.

Externals: Wie der Namen schon sagt, beinhaltet dieser Container eine Vielzahl von externen DLL’s, auf die das Programm zugreift. Ganz stark vertreten sind die Bibliotheken des XNA Frameworks, der Lidgren - und Systemlibrary

Generics: Generics enthält all die Klassen und Vehikel, die das .Net Framework zur Verfügung stellt: z.Bsp. LinkedList, Stack, Queue, Collection…

PacManLibrary: Diese DLL ist die Schnittstelle zwischen Client und Server. Sie kapselt alle Klassen bzw. die Funktionen ein, die für den gemeinsamen („shared“) Zugriff verfügbar sein müssen. Um möglichst wenig Redundanz zu generieren und eine gute Wartbarkeit zu garantieren, wurde das Factory-Design-Pattern verwendet.

PacManContent: Diese „Datei“ enthält zu Recht keine Klassen. Sie wurde als Hilfsprojekt für die Entwicklung konzipiert. In ihr befinden sich sämtliche Audiodateien, Bilder, Fonts und Leveldefinitionen.

PacManImporter Der PacManImporter sorgt beim Laden des Spiels für den korrekten Aufbau der Spielwelt. Wie von Beginn an festgelegt, sollte die Möglichkeit bestehen, auch eigene Levels zu kreieren. Im Spiel selbst ist zwar kein GUI implementiert, jedoch kann die Leveldefinition mittels eines CSV-Editors geändert werden. Beim erneuten Starten des Spiels wird das neu definierte Level dynamisch kreiert.

PacManNetworkShared: Das ist das Herzstück der ganzen Netzwerkkommunikation. In dieser Library werden die Nachrichten für den Versand vorbereitet und an die entsprechende Sende- bzw. Empfangsqueue weitergeleitet. Als wichtigste Klassen sollen die INetworkManager und alle versch. MessageHandler erwähnt werden, die weiter unten präziser beschrieben werden.

PacManServer: Diese Assembly unterscheidet sich von den vorhergehenden darin, dass sie eine EXE-Datei ist und somit ausführbar. Sie enthält die Logik des Servers und stellt die empfangenen Informationen als textliches Resultat im Server-Form dar. Das Form dient eher als Informationsfenster für die Problembehebung. Der Server ist auch ausschlaggebend für die Zeitsynchronisation, denn er setzt die ServerGameTime fest. Die wichtigste Aufgabe ist aber die Konsistenzprüfung der einzelnen Spielinstanzen, indem er die empfangen Richtungsänderungen der Clients in einer separaten Simulation durchtestet und bei Erfolg den Clients die Spielschritte validiert mitteilt.

PacMan: Wie der Server ist auch der Client eine ausführbare Datei. Diese beinhaltet das GUI, in dem entschieden wird, ob eine lokale Einzelspielerinstanz gestartet werden soll oder man sich im Multiplayermodus an einem Server anmelden möchte.

## Entscheide

### Technologien

Ein solches Projekt will gut geplant sein, um erfolgreich ein Ziel zu erreichen. Damit alle vom Gleichen sprechen, mussten wichtige technische Entscheidungen gefällt werden, wie zum Beispiel die Programmiersprache und welche Hilfsmittel wir einsetzen wollen.

Die Programmiersprache war praktisch gesetzt durch den Umstand, dass alle drei Teammitglieder keine praktischen Erfahrungen mit Java hatten. Patrick Joos programmierte bereits seit geraumer Zeit professionell mit C# und Patrick Ravi-Pinto hatte gewisse Grundkenntnisse, da er sich bis vor kurzem in der .Net Welt bewegte. Stefan Zahnd besass kein Know-How in C# oder das .Net Framework, eignete sich dieses jedoch, wie sich später zeigen sollte, in sehr kurzer Zeit an.

Weiter sind folgende Technologien im Einsatz: XNA-Framework, Lidgren Netzwerkbibliothek, Visual Studio 2010, TortoiseSVN, Subversion.

#### XNA-Framework

XNA ist eine Technologie zur Spieleentwicklung für Microsoft Windows, Xbox 360, Microsofts MP3-Player Zune sowie Windows Phone 7. XNA vereint verschiedene Spiele-Entwicklungs-Programmierschnittstellen, unter anderem Direct3D aus DirectX Version 9.0c für die Darstellung von 2D- und 3D-Grafiken, XACT als plattformübergreifende Schnittstelle für die Ausgabe von Audiodaten und XINPUT zur Kommunikation mit allen nötigen Peripheriegeräten, in einem gemeinsamen Framework.

Spiele bestehen nur zu einem Bruchteil aus Programmcode. Der Rest wird von Grafiken, Modellen, Sound etc. belegt. Die Handhabung dieser Spieleinhalte war bisher eher aufwändig und wird durch definierte PlugIns und ein passendes Objektmodell, der sogenannten XNA Content Pipeline, massiv vereinfacht. Einfach gesprochen geht es darum, Inhaltsformate, die XNA bereits kennt, automatisch in .NET-Objekte umzuwandeln.

Konkret handelt es sich um eine Sammlung von .NET-Bibliotheken, die vieles, was bisher aufwändig zu implementieren war, in ein leistungsfähiges objektorientiertes Framework packt. Bibliotheken für Grafik, Sound und Mathematik finden sich darin genauso wie Funktionalitäten zur Speicherung von Daten und der Handhabung von Benutzereingaben. Durch XNA werden Entwicklungswerkzeuge, die bislang nur für die Xbox 360 verfügbar waren, auch für Windows-Programmierer verwendbar.

#### Lidgren-Bibliothek

Möchte man Netzwerkspiele oder sonstige netzwerkfähige Anwendungen schreiben, findet man eine ganze Menge guter Bibliotheken für C++ im Internet. Im .NET Bereich ist die Auswahl jedoch nicht gerade gross, dafür gibt es eine ausgezeichnete C#-Bibliothek namens Lidgren, die fast keine Wünsche offen lässt.

Lidgren setzt auf das Netzwerkprotokoll UDP (User Datagram Protocol), baut jedoch auch Charakterisken einer TCP-Verbindung nach, was die Bibliothek sehr flexibel macht. Man kann als Programmierer selber entscheiden, wie wichtig die Daten sind, die man versendet und ob die Reihenfolge beim Empfänger genau stimmen muss.

Der Nachteil des UDP-Protokolls besteht darin, dass Pakete verloren gehen, in einer anderen Reihenfolge und zu allem Überfluss natürlich auch kaputt ankommen können. Alle diese Besonderheiten haben allerdings den Vorteil, dass diese Übertragungsart sehr schnell ist, denn das UDP-Protokoll muss nicht ständig die Reihenfolge der Pakete überprüfen. Auch werden keine Empfangsbestätigungen verschickt und dann ggf. Pakete mehrmals versendet. Lidgren vereint die Stärken von UDP und TCP. Jede Nachricht, die man versendet, bekommt einen Kanal zugewiesen, der den Nachrichtentyp festlegt. Folgende Kanäle sind möglich:

* **Unreliable unordered**  
   Reihenfolge der Nachrichten ist unbestimmt; Nachrichten können verloren gehen.
* **Reliable unordered** Alle Nachrichten kommen an; Reihenfolge ist unbestimmt
* **Sequenced**  
  Nachrichten können verloren gehen; Reihenfolge wird eingehalten
* **Ordered**  
  Alle Nachrichten kommen in der vorgesehenen Reihenfolge an (früher oder später)

Die Netzwerkbibliothek beherrscht darüber hinaus noch weitere Funktionen wie das kontrollierte Drosseln des Netzwerktransfers (Bandwidth Throttling), Verschlüsselung und die Zeitsynchronisation. Auch sehr interessant ist, dass man – speziell zum Testen – einen künstlichen Lag oder eine Wahrscheinlichkeit für verloren gehende Pakete definieren kann.

## Vorgehensweise

##### Lidgren Bibliothek als Netzwerkschnittstelle

Wir entschieden uns bei der Implementierung des Netzwerks auf die Lidgren Bibliothek zurückzugreifen. Es handelt sich dabei um eine opensource Netzwerkbibliothek für das .Net Framework, die eine einfache API für das Verbinden eines Clients an einen Server sowie das Senden und Empfangen von Nachrichten bietet. Wir entschieden uns für diese Bibliothek, da es uns sinnlos erschien, das Rad neu zu erfinden, zudem die Entwicklung einer Netzwerkbibliothek nicht Kernaufgabe des Projekts war und für den Einsatz in einer Client/Server Umgebung bereits Dokumentationen im Internet existierten.

##### Funktionsweise der Bibliothek

Damit einige Vorgänge besser verstanden werden, hier einige Erläuterungen zu der Funktionsweise der Bibliothek.

Bevor Nachrichten versendet bzw. empfangen werden können, müssen Server und Client gestartet werden. Dazu erzeugt man zwei Objekte, den NetClient und den NetServer. Bei der Initialisierung des Servers muss eine Server-ID (zum Beispiel "pacman") und ein Port, auf welchem zugehört wird, angegeben werden. Der Client benötigt nur die gleiche ID wie der Server, mit dem er kommunizieren will.

Es wäre möglich, Nachrichten an den Server zu senden, ohne vorher eine Verbindung aufgebaut zu haben. Wir entschieden uns jedoch für die andere Variante mit Verbindungsaufbau. Dies ermöglicht uns, auf dem Server eine Liste nachzuführen, in der jeder Client mit seiner zugehörigen Verbindung (connection) und der eindeutigen ID, die aus den Verbindungsparametern generiert wird, gespeichert wird. So können wir Nachrichten dediziert an einzelne Clients schicken, was die Netzwerklast reduziert. Um eine Verbindung zum Server aufzubauen, sendet der Client eine Discovery-Anfrage an die Broadcastadresse (\*.255) und den entsprechend definierten Serverport, den angegeben werden muss und wartet auf eine Anwtort. Man könnte auch eine explizite IP-Adresse angeben, jedoch verzichteten wir darauf, um das Handling für den Benutzer möglichst einfach zu halten. Die dadurch entstandene Limitierung des Einsatzbereiches (lokales Subnetz) nahmen wir in Kauf. Der Server empfängt nun diese Anfrage und sendet dem Client alle nötigen Informationen, um eine Verbindung herzustellen. Erhält der Client diese Nachricht, verbindet er sich mit dem Server. Dieser Schritt wird nicht automatisch ausgeführt, sondern muss entsprechend behandelt werden.

Um Nachrichten untereinander auszutauschen, muss man lediglich eine leere Nachricht erstellen, diese mit Daten abfüllen und absenden. Das Erstellen der Nachricht ist auf dem Server und Client gleich, beim Versenden benötigt der Server zusätzlich zur Nachricht und der Versandart die Verbindungs-ID des Empfängers. Das Versenden von Nachrichten an alle verbundenen Clients (Multicast) mit einem Befehl ist nicht möglich, sondern muss mit einer Schleife gelöst werden. Das Abfüllen der Nachricht mit Informationen ist sehr simpel. Mit dem Befehl Write() können Strings, Integers, Bytefolgen etc. hinzugefügt werden. Zum Auslesen der Inhalte gibt es für jeden unterstützten Datentyp wie String, Integer, Double etc. Methoden, die nur diesen Bereich der Nachricht auslesen. Folgendes Codebeispiel soll dies verdeutlichen:

//Client

NetClient client = new NetClient(new NetPeerConfiguration(„pacman“));

..

NetOutgoingMessage msg = new client.CreateMessage();

msg.Write(„Hallo Welt“);

msg.Write(1234);

client.SendMessage(msg, NetDeliveryMethod.ReliableOrdered);

//Server (die Initialisierung des Servers wird weggelassen)

string msgStr = msg.ReadString(); //msgString hat den Wert „Hallo Welt“

int msgInt = msg.ReadInteger32(); //msgInt hat den Wert 1234

Codebeispiel 1 - Einfaches Senden und Empfangen mit Lidgren

Damit der Inhalt der Nachrichten schlussendlich in der Anwendung verwendet werden kann, müssen die Nachrichten vom NetClient-, oder NetServer-Objekt abgerufen werden. Beide Objekte scheinen im Hintergrund ebenfalls eine Queue zu betreiben, in denen die eingegangenen Nachrichten aufbewahrt werden. Um die Nachrichten auszulesen, verwendet man den Befehl ReadMessage(). Bei einem Aufruf wird eine Nachricht von der Queue geholt. Somit muss der Befehl in einer While-Schlaufe ausgeführt werden, damit die eingehenden Nachrichten immer an die Anwendung weitergeleitet werden. Sobald die Queue jedoch leer ist, endet die Schlaufe und neue Nachrichten werden nicht mehr an die Anwendung weitergereicht, wohl aber empfangen, da der NetClient bzw. NetServer als Thread im Hintergrund laufen. Darum haben wir die IncomingMessageHandler implementiert, welche weiter unten beschrieben werden.

Für die Umsetzung der Clientside-Prediction benötigten wir die Möglichkeit, Objekte über das Netzwerk zu versenden. Nun kann man Lidgren nicht einfach ein Objekt mit der Write() Methode übergeben, dafür aber Bytefolgen. Somit muss das Objekt zuerst serialisiert werden. Dies implementierten wir in der Abstrakten Klasse INetworkManager, da die Methoden Serialize() und Deserialize() auf dem Client und Server gleich sind. Beim serialisieren wird das Objekt in den Arbeitsspeicher geschrieben und dabei der Stream aufgezeichnet. Nun kann man das Objekt wie gewohnt mit Write() in eine Nachricht schreiben, da es sich neu um eine Bytefolge handelt. Beim deserialisieren wird gleich verfahren. Die ganze Nachricht wird in den Arbeitsspeicher geschrieben und der Stream aufgezeichnet. Da wir jetzt aber zusätzlich zum serialisierten Objekt, auch noch einen Identifikator der Nachricht mitgeschickt haben (wir identifizieren die Nachrichten mit Strings, zum Beispiel "newDirection" für einen Directionupdate), muss dieser von der Bytefolge abgezogen bzw. da dieser am Anfang der Nachricht steht, mit einem Offset übersprungen werden. Anschliessend kann die Bytefolge gecastet und als das ursprüngliche Objekt verwendet werden.

##### INetworkManager

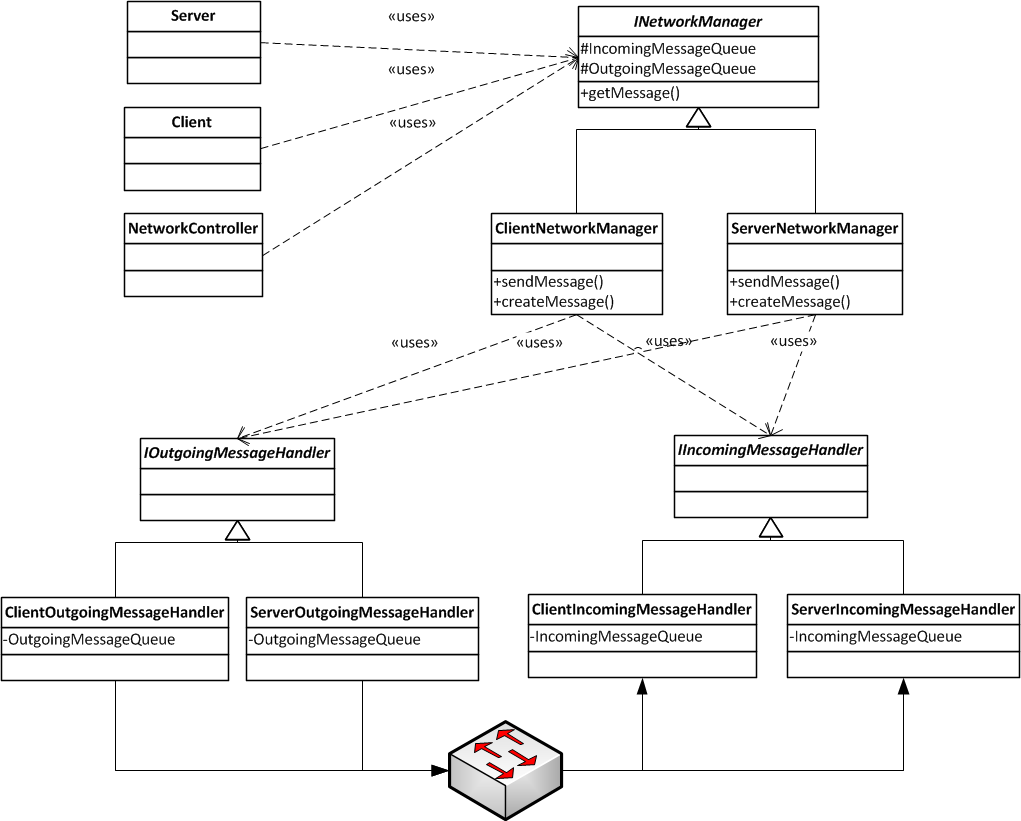
Das Herzstück der Kommunikation unserer Applikation basiert auf zwei Warteschlangen, die gesendete bzw. empfangene Nachrichten enthalten und unabhängig vom Spielcode abgearbeitet werden. Eine Eigenschaft der Lidgren Bibliothek, dass Client und Server unterschiedliche Objekttypen (der Server ist vom Typ NetServer und der Client vom Typ NetClient) haben, und Vorgänge wie Nachrichten senden, empfangen oder erstellen jeweils mit dem entsprechenden Objekt ausgeführt werden müssen, zwang uns dazu, die Komponenten des NetworkManagers in doppelter Ausführung zu implementieren, da er sowohl auf dem Client, als auch auf dem Server eingesetzt werden soll. Dies konnten wir mit Hilfe des Strategy Patterns und Abstrakten Klassen relativ gut lösen. Client und Server verwenden nun dieselbe Abstrakte Klasse INetworkManager.

Abbildung - Schematischer Klassenaufbau der Netzwerkinfrastruktur

##### Die Message Handler

Die Message Handler sind dafür zuständig, Nachrichten, die vom Client oder Server abgeschickt oder empfangen werden, in die entsprechenden Queues zu legen, bzw. diese zu entfernen. Die Queue für die eingehenden Nachrichten ist beim Server und Client gleich und enthält ein Element, die Nachricht. Die Queue für ausgehende Nachrichten hingegen ist unterschiedlich. Beim Client reicht eine einfache Queue mit der Nachricht als Element, da er für das Senden keine Angabe der Verbindungs-ID benötigt. Der Server muss jedoch zum Versenden des Pakets die Verbindungs-ID des Empfängers kennen. Um dies zu realisieren, haben wir die Queue nicht mit einem Element abgefüllt, sondern mit einem KeyValuePair bestehend aus der Nachricht und der Verbindungs-ID. Zuerst implementierten wir beide Komponenten als Threads, die ständig aktiv waren und mit einer unendlichen While-Schlaufe auf neue Nachrichten in den NetClient- und NetServer-Queues prüften. Dabei ergab sich das Problem, dass die Systemresourcen, während dem das Spiel lief, total ausgeschöpft wurden. Uns erschien das sehr merkwürdig. Es konnte ja nicht sein, dass eine solch kleine Anwendung so viel Ressourcen braucht. Es stellte sich heraus, dass wir nicht berücksichtigen, dass die While-Schlaufe im Thread die Ausführung der anderen Threads (das Spiel) störten, da sie nie pausierten. Als Folge änderten wir die Message Handler so ab, dass für ausgehende Nachrichten nur im Fall des Einfügens neuer Nachrichten in die Queue ein Thread gestartet und nach der Verarbeitung der Nachrichten wieder beendet wird. Dem Thread für die eingehenden Nachrichten sagten wir einfach, dass er nach jeder Ausführung kurz pausieren soll.

##### Die NetworkManager

In den Networkmanager werden die Sendemethoden implementiert und die Nachrichten an die entsprechenden MessageHandler übergeben. Hier wird auch der UID für den Client erzeugt. Wie zuvor bereits erwähnt, besteht dieser aus Verbindungsparametern, genauer aus einem Teil des Unique-Identifiers des NetClient-Objekts. Die UID wird verwendet, um die Spielobjekte PacMan und Ghost eindeutig zu identifizieren. Da dieser Wert bei jedem Directionupdate über das Netzwerk übertragen wird, haben wir die Länge des ursprünglichen Identifiers auf die letzten fünf Zahlen verkürzt, um die Paketgrösse geringer und somit die Netzwerklast niedriger zu halten.

##### Server/Client Kommunikation

Unser Konzept, wie der Client mit dem Server und umgekehrt kommuniziert, hat sich eigentlich nach und nach entwickelt und wurde vorher nicht sauber geplant. Klar bildet das nicht die besten Voraussetzungen für eine sauber programmierte Anwendung, jedoch muss man berücksichtigen, dass in unserer Gruppe kaum Know-How über Client/Server Programmierung existierte und nicht alle Mitglieder über viel Programmiererfahrung verfügten, da sie im beruflichen Alltag nichts damit am Hut haben.

Nun zur Umsetzung: Als Server entschieden wir uns für ein simples Formular. So halten wir den Sourcecode schlank und haben die Möglichkeit, Servernachrichten zur Laufzeit auszugeben. Der Server verfügt über zwei Timer, der eine ist für das Handling der Nachrichten zuständig und der andere für die Simulation des Spiels. In diesem Abschnit wird lediglich auf den Nachrichten-Timer eingegangen. Das Prinzip ist einfach: Der Timer wird alle 100 Milisekunden ausgeführt und bei jedem Durchlauf verarbeitet er alle Nachrichten, die in der Queue vorhanden sind. Die 100 Milisekunden sind ein Standardwert und haben keine spezielle Bedeutung. Die ausgelesene Nachricht wird nach ihrem Nachrichtentyp unterschieden und dementsprechend behandelt. Nachrichten, die Kommunikationsdaten enthalten, sind vom Typ NetIncomingMessageType.Data. Zwei weitere wichtige Typen sind "StatusChanged", wenn sich ein Client verbunden hat und "DiscoveryRequest", wenn der Client nach einem Server sucht. Jeder Nachricht geben wir einen Identifikationsstring mit, der bestimmt, wozu das Paket gedacht ist, zum Beispiel "startGame", um den Spielstart zu initialisieren, "newDirection" für ein Direction Update eines Spielobjektes oder "syncTime", um den Zeitunterschied zwischen Client und Server zu ermitteln (dazu später mehr). Einige Erläuterungen zu den jeweiligen Vorgängen:

##### DiscoveryRequest

Dem anfragenden Client wird eine Discovery-Antwort gesendet, welche alle Informationen enthält, damit sich der Client verbinden kann. Dies wird von der Lidgren Bibliothek gehandhabt.

##### syncTime

Auf allen Systemen (Server und Client) laufen unterschiedliche Uhren (beginnen beim Programmstart zu laufen). Für die Clientside-Prediction verwenden wir die Serverzeit als Referenz und daher müssen sich alle Clients mit dieser Zeit abgleichen. Die Zeit auf den Clients wird dabei nicht angepasst, sondern ein Offset zur Serverzeit berechnet, welcher jeweils auf die TimeStamps vom Server (Spiel wird auf dem Server mit der Serverzeit simuliert) angewendet wird, um den Zeitunterschied anzupassen. Dies geschieht ganz einfach, indem wir dem anfragenden Client die aktuelle Serverzeit als TimeSpan (Objekt, das die aktuell verstrichene Programmzeit enthält) schicken. Der Client vergleicht anschliessend seine eigene Zeit und die des Servers. Wird der Client vor dem Server gestartet, so ergibt dies einen negativen Offset, sonst einen positiven. Die Übertragungszeit des Pakets wird dabei auch berücksichtigt. Nacheinander werden fünf Anfragen an den Server gesendet, wobei jedes Mal die Sendezeit gespeichert wird und beim Empfangen der Antwort von der aktuellen Zeit abgezogen wird. Dies ergibt die gesamte Übertragungszeit. Diese Zeit wird halbiert, da der Server die Antwort und somit seine Systemzeit nach Ablauf der Hälfte der gesamten Übertragungszeit abschickt. Nachdem alle Nachrichten empfangen und die jeweiligen Übertragungszeiten addiert wurden, wird der Durchschnitt berechnet, was die benötigte Sendezeit ergibt, die mit dem Offset verrechnet wird. Dieser Vorgang wird während dem Spiel alle 10 Sekunden periodisch wiederholt.

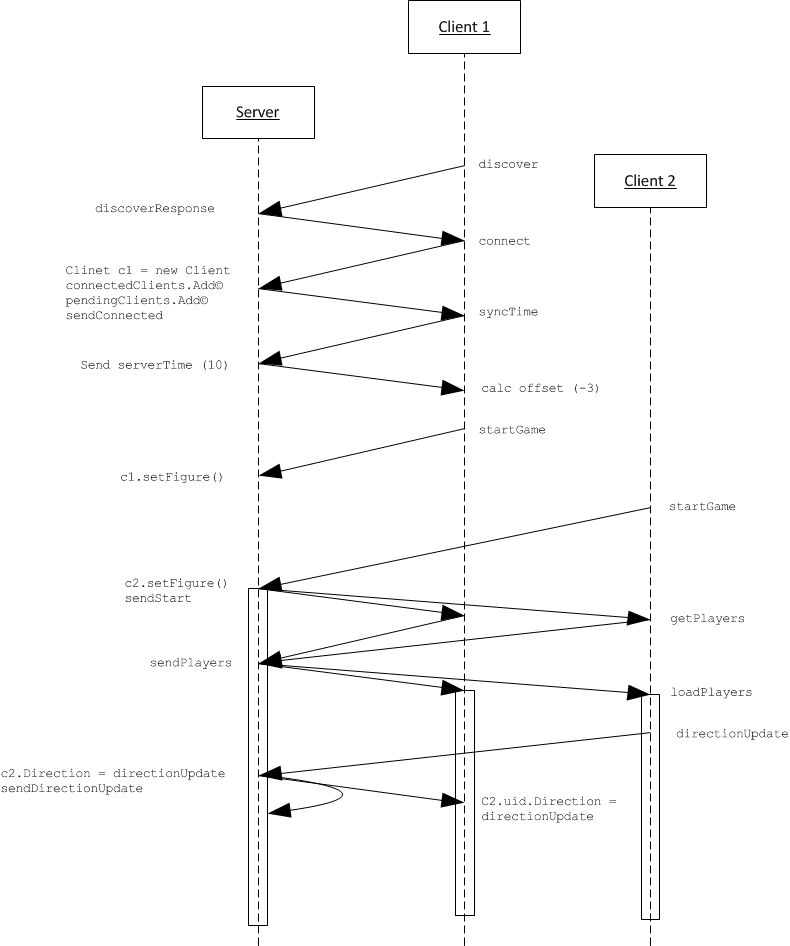


Abbildung 6 - Sequenzdiagramm Client/Server-Kommunikation

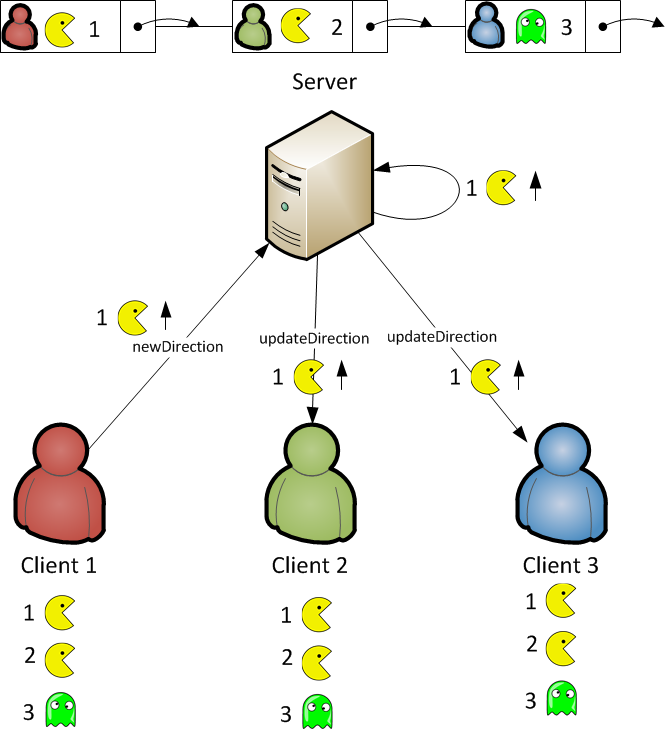
##### startGame

Sobald der Client alle Einstellungen für sein Multiplayerspiel gemacht hat und das Spiel startet, wird der Server informiert. Dabei wird auch gleich der Typ des Spielobjekts, das der Client gewählt hat, an den Server übertragen und dort in der Liste der verbundenen Clients aktualisiert. Nachdem der Server das Spielobjekt bei seiner Simulation erzeugt hat, wird der entsprechende Client aus der Liste der pendenten Clients entfernt. Es gibt keine Rückmeldung an den Client. Dieser muss nun warten, bis alle anderen Clients ebenfalls das Spiel gestartet haben. Der Client wird beim Verbinden mit dem Server in die Liste der pendenten Clients eingetragen. Hat jeder Client das Spiel gestartet und ist somit die Liste der pendenten Clients leer, wird eine Startnachricht an alle Clients gesendet, die diesen befiehlt, das eigentliche Spiel zu starten.

##### getPlayers

Wenn der Client das Spiel nun startet, wird zuvor beim Server angefragt, wie viele andere Spieler existieren. Der Server sendet jedem Client eine Liste der Spieler mit UID und Spielobjekttyp. Diese Information ist als reiner String mit Trennzeichen (; und /) kodiert und muss auf dem Client entpackt werden. Der Client kann so alle Spielobjekte initialisieren und das Spiel starten.

##### newDirection

Bei jedem Richtungswechsel eines Spielobjekts wird diese Information an den Server weitergeleitet. Dieser aktualisiert wiederum seine eigenen Spielobjekte in der Simulation und sendet anschliessend die Aktualisierung an alle Clients, ausser dem Sender. Dazu iteriert er durch die Liste verbundener Clients und vergleicht die Verbindungsparameter. Auf dem Client wird nur dasjenige Objekt aktualisiert, für das das Update gedacht ist (Identifikation mittels UID).

##### NetworkController

Jedes Spielobjekt besitzt einen NetworkController bzw. einen PlayerController (je nachdem ob es sich um ein lokales, oder netzwerkbasiertes Spiel handelt), der das Objekt steuert und die Directionupdates an den Server übermittelt, empfängt und anwendet. Zu Beginn hatten wir die Methode, welche die Directionupdates empfing und dem Controller übergab, im Controller selbst implementiert. Dabei stiessen wir auf das Phänomen, dass sich bei allen Clients nur dasjenige Spielobjekt bewegte, das dem Spieler entsprach, der sich als erstes mit dem Server verbunden hat. Die Ursache des Problems lag darin, dass es pro Client so viele Controller wie Spielobjekte gibt. Wenn jetzt ein Controller eine Nachricht aus der Queue herausnimmt, ist diese nur für diesen Controller, also auch nur dieses Objekt verfügbar. Stimmt der UID des Controllers mit demjenigen des DirectionUpdates aus der Nachricht überein, wird die Aktualisierung angewendet und sonst nicht. Ist letzteres der Fall gewesen, war der Update für den richtigen Controller nie sichtbar. Daher haben wir die Methode eine Stufe nach oben, in den MultiplayerScreen verschoben.

Abbildung - Richtungsupdate eines Objekts

# IST-Zustand

Der Singleplayer Modus ist ohne Einschränkungen spielbar. Die Geister jagen, je nach ihrer programmierten künstlichen Intelligenz, den PacMan und machen ihm das Leben schwer. Die Kekse (Crumbs) werden gefressen und sorgen für das Anwachsen des Highscore. Die roten Powerpillen setzen PacMan in einen Sonderstatus, so dass er 50% schneller laufen kann. Dieser Effekt endet nach XX Sekunden. Ab und an erscheint eine Kirsche in der unteren Spielfeldhälfte. Wird diese gefressen, erhält der Spieler weitere XX Punkte auf sein Konto gutgeschrieben.

Gemäss aktueller Implementation existiert keine Einschränkung der Anzahl Teilnehmer. Man müsste jedoch genau prüfen, ab wie vielen Spielern die Netzwerkbelastung bzw. die Datenmenge für den Server zu gross wird. Die Lidgren-Bibliothek stellt die Simulation eines langsamen Netzwerks von Haus aus zur Verfügung. Aus Zeitgründen haben wir jedoch darauf verzichtet dem nachzugehen.

Ein weiteres Verhalten, dem nachgegangen werden muss, ist die Tatsache, dass der Programmprozess weiter läuft, obwohl das Spielfenster geschlossen wurde (Client). Der Prozess muss von Hand über den Task-Manager geschlossen werden. Wo die Ursache dieses Problems liegt, konnten wir noch nicht klären. Eventuell könnte es damit zusammenhängen, dass das Spiel während der Ausführung relativ viele Systemressourcen verschlingt oder einige Threads nicht ordnungsgemäss beendet werden. Die Ursache dessen wurde noch nicht geklärt.

Folgende Features sind nur teilweise implementiert oder pendent:

* **Geister im Multiplayer**: Die funktionierende Implementierung vom singleplayer-Modus muss noch in den multiplayer-Modus übertragen werden.
* **TimeStamp**: Das TimeStamp-Objekt wird auf dem Client und Server ordnungsgemäss erstellt, die Pakete können auch zwischen den Computern korrekt übertragen werden (Serialisierung und Deserialisierung funktionieren) und der Algorithmus zur korrekten Simulation ist auch implementiert. Die Zuweisung des TimeStamps auf den aktuellen Spielzustand des Clients funktioniert jedoch nicht. Nach stundenlangem debuggen haben wir die Lösung leider noch nicht gefunden.
* **Spielverhalten bei Ende**: Was passiert, wenn alle Punkte aufgefressen werden oder ein Spieler keine Leben mehr hat?

Obschon die Ausführung sehr minutiös geplant wurde, ergaben sich aus verschiedensten Überraschungen und Unbekannten Zeitverzögerungen, die unmöglich vorausplanbar gewesen wären. Dies brachte das Projekt in den Rückstand und gewisse Dinge konnten nicht mehr fertig implementiert werden.

# Ausblick

Aus Zeitgründen konnten nicht alle Kriterien wunschgemäss implementiert werden. Die geschätzte Zeit, diese noch zu Erledigen, ist auf eine Woche veranschlagt. Ist dies geschafft, liegt eine PacMan Version vor, die wahrlich vielen anderen das Wasser reichen kann. Durch minimale Änderungen ist es zudem möglich, das Spiel so zu modifizieren, dass dieses auf der Microsoft Xbox Spielkonsole läuft. Abgesehen von markenschutzrechtlichen Belangen, die zur Zeit noch nicht abgeklärt wurden, könnte das Spiel sogar in den Xbox Live Marktplatz gestellt werden, und gegen eines kleinen Entgeltes von der Öffentlichkeit heruntergeladen werden.   
Der Träumer der Gruppe rechnete vor, dass, wenn nur 10% aller Xbox Besitzer dieses Spiel zu einem Franken kaufen würden, dem Team eine Summe von 2‘000‘000 Franken zur Verfügung stünde.

# Fazit

##### Patrick Joos:

Dem Projekt fehlte anfangs eine konkrete Richtung. Wir legten sehr schnell mit der Entwicklung los, ohne einen konkreten Plan zu erstellen. Durch unser Ziel erst ein funktionierendes Spiel und danach die Netzwerkfunktion einzubauen, waren wir zu stark auf das Spiel selbst fokussiert. Wir versuchten aber das Spiel leicht erweiterbar zu machen, was es uns letzten Endes den Einbau der Netzwerkfunktion erheblich erleichterte. Eine Woche mehr und es wäre perfekt gewesen.  
Ich habe bei dem Projekt viel gelernt, was Software-Design und Projektplanung angeht und hoffe die Fehler bei diesem Projekt nicht mehr zu machen. Ich bin insgesamt sehr zufrieden mit dem was wir erreicht haben.

##### Patrick Ravi-Pinto:

Als ehemaliger VB.Net Entwickler war mir die eigentliche Programmierarbeit nicht ganz fremd. Ich kannte zwar C# erst seit einem Monat, aber die .Net Grundprinzipien ändern sich nicht. Unterschätzt wurde meinerseits die Einbindung des XNA Frameworks. Ich war oft froh, dass ich Patrick Joos fragen konnte, wenn Unsicherheiten auftraten - und von denen gab es eine Menge. Die Lösungsansätze waren im Internet sehr gut beschrieben, diese jedoch dann zu implementieren erwies sich komplizierter als erwartet. Ich glaube, wir hatten einfach viel zu hohe Erwartungen und Wünsche, was wir alles implementieren wollten und verzettelten uns in den Details. Eine Feinplanung mit Schätzungen, wie lange wir für welche Funktionen bräuchten, hätte uns sehr schnell gezeigt, wenn der Zeitplan des Projekts aus dem Ruder gelaufen wäre.  
Ich habe enorm viel gelernt: sowohl im Projektmanagement wie auch in den grundlegenden Synchronisations- bzw. Konsistenzwahrungsmechanismen. Diese Erkenntnisse kann ich in Zukunft auch auf andere, nicht rein spielorientierte Projekte anwenden. Ich möchte auch meinen zwei Projektkameraden für die tolle Zeit danken.

##### Stefan Zahnd:

Ich habe mit Programmieren im Alltag praktisch nichts am Hut und hatte daher zu Beginn des Projekts etwas Bange, ob das nicht in einem Desaster endet, da wir zu allem anderen, noch entschieden, das ganze in einer Sprache zu entwickeln, die ich zuvor noch nie gesehen hatte. Somit bestand das Projekt für mich zu 60% aus lernen und 40% implementieren ☺. Patrick Joos, als erfahrenerer Entwickler, übernahm zu Beginn den Lead und implementierte relativ schnell die Grundlogik des Spiels. Das war für mich eine grosse Hilfe, da ich seinem Programmcode viel abschauen konnte und er mir bei Fragen auch immer zur Seite stand. Wo ich etwas mehr Know-how als die anderen hatte war das Thema Netzwerk. Somit konnte ich mich dort gewinnbringend integrieren und die Netzwerkkommunikation zwischen Client und Server implementieren. An die ersten Wochen kann ich mich noch gut erinnern, bei fast jedem Problem musste ich bei Joos nach der Lösung fragen. Gegen Ende war das Verständnis grösser und ich konnte die meisten Probleme selbst bewältigen. Dabei gab es einige Aha-Momente. Zum Beispiel setzten wir einige Design Patterns ein, die wir im vorigen Semester behandelt hatten und ich damals keine Vorstellung davon hatte, wozu die gebraucht werden könnten. In diesem Semester habe ich im Bereich programmieren so viel gelernt, wie im ganzen vorherigen Studium nicht. Daher möchte ich abschliessend sagen, dass ich mit dem Resultat, obwohl das Spiel nicht fertig ist und den gemachten Erfahrungen sehr zufrieden bin.

## Zusammengefasst

Unsere Erfahrungen waren in diesem Projekt unterschiedlich und konzentrierten sich in verschiedenen Kernpunkten. Obschon unser Hauptprogrammierer ein erfahrener Coder ist, starteten wir in der Thematik „Konsistenzwahrung und Synchronisation“ mit einer Tabula rasa. Wir stellten uns alles viel einfacher vor. Bereits nach dem Projektstart stellten sich die ersten Fragen. Da aber zu diesem Zeitpunkt noch keine Synchronisation vonnöten war, hielten sich diese in bescheidenem Ausmass. Stefan Zahnd musste sich zuerst in C# einlesen und Patrick Ravi-Pinto haderte mit dem XNA Framework. Grobe Meilensteine waren zwar gesetzt, aber weder Feinplanung noch Kontrolle eben dieser Meilensteine wurden je gemacht. Als die Netzwerkfunktionalität implementiert werden musste, konnte Stefan Zahnd mit seiner beruflichen Erfahrung auftrumpfen. Gewisse unerklärliche Phänomene liessen bei der Suche eben derer die schon spärliche Zeit schmelzen und wir kamen in Zeitnot. Aus diesem Grunde konnten auch nicht alle Punkte aus der Wunschliste implementiert werden.

Es war aber nicht alles so schwarz wie es sich vielleicht jetzt gerade anhört. Wir sind überzeugt, im Projektmanagement- und Zeitmanagementbereich einiges gelernt zu haben. Wir hatten sehr viel Spass und konnten endlich das in den vorigen Semester erlernte praktisch anwenden.

# ANhang

## Literaturverzeichnis

**BAILEY Robert W.** Human Performance Engineering: A Guide for System Designers [Book]. - [s.l.] : Prentice Hall Professional Technical Reference, 1982. - p. 672. - ISBN 0-1344-5320-4.

**BOOTH SIMPSON Zachary** A Stream-based Time Synchronization Technique For Networked Computer Games [Online]. - März 1, 2000. - http://www.mine-control.com/zack/timesync/timesync.html.

**CHEN Kuan-Ta, HUANG Polly and LEI Chin-Laung** How sensitive are [Article] // Communication of the ACM. - November 2006. - Nr. 11 : Vol. Volume 49. - pp. 34-38.

**FIEDLER Glenn** Glenn Fiedler's Game Development Articles and Tutorials [Online] // What every programmer needs to know about game networking. - Januar 24, 2010. - http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/.

**GAUTIER Laurent and DIOT Christophe** MiMaze Publications [Online] // The Multicast internet Maze. - Juni 28, 1998. - http://www-sop.inria.fr/rodeo/MiMaze/ReportInt.html.

**LAMPORT Leslie** Recent Discoveries from Paxos [Conference] // International Conference on Dependable Systems and Networks (DSN). - 2004. - p. 3.

**Society Standards Committee on Interactive Simulation (SCIS) of the IEEE Computer** IEEE standard for distributed interactive simulation communication services and profiles [Patent] / ed. SOCIETY IEEE COMPUTER and Simulation USA Standards Committee on Interactive. - 1996.

**SWEENEY Tim** Advanced UnrealScript Programmers, C++ Programmers [Online]. - Juli 21, 1999. - http://unreal.epicgames.com/Network.htm.

**UNBEKANNT** Lidgren Netzwerkbibliothek [Online] // Lidgren Netzwerkbibliothek. - Juni 26., 2008. - http://georgwaechter.wordpress.com/2008/06/26/lidgren-netzwerkbibliothek/.

## Bilderverzeichnis

[Abbildung 2 - Client / Server Spielengine Architektur 5](#_Toc283668648)

[Abbildung 1 - Client / Server Architektur 5](file:///C:\PacMan\Dokumentation\Projektbeschrieb.docx#_Toc283668649)

[Abbildung 3 - Peer-to-Peer Architektur 6](file:///C:\PacMan\Dokumentation\Projektbeschrieb.docx#_Toc283668650)

[Abbildung 4 - Einfaches Klassendiagramm (nach Assembly zusammengefasst) 10](#_Toc283668651)

[Abbildung 5 - Schematischer Klassenaufbau der Netzwerkinfrastruktur 16](file:///C:\PacMan\Dokumentation\Projektbeschrieb.docx#_Toc283668652)

[Abbildung 6 - Sequenzdiagramm Client/Server-Kommunikation 19](#_Toc283668653)

[Abbildung 7 - Richtungsupdate eines Objekts 20](file:///C:\PacMan\Dokumentation\Projektbeschrieb.docx#_Toc283668654)

## Code Snippets

[Codebeispiel 1 - Einfaches Senden und Empfangen mit Lidgren 15](#_Toc283631921)

## Revisionsverlauf

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Description** | **Author** |
| **0.1** | **01122010** | **Erster Entwurf, Vorlage** | **Ravi-Pinto** |
| **0.2** | **17122010** | **Ausarbeitung der einzelnen Details** | **Ravi-Pinto** |
| **0.3** | **17012010** | **Technologiebegriffe** | **Ravi-Pinto** |
| **0.4** | **20012010** | **Netzwerkmanager** | **Zahnd** |
| **0.5** | **21012010** | **Fazit** | **Zahnd / Joos** |
| **0.6** | **23012010** | **Schlussschliff & Korrektur** | **Ravi-Pinto / Zahnd** |