Semesterprojekt „Kommunizierende Systeme“

„Lasertag – Nerd Style“

Wintersemester 2017/2018

Humboldt-Universität zu Berlin
Lehrstuhl für Technische Informatik

Abschlussbericht

Teilnehmer:
David Bachorska
Kevin Cornelius
Rafael Robert Hadamik
Angelina Jellinek
Pascal Jochmann
Tom Kieseling
Dennis Ness
Manuel Radatz
Tim Sikatzki
Jan Arne Sparka
Kevin Marc Trogant
Lennart Weiß

Rev.: PDF-Dokument erstellt am 16. März 2018 um 21:12 Uhr
(Dieser Hinweis sollte vor der Abgabe noch entfernt werden!)

Inhaltsverzeichnis

1 Einleitung

Das Semesterprojekt „Kommunizierende Systeme" wurde vom Lehrstuhl der Technischen Informatik an der Humboldt-Universität zu Berlin organisiert und umfasste 12 studentische Teilnehmer. Das Projekt begann im Oktober 2017 und die Endpräsentation fand am 16. Februar 2018 statt.

Das Ziel war es, das populäre Spiel Lasertag mit modifizierten Regeln nachzubilden. Die besondere Anforderung war dabei, dass die Spielart der Funktionsweise eines Peer-to-Peer-Netzwerkprotokolls entsprechen sollte. Der Entwurf und die Funktionsweise des technischen Systems, das dies bewerkstelligt, wurde komplett den Teilnehmern überlassen.

2 Anforderungsanalyse

Bereits die Forderung, dass das Spiel Lasertag nachgebildet werden soll, hatte weitreichende Konsequenzen. Wie im Original muss jeder Spieler ein Gerät bekommen, mit dem er andere Spieler abschießen kann und entsprechend muss ein Abschuss detektiert werden. Die Spieler sollen dabei möglichst mobil und frei beweglich sein. Daraus resultiert, dass irgendeine Form von Sender-Empfänger-Kombination verwendet werden muss.

Implizit war auch der Spielspaß eine Anforderung. Dies ist weniger eine technische Anforderung. Es verlangt eher, dass die technische Funktionsweise des Systems nicht nur demonstriert werden kann, sondern auch ausgereift und zuverlässig ist. Das Spiel sollte so aufbereitet sein, dass jemand mit möglichst wenig Vorkenntnissen ein intuitives Verständnis entwickeln kann. Da das Spiel als „Lasertag Nerd Style“ bezeichnet wird, können von der primären Zielgruppe zumindest grundlegende Vorkenntnisse zu Netzwerkpotokollen erwartet werden. Dennoch muss das P2P-Protokoll der Wahl in die Form eines spielgerechten Regelwerks transformiert werden.

Es wurden keine genaueren Anforderungen an die Spielgeräte gestellt. Daher kann sich das Team eine beliebige Architektur aussuchen, die das Spiel unterstützt. Um den Spielspaß zu gewärleisten, muss jedoch die Übertragung von Treffern zuverlässig sein und die Geräte sollten möglichst handlich und leicht zu bedienen sein. Die Spieler brauchen insbesondere eine Möglichkeit, Feedback darüber zu erlangen, ob sie einen Spieler getroffen haben und wie das Spiel derzeit verläuft. Auch gilt, dass eine möglichst bequeme Lösung für den Spieler den Spaß unterstützt.

Aus den Anforderungen für Bequemlichkeit ergibt sich, dass drahtlose Kommunikationsmethoden verwendet und die Spielgeräte über Akkumulatoren betrieben werden sollten. Die Sensoren sollten entweder am Spielgerät befestigt oder möglichst einfach am Körper anbringbar und robust sein.

Da drahtlose Kommunikation verwendet wird und das Spiel möglichst reibungslos ablaufen soll, muss insbesondere bedacht werden, dass es zu Störungen kommen könnte und somit Informationen verloren gehen. Es ist also notwendig, dass die Geräte von einem unzuverlässigen Übertragungskanal ausgehen und mit daraus resultierenden Problemen umgehen können. Es ist außerdem zu beachten, dass es zu Schwierigkeiten mit konkurrierenden Prozessen kommen kann, weil z.B. Signal X schneller als Signal Y empfangen wurde, obwohl Signal Y früher abgeschickt wurde.

3 Systemarchitektur

In diesem Abschnitt wird das geplante System beschrieben, mit dem die Gruppe die Anforderungen erfüllen möchte. Die Abb. **1** stellt die Systemarchitektur dar.

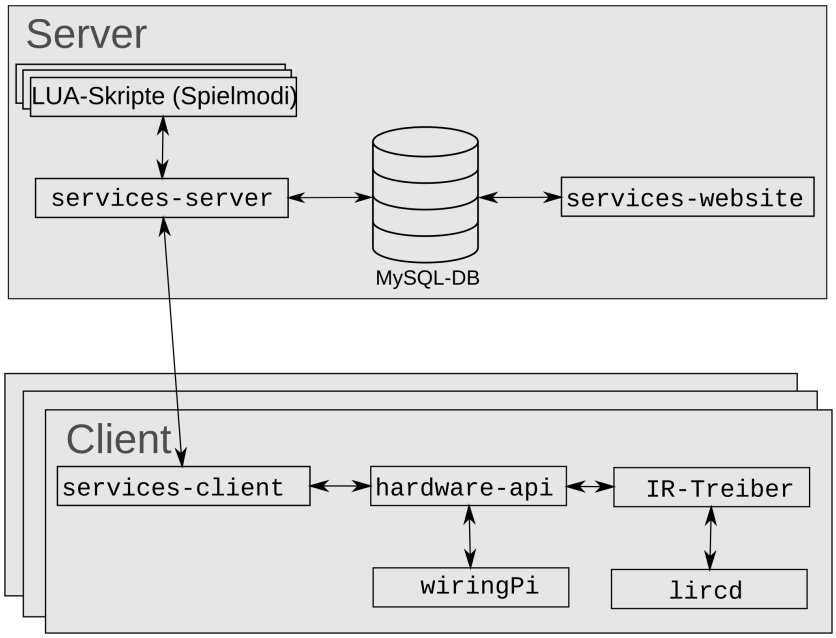

Abbildung 1: Ein Überblick über die Systemarchitektur

3.1 Die Spielgeräte

Wie oben bereits beschrieben, war für das Nachbilden von Lasertag eine drahtlose Übertragungsmethode notwendig, die möglichst zuverlässig ist. Im realen Lasertag wird dafür Infrarot verwendet und es werden LEDs benutzt, um den Abschuss darzustellen. Auch in diesem Projekt fiel die Wahl auf Infrarot. Es ist im Gegensatz zu Laser nicht schädlich für die Augen und es kann über eine größere Distanz zielgerichtet gesendet werden. Es gibt kostengünstige Komponenten wie LEDs und Sensoren sowie Open-Source-Treiber für die Modellierung für Daten. Es reicht nicht aus, einfach nur zu erkennen, dass man getroffen wurde, weil es für die Spielregeln notwendig ist zu wissen, von welchem Spieler der Schuss kam. Diese Notwendigkeit ergibt sich daraus, das zwei Systeme, die über das Netzwerk kommunizieren, dazu in der Lage sein müssen, sich eindeutig zu identifizieren. Daher war es sehr nützlich, dass über Infrarot Daten versendet werden können wie zum Beispiel bei Fernbedienungen.

Die gegenseitige Identifizierung wurde so gelöst, dass das Spielgerät eines Spielers beim Abschuss eine ID versendet. Wenn der Sensor eines Spielers diese ID empfängt, weiß das Gerät, von wem der Schuss kam, und es weiß ebenso seine eigene ID. Daher erhält es eine Information der Form: „*Spieler X hat Spieler Y getroffen*“.

Als Gerät fiel die Wahl auf den Raspberry Pi Model Zero W. Die Entscheidung wurde getroffen, weil ein leichter, stromsparender Computer benötigt wurde, der freiprogrammierbar ist und mit Infrarotkomponenten erweitert werden kann. Für den kabellosen Betrieb werden Powerbanks verwendet, um die Geräte mit Strom zu versorgen.

3.2 Netzwerkarchitektur

Informationen über Abschüsse müssen eine entsprechende Auswirkung auf den Spielstand haben. Eine Option dafür wäre, dieselbe Applikation auf jedem Gerät zu installieren und ein

P2P-Netzwerk zu errichten, in dem sich die Geräte konstant über den gegenwärtigen Spielstand austauschen. Daraus resultiert jedoch ein sehr kompliziertes Protokoll zur Synchronisierung, insbesondere weil ein Gerät mitten im Spiel die Verbindung verlieren könnte. Stattdessen wurde eine simplere Client-Server-Architektur gewählt, in der Clients lediglich Informationen sammeln und diese an den Server weiterreichen. Der Server wertet sie aus und gibt Informationen über den Spielstand an die Clients zurück. Durch diese Architektur werden viele potenzielle Logikfehler, die sich durch konkurrente Nachrichten ergeben, eliminiert.

Die Clients und der Server verwenden zur Kommunikation WLAN. Dies ist die naheliegendste Lösung, da die Technik zuverlässig ist und Verbindungsprobleme größtenteils durch den TCP/IP-Stack des Betriebssystems gelöst werden. Um ein weitaufspannendes Feld zu errichten, wäre es notwendig, mehrere Access Points (AP) zu errichten, die sich mit Roaming austauschen. Es wurde allerdings entschieden, dass dies den Rahmen des Projekts sprengen würde und es wurde der Einfachheit halber entschieden, den Server auch gleichzeitig als AP einzusetzen, womit theoretisch ein Spielfeld unterstützt wird, dass mehrere Räume umspannt.

Für den Server wurde ebenfalls ein Raspberry Pi (Model B+) verwendet, ein Notebook wäre jedoch genauso geeignet gewesen. Der Raspi wurde verwendet, weil es für die Entwicklung praktisch war, einen Minicomputer zu haben, der leicht zu transportieren ist und als WLAN AP verwendet werden kann.

3.3 Das Spiel

Zuvor wurde die grundlegende Infrastruktur beschrieben, mit der Informationen über Abschüsse ausgetauscht werden können. Das allein reicht für das Spiel noch nicht aus, weil die Spieler eine Möglichkeit brauchen, eine Übersicht über den Spielstand zu bekommen. Dafür gibt es zwei Ansätze, die verfolgt wurden:

- eine Anzeige auf jedem Gerät über den Status des Spielers und
- eine Anzeige für den Server über den aktuellen Spielstand.

Die Anzeige für die Spielgeräte wurde mit LEDs realisiert, die mit verschiedenen Farben anzeigen, ob ein anderer Spieler getroffen wurde, ob man unverwandbar ist, etc. Dies ist eine preiswerte Möglichkeit, die durch die GPIO-Anschlüsse der Raspberry Pis unterstützt wird. Mehr zu den LEDs ist im Abschnitt **4.1.5** zu finden.

Die Anzeige des Servers wurde mit einer Website umgesetzt, die tabellarisch das Scoreboard anzeigt. Die Webseite soll auf einem oder mehreren Bildschirmen dargestellt werden, die auf das gesammte Spielfeld verteilt werden.

In dieser Entscheidung spiegelt sich wider, dass Spieler regelmäßig erfahren wollen, ob sie noch im Spiel sind oder ob sie einen anderen Spieler getroffen haben. Daher muss es eine Möglichkeit geben, die entsprechende Anzeige sehr schnell und einfach zu überprüfen. Der aktuelle Spielstand muss seltener angesehen werden, weshalb es in Ordnung ist, wenn der Spieler einige Sekunden braucht, um auf die Webseite zu schauen.

Letztendlich muss auch auf Fairness geachtet werden. Daher ist es wichtig, dass die Fähigkeit, andere Spieler abzuschießen, limitiert ist. Dies muss auf mehrere Weisen geschehen:

1. Die Streuung von Abschüssen muss eingeschränkt sein. Dies wurde gelöst, indem Strohhalme über die LEDs gestülpt wurden, die das Aussenden des Signals in eine bestimmte Richtung lenken.

2. Damit ein Spieler nicht mehrmals hintereinander getroffen werden kann, gibt es bei vielen Spielmodi eine kurze Unverwundbarkeitszeit, nachdem ein Spieler getroffen wurde. In dieser Zeit hat der Getroffene die Möglichkeit, sich so zu positionieren, dass er nicht erneut getroffen wird.

3.4 Die Spielmodi

Es wurde entschieden, dass die Spiellogik möglichst vom Backend getrennt sein soll. Deswegen wurde sie in Form von LUA-Skripten gekapselt, die vom Server eingebunden werden. Die Schnittstelle zwischen diesen beiden Komponenten ist ein Informationsaustausch. Der Server gibt Spielparameter und Abschussinformationen an die LUA-Skripte weiter und die Spiellogik gibt die resultierenden Auswirkungen auf den Spielstand zurück, damit dieser auf Webseite und Spielgeräten angezeigt werden kann.

Als zu spielendes Netzwerkprotokoll haben wir eine vereinfachte Version von Bittorrent gewählt. Diese Wahl fiel unter anderem, da dieses Original auch in der Realität spielerische Elemente bietet.

Tatsächlich wurden mehrere Spielmodi implementiert. Dies geschah aus verschiedenen Gründen:

1. Es zwang uns, eine modulare Architektur zu errichten, die völlig unabhängig vom gewählten Spielmodus funktioniert. Dadurch haben wir eine garantierte Flexibiltät, Änderungen an Spielmodi vornehmen zu können.

2. Die Spielmodi haben eine unterschiedliche Reichhaltigkeit an Funktionen. Dies ist nützlich für das Debugging, weil somit gezielt Features getestet und gewisse Rahmenbedingungen je nach Spielmodus vernachlässigt werden können.

4 Komponenten

Aus der Anforderungsanalyse ergeben sich drei abstrakte Komponenten:

1. die Spielgeräte und hardwarebasierte Infrastruktur
2. Kommunikation der Spielgeräte und softwareseitige Infrastruktur
3. das Entwerfen eines Spielkonzepts und die Umsetzung in Software

Es handelt sich hierbei um ein System mit mehreren Schichten, wobei jede Schicht Dienste für die darüberliegende Schicht anbietet. Man kann feststellen, dass die Trennung der Komponenten nicht sauber ist. Es ist zum Beispiel unklar, wo die hardwarebasierte Infrastruktur in die softwarebasierte übergeht, etwa bei Netzwerkprotokollen.

Bei der Arbeit am Semesterprojekt haben wir den Komponenten entsprechend die Teilnehmer in drei Gruppen eingeteilt, welche abgekürzt als „Hardware“, „Services“ und „Spiellogik“ bezeichnet werden. In den folgenden Abschnitten werden die jeweiligen Gruppenmitglieder aufgezählt und die Arbeitsergebnisse derselben erläutert.

4.1 Hardware

Mitglieder und Aufgaben:

- Kevin Cornelius (Schaltungen, Spielgeräte)
- Lennart Weiß (Installation, Betriebssystemkonfiguration)
- Manuel Radatz (API, Spielanzeige)
- Rafael Robert Hadamik (Infrarottreiber, Spielgeräte)

4.1.1 Betriebssysteminstallation

Dieser Abschnitt erläutert, wie die Grundinstallation auf den Spielgeräten und dem Server vorgenommen wird. Für diese Aufgabe war Lennart Weiß verantwortlich.

Der Server wurde auf einem Raspberry Pi Model B+ eingerichtet. Strenggenommen sind für den Server nur Strom und ein Wifi Netzwerk absolut erforderlich. Per Wifi kann der Server als Access Point dienen und der Administrator kann sich mit SSH auf dem Gerät einloggen und Änderungen vornehmen. Das gewählte Gerät bringt allerdings noch einige Funktionen mit, welche die Wartung erleichtern. Der zusätzliche Ethernetanschluss wird genutzt, um ihn mit einem Netzwerk zu verbinden, das ihm eine DHCP-Adresse zuweist und Internetzugang gewährt. Dadurch können der Server sowie alle mit dem Wifi-Netzwerk verbundenen Geräte Pakete aus dem Internet herunterladen. Sollte aus etwaigen Gründen das Wifi-Netzwerk abstürzen, kann man den Server über HDMI an einen Bildschirm anschließen und mit einer USB-Tastatur warten. Sollte ein Bootvorgang nicht mehr möglich sein, kann man einfach die SD-Karte entfernen und auf einem anderen Gerät das Dateisystem überprüfen und reparieren. Natürlich hätte man als Server auch einen gewöhnlichen Desktop-PC verwenden können, dieser lässt sich jedoch nicht so leicht transportieren. Der kleine und leichte Raspberry Pi eignet sich also besser für den Fall, dass es kein festgelegtes Spielfeld gibt.

Auf dem Server ist Raspbian Lite installiert. Raspbian wurde vor allem wegen seiner weiten Verbreitung und Stabilität gewählt. Für Probleme bei der Serverinstallation kann im Regelfall durch eine Suche in Debianforen und ähnlichem eine Lösung gefunden und auf Raspbian übertragen werden. Außerdem ist das Angebot an Paketen sehr umfangreich. Die Lite-Version wurde gewählt, weil eine grafische Oberfläche für den Betrieb nicht notwendig ist und die gesamte Installation und Wartung des Geräts über die Kommandozeile erledigt wird. Durch den

Verzicht auf die grafische Oberfläche und weitere unnötige Pakete werden Ressourcen gespart und es gibt weniger Angriffsfläche für Fehlfunktionen.

Auf dem Server ist ein Puppetmaster eingerichtet. Damit kann genau festgelegt werden, welche Software auf die Clients verteilt werden soll und die Installation kann automatisiert werden. Für jeden Client muss auf dem Server ein Zertifikat signiert werden. Dieser manuelle Schritt wird jedoch nur einmal bei der Erstinstallation des Clients vorgenommen. Für die möglichst schnelle Installation der Spielgeräte wurde ein Image mit Raspbian vorbereitet, in dem eine deutsche Lokalisierung mit englischer Sprache eingestellt wurde und ein Puppet Client installiert ist. Außerdem wurde die WLAN SSID und das Passwort eingerichtet und der Client verbindet sich nach dem Boot-Vorgang automatisch per `wpa_supplicant` mit dem Server. Um den Kopiervorgang zu beschleunigen, wurde das Image auf die minimale Größe (ca. 2 Gigabyte) gehalten. Dadurch kann das Image in wenigen Minuten auf eine SD-Karte geschrieben und in den neuen Raspberry Pi eingesetzt werden. Außerdem kann die SD-Karte eine beliebige Speicherkapazität haben.

Um den Installationsvorgang zu beenden, muss auf man sich per SSH auf dem Client einloggen. Per `raspi-config` wird das Filesystem auf die maximale Größe eingestellt und der Hostname festgelegt. Dieser ist „`client-x`“, wobei `x` die Client ID bezeichnet, welche vorher eindeutig an jedes Gerät vergeben wird. Die Geräte sind entsprechend beschriftet, damit sie identifiziert werden können. Schließlich werden mit Puppet alle benötigten Komponenten installiert und der Client ist einsatzbereit.

4.1.2 Netzwerkkonfiguration

Dieser Abschnitt erläutert die Netzwerkinfrastruktur des Projekts. Für diese Aufgabe war Lennart Weiß verantwortlich.

Der Server ist im wesentlichen ein Wifi Access Point und Router. Die Clients können sich mit ihm über WLAN verbinden und bekommen über DHCP eine IP-Adresse, behalten jedoch ihren Hostname. Der Server fungiert als DNS-Server. Daher können sich die Geräte untereinander mit dem Hostname ansprechen. Die IP-Adressen sind nicht statisch und im gesamten Projekt soll generell mit den Hostnames gearbeitet werden. Damit wird eine Abstraktion geschaffen, die es ermöglicht, die Netzwerkkonfiguration zu verändern, ohne dass entsprechende Änderungen an den Applikationen vorgenommen werden müssen.

Die Routerfunktionalität wurde vor allem deswegen implementiert, weil die Clients über das Internet Pakete und Updates beziehen müssen. Wenn der Router einen entsprechenden Internetuplink besitzt, leitet er die Pakete der Clients nach außen durch und umgekehrt. Durch den Einsatz von NAS benötigt der Server nur eine eigene IP Adresse, aber keine für die Clients. Das Arbeiten und Debuggen am Server und den Clients macht es oft notwendig, dass sich der Entwickler in das WLAN-Netzwerk einloggt. Daher ist es ein nützlicher Nebeneffekt, dass er weiterhin auf das Internet zugreifen kann, wenn er mit dem Netzwerk verbunden ist. Damit der Server sowohl lokale als auch globale URIs auflösen kann, muss er auf dem Client als Nameserver hinzugefügt werden. Er fungiert als DNS Cache, Anfragen an externe Ressourcen werden an einen anderen Nameserver weitergeleitet.

4.1.3 Installation des Projekts

Dieser Abschnitt erläutert, wie die einzelnen Komponenten des Projekts auf die Geräte verteilt werden. Für diese Aufgabe war Lennart Weiß verantwortlich.

Der gesamte Source-Code für die verschiedenen Komponenten befindet sich in einem Git-Repository. Um die neueste Version des Projekts auf die Geräte zu verteilen, wird auf dem Server ein Skript aufgerufen, welches zuerst die einzelnen Komponenten kompiliert und die erforderlichen Dateien installiert.

Dies sind die Komponenten die installiert werden müssen: `services-server`, `services-website` und `mysql` auf dem Server sowie `services-client`, `hardware-api` und `lirc` auf dem Client. Dabei sind `mysql` und `lirc` externe Komponenten, die anderen wurden von uns entwickelt.

Für jede Komponente wird eine Systemd Unit definiert und entweder auf dem Server oder auf allen Clients verteilt. Es ist erwünscht, dass die verschiedenen Dienste nach dem Systemstart automatisch gestartet werden und mit Systemd können Abhängigkeiten und die richtige Reihenfolge festgelegt werden. Auf den Clients ist `services-client` von `hardware-api` abhängig und Letzteres von `lircd`. Auf dem Server hängt `services-server` von `services-website` und diese wiederum von `mysql` ab. Die Prozesse werden also durch das Betriebssystem verwaltet und sollte einer abstürzen, lassen sich die Logs mit `journalctl` nachlesen. Das Verwalten vieler Prozesse auf mehreren Geräten lässt sich somit besser handhaben.

Um die benötigten Binaries zu kompilieren, wird die benötigte Version aus dem Git-Repository gepullt. Dann wird ein Skript aufgerufen, welches in die verschiedenen Ordner der Komponenten geht und Makefiles aufruft. Diese erzeugen dann die erforderlichen Dateien. Wenn diese auf dem Server benötigt werden, werden sie direkt in `/usr/bin` oder `/usr/lib` kopiert, je nachdem welchen Zweck sie erfüllen. Die entsprechenden Systemd Units werden dann neu gestartet. Werden die Dateien auf den Clients benötigt, werden sie in das Puppet-Code-Verzeichnis kopiert. Wurden diese Dateien noch nicht zuvor verwendet, muss darüber hinaus noch die Konfiguration von Puppet angepasst werden. Die Verteilung wird dann über Puppet automatisiert.

Es gibt auch einige Dateien, die verteilt, jedoch nicht kompiliert werden müssen. Die Webseite wird einfach vom Git-Verzeichnis als Ordner nach `/usr/bin/services-website` kopiert. Die Systemd Unit startet Flask und der Pfad der Flask-Applikation wird exportiert. Die Spielmodi werden ebenfalls aus Git heruntergeladen und nach `/var/lib/spielmodi` kopiert.

4.1.4 IR-Treiber

Um die einzelnen Spieler zu unterscheiden, war es notwendig, dass sie unterschiedliche Informationen aussenden, um sie eindeutig im Spielkontext zu identifizieren. Wir hatten drei verschiedene Ideen, wie man diese Informationen modellieren kann, um sie schnell und korrekt zu übertragen. Dabei war die Grundlage, dass jede gesendete Information in eine eindeutige Abfolge von 2 verschiedenen Zuständen – im Folgenden als Zustand 0 (kurz: 0) bzw. Zustand 1 (kurz: 1) bezeichnet – übersetzt wird und dann diese Abfolge von Zuständen gesendet wird. Diese Abfolge von Zuständen musste ebenfalls modelliert werden, um das Gebot der Korrektheit zu erfüllen, da ein einfaches Absenden der Information – mit Signal an = 1 und Signal aus = 0 – sehr rauschanfällig wäre. Des Weiteren wollten wir auf selbstkorrigierende Codes verzichten, um eine möglichst schnelle und genaue Treffererkennung zu schaffen, und um zu verhindern, dass wenn zwei Spieler nahezu gleichzeitig schießen, der Spieler, der zuerst schießt, den Treffer nicht gewertet bekommt, da sein Schuss erst korrigiert werden musste, wohingegen der Schuss des langsameren Spielers zuerst gewertet wird, da sein Schuss keine Korrektur hatte. Dazu hatten wir folgende drei Ideen, wie man diese Modulation umsetzen könnte:

1. Man gibt Zustand 1 und Zustand 0 unterschiedliche Längen von einem Signal mit festen Pausen dazwischen.

2. Man modeliert die Zustände über Signalflanken.

3. Man gibt beiden Zuständen die gleiche Signalzeit, aber unterschiedliche Pausen zwischen den einzelnen Signalen.

Idee 1 Der Vorteil der ersten Idee ist, dass sie sehr simpel zu implementieren ist. Der Nachteil ist jedoch, dass sie von den drei Ideen am stärksten rauschanfällig ist und somit die geringste Reichweite bietet.

Idee 2 Die Vorteile hier bestehen darin, dass sie sehr einfach zu implementieren wäre, wenn man den Treiber von Grund auf selbst schreibt, und dass sie die schnellste Übertragung von allen bietet. Der größte Nachteil besteht jedoch darin, dass diese Idee selbst sehr rauschanfällig ist, da Flanken in der Theorie überspielt werden könnten.

Idee 3 Der Vorteil besteht hier darin, dass die geringste Rauschanfälligkeit besteht, da man – mit genügend großen Unterschieden zwischen den Pausen bei den beiden Zuständen – sehr großzügig sein kann, was noch als Treffer gilt. Ein weiterer Vorteil war, dass man bereits Geräte im Haus hatte, die genauso funktionieren, was Tests am Anfang stark vereinfacht hatte und dafür gesorgt hat, dass eine Fehlerbehebung später ebenfalls leichter war. An Nachteilen ist hier aufzulisten, dass es von den drei Ideen am schwierigsten zu implementieren gewesen wäre und dass es die langsamste Variante für die Informationsübertragung ist.

Am Ende hatten wir uns für Idee 3 entschieden, da die Informationspakete klein genug sind, dass die Geschwindigkeit nicht leidet. Außerdem haben wir das Problem mit der Implementation dadurch umgangen, dass man den Open-Source-Treiber Lirc benutzt hat, sodass man nicht den Treiber selber schreiben musste, sondern nur über die Lirc-Bibliothek mit dem Treiber kommunizieren musste. Der Vorteil eines externen Testgerätes war dann den Vorteilen der anderen beiden Varianten stark überlegen.

4.1.5 Die LEDs und ihre Bedeutungen

Dieser Abschnitt erklärt die Bedeutung der LEDs. Für diese Aufgabe war hauptsächlich Manuel Radatz verantwortlich.

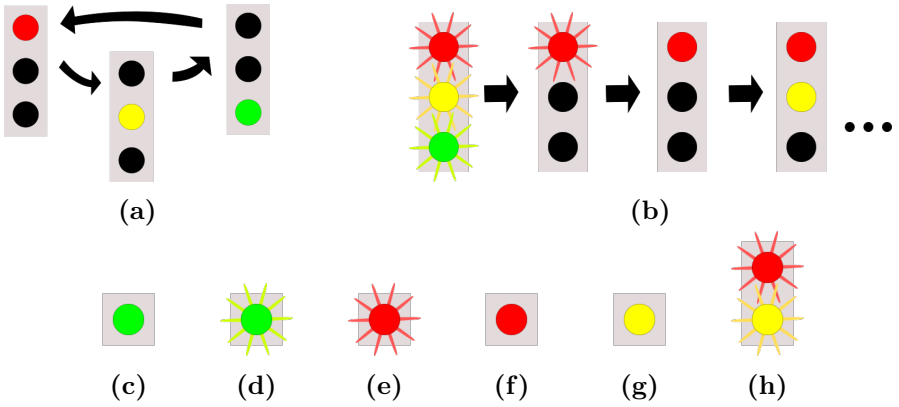

Abbildung 2: LED-Zustände
(a) Initialer Zustand **(b)** Spielstart **(c)** Spiel läuft **(d)** Spielende naht
(e) wurde getroffen (unverwundbar) **(f)** tot **(g)** hat getroffen **(h)** letztes Leben

Jedes Spielgerät hat jeweils eine rote, eine gelbe und eine grüne LED, die wie auf einer Ampel angeordnet sind. Für Menschen mit Rot-Grün-Schwäche könnte man auch Prototypen bauen, bei denen die grüne LED durch eine blaue LED ersetzt wird. Mit diesen LEDs werden die verschiedenen Ereignisse kodiert. Läuft kein Spiel und ist auch kein Spiel angekündigt, so leuchten alle LEDs der Reihe nach auf. Dadurch können defekte, sowie nicht- oder falschangeschlossene LEDs schon direkt nach dem Einschalten der Spielgeräte erkannt werden. Wird ein Spiel angekündigt, so fangen alle LEDs synchron an zu blinken. 10 Sekunden vor dem Spielstart blinkt dann nur noch die rote LED und 4 Sekunden davor schalten die LEDs, analog zu einer deutschen Straßenverkehrsampel, von rot über rot-gelb auf grün. Die grüne LED zeigt an, dass das Spiel läuft. Dies mag zwar kompliziert klingen, ist aber doch sehr intuitiv, denn solange die grüne LED nicht leuchtet, weiß jeder, dass das Spiel noch nicht läuft, und wenn nach rot rot-gelb kommt, weiß jeder aus Erfahrung, dass als nächstes grün kommt.

Neigt sich das Spiel dem Ende zu, fängt die grüne LED 64 Sekunden vorher langsam an zu blinken und blinkt bis zum Ende des Spiels immer schneller. Nach dem Spielende kehren die LEDs zum initialen Zustand zurück. Wenn das Spiel läuft, zeigt das Blinken der roten LED an, dass man getroffen wurde und unverwundbar ist. Auch sie wird immer schneller, wenn die Zeit der Unverwundbarkeit abläuft. Zeigt die LED Dauerrot, dann ist man tot. In diesem Zustand ist es auch nicht mehr möglich, andere Spieler abzuschießen, d.h. der IR-Sender wird deaktiviert. Die gelbe LED zeigt an, dass man einen anderen Spieler getroffen hat. Das letzte Leben wird dadurch angezeigt, dass die rote und die gelbe LED etwa alle 2 Sekunden für 100 Millisekunden synchron aufleuchten. Alle LED-Zustände sind zur Übersicht auch in Abb. **2** dargestellt.

Ursprünglich wollten wir anstelle von LEDs ganze Displays verwenden, um auch individuelle Informationen als Text oder eine Karte des Spielfeldes mit allen Mitspielern anzeigen zu können. Neben des begrenzten Budgets und der Tatsache, dass Informationen dieser Art auch auf der Webseite angezeigt werden können, war auch die begrenzte Zeit ein Grund dafür, dass wir diese Idee verworfen haben. Auch die Karte wurde nicht umgesetzt, da für die zuverlässige und genaue Positionsermittlung eines Spielers in geschlossenen Räumen diverse Probleme zeit- und kostenintensiv gelöst werden müssten und dies jenseits unserer Möglichkeiten in diesem Projekt gewesen wäre. (Trotzdem wurde zunächst viel Zeit in diese Idee investiert.)

4.1.6 Die Hardware-API

Dieser Abschnitt erklärt den Teil der Hardware-Komponente, der für die Kommunikation mit der Services-Komponente sowie für die Ansteuerung des Buttons und der LEDs zuständig ist. Für diese Aufgabe war Manuel Radatz verantwortlich.

Wir haben uns dafür entschieden, dass die Hardware-Komponente und die Services-Komponente jeweils verschiedene Programme darstellen. Auf jeden Client laufen daher die Prozesse `services-client` und `hardware-api`. Die Kommunikation erfolgt somit *nicht* beispielsweise dadurch, dass von der Services-Komponente eine Bibliothek verwendet wird, die die Hardware-Komponente bereitstellt, sondern über Unix-Sockets. Dabei werden Strings im Format von Google Protocol Buffers übertragen, denen jeweils ein Header mit Nachrichten-ID und Nachrichtenlänge vorangestellt ist. Durch die Verwendung von Protocol Buffers ist es möglich, dass die verschiedenen Komponenten in unterschiedliche Programmiersprachen geschrieben werden können. Dieser Flexibilität und einer besseren Wartbarkeit, die neben religiösen Gründen die Anlässe für die Verwendung von Unix-Sockets und Google Protocol Buffers waren, stehen auf der anderen Seite ein größerer Kommunikationsoverhead gegenüber. Wegen der Rechenleistung der Raspberry Pis fällt dieser Overhead kaum ins Gewicht. Da wir die gewonnene Flexibilität jedoch nicht genutzt haben und beide Komponenten in C++14 geschrieben sind, ist der Overhead letztendlich unnötig. Rückblickend wäre es auch einfacher gewesen, wenn die Hardware-Komponente auf den Clients direkt mit der Services-Komponente auf dem Server kommuniziert hätte, da wir uns später für eine Client-Server-Architektur entschieden hatten. Würde man das Projekt noch weiterentwickeln, könnte die jetzige Architektur allerdings noch von Vorteil sein.

Die Hardware-API besteht aus dem in C++14 geschriebenen Programm `hardware-api`, das auf allen Clients, jedoch nicht auf dem Server läuft. C++ war für diese Aufgabe sehr gut geeignet. Die Sprache hat einen sehr geringen Overhead und bietet die notwendige Flexibilität. Hauptsächlich erfolgte diese Entscheidung jedoch wegen bereits vorhandenen Programmierkenntnissen in dieser Sprache. Die Entscheidung für die Version C++14 beruht darauf, dass dies die höchste Version ist, die von der GCC-Version, die Raspbian zur Verfügung stellt, unterstützt wird.

Das Programm kommuniziert nach oben mit `services-client` über Unix-Sockets und Protocol Buffers. Nach unten kommuniziert es mit dem Infrarot-Treiber, indem dessen Quellcode bei der Kompilierung der `hardware-api` direkt eingebunden wird, und sie spricht die LEDs und den Button direkt über wiringPi an.

Für die Kommunikation nach oben wird die Socket-API der POSIX-C-Bibliothek genutzt. Für die Einbettung in C++ und die Nutzbarkeit nach dem in dieser Sprache üblichen RAII-Prinzip wurden für die Unix-Sockets die Wrapperklassen `CSocketWrapper`, `UnixReceiver` und `UnixSender` geschrieben, die die Kommunikation über die Socket-API abstrahieren. Dazu gehört auch das Puffern von unvollständigen empfangenen Nachrichten. Außerdem dient die Klasse `FileDeleter` dazu, dass die für die Kommunikation über Unix-Sockets notwendigen Dateien nach dem Beenden des Programms implizit gelöscht werden. Die Klassen `TCPConnection`, `TCPServer` und `TCPClient` erfüllen dieselbe Aufgabe für die TCP-Kommunikation, die für die später beschriebenen Testprogramme `client-stub` und `server-stub` notwendig ist. Alternativ hätte man auch bestehende Bibliotheken wie boost verwenden können, die ebenfalls eine C++-Schnittstelle für die Netzwerkkommunikation zur Verfügung gestellt hätten. Dafür hätte man sich zuerst in diese Bibliothek einarbeiten müssen und da Manuel Radatz, der diese Komponente geschrieben hat, bereits Erfahrungen mit der C-Socket-API hatte, wäre das aufwendiger gewesen als die Implementierung der Wrapperklassen, weshalb sie am Ende nicht genutzt wurde. Da diese Wrapperklassen am Ende stabil liefen, war diese Entscheidung auch richtig.

Eine weitere Klasse (`InterfaceToServices`) abstrahiert die Kommunikation mit `services-client`. Sie ist hauptsächlich dafür zuständig, dass diese Kommunikation stabil bleibt und nach einem Verbindungsabbruch (z.B. wenn `services-client` abgestürzt ist oder zu Testzwecken beendet wird) wieder hergestellt wird. Bei dem Empfangen von Nachrichten werden die zuvor von der `main()`-Funktion gesetzten Callback-Funktionen aufgerufen.

Für die LED-Ansteuerung empfängt die Hardware-API von `services-client` Nachrichten, in denen mit drei ganzen Zahlen kodiert ist, welches Ereignis wann und wie lange angezeigt werden soll. Die Ereignisse, die übertragen werden, sind:

- Spieler wurde getroffen und ist unverwundbar – mit der Information, wie lange der Spieler unverwundbar ist
- Spieler hat einen anderen Spieler getroffen – optional mit der Information, wie lange das dem Spieler angezeigt werden soll
- Spiel startet – mit der Information, wann das Spiel startet und wie lange das Spiel läuft
- Spiel endet – mit der Information, wann
- Spieler tot
- letztes Leben

Nicht alle Ereignisse werden in allen Spielmodi genutzt. So gibt es beispielsweise Spielmodi, in denen es das Konzept von „Leben“ nicht gibt. Das Ereignis „Spiel endet“ ist auch nur dann notwendig, wenn bei dem Ereignis „Spiel startet“ nicht übertragen wurde, wie lange das Spiel läuft.

Die LED-Ansteuerung erfolgt ebenfalls in einer eigenen Klasse namens `LEDEventStatus`. Diese ruft direkt Funktionen von wiringPi auf, verwaltet alle Daten, die zur korrekten LED-Ansteuerung notwendig sind (auch die Timer) und interpretiert als Eingabe das von `services-client` empfangene Zahlentripel.

Der Status des Buttons wird direkt in der `main()`-Funktion durch Aufrufe der wiringPi-Funktionen abgefragt.

Die `main()`-Funktion besteht abgesehen von der Initialisierung aus einer Schleife, die erst dann verlassen wird, wenn das Programm `SIGTERM` empfängt oder eine Exception auftritt, die auf größere Probleme hindeutet. Dort besteht jede Iteration aus:

- das Abfragen, ob Nachrichten von `services-client` angekommen sind, welche dann auch durch das Aufrufen von Callback-Funktionen interpretiert werden,

- die Aktualisierung des LED-Status,
- die Abfrage des Buttons, ob dieser gedrückt wird, wobei in diesem Fall die eigene Client-ID an den IR-Sender gesendet wird und
- das Abfragen des IR-Sensors, ob Daten empfangen wurden sowie die Weiterleitung dieser an `services-client`.

Danach wartet das Programm 0,5 Millisekunden, um die CPU-Auslastung gering und folglich die Temperatur des Prozessors niedrig zu halten und die Lebensdauer des Akkumulators zu erhöhen. Die dadurch entstehende Latenz von bis zu 0,5 Millisekunden ist so niedrig, dass sie keine spürbaren Auswirkungen auf das Spiel haben sollte. Wird der Button gedrückt, werden auch nicht in jeder Iteration IR-Daten gesendet. Dies würde zu einer Überbeanspruchung der IR-LED führen. Stattdessen werden ab dem Moment, in dem der Button heruntergedrückt wird, nur alle 100 Millisekunden IR-Daten an den IR-Treiber übergeben.

Neben diesem regulären Ablauf wurden auch zahlreiche Features zum Testen implementiert, welche über Programmparameter ein- oder ausgeschaltet werden können. Ist kein Button vorhanden, so besteht die Möglichkeit, dass ohne Unterbrechung alle 500 Millisekunden die eigene Client-ID an den IR-Sender gesendet wird. Wenn kein IR-Empfänger funktionsfähig ist, das Programm auf anderer Hardware ausgeführt wird oder nur die Kommunikation mit `services-client` getestet werden soll, gibt es zusätzlich einen automatischen und einen manuellen Testmodus. Im automatischen Testmodus werden regelmäßig zufällige Daten an `services-client` gesendet, als wären diese von dem IR-Sensor empfangen worden. Im manuellen Testmodus kann man manuell Testdaten eingeben, die gesendet werden sollen. In beiden Testmodi wird der LED-Status auch auf der Konsole ausgegeben. Dadurch sind Tests auch ziemlich einfach in virtuellen Umgebungen ohne die Hardware möglich.

Außerdem wurden Stubs implementiert, welche `services-client` ersetzen können, sodass auch andersherum die Hardware ohne Services getestet oder demonstriert werden kann. Dazu gehört der Stub `led-trigger`, mit dem man manuell LED-Ereignisse an die `hardware-api` senden kann, sowie das Paar von Stubs `client-stub` und `server-stub`, von denen einer auf den Clients und einer auf dem Server läuft, der alle empfangenen IR-Daten in Echtzeit auf dem Server ausgibt, was das ansonsten erforderliche Nachschauen in Log-Dateien beim Debuggen erspart.

Das Programm `hardware-api` läuft im fertigen System als Daemon. Es kann aber auch als gewöhnliches Programm gestartet werden, was zum interaktiven Testen auch notwendig ist.

Der build-Prozess erfolgt durch eine einzige Makefile. Wegen der Einfachheit des Programms und der Tatsache, dass die Programme gezielt für ein bestimmtes System entwickelt worden sind, wäre die Nutzung von cmake oder des GNU Build Systems überflüssig gewesen.

4.2 Services

Mitglieder und Aufgaben:

- Angelina Jellinek (Design der Webseite)
- Jan Arne Sparka (Integration von LUA und Server)
- Kevin Marc Trogant (Integration von Hardware und Server)
- Pascal Jochmann (Website Backend)
- Tim Sikatzki (Website Backend)

4.2.1 Kommunikation

Der Spielserver `services-server` nutzt einen eigenen Thread für die Kommunikation mit den Clients und der Webseite. Clients und Server kommunizieren über BSD Sockets. Als Alternative kam Googles gRPC in Frage. Dieses wurde verworfen, weil:

- gRPC erlaubt Kommunikation nur in eine Richtung. Der Server kann nur dann Nachrichten an den Client verschicken, wenn ein Request eingegangen ist.
- gRPC erlaubt nur einen Nachrichtentypen pro Kommunikationsrichtung. Da wir mehrere verschiedene Nachrichtentypen benötigen, hätten wir das Konzept eines Unions in gRPC nachbilden müssen.

Es wäre möglich gewesen Bibliotheken, zum Beispiel boost.asio, die eine einfachere Schnittstelle anbieten zu verwenden. Da wir zu dem Zeitpunkt, an dem wir entschieden haben die Socket API direkt zu verwenden, noch nicht genau wussten wie die Netzwerkkommunikation funktionieren würde, wollten wir uns möglichst viele Optionen offen halten. Außerdem hatten wir im Verlauf der Entwicklung mehrmals Probleme mit inkompatiblen Bibliotheksversionen, entweder durch andere APIs oder - wesentlich lästiger - durch unterschiedliches Verhalten. Deshalb waren wir später sehr zurückhaltend beim Einführen neuer Abhängigkeiten.

An die Datenbank ist der Server über die mysql c connector Schnittstelle angebunden. Die Schnittstelle bietet eine relativ simple API: Zuerst verbindet sich das Programm mit dem MySQL Datenbankserver (entweder über TCP oder über Unix Sockets) und dann können beliebige SQL Anfragen als String an den Datenbankserver geschickt werden, der daraufhin Zeilenweise die Ergebnisse zurückliefert. Diese Schnittstelle ist in einem C++ Singleton gekapselt `DatabaseConnection`, die Methoden für alle Anfragen bereitstellt die der Server stellen muss (bswp `get_runnable_games()`).

Der Thread für die Kommunikation führt in einer Endlosschleife folgende Schritte aus:

1. Falls sich neue Clients verbunden haben: Lege die passenden Datenstrukturen an und setze den Status des Clients in der Datenbank auf verbunden.
2. Hole eine Liste noch nicht beendeter Spiele von der Datenbank. Falls neue dazu gekommen sind, lege ein entsprechendes Objekt an.
3. Frage den aktuellen Zustand aller Spiele die dem Server bekannt sind ab. Falls ein Spiel gestartet wurde, starte den entsprechenden Thread für die LUA API des Spiels.
4. Sende den Spiellogik-Threadd aller neu beendeten Spiele eine Nachricht die diese dazu auffordert sich zu beenden.
5. Empfange und Parse Nachrichten von Clients. (Siehe weiter unten)
6. Zurück zu 1.

Das Nachrichtenformat ist mittels Google Protocol Buffers, Googles Lösung der Datenformatserialisierung umgesetzt. Dies ist eine sehr simple Lösung der Frage, wie wir die Nachrichten einheitlich und verständlich strukturieren. Protocol Buffers ist dabei im Vergleich zu den betrachteten alternativen wie XML am simpelsten mit dem wenigsten Overhead, während es trotzdem unseren Anforderungen genügt.
Da Protocol Buffers jedoch selbst keine Möglichkeit implementiert um aus einem empfangenem Byte-String den Nachrichtentyp zu rekonstruieren, wird vor jeder Nachricht ein Header verschickt der den Nachrichtentyp und die Länge der Nachricht in Bytes enthält. Die Nachrichten

selbst können einfach mittels der Protocol-Buffers Bibliothek zu entsprechenden C++ Objekte geparst werden, die dann Nachrichtenfelder als „Getter- und Setter-Funktionen“ verfügbar machen.

Die Kommunikation mit der Hardware-Komponente läuft über einen eigene Komponente (`services-client`) die auf den Spielgeräten ausgeführt wird. Diese empfängt Nachrichten von der Hardware-Schnittstelle über einen Unix-Socket, wandelt diese in das Services-Interne Format um und sendet sie dann über einen TCP Socket an den Server. Außerdem kümmert sich die Komponente darum in regelmäßigen Abständen (ca. 5 Sekunden) eine kurze „Keep-Alive“ Nachricht an den Server zu senden um zu signalisieren das der Client noch verbunden ist und am Spielbetrieb teilnimmt.

Da für ein Lasertag das Treffen von anderen Spielern von relativ hoher Bedeutung ist, soll hier noch kurz ausgeführt werden, was aus Sicht der Services-Komponente passiert wenn ein Spieler einen anderen trifft.

Zuerst erhält der Client eine Nachricht von der Hardware-API, die mitteilt das der Client getroffen wurde. In der Nachricht ist die ID des Clients enthalten, von der das Gerät getroffen wurde. Die eigene ID erhält der Client beim start. Diese beiden Informationen werden in eine sog. „Hit-Message“ verpackt und an den Server gesendet. Der Server erhält die Nachricht, parst diese und sucht nach einem Spiel das beide Clients - Schießenden und Getroffenen - enthält. Findet er kein passendes Spiel, wird die Nachricht verworfen. Falls ein passendes Spiel gefunden wurde, werden die Client-IDs zu den entsprechenden Spieler-IDs aufgelöst (die Zuordnung ist pro Spiel eindeutig) und eine passende Nachricht an den entsprechenden Spiel-Thread geschickt.

4.2.2 LUA API (services)

Um Spielmodi zu beschreiben und sauber von den anderen Komponenten abzutrennen brauchten wir eine API. Die erste Entscheidung war zwischen LUA, c-Code und einer eigenen Spielmodus-DSL. Da unsere Spiellogikgruppe die Entscheidung für LUA traf und sogar schon einen funktionierenden Simulator für Spiele in sehr schneller sukzession herausbrachte, war diese Entscheidung für uns damit schnell getroffen.

Die LUA API auf services Seite war sehr schwierig einzubauen. Die erste Option war es die API als Singleton bereitzustellen, dies stellte sich aber sehr schnell als schlechter Ansatz heraus, da der Start einer weiteren Runde hierdurch zu einem komplizierten und unschönen Hack geworden wäre. Daher und da wir die Architektur für potentiel mehrere Spiele die Parallel laufen offen halten wollten, entschieden wir uns für den jetztigen Ansatz, die API läuft in einem eigenen Thread.

Der Logik-Thread sammelt sich erstmal die relevanten Spieldaten zusammen, initialisiert dann die Felder auf welche LUA zugreift und geht dann in eine Event Loop über. In dieser Gameloop wird zuerst überprüft ob Schussnachrichten durchgereicht wurden, welche wiederum an LUA weitergereicht werden. Danach werden alle potentiellen Timer weitergetickt um die vergangene Zeit. Als letzte wird noch abgeprüft ob der Server uns ein AskedToExit Event schickt, welches mit Aufräumen und beenden des Spiels quittiert wird.

Der Hauptteil der API Arbeit auf der service Seite war die vorhandene Spiellogik API (den Simulator) an die real genutzte Hardware und Software anzubinden. Der Großteil der Input/Output Operationen wurde über unser Datenbankconnection Singleton umgeleitet. Das Scoreboard wird dabei permanent mitgeführt, wodurch Erweiterungen, wie z.B. spielerseitige Displays denkbar sind und was uns die Probleme von Score Serialisierung ignorieren lässt, da dieser sowieso von der Datenbank serialisiert wird. Da dann letztendlich die Relevanz eines Timers für LUA Skripte aufkam, welcher nicht den Logik-Thread vom Arbeiten abhält, haben wir einen einfachen Timer entwickelt der mit Funktionen Callbacks arbeitet. Die letzte Erweiterung der API war das hinzufügen der LED Events und der Abstraktion dieser in Funktionen, die Teil der LUA API wurden.

Die services Seite der LUA API hatte als Hauptproblem, das sie aufgrund der sehr zentralen Stellung (Verknüpfung von 3 Komponenten), sehr viele Abhängigkeiten hatte und sehr schwierig zu testen war. Dies ist dem Projekt letztendlich zum Verhängnis geworden, da wir in der Interaktion dieser vier Komponenten einen Fehler haben, den wir bisher noch nicht lokalisieren konnten. Das Debugging ist auf Grund von LUA auch nur sehr begrenzt möglich und wird uns noch einiges an Aufwand einbringen.

4.2.3 Website

Unsere Website bildet das Herzstück in der Interaktion zwischen dem Nutzer und der Software. Sie gibt ihm die Möglichkeit, die Konfigurationen für das Spiel einzustellen, wie zum Beispiel die Anzahl der Teams, dem Spielmodus und den Spielgeräten Namen zuzuteilen. Außerdem haben die Spieler durch sie die Möglichkeit, ihre Leistungen während des Spiels in Echtzeit einsehen zu können.

Zur Entwicklung der Website haben wir "Flask"benutzt, ein Webframework auf Python-Basis. Wir haben uns für dieses Framework entschieden, da es zunächst sehr leicht zu benutzen ist und uns gleichzeitig alles geboten hat, um die erwarteten Funktionalitäten der Website implementieren zu können, wie zum Beispiel ein MySQL-Package, welches wir zur Interaktion zwischen Webapplikation und Datenbank benutzt haben. Um die oben genannten Funktionen gewährleisten zu können, besteht unsere Webapplikation aus mehreren HTML-Seiten.

Die erste Seite (Abb.1) ist der Kern der Website. Sie wird benutzt, um das eigentliche Spiel zu erstellen. Nicht nur nimmt sie die eingegebenen Daten der Spieler auf, sondern überträgt die Einstellungen direkt auf die Datenbank und erstellt dort direkt Einträge, die für das Scoreboard relevant sind. In der Sektion Datenbank werden die Einträge und Tabellen genauer erläutert. Die möglichen Einträge der Spielernamen sind hierbei abhängig von der Anzahl der verbundenen Clients. Sind nur drei Clients zum Server verbunden, so können wir auch nur drei Namen zuweisen. Auch die Spielmodi, die ausgewählt werden können, werden dynamisch erstellt. Hierbei wird aus einem Ordner ausgelesen, in dem die Lua-Skripte abgelegt werden können.

Sobald ein Spiel gestartet wird, leitet die Seite auf das Scoreboard weiter.
Das Scoreboard zieht sich seine Einträge direkt aus der Datenbank und aktualisiert dieses asynchron unter der Verwendung von jQuery und ajax. Es stellt die Einträge in Form von Tabellen da, hierbei gibt es eine Tabelle die Spieler und eine Tabelle für die Teams (Abb.2). Auf der Seite haben wir dann die Möglichkeit, das Spiel manuell zu pausieren oder auch zu beenden. Wenn die Dauer einer Runde ausgelaufen ist, wird das Spiel automatisch in den pausierten Zustand gesetzt (Abb.3). Wir können nun jederzeit eine mithilfe des Links SSpiel fortfuehrenËine neue Runde starten bzw. das pausierte Spiel weiterspielen.
Der Link SSpiel beenden"funktioniert dementsprechend genauso, allerdings ist es danach nicht mehr möglich, das Spiel weiterzuführen. Der Nutzer dann danach auf einen Button klicken, um zurück zur Startseite zu gelangen und ein neues Spiel zu starten (Abb.4).

Das Aussehen der Website ist dynamisch gestaltet. Die Größe des Scoreboards passt sich individuell an die Anzahl der Einträge an. Gleichzeitig gibt es auch nur ein extra Scoreboard für Teams, wenn es Teams auch wirklich gibt.
Sowohl der Hintergrund, als auch die Schriftart und die Farben sind so gewählt, dass sie zum Thema Lasertag und Nerd Style passen. Gleichzeitig ist das Design dunkel genug und wenig belastend für die Augen, um es in einer dunklen Halle auf eine Leinwand zu übertragen, ohne dabei die Spieler von ihrem Spiel abzulenken.

Insgesamt ist die Webapplikation seitens Funktionalität vollständig. Es könnten noch Verbesserungen hinsichtlich der Anzeige der verschiedenen Dateifragmente und der Effizienz der Anzeige des Scoreboards gemacht werden.

4.2.4 Datenbank

Unsere Datenbank ist die ermöglichte uns die Interaktion innerhalb der services-Gruppe. Wir haben uns für eine Datenbank entschieden, da wir eine Spielhistorie aus den vergangenen Spielen erstellen wollten. Gleichzeitig bot sich die Datenbank auch für die Darstellung des Scoreboard an, da die SQL Anfragen für eben dieses sehr trivial waren.

Um die Fehlerquelle einer Datenbank in Form einer Datei zu umgehen, haben wir uns nach einiger Zeit dafür entschieden von SQLite auf MySQL, als Datenbankmanagementsystem, umzusteigen.

In der obrigen Abbildung sehen wir den Aufbau unserer Datenbank. Wie wir sehen können, ist sie in drei Tabellen unterteilt.
Die client - Tabelle ist relevant für das Anzeigen der möglichen Eingabefelder für die Spielernamen im Menü der Website. Hierbei wird unterschieden, ob ein Client mit dem Server verbunden ist oder nicht.

Die game - Tabelle ist vorallem interessant für die LUA API, da sie die Einträge enthält, die von dem Spieler vor Beginn des Spieles mithilfe des Menüs übergeben werden, wie zum Beispiel der Spielmodus und die Rundendauer. Gleichzeitig beinhaltet die Tabelle auch den Status, den das jeweilige Spiel zu jeder Zeit hat. Dieser Status wird von allen Parteien benutzt: die LUA API muss das Spiel nach Überschreiten der Rundendauer pausieren oder es beenden, die Netzwerkkommunikation muss bei einem gestoppten Spiel aufhören, Nachrichten weiterzuschicken, damit keine Punkte während eines pausierten oder beendeten Spieles verteilt werden.

Die playing - Tabelle wird für unsere Historie und gleichzeitig der Darstellung des Scoreboards benutzt. Sie teilt jedem Spieler eine gameID, den jeweiligen Score, die Dateifragmente usw. zu und macht es möglich die Tabelle nach der gameID zu sortieren um einzelne Spiele wieder einsehen zu können.

4.3 Spiellogik

Mitglieder und Aufgaben:

- David Bachorska (Entwicklung eines Spielmodus, Erweiterung des Simulators)
- Dennis Ness (Entwicklung eines Spielmodus, grundlegende Implementierung des Simulators)
- Tom Kieseling (Entwicklung eines Spielmodus, Erweiterung des Simulators)

4.3.1 Spielmodi allgemein

Die wichtigste Anforderung seitens des Kunden für die Umsetzung der Spielmodi war, dass eines oder mehrere Ad-Hoc Multi-Hop Netzwerkprotokolle diesen zugrundeliegen sollten. Dabei sollen die Entscheidungen für Interaktionen von den Spielenden möglichst selbst getroffen werden,

damit die Spielenden lernen können, wie das jeweils gespielte Netzwerkprotokoll funktioniert. Darüberhinaus war dem Kunden jedoch auch die Spielbarkeit (Fairness, Spannung, Unterhaltung) wichtig. Als Vorbild in diesem Zusammenhang sollte Lasertag dienen. Es war dem Kunden und auch uns dabei klar, dass das Modifikationen am jeweiligen Netzwerkprotokoll erforderlich machen würde.

Als Ideen für die Umsetzung dieser Anforderungen standen zwei grobe Szenarien zur Diskussion: ein Protokoll aus dem Bereich der Wireless Sensor Networks oder der Peer-to-Peer Content Distribution. Nach eingehender Beschäftigung mit einigen Protokollen aus dem Bereich der Wireless Sensor Networks gab es erhebliche Zweifel an der Umsetzbarkeit eines Spiels auf deren Basis unter Berücksichtigung der genannten Spielbarkeitskriterien Spannung und Unterhaltung und nach dem Vorbild von Lasertag. So besteht ein Wireless Sensor Network meist aus vielen Wireless Sensor Nodes, die gut von den Spielenden repräsentiert hätten werden können. Jedoch arbeiten diese Knoten in der Praxis oft zusammen an der Erfüllung eines gemeinsamen Ziels. Wir haben keinen Konkurrenzgedanken, keine egoistischen Motive der einzelnen Netzwerkknoten in Wireless Sensor Networks gesehen, die sich gut zur Umsetzung eines Spiels nach dem Vorbild von Lasertag geeignet hätten. Stattdessen gibt es Ziele in Wireless Sensor Networks wie die Energiesparsamkeit der einzelnen Geräte, deren Umsetzung für uns im Widerspruch zu einem action-geladenen Spiel wie Lasertag standen. Das Spiel auf dieser Basis wäre eher statisch geworden.

Nach Beschäftigung mit Peer-to-Peer Content Distribution erschien diese uns deutlich besser als Grundlage für das Spiel geeignet. Wir haben uns dabei für den bekanntesten Vertreter seiner Art, BitTorrent, entschieden, denn wir haben folgende Analogien zwischen dessen Funktionsweise und Lasertag gefunden, die sich durch alle von uns erstellten Spielmodi ziehen:

1. Bei Lasertag interagieren die Spielenden miteinander, indem sie mit ihrer Spielwaffe auf die Weste eines anderen Spielenden schießen. Diese Interaktion, der Schuss, entspricht in unserem Spiel im Sinne von BitTorrent der Initiierung einer Datenübertragung zwischen den beiden Spielenden.

2. Bei Lasertag hat jeder Spielende standardmäßig das (egoistische) Ziel, möglichst viele seiner Gegner zu treffen bzw. abzuschießen, und er bekommt dafür Punkte. Bei BitTorrent hat jeder Teilnehmer im Netzwerk das (ebenfalls egoistische) Ziel, möglichst schnell die gewünschte Datei(en) von seinen Peers vollständig zu erhalten. Entsprechend vergeben wir im Allgemeinen (wenn nicht anders beim jeweiligen Spielmodus erklärt) auch in unserem Spiel Punkte für erfolgreiche und für das Erreichen des egoistischen Ziels sinnvolle Interaktionen, also Datenübertragungen zwischen den Peers.

3. Bei Lasertag gibt es das Konzept der Unverwundbarkeit(szeit). Wenn ein Spielender einen anderen Spielenden getroffen hat, dann ist der Getroffene üblicherweise für eine gewisse Zeit (einige Sekunden) vor weiteren Schüssen auf ihn geschützt. Das verhindert unfairen Mehrfachbeschuss von einem Spieler innerhalb kurzer Zeit. Diese Zeit kann der Getroffene nutzen, um sich selbst wieder in Deckung zu begeben. Wir haben dieses Konzept im Prinzip aus denselben Gründen in unsere Spielmodi übernommen. Im Kontext von BitTorrent lässt sich diese Unverwundbarkeitszeit jedoch auch einfach als die Zeit erklären, die die durch den Schuss initiierte Datenübertragung (für ein Dateifragment) benötigt. In unserem Spiel kann dieser Interpretation folgend jeder Schütze zwar mehrere Dateifragmente gleichzeitig erhalten, jedoch kein Getroffener mehr als ein Dateifragment gleichzeitig an einen oder mehrere Schützen übertragen. Natürlich wären in der Realität bei BitTorrent mehrere Datenübertragungen in beide Richtungen gleichzeitig möglich und sogar anzustreben. Die genannte Interpretion ist also gleichzeitig eine der Vereinfachungen, die wir zur besseren Spielbarkeit vorgenommen haben.

4. Bei Lasertag gibt es Spielmodi in Teams. Dieses Spiel in Teams lässt sich gut zur Darstellung des kollaborativen Ansatzes von BitTorrent nutzen. Denn trotz der egoistischen Motive der Peers im Netzwerk braucht jeder Peer die anderen Peers und trägt entsprechend auch selbst etwas zur Erreichung der Ziele anderer Peers bei, indem er ihnen bereits erhaltene Dateifragmente zur Verfügung stellt.

Allgemein folgen all unsere Spielmodi den folgenden Annahmen. Diese sind größtenteils Vereinfachungen, die die Komplexität des Spiels reduzieren und es damit leichter spielbar machen.

1. Jeder Spielende besitzt zum Start ein Dateifragment, welches keiner seiner "Peers" (alle anderen Spielenden) besitzt.

2. Es geht insgesamt um eine Datei, die in entsprechend viele Dateifragmente zerlegt ist, wie es Spielende im Spiel gibt.

3. Ziel ist allgemein, bis auf Ausnahmen, die beim entsprechenden Spielmodus beschrieben sind, die eine Datei als Erster vollständig von den Peers zu erhalten.

   Die initiale Verteilung der Dateifragmente auf die Peers sorgt für einen schnellen und relativ einfachen Einstieg in das jeweilige Spiel, weil sich so zu Beginn ein Schuss auf jeden Peer lohnt. Entsprechend nimmt die Komplexität des Spiels für jeden Spielenden mit der Zeit zu. So muss jeder Spielende zunehmend im Auge behalten, welche Dateifragmente er schon hat und von welchem Peer er die fehlenden bekommen kann.

   Wir haben verschiedene Spielmodi entworfen, die grundsätzlich auf den genannten Analogien und Annahmen beruhen. Sie unterscheiden sich teilweise in der Komplexität ihrer Regeln und haben teilweise verschiedene Szenarien als Ausgangspunkt, die entsprechenden Einfluss auf das Spiel und dessen Regeln nehmen.

4.3.2 Spielmodi speziell

Die verschiedenen Spielmodi wurden mittels eines von uns als Simulator bezeichneten Programms entwickelt und ausprobiert. So konnten wir u.a. prüfen, ob die Spielmodi wie erwartet funktionieren und ob bei realistischen Eingaben (also entsprechenden Schusswechseln zwischen den Spielern) Szenarien auftreten können, die entsprechend der Bedeutung des jeweiligen Spielmodus nicht sinnvoll sind. Mehr zu den technischen Aspekten des Simulators ist im Abschnitt **4.3.3** zu finden.

Solo Easy Bei dem Spielmodus „Solo Easy" handelt es sich um einen Spielmodus, welcher schnell implementiert werden sollte. Demnach soll der Spielmodus einfach spielbar sein. Der Spielmodus funktioniert wie folgt: Jedem Spieler wird ein Dateifragment zugewiesen. Wird ein Spieler durch einen anderen Spieler getroffen, bekommt der Schütze das ursprüngliche Dateifragment des Getroffenen. Dabei ist zu beachten, dass der getroffene Spieler sein Dateifragment jedoch behält, d.h. Dateifragmente werden bei Treffern somit kopiert. Das Spiel endet sobald ein Spieler alle existierenden Dateifragmente besitzt.

Solo Advanced Der Spielmodus „Solo Advanced" baut auf dem Spielmodus „Solo Easy" auf. Der Unterschied hierbei liegt darin, dass bei einem Treffer der getroffene Spieler sein Dateifragment jedoch nicht behält, d.h. Dateifragmente werden nun bei Treffern abgegeben. Das bedeutet also insbesondere, dass Dateifragmente nicht kopiert werden. Ein Dateifragment zählt hierbei wie ein Punkt. Des Weiteren kann der Spielmodus in mehreren Runden gespielt werden, wobei eine Runde durch ein Zeitlimit begrenzt wird. Eine Runde gewinnt derjenige Spieler, welcher die meisten Punkte, also die meisten Dateifragmente am Ende einer Runde besitzt.

trifft	fair	unfair
fair	+10 für beide	+20 für Schützen −20 für Getroffenen
unfair	+20 für Schützen −20 für Getroffenen	+10 für Schützen −10 für Getroffenen

Tabelle 1: Punktevergabe im Spielmodus „Share Or No Share“
Zu lesen ist die Tabelle in der Form „*<Eintrag aus erste Spalte> trifft <Eintrag aus erste Zeile>*“.

Teams Advanced Als Variation kann der Spielmodus „Solo Advanced“ auch teambasiert gespielt werden, wobei die Teamzuweisung manuell erfolgt. Gewonnen hat demnach das Team, welches am Ende einer Runde die meisten Punkte hat. Teambeschuss wird hierbei nicht beachtet (ist nicht erlaubt) und führt demnach zu keinem Ereignis (kein Punktabzug o.Ä.).

No Carry No Win „No Carry No Win“ ist ein Spielmodus, der ausschließlich teambasiert gespielt wird, wobei die Teamzuweisung manuell erfolgt. Hierbei wird pro Team ein Spieler zufällig ausgewählt, welcher alle Dateifragmente einer begehrten Datei besitzt, der sogenannten „Carry“. Er hat besondere Bedeutung für das BitTorrent-Netzwerk in dem zugrundeliegenden Szenario, denn da er bereits alle Dateifragmente hat, tritt er nur noch als Seed auf und ist für das Netzwerk (sein Team) besonders wichtig. Nun bricht ein Kampf zwischen den Teams aus, ganz im Stile von Lasertag. In diesem Kampf ist folglich das Ziel jedes normalen Spielers (jedes Spielers, der kein Carry ist), seinen Carry zu verteidigen, ihn also im BitTorrent-Netzwerk zu halten, denn sie wollen von ihm in Zukunft auch noch Dateifragmente erhalten können. Jeder normale Spieler besitzt zu Beginn wie üblich ein Dateifragment. Zu beachten ist hierbei, dass in diesem „Kampf“ Dateifragmente nicht kopiert oder übertragen, sondern nur als Leben gewertet werden. Ein Spieler „stirbt“, scheidet also aus einer Runde aus, wenn er kein Dateifragment mehr besitzt. Das bedeutet, dass ein normaler Spieler nach einem Treffer, der Carry jedoch erst nach mehreren Treffern „stirbt“. Der Spielmodus kann in mehreren Runden gespielt werden, wobei eine Runde dann endet, wenn nur noch ein Carry übrig ist. Das Team, dessen Carry zuletzt übrig ist, gewinnt die jeweilige Runde. Bei diesem Spielmodus wird ebenfalls kein Teambeschuss toleriert.

Share Or No Share Der Spielmodus „Share Or No Share“ soll ein fundamentales Prinzip von BitTorrent besonders hervorheben: die Kollaboration. Jeder neue Peer im Netzwerk profitiert zunächst von den bereits angebotenen Dateifragmenten und sollte anschließend, nach Erhalt des ersten Dateifragments, zur Weiterverteilung der entsprechenden Datei beitragen, indem er seinerseits Dateifragmente anderen Peers zur Verfügung stellt. „Share Or No Share“ wird dafür ausschließlich teambasiert gespielt, wobei die Teamzuweisung zufällig erfolgt. Hierbei werden die Spieler in zwei Teams aufgeteilt, ein faires Team und ein unfaires Team. Wieder wird davon ausgegangen, dass jeder Spieler zu Beginn des Spiels bereits ein Dateifragment erhalten hat. Im fairen Team befinden sich „normale“ Peers, die sowohl Dateifragmente von anderen Peers erhalten als auch selbst Dateifragmente zur Verfügung stellen wollen. Im unfairen Team hingegen befinden sich sogenannte „Power-Leecher“, Peers, die also möglichst schnell alle benötigten Dateifragmente bekommen möchten, jedoch keine zur Verfügung stellen wollen. Die Aufteilung in Teams erfolgt bestmöglich in dem Verhältnis 2:1 (faire:unfaire Spieler), weil uns realistisch erscheint, dass es mehr faire als unfaire Peers gibt. Es gewinnt das Team, welches zum Ende des Spiels die meisten Punkte hat. Dabei werden die Punktzahlen nach Tab. **1** vergeben. Die Motivation für die Punktzahlen ist wie folgt. Offensichtlich werden Punkte nur auf der fairen Seite „erzeugt“. Das soll unterstreichen, dass BitTorrent prinzipiell nur kollaborativ funktioniert, und nur bei fairem Dateiaustausch ein Mehrwert für alle Peers entsteht. Ansonsten spiegeln die Punktzahlen ganz einfach die Motivation der jeweiligen Spieler wider, entsprechend zu treffen

oder getroffen zu werden. So wollen faire Spieler beispielsweise, dass alle anderen Peers im Netzwerk auch fair spielen, also Dateifragmente zur Verfügung stellen. Deshalb bekommen sie mehr Punkte, wenn sie einen unfairen Spieler treffen und ihn damit zum Teilen „zwingen“, als wenn sie einen fairen Spieler treffen. Entsprechend sieht es auf der anderen Seite aus. Unfaire Spieler nutzen also gern die fairen Spieler aus und werden damit mit entsprechend mehr Punkten belohnt. Und für Schüsse innerhalb des unfairen Teams gibt es beim getroffenen unfairen Spieler Punktabzug, weil ein unfairer Spieler mit niemandem „teilen“ möchte. Der Spielmodus hat nur eine Runde, wobei das Spiel endet, sobald ein Spieler alle Dateifragmente gesammelt hat. Wir haben die Spielmodi nur mit dem Simulator und nicht im Praxiseinsatz, also auf einem echten Spielfeld mit echten Spielern, testen können. Dadurch könnten uns Faktoren, die den Spielverlauf in der Praxis beeinflussen und damit eventuell Anpassungen an den Annahmen und bspw. Punktzahlen der Spielmodi nötig machen würden, völlig unbekannt geblieben sein. Besonders für das Balancing von Parametern wie Punktzahlen und die Zuweisung der initialen Dateifragmente hätte ein automatisches Testen sinnvoll sein können. Andererseits hätten auch bei automatischen Tests Rahmenbedingungen bzw. Testcases festgelegt werden müssen, die ohne Daten aus der Praxis nur schwer realistisch festgelegt hätten werden können.

Alle drei Mitglieder der Spiellogik Gruppe haben Ideen für Spielmodi vorgeschlagen, die gemeinsam diskutiert wurden. Die Spielmodi „Solo Advanced“ und „Teams Advanced“ sind Ideen von David Bachorska. Der Spielmodus „No Carry No Win“ wurde von Dennis Ness und „Share Or No Share“ von Tom Kieseling entwickelt.

4.3.3 API zu Services Komponente

Als grundsätzliche Designentscheidung verwendet jeder Spielmodus ein eigenes Lua Skript. Dies ist sinnvoll, da sich die Spielmodi teils massiv unterscheiden, denn es gibt z.B. je nach Spielmodus vollkommen verschiedenen Start- und Siegbedingungen. Somit muss es zwischen der Spiellogik und der Services Komponente klar sein, welches Lua Skript bzw. Spielmodus gerade gespielt wird. Dieses Problem wird gelöst durch die Art und Weise, wie ein Lua Skript in C++ eingebunden wird.

Denn die API zu der Services Komponente nutzt die von Lua bereitgestellte C++ API. Somit ist es problemlos möglich, sowohl Funktionen von C++ heraus in den Lua Skripten aufzurufen, als auch von den Lua Skripten aus C++ Funktionen aufzurufen. Hier wurden beide Richtungen genutzt und gebraucht, wobei über das gesamte Spiel und auch der Anzahl der Funktionen hinweggesehen die meisten Aufrufe von den Lua Skripten zu den C++ Funktionen laufen.

Die Entwicklung der API ging stark einher mit der Entwicklung des Simulators, da dieser die API verwendet. Es war also erforderlich möglichst schnell eine stabile API zu haben, um mit der Entwicklung der Spielmodi beginnen zu können. Aufgrund dieser frühen Festlegung blieb zwar die grundlegende API im Laufe der Entwicklung erhalten, hat aber einige Erweiterungen erfahren.

Um einen Überblick über die API zu erhalten, werden hier nun die drei Hauptbestandteile, die die API abdeckt, erläutert: Die Spielinitialisierung, der normale Spielablauf und zum Schluss das Spielende. Wie bereits erwähnt, läuft die meiste Kommunikation von den Lua Skripten zu der Services Komponente. Ausnahmen bilden hier der Start des Spiels, wenn die Zeit einer Runde abläuft und wenn ein Spieler einen anderen Spieler trifft. Ein Großteil der Funktionen der API beschäftigt sich mit der Initialisierung des Spiels. Dazu werden eine Reihe von Spielparametern abgefragt, wie z.B. die Anzahl der Spieler, die Rundendauer oder auch die Zeit der Unverwundbarkeit eines Spielers. Diese Informationen werden in den entsprechenden Lua Skripten verarbeitet und weitere Funktionen in C++ werden zur Dateiverwaltung aufgerufen. Ein gutes Beispiel dafür ist die Festlegung der Anzahl der Dateifragmente jedes Spielers. Um diese Aufgabe zu bewältigen wird zunächst über die API die Anzahl der Spieler abgefragt. Dann wird ebenfalls über die API der C++ Komponente mitgeteilt, wie viele Dateifragmente

jeder Spieler maximal besitzen kann. Anschließend bestimmt das Lua Skript, welcher Spieler welche Dateifragmente besitzt. Diese Information wird dann ebenfalls über die API an die C++ Komponente übergeben.

Nachdem die Spielinitialisierung abgeschlossen ist, wird das Spiel von der C++ Komponente aus gestartet. Von nun an besteht die Hauptaufgabe der API darin, auf das Aufrufen der shootPlayer(source, target) Funktion zu warten. Sobald ein Schuss von der Hardware ausgelöst und von der Services Komponente vorverarbeitet wurde, wird diese shootPlayer Funktion aufgerufen. Diese Funktion ist der Hauptteil eines jeden Lua Skripts, dann hier wird entschieden, ob der Schuss zulässig war, z.B. ob der Gegner überhaupt verwundbar ist, aber auch wer wie viel Punkte und/oder Dateifragmente erhält. Deshalb ist diese Funktion recht komplex und enthält letztendlich die gesamte Logik eines Spielmodus.

Grundsätzlich gibt es zwei Möglichkeiten, wie ein Spiel enden kann und somit auch wie die API genutzt wird: 1.) Nach einem Abschuss tritt eine beliebige Siegbedingung ein und es werden entsprechende C++ Funktionen von dem Lua Skript aufgerufen, um das Ende des Spiel zu signalisieren und die Auswertung des Gewinners zu starten. 2.) Es läuft ein Zeitlimit ab. In diesem Fall ruft ein Timer aus der C++ Komponente heraus eine spezielle Lua Callback Funktion auf. Diese spezielle Funktion hält dann das Spiel an und ruft weitere Funktionen auf. Diese zweite Variante ist ein gutes Beispiel für eine Erweiterung der API, die sich sehr gut in die bestehende API integrieren lassen hat.

Es stellt sich also die Frage, inwiefern Verbesserungen an der API und allen davon unmittelbar betrofenen Bestandteilen möglich gewesen wären. Letztendlich ist die API sehr umfassend geworden und deckt alle Funktionalitäten ab, die für die Spielmodi notwendig sind. Allerdings verwendet die API auch sehr viele sehr kleinteilige Funktionen, die zu größeren, umfassenderen Funktionen zusammengefasst werden könnten. Dadurch wäre die API übersichtlicher geworden, ohne an Funktionalität und Flexibilität einzubüßen. Eine weitere Verbesserungsmöglichkeit setzt nicht direkt bei der API an, sondern bei den Funktionen innerhalb der Lua Skripte. Dort hätte man für eine bessere Übersicht mehr Funktionen auslagern können und auch von dem Konzept abrücken können, immer nur ein Skript für einen Spielmodus zu verwenden. Denn dadurch wären die Skripte noch sehr viel modularer gestaltbar gewesen. Dies alles wäre der Übersicht zugute gekommen und hätte beim Debugging helfen können.

Die grundsätzliche Entwicklung der API wurde gemeinsam von allen drei Mitgliedern der Spiellogik Gruppe durchgeführt. Die erste Umsetzung der API in Form des Simulators geschah durch Dennis Ness. Im Verlaufe des Projekts wurden jedoch eine Reihe von Erweiterungen von David Bachorska (z.B. Teamhandling) und Tom Kieseling (z.B. Teamzuweisung von Lua aus) zu der API hinzugefügt. Allerdings gab es auch Erweiterungen der API, die von Personen der Services Gruppe ausgingen. So hat z.B. Jan Arne Sparka eine Erweiterung zur Ansteuerung der LEDs hinzugefügt.

5 Zusammenfassung

Im Abschnitt zur Systemarchitektur wurde unser Plan vorgestellt, mit dem wir die Aufgabenstellung lösen wollten. In diesem Kapitel wird erläutert, welche Teile dieses Plan tatsächlich umgesetzt wurden, und diskutiert, inwieweit diese Lösung die Aufgabe erfüllt.

5.1 Auswertung

Es gibt zwei Sichten darauf, wieviel von der geplanten Architektur umgesetzt wurden: zum einen die Implentierung der einzelnen Komponenten, zum anderen die Integration derselben. Um eine technische Machbarkeit der Lösung zu demonstrieren, reicht es aus, mit Unit Tests die Funktionalität der Einzelkomponenten zu beweisen. Allerdings war als Endprodukt ein Spiel gefordert, wofür es notwendig ist, dass die Integration reibungslos funktioniert. Das heißt, ein beliebiger externer Spieler kann das System verwenden, ohne Kenntnisse über deren interne Funktionsweise zu haben.

Die einzelnen Komponenten, die geplant waren, wurden alle implementiert und zumindest teilweise getestet. Die Spielelogikgruppe konnte mit einem Simulator demonstrieren, dass die einzelnen Spielmodi das Soll-Verhalten generieren und die Hardware-Gruppe konnte mit Stubs demonstrieren, dass Infrarotabschüsse an den Services-Client durchgereicht und LED Events korrekt angezeigt werden. Die Services-Gruppe hat die Funktionalität der Webseite demonstriert, indem sie manuell Einträge in die Datenbank hinzufügten und bestätigten, dass jene Änderungen auf der Webseite übernommen werden.

Die Integration der Komponenten ist unvollständig. Es lässt sich nachweisen, dass der Server mit LUA kommuniziert. Allerdings lassen Logs darauf schließen, dass Parameter in der falschen Reihenfolge an LUA weitergegeben werden. Die Webseite-zu-Server-Integration über die Datenbank funktioniert für einfache Spielmodi, bei komplexeren bringen jedoch Deadlocks von Mutex-Objekten den Server zum Absturz. Es kam außerdem zu diversen Fehlern bei Zugriffen des Servers auf die Datenbank. Der Services-Server und der Services-Client können sich miteinander verbinden und Abschussinformationen werden ausgetauscht. Allerdings ist die Verbindung selbst teilweise instabil und die Clients verlieren die Verbindung oder können sich nicht erneut verbinden. Diese Problem machen derzeit einen Neustart der Systemd-Units auf allen Geräten notwendig, wenn ein neues Spiel gestartet wird. Die Verbindung der Client/Server-Programme mit Systemd ist noch instabil. Es dauert teilweise Minuten, bis eine Unit gestoppt werden kann. Die Hardware-API ist in der Lage, zuverlässig Abschussinformationen an den Services-Client weiterzugeben. Allerdings werden derzeit keine LED-Events an die Hardware-API gesendet. Die Integration der Hardware-API mit der darunter liegenden Hardware und Systemd ist stabil und verursacht keine Probleme.

5.2 Ausblick

Aus dem letzten Abschnitt geht hervor, dass ein Spiel bis zu einem gewissen Punkt demonstriert werden kann. Ein ausgereiftes Spiel ist jedoch noch nicht möglich und von Spaß kann nicht die Rede sein. Die geplante Architektur erfüllt dennoch die Aufgabenstellung und sie wurde größtenteils umgesetzt. Viele der genannten Probleme könnten nach einigen Tagen Bugfixing behoben werden und sobald die Integration der Komponenten stabil ist, kann ein richtiges Spiel gespielt werden. Ob man dabei auch Spaß hätte, bleibt offen. Dies lässt sich mit einem Simulator nicht demonstrieren und wahrscheinlich wären viele weitere Tests notwendig, bis das Produkt ausgereift und tauglich für die Allgemeinheit wäre.