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 13. März 2018 um 18:59 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 Spiel 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 verwendendet werden sollten und die Spielgeräte über Akkumulatoren betrieben werden sollten. Die Sensoren sollten entweder am Spielgerät befestigt sein, oder es sollte 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.

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 können 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 hat 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.

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. So gibt es im Netzwerk etwa Teilnehmer, die möglichst viel Datenvolumen von anderen beziehen wollen, ohne selber eine entsprechende Upload-Bandbreite anzubieten. Dieses Element der Fairness sollte eine zentrale Komponente im Spiel sein.

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 habe 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.

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, die Technik ist zuverlässig und Verbindungsprobleme werden größtenteils durch den TCP/IP Stack des Betriebssystems gelöst. 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 Desktop PC 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 behaupten. Dafür gibt es zwei Ansätze die verfolgt wurde: 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 gelöst, die mit verschiedenen Farben anzeigen, ob ein anderer Spiel getroffen wurde, ob man unverwandbar ist, etc. Dies ist eine preiswerte Möglichkeit, die durch die GPIO Anschlüsse der Raspberry Pis unterstützt wird.

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

In diesen Entscheidung spiegelt sich wieder, 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, daher ist es in Ordnung, wenn der Spieler einige Sekunden braucht, um auf die Website 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. Ein Spieler darf nicht konstant schießen, es muss also eine Art Abklingzeit geben. Dafür wurde an den Geräten ein Button angebracht, sodass das Gerät nur schießt, wenn der Button gedrückt wurde und nicht mehr schießen kann, bis er wieder losgelassen wurde.

3. 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 Spielelogik möglichst vom Backend getrennt sein soll. Deswegen wurde sie in Form von LUA-Skripts gekapselt und wird vom Server eingebunden. Die Schnittstelle zwischen diesen beiden Komponenten ist ein Informationsaustausch. Der Server gibt Spielparameter und Abschussinformationen an LUA weiter und die Spielelogik gibt die resultieren Auswirkung auf den Spielstand zurück, damit dieser auf der Website und den Geräten 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 habe 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 „Spielelogik“ 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 ist ein Raspberry Pi Model B+. Er hat zwei physikalische Netzwerkinterfaces: WLAN und Ethernet. Aufgrund der geringen Größe lässt er sich leicht überall hin transportieren. Für ein Spiel benötigt er lediglich eine Stromversorgung, um als internetfähiger Access Point zu dienen braucht darüber hinaus einen Ethernet Uplink, über dem er per DHCP eine IP Adresse beziehen kann und Internetzugang hat. Zum Debugging sind darüber hinaus Bildschirm und Tastatur nützlich.

Auf dem Server ist Raspbian Lite installiert. Das Gerät lässt sich auch durch einen beliebigen anderen PC austauschen, dass besagte Netzwerkschnittstellen besitzt und auf dem Debian installiert werden kann (was den meisten gängigen Prozessorarchitekturen möglich ist). Raspbian wurde vor allem wegen seiner weiten Verbreitung und Stabilität gewählt. Für Probleme bei der Serverinstallation kann im Regelfall eine Suche in Debianforen und ähnlichem eine Lösung gefunden und auf Raspbian übertragen werden.

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 Client 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. 2G) 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 Größe 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 eingestellt. 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 im WLAN Netz 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 den 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, mysql auf dem Server und 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 Services 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 erfordlichen Dateien. Wenn die 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 werden müssen und nicht kompiliert werden müssen. Die Website 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 3 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 2 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 3 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 3 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 selber sehr rauschanfällig ist, da Flanken in der Theorie überspielt werden könnten.

Idee 3 Der Vorteil besteht hier darin, dass die wenigste 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 sie dafür gesorgt hat, dass das Troubleshooting später ebenfalls leichter war.

An Nachteilen ist hier aufzulisten, dass es von den 3 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 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. Die API kommuniziert nach oben mit `services-client` über Unix-Sockets und Protocol Buffers. Nach unten kommuniziert sie mit dem Infrarot-Treiber, indem dessen Source-Code 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 3 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.

Wir haben jeweils eine rote, eine gelbe und eine grüne LED, die wie auf einer Ampel angeordnet sind. Für Menschen mit Rot-Grün-Schwäche könnte man 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 falsch-angeschlossene 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 Ampel, von rot über rot-gelb auf grün. Die grüne LED zeigt an, dass das Spiel läuft. 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, wenn 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.

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.)

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.

Die Hardware-API läuft im fertigen System als Daemon. Sie kann aber auch als gewöhnliches Programm gestartet werden, was zum 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)

TODO

4.3 Spielelogik

Mitglieder und Aufgaben:

- David Bachorska
- Dennis Ness
- Tom Kieseling

TODO

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, für dieses ist notwendig, 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 wurde alle implementiert und zumindest teilweise getestet. Die Spielelogikgruppe konnte mit einem Simulator demonstrieren, dass die einzelnen Spielmodi das erwartete Verhalten generieren und die Hardwaregruppe konnte mit Stubs demonstrieren, dass Infrarot Abschüsse an den Services Client durchgereicht werden und LED Events korrekt angezeigt werden. Die Servicesgruppe hat die Funktionalität der Website demonstriert, indem sie manuell Einträge in die Datenbank hinzufügten und bestätigten, dass jene Änderungen auf der Website 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 Website zu Server Integration über die Datenbank funktioniert für einfache Spielmodi, bei komplexeren bringen allerdings Deadlocks von Mutex Objekten den Server zum Absturz. Es kam außerdem zu diversen Fehlern bei Zugriffen des Servers auf die Datenbank. Der Server und der 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 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, jedoch ist ein ausgereiftes Spiel noch nicht möglich, von Spaß kann nicht die Rede sein. Die geplante Architektur erfüllt jedoch 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.