

## **WEBSOCKETS OCH LONG POLLING**

För nätverkskommunikation i situationer med hög trafik och realtidskrav

## **WEBSOCKETS AND LONG POLLING**

For network communication in situations of high traffic and real-time requirements

Examensarbete inom huvudområdet Datalogi  
Grundnivå 30 Högskolepoäng  
Vårtermin 2012

Christian Cromnow

Handledare: Henrik Gustavsson  
Examinator: Thomas Fischer

# Sammanfattning

Då webben nu består av dynamiska hemsidor och kraftfulla applikation blir även kraven på kommunikationshastigheter större. Detta arbete har tittat på den äldre och populära tekniken Long Polling och ställt den i förhållande till HTML5s nya websocket API. Igenom att bygga ett multiplayer spel för webbläsaren utan några pluggins ställdes teknikerna mot varandra för att se vilken som presterade bäst och visade sig mest effektiv för användning i den typen av applikation. WebSockets visade sig klara av alla tester med marginal mot de värden relaterad forskning visat på är minimum kraven för att kunna realisera realtidsapplikationer. Long Polling föll kort och visade sig vara svagare på alla punkter i förhållande till WebSockets.

**Nyckelord:** Realtid, Webbapplikationer, HTML5, WebSockets, Long Polling



# Innehållsförteckning

Innehållsförteckning .....	0
<b>1</b> <b>Introduktion</b> .....	<b>1</b>
<b>2</b> <b>Bakgrund</b> .....	<b>2</b>
2.1 <b>HTML5, applikationer och spel på webben</b> .....	<b>2</b>
2.2 <b>Kvalitet</b> .....	<b>3</b>
2.2.1 <b>Quality of Service och latens i spel</b> .....	<b>4</b>
2.2.2 <b>Quality of Service i Webbapplikationer</b> .....	<b>5</b>
2.3 <b>Komunikationstekniker för Webbapplikationer</b> .....	<b>5</b>
2.3.1 <b>Long Polling</b> .....	<b>6</b>
2.3.2 <b>WebSockets</b> .....	<b>7</b>
<b>3</b> <b>Problem</b> .....	<b>9</b>
3.1 <b>Delmål</b> .....	<b>9</b>
<b>4</b> <b>Metod</b> .....	<b>10</b>
4.1 <b>Mätning</b> .....	<b>10</b>
4.1.1 <b>Testmiljö</b> .....	<b>10</b>
4.1.2 <b>Enkät</b> .....	<b>11</b>
<b>5</b> <b>Genomförande</b> .....	<b>12</b>
5.1 <b>Klientsidan</b> .....	<b>12</b>
5.1.1 <b>Dead Reckoning</b> .....	<b>13</b>
5.2 <b>Serversidan</b> .....	<b>14</b>
5.2.1 <b>WebSockets med Node.js och Socket.io</b> .....	<b>15</b>
5.2.2 <b>Long Polling med jQuery, PHP och MySQL</b> .....	<b>15</b>
5.3 <b>Kommunikation mellan klient och server</b> .....	<b>16</b>
5.3.1 <b>Uppdateringar mellan klient och server</b> .....	<b>16</b>
5.3.2 <b>WebSockets medelände struktur</b> .....	<b>18</b>
5.3.3 <b>Long Pollings medelände struktur</b> .....	<b>19</b>
<b>6</b> <b>Analys</b> .....	<b>20</b>
6.1 <b>Latens/Round Trip Time (RTT)</b> .....	<b>20</b>
6.1.1 <b>LAN, 1 Klient</b> .....	<b>21</b>
6.1.2 <b>Internet, 1 Klient.</b> .....	<b>21</b>
6.1.3 <b>Internet, 3 Klienter.</b> .....	<b>22</b>
6.2 <b>Nätverkspaket och overhead</b> .....	<b>23</b>
6.3 <b>Enkät</b> .....	<b>23</b>
<b>7</b> <b>Resultat</b> .....	<b>25</b>
7.1 <b>Resultatsammanfattning</b> .....	<b>25</b>
7.2 <b>Diskussion</b> .....	<b>26</b>
7.3 <b>Framtida arbete</b> .....	<b>27</b>
<b>Referenser</b> .....	<b>29</b>

# 1 Introduktion

Med all ny innovation inom webben skapas nya krav hos utvecklare. En av webbens största svagheter är enligt Carl Gutwin et al. (2011) att den saknar bra stöd för realtidsapplikationer. Carl Gutwin et al. (2011) menar på att AJAX har fungerat bra för att skapa dynamik i webbapplikationer men att det är ett för stort slöseri på resurser. För att kunna skapa en dubbelriktad kommunikation mellan en klient och en server för webbapplikationer har W3C implementerat WebSockets (W3C, 2011 och Fette & Melnikov, 2011).

Detta arbete avser att bygga en realtidswebbapplikation för att undersöka om det är möjligt att bygga en webbapplikation som kan köras och få uppdateringar i realtid. Arbetet kommer behandla teknikerna WebSockets och Long Polling för att jämföra prestandan och hitta styrkor och svagheter hos teknikerna. Webbapplikationen kommer vara ett enkelt spel som använder 2D grafik och tilemaps. Användare kommer ansluta till spelet och spela mot varandra i realtid. Detta arbete kommer även använda Dead Reckoning för att minimera skadan av latenser över 200ms i nätverkskommunikationen (Pantel & Wolf, 2002a). Det kommer även genomföras en simpel användarstudie för att undersöka om användare av spelet upplever en synkroniserad nätverksmiljö och hur användare uppfattar dess påverkan på spel mekaniken.

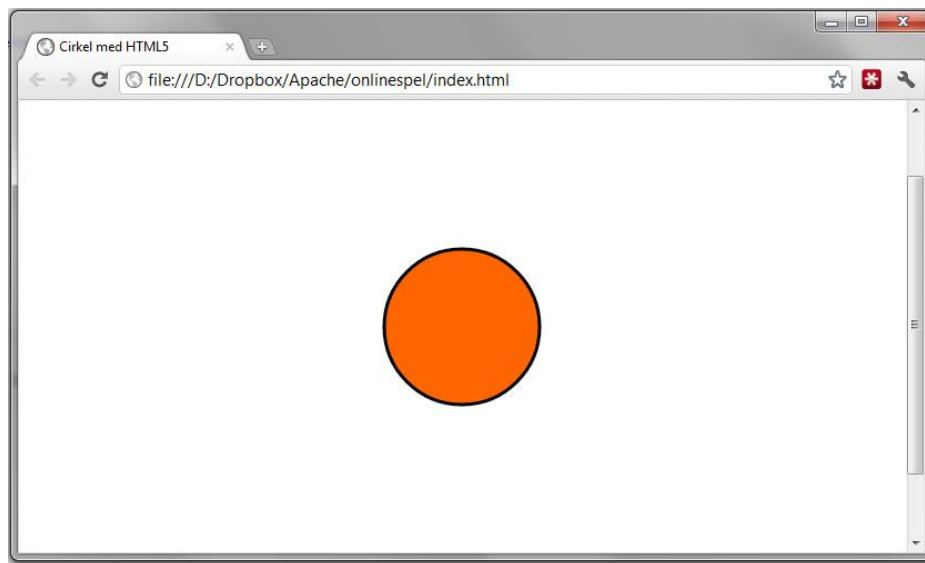
## 2 Bakgrund

Webben har från att vara en tjänst för få, växt till ett globalt nätverk. Ett nätverk som gått från statiska till dynamiska hemsidor. Från hemsidor till applikationer och enorma program som förut enbart kunde köras på en dator efter en installationsprocess.

Taivalsaari och Mikkonen (2011) menar på att det har funnits många svagheter hos webben och pekar främst på de som har med prestanda att göra. Carl Gutwin et al. (2011) menar också på att mer avancerade program även ställer högre krav. Till exempel på Quality of Service. Dvs att ett realtidsprogram faktiskt körs i realtid. Bland program som kräver låg latens och i vissa fall hundradelsprecision på uppdateringar tillhör spel. För att förstå Quality of Service för webben behöver vi först förstå vilka typer av applikationer det finns, både på webben och lokalt på datorn. Och hur de program och spel som vi normalt har behövt installera lokalt på hårddisken kan realiseras på webben.

### 2.1 HTML5, applikationer och spel på webben

Med HTML5 kom nya funktioner och tillämpningar för webben. Med hjälp av element som canvas, video och audio kan utvecklare skapa spel och multimedia sidor helt utan att behöva använda någon typ av plugin. Taivalsaari och Mikkonen (2011) hävdar även att webbapplikationer inte bara fått så stor slagkraft för att de inte behövs installeras på datorn, utan även för att utvecklarna kan garantera en plattformsoberoende applikation. Istället för att behöva koda programmet för olika operativsystem kodas programmet för att fungera i olika webbläsare.



**Figur 1** En cirkel i ritad i canvas-elementet på en hemsida.

Carl Gutwin et al. (2011) listar några vanliga krav hos grupprogram och om webbt teknologier kan hantera dessa krav eller inte:

- *Varierande QoS*. Applikationen ska kunna växla mellan olika typer av QoS beroende på om det är lates eller pålitlighet som ska prioriteras. Problemet med webben är att den enbart stöder TCP. För att kunna sänka latensen skulle eventuellt en UDP anslutning fungera bättre.

- *Anpassa upplösningen* för olika maskiner. Även om gruppprogram körs på datorer eller mobiltelefoner bör gränssnittet anpassa sig efter skärmstorleken. För att lösa detta för webbsidor kan CCS tekniker användas eller till exempel verktyg som jQuery Mobile användas.
- *Synkronisering av strömmar*. När gruppprogram använder flera typer av interaktion är det ofta viktigt att dessa kan synkroniseras med varandra, som exempel video och ljud. Om någon användare visar någonting i videon och samtidigt förklarar detta med hjälp av en mikrofon är det viktigt att försöka sträva efter en synkroniserad ström.
- *Grafik*. I dagsläget finns det ett antal spel som både har mycket hög nätverksprestanda och samtidigt väl detaljerad och avancerad grafik. Webben har i nuläget inte stöd för att rendera och köra lika avancerad grafik som vanliga skrivbordsapplikationer, men det kommer allt närmre med tekniker som WebGL och canvas-elementet. Se figur 1.
- *Cross-plattformstöd*. Detta är ett vanligt problem för utvecklare av gruppprogram. Eftersom det finns flera användare som ofta använder olika plattformar behöver programmet ofta fungera i Windows, Linux, iOS och även på andra plattformar. Detta är den standardiserade webbens starkaste punkt, det som funkar i Firefox i windows funkar även i Firefox i iOS. Men här är de olika webbläsarnas teknologi problemet istället, dvs. det som fungerar i Google Chrome fungerar inte alltid i Microsoft Internet Explorer.

Taivalsaari och Mikkonen (2011) diskuterar även andra stora landvinningar för HTML5 i form av stöd för offline applikationer, localstorage, Asynkron skriptladdning och drag-and-drop support. Mycket av de nya i HTML5 är utvecklat för att hjälpa utvecklare att skapa applikationer som liknar vanliga program. Med hjälp av det nya canvas-elementet kan utvecklare nu skapa riktig vektorgrafik direkt i webbläsaren. Med denna nya teknik har flera utvecklare skapat flera sorters olika spel för webben. Utan att använda något plugin är det i nuläget svårt, omöjligt, att skapa avancerad grafik i webbläsaren som skulle kunna tävla med de spel som finns att installera lokalt på datorn. Webben har utvecklats kraftigt under den senaste tiden och blir mer och mer avancerad. Några exempel på detta är till exempel spelen Angry Birds™ och Cut the Rope™ som går att spela direkt i webbläsaren (Rovio Entertainment Ltd, 2011 och ZeptoLab, 2012).

Då grafiken för webben gått framåt är det ändå bara en av delarna som krävs för att kunna skapa ett riktigt spel. Användare nöjer sig inte med att det är fint, det bör även fungera.

## 2.2 Kvalitet

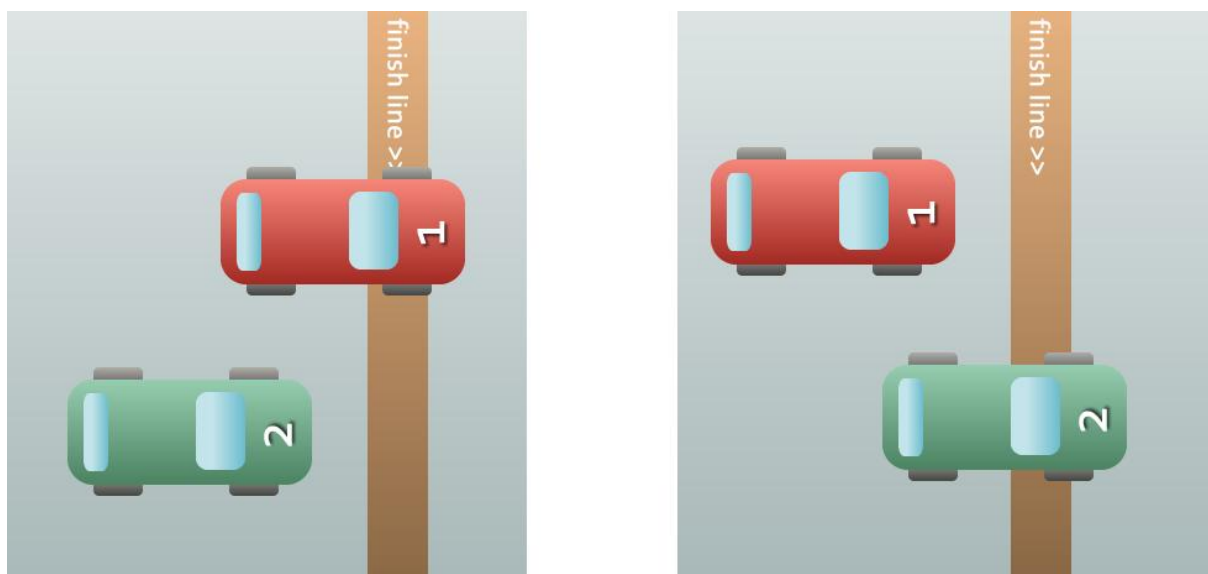
För att till exempel spel ska fungera krävs det att det kör i en viss bilduppdateringsfrekvens. Hur hög bilduppdateringsfrekvensen skall vara beror ofta på vilken typ av spel det gäller. Enligt Mark Claypool et al. (2006) har bilduppdateringsfrekvensen en stor inverkan på hur väl ett spel fungerar. Ett spel som körs i 3 FPS (Frames per second, bilduppdateringar per sekund) är i princip ospelbart och att förändringen är märkbar i upp till 60 FPS. Denna undersökning gjordes på spelet Quake III.

I multiplayer spel blandas även nätverksstrukturen in i bilden och ett krav på Quality of Service uppstår. Det betyder att till exempel om två användare spelar mot varandra och den ena användaren väljer att hoppa med sin karaktär, bör den karaktären även hoppa på den andra användarens klient. Och detta bör ske inom rimlig tid. Carl Gutwin et al. (2011) hävdar

att i ett vanligt FPS (First-Person Shooter) spel sker ca 20 nätverksuppdateringar per sekund från klient till server. Storleken på dessa uppdateringar varierar beroende på hur många klienter som är anslutna till spel servern och vilket spel det rör sig om.

### 2.2.1 Quality of Service och latens i spel

Enligt Tristan Henderson & Saleem Bhatti (2003) kommer en försämrad QoS märkas av användare i ett nätverksspel och detta leda till att användarna antingen lämnar spelet eller väljer att inte delta över huvud taget. Tristan Henderson & Saleem Bhatti (2003) menar även att i ett FPS (First Person Shoter) spel vill inte användarna spela på servrar där deras lates överskrider 225ms till 250ms. Användarnas tolerans varierar dock från spel till spel, bland annat skriver Pantel & Wolf (2002b) att latensen i ett bil spel inte bör överskrida 100ms. Christian Schaefer et al. (2002) använder en MOS (Mean Opinion Score) och testar ett skjutspel kallat XBlast och hävdar att en tolererad latens för det spelet ligger på 139ms.



**Figur 2** En illustration av två bilar som passerar mållinjen vid olika tillfällen för de olika klienterna. Till höger ser vi hur den röda bilen (1) vinner loppet medan den gröna bilen (2) vinner till vänster. De två bilderna visar de olika spelarnas skärmar och hur de upplever att loppet slutade.

Ett av de största problemen med hög latens är att det tar en stund för informationen att nå alla klienter, till exempel som för nya positioner och händelser att uppdateras. Det vill säga alla objekt i spelet är inte på samma ställe för alla klienter. Pantel & Wolf (2002a) beskriver problemet med ett exempel från ett bilspel, där en två klienter spelar emot varandra. När de kör över mållinjen ligger de precis bredvid varandra, men eftersom det finns en liten latens tar det en stund för motspelarens position att uppdateras, därför hamnar den lite efter. Båda spelarna tror därför att de har vunnit och deras installation av spelet har ingen bättre information och sätter sin respektive klient som vinnare av loppet (Se figur 2).

För att minska skadan av detta har en konstant och mer pålitlig teknik tagits fram. Denna teknik kallas för Dead Reckoning. Men hjälp av Dead Reckoning går det att förutspå var objektets nästa position kommer vara. För att kunna förutse så riktigt som möjligt används olika sorts scheman beroende på vad det är för typ av spel och vad utvecklarna av spelet tror kommer stämma mest överens med vad användarna förväntas göra. Pantel & Wolf (2002a)

testade olika typer av scheman för Dead Reckoning i olika typer av spel och kom fram till att ett bra förväntat utfallsschema ger ett bra resultat för alla spelstilar om de jämförs med andra scheman.

### 2.2.2 Quality of Service i Webbapplikationer

Många webbapplikationer har inte brytt sig inte om QoS. Enligt Conti & Kumar (2001) kräver webbapplikationer att multimedia och data ska kunna skickas i realtid. Med detta krav skapas en QoS upplevelse för användarna. Denna QoS blir en dominerande faktor för hur väl webbapplikationen kommer tas emot och uppskattas av användarna.

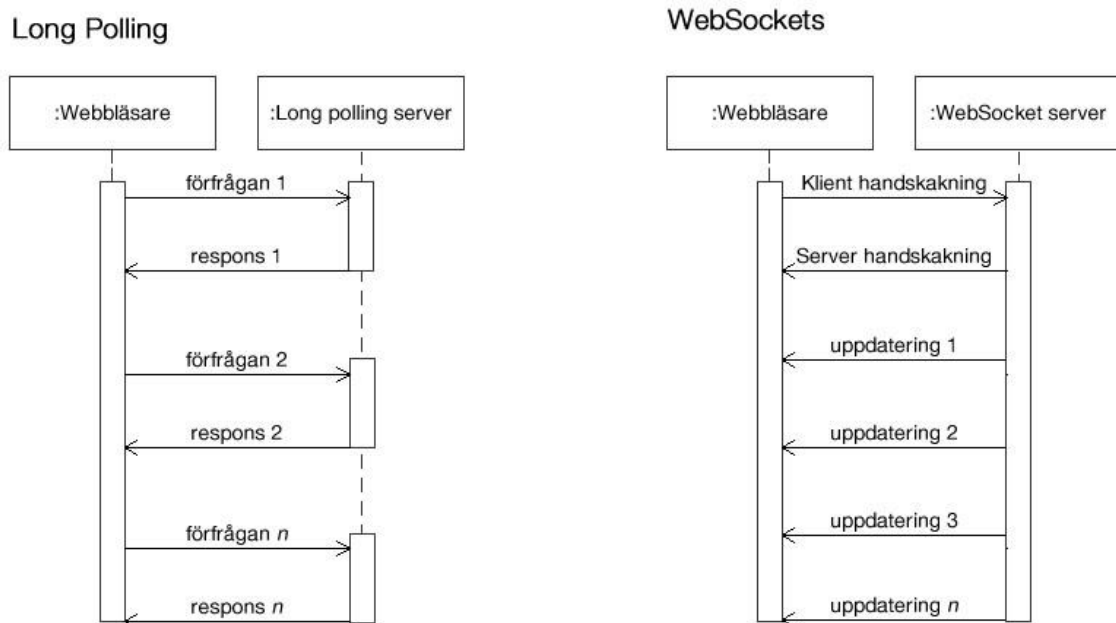
Carl Gutwin et al (2011) skriver att det ofta finns olika typer av data meddelanden med olika QoS krav i realtidssystem. Författarna beskriver de olika grupperna som:

- *Sekventiell turordning.* Detta skulle gälla för kort- och brädspel där en process körs och de andra väntar. Författarna hävdar även att dessa processer oftast är ganska få och att det sällan sker flera processer fort i följd. Detta skulle även fungera bra vid fler användare eftersom det fortfarande enbart är en användare som kan utföra någonting åt gången.
- *Text chat.* För nästan alla moderna multiplayer spel finns det någon typ av chatfunktion. Författarna menar på att texten i chatten antingen kan skickas direkt från servern till användarna när texten nått servern eller att texten skickas i bitar med hjälp av en timer.
- *Muspekarrörelser.* Att få musrörelser med mjuka rörelser över nätet är svårt, det kräver många meddelanden (20-30 i sekunden). Det är även svårt att förutspå vart muspekaren kommer att ta vägen. Författarna skriver även att det går att förfinas med till exempel motion blur men att det bästa är om det går att få informationen i realtid.
- *Karaktärrörelser.* I spel används ofta en avatar som användaren navigerar igenom olika miljöer. Här skickas också små meddelanden ofta, men när det finns flera spelare på en server så blir trafiken ändå hög och tekniker behövs implementeras för att förminska skadorna av latensen som uppstår till följd av den höga trafiken. Till exempel Dead Reckoning.
- *Ljud och bild.* Exempelvis video- och eller ljudkommuniceringsprogram ställer höga krav på nätverksstrukturen. Detta är just nu inte relevant för webben i mån om webbkamera och mikrofon, då HTML i nuläget inte har någon standard som stödjer användandet av dessa.

Hög QoS i kombination med Dead Reckoning teknik är att föredra. Det kan fortfarande förekomma att användarens personliga nät är belastat och det uppstår en fördröjning. För att kunna förstå vilka möjligheter vi har för realtid på webbapplikationer behöver vi titta på vilka kommunikationstekniker det finns och hur väl dessa kan uppnå kraven på god QoS.

## 2.3 Kommunikationstekniker för Webbapplikationer

För att kunna skapa applikationer som klarar av att möta de olika QoS krav som ställs på realtidssystem tittar vi på av som finns för alternativ på webben. De flesta webbapplikationerna lider inte av kravet att de måste vara blixtnabb uppdatering och har därför använt polling och Long Polling för att skapa ett dynamiskt innehåll.



**Figur 3** Är ett UML diagram som visar nätverksströmmar för en webbapplikation som konstant skickar ny data från en server till en klient.

### 2.3.1 Long Polling

HTTP saknar stöd för att sprida data till klienterna så fort det finns ny data tillgänglig. Ett sätt att lösa detta är att använda tekniken AJAX. AJAX gör det möjligt för klienten att få ny data från servern utan att ladda om sidan. XHR (XMLHttpRequest) är en viktig del av AJAX. XHR är ett API som har klient funktioner för att transportera data mellan en klient och en server enligt W3C (2011). JavaScript funktioner hämtar sedan data fortlöpande från en server. Detta kallas polling (Se figur 3). Carl Gutwin et al (2011) menar på att AJAX har fungerat bra för att skapa dynamik i webbapplikationer men att en ny socket anslutning och ett nytt http-meddelande måste skapas för varje uppdatering. Dessa uppdateringar menar författarna på att det är ett slöseri på resurser.

Detta går delvis att lösa igenom att använda någonting som heter Long Polling. Med Long Polling skapas en liknande effekt där applikationen skickar en förfrågan till servern där den sedan ligger och väntar på att en uppdatering skett, sedan skickas svaret tillbaks och behandlas av klienten och därefter skickas förfrågan till servern igen. Det finns även ett annat problem med denna metod och det är skalbarhet. Eftersom klienterna skickar förfrågningar, i form av loopar som ligger och kör på servern tills en uppdatering sker, leder detta till att desto fler klienter som använder applikationen desto fler loopar kommer köras på servern. Carl Gutwin et al. (2011) hävdar också att när trafiken blir hög, i ett realtidssystem sker uppdateringarna så pass fort att det blir precis som att använda vanlig polling.

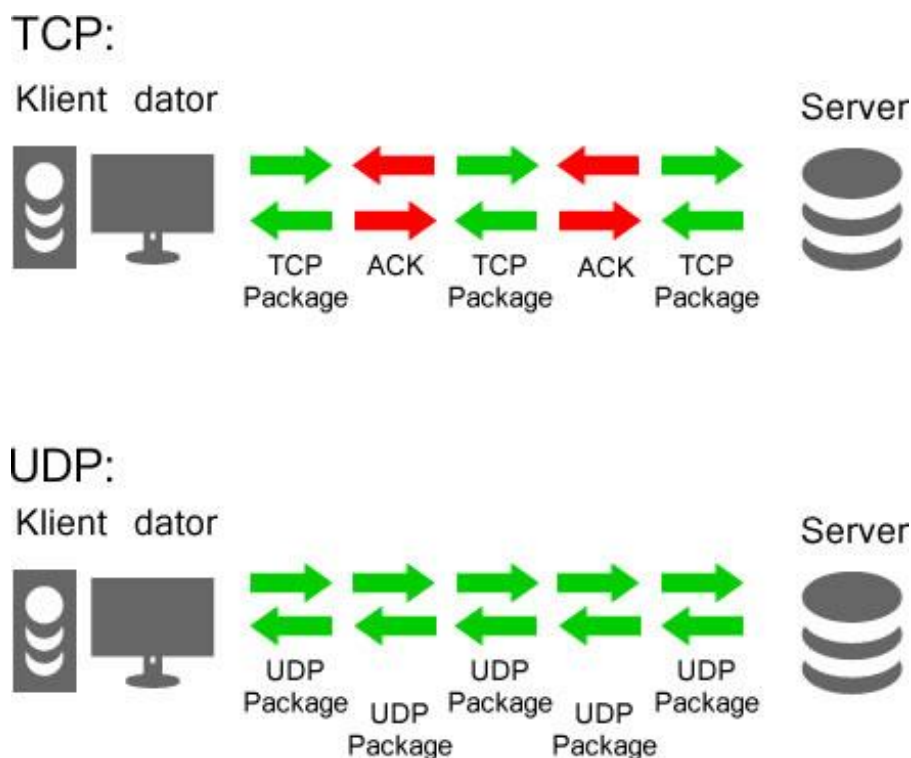
I en undersökning av Johannes Morgenroth et al. (2011) testade författarna att sätta upp en klient och en server som skickade data från klienten till server och sedan tillbaks till klienten med hjälp av Long Polling. Det visade sig i deras resultat att deras RTT (round trip time) var i princip oförändrad när datastorleken låg under 100kb. Round trip time är tiden det tar för ett paket att gå från en användare till servern tillbaks till användaren. Av resultatet drog de

slutsatsen att det inte var att skicka som tog upp tid, utan bearbetningen av data. Författarna undersökta detta ytterligare och kom fram till att det tog ca 60ms att enbart bearbeta datamängden.

### **2.3.2 WebSockets**

För att kunna skapa en full-duplex (dubbelriktad kommunikation) mellan en klient och en server för webbapplikationer har W3C implementerat WebSockets. WebSockets är en del av HTML5 och används med hjälp av JavaScript. JavaScript funktioner kan kallas automatiskt när användaren eller server tar emot data via en WebSocket (Se figur 3). Nu är det alltså fullt möjligt att ha en kommunikation där alla kan skicka data samtidigt även på webben. Detta skulle helt kunna ersätta polling för applikationer som kräver realtidsstöd. Dock är detta inte den enda fördelen. Enligt Andrew Wessels et al (2011) tar även WebSockets bort onödig overhead som skapas vid användandet av polling, samt att servern och klienten förblir anslutna även när det inte skickas några meddelanden under en period. Med polling är det lätt hänt att kopplingen mellan servern och klienten bryts om servern inte skickar något svar till klienten under en period.

Enligt Carl Gutwin (2011) har inte utvecklare lika bred kontroll av WebSockets jämfört med socket protokoll för andra programmeringsspråk, med WebSockets får utvecklare endast tillgång till basfunktioner. En annan nackdel med WebSockets är att det enbart kan kommuniceras enbart via TCP (Transmission control protocol) som är ett transportprotokoll som använder sig av nätverksprotokollet IP (Internetprotokoll). TCP är det mest använda förbindelseorienterat dataöverföringsprotokollet och används för HTTP, FTP och email m.m. TCP fungerar igenom att med hjälp av IP skicka data. Med denna information skickas även information om vilken dator som skickat det och vilken dator informationen är avsedd för (DARPA, 1981).



**Figur 4** Visar datapacket skickade mellan en klient och en server. Den övre med transportprotokollet TCP och den nedre med UDP.

Enligt Carl Gutwin et al. (2011) är TCP inte alltid lämpligt för realtidsapplikationer. Anledningar till att det är olämpligt är att när ett packet med data går förlorat måste denna datan skickas på nytt. TCP uppfattar även tappade datapacket som att nätet är överbelastat och då drar ner sändningstakten på packeten för att inte belasta nätet ytterligare. För att kontrollera om ett paket kommit fram eller inte skickas ett ACK (acknowledgment, *kvittering*). Dessa ACKs kan också ses som en typ av overhead då det inte alltid är nödvändigt för avsändaren att veta att packeten kom fram eller inte.

För att slippa ACK och kravet på att alla paket kommer fram kan ett annat transportprotokoll som kallas UDP användas (J. Postel 1980). UDP är mindre säkert men fungerar bättre för applikationer som stävar efter lägre latens, till exempel spel (Se figur 4). Enligt Kuan-Ta Chen (2006) finns det spel som använder TCP som till exempel World of Warcraft™, Guild Wars™ och Lineage II™. Detta är ett bevis på att det inte är omöjligt att lösa med TCP även om UDP är att föredra för spel.

## 3 Problem

Det finns många fördelar att skapa webbapplikationer som till exempel plattformsoberoende, bred spridning, att användare behöver inte installera programmet osv. Det är många krav som ställs på webben för att klara av att köra och hantera program som normalt skulle behövas installeras på en dator. Med hjälp av alla nya funktioner och möjligheter i HTML5 har webben kommit en bit på vägen. En av webbens stora svagheter har varit stöd för realtidsapplikationer (Carl Gutwin et al, 2011).

Problemet med realtidswebbapplikationer grundar sig i att få pålitliga och korrekta uppdateringar från flera simultananvändare, detta på grund av långsam nätverkskommunikation. Detta arbete skall undersöka om en applikation som använder WebSockets eller Long Polling kan möta användares krav på uppdateringshastighet, så en realtidsapplikation kan realiseras.

Enligt Kuan-Ta Chen (2006) är UDP att föredra för att uppnå en så låg latens som möjligt i applikationer. Men eftersom WebSockets endast stödjer TCP, kommer det även undersökas om TCP kommer räcka för att realisera en realtidsapplikation på webben. Eftersom TCP inte är att föredra för spel kommer det även undersökas om med hjälp av tekniker så som Dead Reckoning kommer gå att kunna minimera skadan av ett långsammare kommunikationsnätverk för webben. Att skapa bra nätverkskod för applikationer är en viktig del av bra realtidsapplikationer och med WebSockets får utvecklare enbart tillgång till basfunktioner och har inte alls lika bred kontroll som i andra programmeringsspråk. Det kommer undersökas hur väl det går att utnyttja dessa funktioner för att skapa bästa möjliga resultat för test applikationen. Det bör undersökas hur väl webbapplikationen fungerar i olika webbläsare och anpassa applikationen för att få den att fungera i så många webbläsare som möjligt. Enligt Carl Gutwin et al. (2011) är just plattformsoberoendet en av webbens starka sidor. Dock faller plattformsoberoendet lite på krav på rätt webbläsare.

### 3.1 Delmål

För att kunna genomföra projektet kommer ett antal delmål presenteras. Detta för att tydligt presentera vad som behövs för att kunna realisera testmiljön.

- Skapa ett spel skrivet i JavaScript för HTML5's canvas-element. Detta med en tydlig och simpel mekanik som fungerar för en spelare.
- Skapa en servermiljö som kommer hantera hemsidan och spelet. Den kommer även hantera Long Polling och WebSockets. Servermiljön ska även ha stöd för mätning av nätverkskommunikation och hastighet.
- En del av servern och spelet kommer byggas ut för att stödja flera spelare i samma spel samtidigt för att kunna göra mätningar på hur flera spelare påverkar latensen.
- En liknande utbyggnation kommer även användas för att stödja flera spelare i samma spel fast med hjälp av WebSockets istället.
- Skapa ett användarformulär för insamling av information från användare som testat spelet.

## 4 Metod

För att genomföra arbetet krävs mätningar av nätverkstrafiken med Long Polling och WebSockets. Mätningarna kommer göras på ett experiment i form av ett spel skapat i HTML5. Resultaten från mätningarna kommer styrkas med en enkät där användare av spelet kommer få svara på ett fåtal enklare frågor.

### 4.1 Mätning

Tom Beigbeder et al. (2004) och Nathan Sheldon et al. (2003) använde båda ett program som analyserar nätverkspaket. Ett liknande program kommer användas i det här experimentet för att samla in data från WebSockets och Long Polling som sedan kan bearbetas och presenteras. Carl Gutwin et al. (2011) gör en mätning där han tittar på hur mycket overhead som följde med nätverkspaketen, det vill säga all information som inte är relevant information för applikationen. Detta då det är en intressant skillnad mellan WebSockets och Long Polling som även kommer undersökas i det här arbetet.

Det bör även undersökas hur lång tid det tar för WebSockets att skicka data från en klient via servern till en annan klient eller tillbaka till sig själv, RTT (round trip time), och sedan göra ett RTT test på Long Polling och undersöka skillnaden. Detta kommer skrivas i JavaScript och köras lokalt på klientdatorerna. Efter testet kommer datan automatiskt skickas in via en funktion och sparas i en databas.

#### 4.1.1 Testmiljö

För att testa realtidsmöjligheterna för webbapplikationer kommer ett tilemap baserat 2D spel att utvecklas. Det kommer ha stöd för flera spelare i samma spel samtidigt, detta för att kunna göra mätningar på hur flera spelare påverkar latensen mellan klienten och servern. Spelet kommer ritas ut med hjälp av HTML5s canvas-element och vara skrivet i JavaScript. Spelet kommer att vara baserat på det gamla spelet *Snake*. *Snake* går ut på att spelaren skall navigera en orm på en spelplan och äta frukter för att samla poäng. För varje frukt som ormen äter växer ormen. Spelet tar slut om spelaren krockar med sidan av spelplanen eller med sig själv. För att se till att ingen användare möts av stötande innehåll kommer testspelet att följa PEGIs (Pan European Game Information) standard för åldersgräns på dataspel. Spelet kommer sedan få ett multiplayerläge där spelare kommer kunna spela mot varandra. Carl Gutwin et al. (2011) hävdar att i ett realtidsspel sker ca 20 nätverksuppdateringar per sekund och detta spel kommer ha en liknande nätverksuppdateringsfrekvens. Grafiken och spelmekanismen kommer hållas simpel för att undvika att spelmekanik eller grafik blir flaskhalsen.

En server kommer användas för köra applikationen och lagra data från testerna. Två lösningar av nätverkskommunikationen för testspelet kommer använda en som stödjer WebSockets och en som stödjer Long Polling. Testspelet kommer se ut och fungera på samma sätt för de båda lösningarna, alltså enbart nätverkskommunikationsstrukturen kommer förändras. Servern som kommer ansvara för Long Polling och WebSockets är en *Acer Aspire 7540G 17,3" WXGA* med operativsystemet Ubuntu 11.10. Testet kommer genomföras med hjälp av användare som kommer kunna ansluta till testspelet från sin hemdator över internet. Det kommer även göras tester på hur väl applikationen fungerar över LAN.

### **4.1.2 Enkät**

Tristan Henderson & Saleem Bhatti (2003), Pantel & Wolf (2002b) och Christian Schaefer et al. (2002) presenterar alla olika maxlatenser för olika spel. Av detta dras slutsatsen att spel har en unik tolerans för latens och det är därför svårt att döma hur väl ett spels nätverkskommunikation fungerar enbart igenom att mäta latensen. Därför kommer en enkät för användare att användas som en validering av mätningarna. En liknande teknik användes även av Claypool et al. (2006). Ett webbformulär med ett par frågor kommer ställas där användare som testat spelet kommer att få fylla i hur väl de tyckte det fungerade. Formuläret kommer hållas kort och enbart fungera som ett komplement till mätningen av data då datamätningen ligger i fokus för det här arbetet.

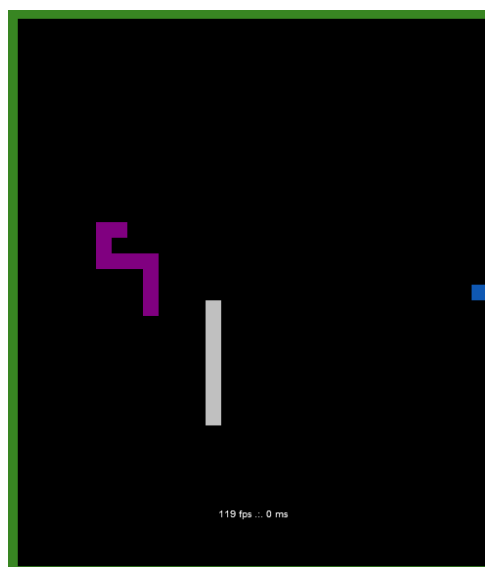
## 5 Genomförande

Det här kapitlet handlar om utvecklingsprocessen för testapplikationen och val av programmeringsspråk och bibliotek för kommunikationsdelen. Hur testapplikationen fungerar och kommunikationen mellan klienter och server förklaras samt en jämförelse mellan Long Polling och WebSockets och de olika skillnaderna som finns mellan de olika lösningarna. Eftersom arbetets fokus ligger på Long Polling och WebSockets kommer det också beskrivas hur de olika teknikernas medelände struktur ser ut. Koden för WebSockets implementationen finns i Appendix B för klienten och Appendix C för server koden. Long Pollings klient finns i Appendix D och PHP koden i Appendix E.

Andra spel som använder HTML5s canvas element är till exempel Angry Birds™ och Cut the Rope™ (Rovio Entertainment Ltd, 2011 och ZeptoLab, 2012) ett annat spel som utöver canvas elementet också använder WebSockets är BrowserQuest. BrowserQuest är ett open source spel utvecklat av Mozilla (Lecollinet & Lecollinet, 2012). Spelet som utvecklades för den här testapplikationen kan liknas med test applikationen som Carl Gutwin et al. (2011) använde sig av. Carl Gutwin et al. (2011) utvecklade en ritapplikation där flera användare kunde rita på en canvas samtidigt. Test applikationen i det här arbetet är istället mer likt ett riktigt spel.

Spelet använder en så kallad Klient/Server arkitektur där det viktigaste för varje klient är anslutningen till servern och de behöver inte bry sig om andra klienter, Jouni Smed et al. (2002). Test spelet bygger på Snake spelet känt från Nokias telefoner och går ut på att spelaren spelar en mask och äter frukter för att bli längre. Spelet består av två grundläggande objekt, masken och dess omgivning. Omgivningen består av en spelplan och en frukt. Målet för spelaren är att navigera masken med hjälp av piltangenterna för att nå frukten. Efter att masken ätit frukten kommer en ny frukt att placeras ut på spelplanen, (Verma & McOwan, 2005). Verma & McOwan (2005) skriver i sin artikel om Snake spelet anpassat för mobiltelefoner medan applikationen i detta arbete är anpassat för datorer, dock är spelmekaniken den samma.

### 5.1 Klientensida



**Figur 5** Visar en skämdump av klientsidan när två spelare spelar test spelet.

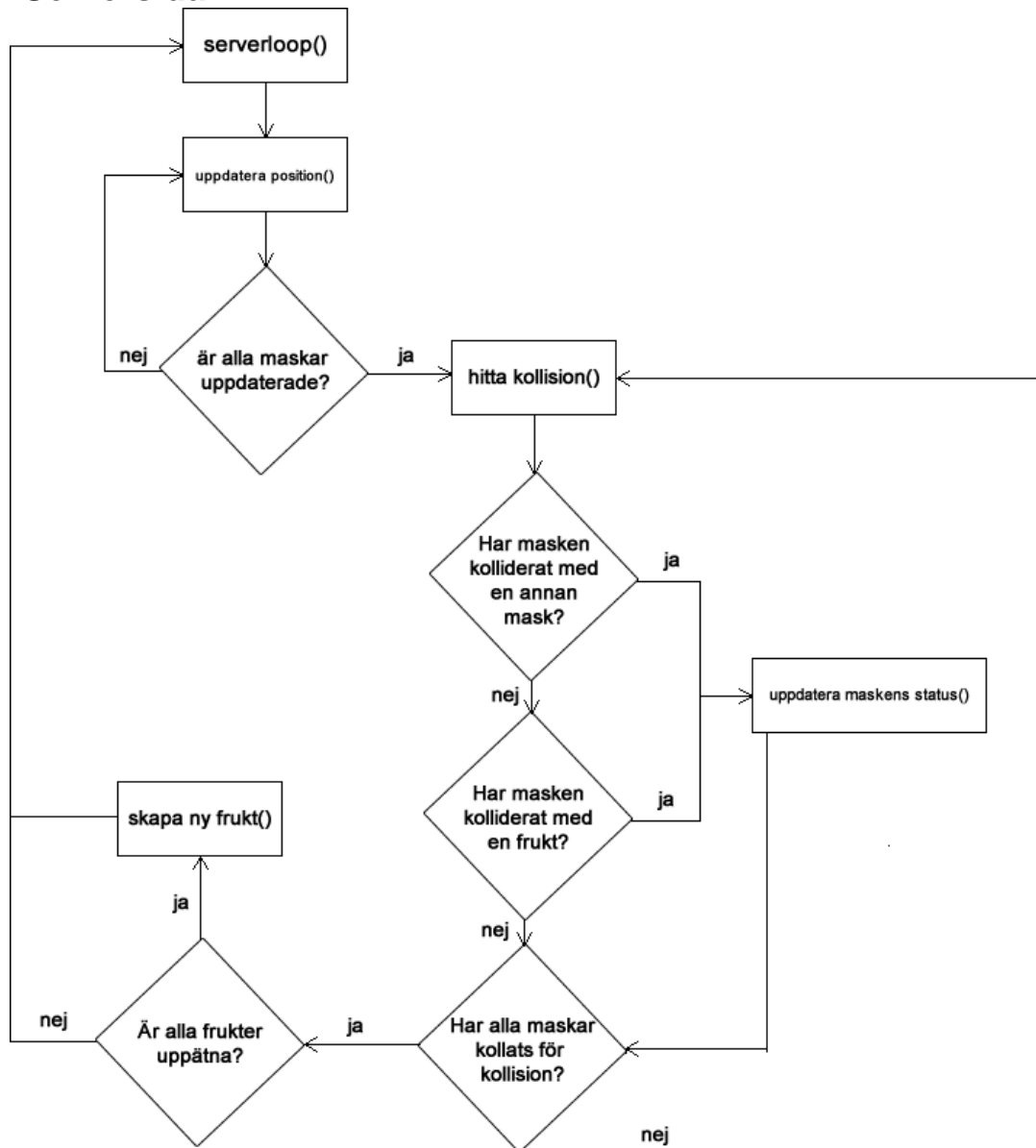
Klienterna fungerar som en simulering av server-sidan. Klienterna har också, liksom servern, en loop som körs konstant och uppdaterar positioner men ingen funktion för kollidering, det är helt överlåtet åt servern att hålla reda på. Klientens huvudansvar är att rita ut spelet på en canvas och samla in data från användarna. Klienten använder sig av tre JavaScript filer. En som ansvarar för speluppdateringarna, det vill säga uppdatera maskarnas positioner, och en som ansvarar för all grafik. Sist även en som ansvarar för kommunikationen mellan servern och klienten. Figur 5 visar en bild från spelet.

### 5.1.1 Dead Reckoning

I den tidigaste implementationen av nätverks kommunikation i spelet skickade servern konstant nya positioner för masken. Varje gång servern flyttade masken skickades koordinaterna till klienten vars position uppdaterades. Det fick dock katastrofala påföljder i form av att maskarna hoppade fram och tillbaka på canvasen och det var fullkomligt omöjligt att med hjälp av att bara titta på canvasen urskilja var masken egentligen var. Detta ersattes med att endast uppdatera masken när en klient svänger och använda Dead Reckoning för att räkna ut var masken borde vara. Baserat på allt klienten vet om masken gissar klienten att masken kommer fortsätta på samma sätt fram tills servern säger annorlunda. När en användare svänger, svänger också användarens lokala mask direkt. Klienten gissar alltså att den lokala masken är synkroniserad med serverversionen av masken och tar för givet att i samma ögonblick användaren svänger lokalt svänger även användarens mask på servern. Detta är dock inte alltid sant utan i vissa fall kan fördröjningen mellan klienten och servern leda till att klienten ligger något efter servern och masken svänger då för tidigt hos klienten. För att reparera detta returnerar servern ett X och Y värde för masken när en uppdatering skett. Om X och Y värdet skiljer sig från den lokala maskens X och Y position så flyttas den lokala masken till X och Y positionen som servern gav den.

Eric Cronin et al (2002) pratar om vikten av synkronisering för olika typer av realtidsapplikationer och använder som exempel spelet Quake 3. Det här spelets synkronisering liknar tekniker som diskuteras av Eric Corin et al (2002). En av teknikerna kallas Time Warp och är en optimistic algorithm vilket betyder att spelet litar på att det gör rätt och tillrättavisar sig själv när det blir fel. Eric Corin et al (2002) menar att en annan typ av optimistic algorithm, kallad trailing state synchronization (TSS), skulle kunna göra sig mer lämpad. TSS har flera instanser av applikationen som kör och upptäcker fel i synkroniseringen och kan då backa till dessa tidigare instanser och på så sätt reparera synkroniseringsfelet. Eftersom spelet som utvecklats för det här arbetet inte är lika krävande som spelet Quake 3, som användes av Eric Corin et al (2002), var det inte lika mycket data som behövdes synkroniseras. Resultatet blev en synkronisering som använder servern som den korrekta versionen av spelet. När maskar svänger synkroniseras de med det värdet de har på servern och om klienterna gör fel i sitt antagande flyttas klientmasken till positionen som servern angivit. I ett försök att nå ytterligare synkronisering synkroniserades spelloopen mellan servern och klienterna varje gång en ny användare anslöt sig till spelet.

## 5.2 Serversidan



**Figur 6** Visar en illustration av flödet i serverloopen för spelet.

Serversidan ansvarar för hur meddelanden från klienterna ska hanteras och uppdaterar spelet. Serversidan innehåller även koder för de olika objekten i spelet så som player (masken) och apple (som är frukterna). Alla maskar och frukter genereras av servern och skickas sedan ut till alla klienterna, detta för att alla klienter ska få samma värden för alla objekt.

*Serverloop()* är den funktionen som ansvarar för att uppdatera maskarnas positioner, kolla om någon mask krockat med en annan mask eller en frukt, samt uppdatera maskarnas status. *Serverloop()* körs konstant från att servern startas tills att den stängs av. Funktionen körs igång igen så fort den är klar och använder sig av en deltatimer för att räkna ut hur lång tid det har tagit att köra funktionen. Maskarnas position uppdateras så fort serverns deltatimer är 0.2, det vill säga 200ms. Det betyder att masken hinner röra sig fem

positionsenheter på en sekund. Funktionen *uppdatera\_position()* uppdaterar maskarnas positioner.

Genom att använda maskens riktning samt nuvarande position räknas nästa position ut och det värdet läggs till i maskens positions-array. Om masken inte har ätit en frukt tas då det sista värdet bort från positions-arrayen annars lämnas den kvar och det leder då till att masken växer med en position. Maskens längd är baserad på hur många positioner den har lagrat i sin positionsarray. Precis som Moore & Wilhelms (1988) skriver så handlar det om att hitta en och ha en respons för den inträffade kollisionen. För att lösa detta kontrolleras sedan alla positionsuppdateringar av funktionen *hitta\_kollision()*. *Hitta\_kollision()* plockar ut maskens huvud ur dess positionsarray och kontrollerar med alla maskar, även sig själv, om huvudet har kolliderat med någon annan mask. Funktionen kontrollerar även om masken har kolliderat med en frukt. Den ätna frukten tas även bort ur serverns fruktarray och när serverloopen ser att fruktarrayen är tom skapas en ny frukt på en ny position på spelplanen. Se figur 6 för en illustration av serverloopen.

### 5.2.1 WebSockets med Node.js och Socket.io

I det här arbetet användes Node.js för att skapa en händelsedrivna TCP spelservar för WebSockets. Node.js är ett system som är designat för att underlätta utvecklingen av webbapplikationer. Node.js använder JavaScript och styrs av händelsedrivna asynkron I/O. Node.js skrevs av Ryan Dahl 2009 och är licensierat under MIT-licensen. I kombination med Node.js användes även Socket.io. Socket.io användes för att underlätta användandet av WebSockets, då Socket.io bidrar med en tydlig struktur för kommunikationsfunktioner. Socket.io har även ett bredare crossbrowserstöd för WebSockets. (Joyent Inc, 2012 och LearnBoost, 2012).

- Node.js version 0.6.15
- Socket.io version 0.9.3

### 5.2.2 Long Polling med jQuery, PHP och MySQL

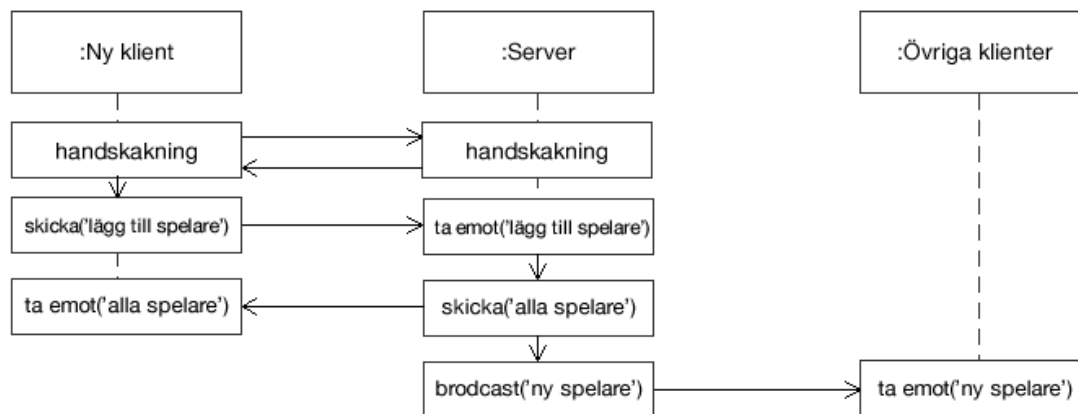
För Long Polling lösningen användes jQuery. jQuery har färdiga funktioner för AJAX i form av \$.post() och \$.get(). jQuery är ett av de mest använda JavaScript biblioteken och har varit under utveckling sedan 2006. jQuery används i dagsläget på 55% av de 100.000 mest populära hemsidorna (The jQuery Foundation, 2012). Server koden skrevs i PHP och databasen som användes för att lagra information var MySQL. En alternativ lösning för lagring kunde varit att skriva datan till en fil istället för att lagra den i en databas men för att undvika att det uppstår problem när flera klienter försöker skriva till samma fil samtidigt valdes istället en databas. Databasen ger även en mer strukturerad data som är lättare att tolka om någonting blir fel. Long Polling kan användas med andra språk så som till exempel .net. PHP och MySQL valdes eftersom min personliga expertis ligger i utveckling i dessa språk så valdes dem för att ge en så välskrivna server kod som möjligt.

- jQuery version 1.7.1
- PHP version 5.4.0
- MySQL version 5.5.23

## 5.3 Kommunikation mellan klient och server

I det här del kapitlet förklaras det hur Long Polling och WebSockets kommuniserar med servern. Hur de olika teknikerna växlar uppdateringar mellan klient och server kommer presenteras först följt av en genomgång av hur respektive tekniks meddelande struktur ser ut.

### 5.3.1 Uppdateringar mellan klient och server



**Figur 7** Visar hur en ny anslutning från en klient till servern hanteras med hjälp av WebSocket.

Skillnaden mellan WebSockets och Long Polling är att WebSockets först behöver genomföra en handskakning med servern där servern accepterar klienten och vice versa. När anslutningen etablerats kan klienten och servern skicka meddelanden mellan varandra via WebSockets. Se figur 7 för en tydligare bild av hur anslutningen med WebSockets går till.

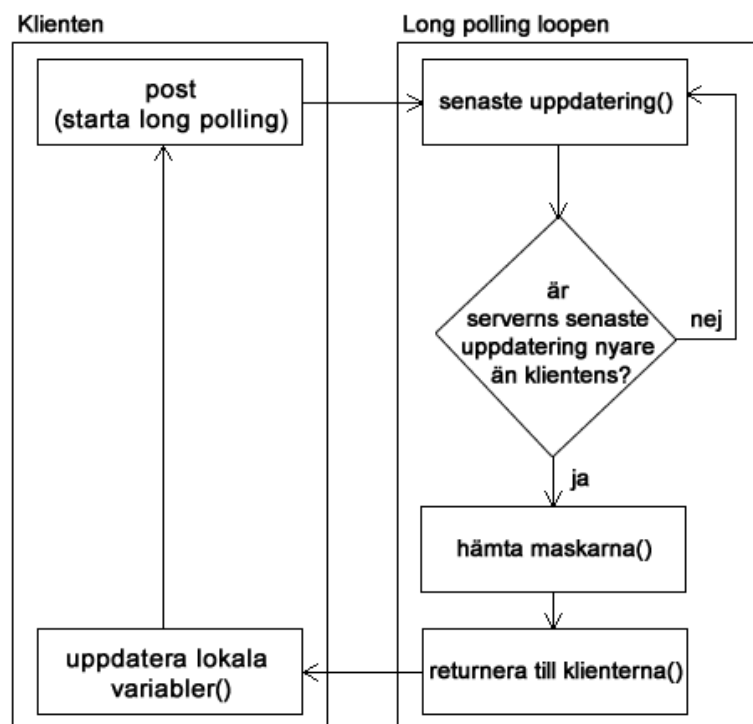
För Long Polling krävs det ingen handskakning, utan så fort användaren ansluter till webbservern går det att börja använda AJAX funktionerna. Både WebSocket och Long Polling börjar med att skicka ett meddelande till servern där de talar om att en ny klient har anslutit sig till spelet. Servern behandlar sedan informationen, skapar en ny mask för den nya användaren och uppdaterar serverns maskar. Servern sparar ner användarens namn som en nyckel, i WebSockets lösningen som en nyckel i en array och i Long Polling versionen som en nyckel i MySQL databasen. Sedan skickas alla maskar ut till klienterna. När servern returnerat maskarna till klienten placeras de in i den lokala spelararrayen och klienten är redo att delta i spelet. Long Polling skickar direkt sin server loop som väntar på nya uppdateringar. WebSocket väntar passivt på uppdateringar från klienten eller servern.

När klienten sedan lämnar servern (lämnar webbplatsen) har socket.io en funktion som ser att anslutningen gått förlorad och kallar automatiskt socket.on('disconnect'). Den funktionen ger möjligheten att på servern köra kod som kan hantera att användaren lämnat sidan och stänger WebSocket anslutningen mellan klienten och servern. I Long Polling versionen krävs en annan metod för att identifiera om användaren lämnat sidan. När en användare lämnat servern tas masken först bort från servern. I WebSocket lösningen skickas ett meddelande till alla klienter som talar om att masken är borta och i Long Polling lösningen uppdateras alla maskar och returneras som vanligt via Long Polling loopen.

När en spelare svänger skickar klienten ett meddelande till servern. Servern får från meddelandet ett namn och en riktning, och en funktion körs som uppdaterar serverns

maskversion. I WebSocket versionen skickar servern ut ett meddelande som talar om för alla klienter vilken mask som svängt, vilken ny riktning den har samt vilka X och Y positioner den befinner sig på. Long Polling versionen uppdaterar istället alla maskar och sparar ner de uppdaterade maskarna i databasen, uppdaterar tiden för senaste serveruppdateringen till nu och låter Long Polling loopen upptäcka att det finns ny data på servern samt hämta in informationen.

Hur servern räknar ut maskens position med Long Polling och WebSockets fungerar också på olika sätt. WebSockets kör en konstant serverloop där maskarnas position uppdateras konstant vilket betyder att när servern får reda på att en klient ändrar riktning på masken kan servern direkt byta riktning på sin mask och returnera resultatet till klienten. I Long Polling lösningen så existerar inte den konstanta serverloopen utan när en klient uppdaterar sin riktning tittar servern hur lång tid som passerat sedan den senaste uppdateringen ägde rum. Därefter delar servern den tiden med positionsuppdateringsfrekvensen och utifrån resultatet kör spelloopen så många gånger. Servern ansvarar för all kollision i spelet och behöver därför kunna tala om för klienterna när en kollision inträffat. WebSockets gör detta genom att direkt skicka ett meddelande till alla klienter när den konstanta serverloopen upptäckt en kollision. Long Polling versionen uppdaterar istället serverns maskstatus till död. Samma gäller även när en mask äter en frukt och växer.



**Figur 8** Visar hur Long Polling loopen ser ut.

Med WebSockets kan servern direkt skicka dessa uppdateringar till klienterna genom att skicka ett specifikt meddelande och klienten kör funktionen för meddelandet och uppdaterar specifika delar av spelet. Long Polling får inte uppdateringar direkt från servern utan använder sig av en loop som ligger och kör på servern och konstant frågar servern om en uppdatering ägt rum. När loopen startas skickar klienten när den senast fick sin uppdatering från servern. Klientens senaste uppdatering jämförs sedan med serverns värde för när den

senaste uppdateringen ägt rum. Om servern har en nyare tid än klienten betyder det att det finns ny information tillgänglig och servern skickar alla maskar, frukter och den nya tiden tillbaka till klienten. Long Polling, till skillnad mot WebSockets, uppdaterar då alla objekt från en och samma funktion. Efter att informationen på Long Polling klienten uppdaterats skickas direkt en ny förfrågan till servern med den nya synkroniseringstiden.

### 5.3.2 WebSockets medelande struktur

När användare anslutit sig till spelet måste en ny mask genereras. Den här masken ska sedan delas ut till alla andra klienter som också deltar i spelet. Den nya användaren behöver även få alla andra klienters maskar skickade till sig. För att se skillnaden mellan hur WebSockets och Long Polling hanterar uppdateringar till och från servern kommer spelets anslutning för nya användare användas som exempel. I det här delkapitlet kommer vi att titta på hur WebSockets utför det här anropet och hur servern och klienten kommunicerar med varandra via WebSockets i applikationen. Flödet för den här kommunikationen är samma som illustreras i figur 7.

```
socket.emit('addplayer', yourname);
```

Med hjälp av `.emit()` funktionen skickas meddelandet `'addplayer'` tillsammans med variabeln `yourname` till servern. På servern finns då en funktion som ansvarar för alla meddelanden `'addplayer'`.

```
socket.on('addplayer', function(username){
    snakes[username] = new player(username, colors.pop(),
    15, 15);
    var activeplayers = JSON.stringify(snakes);
    io.sockets.emit('remote_players', activeplayers);
    io.broadcast.emit('newSnake', username,
    snakes[username]);
});
```

Servern svarar genom att skapa en ny mask för den nya användaren. Den masken sparas ner i en array som ligger på servern och håller reda på alla maskar i spelet. Servern svarar sedan klienten med att skicka ett nytt meddelande tillbaka kallat `'remote_players'`. Med meddelandet bifogas serverns array av alla spelare. Servern skickar även ett till meddelande där den istället använder `.broadcast.emit()`. Det här meddelandet talar om för alla andra klienter att en ny mask har anslutit sig till spelet och skickar dem meddelandet `'newSnake'`, tillsammans med det nya användarnamnet och den tillhörande masken.

```
socket.on('remote_players', function(data){
    game.players = JSON.parse(data);
});
```

Tillbaka på den nya klienten, som tar emot `'remote_players'` meddelandet, plockas datan från meddelandet in och sparas ned lokalt så att alla maskar kan bearbetas av klienten och ritas ut på canvasen. De klienter som redan var anslutna till servern när den nya klienten anslöt sig till spelet bearbetar meddelandet `'newSnake'`.

```
socket.on('newSnake', function(username, newSnake){
    game.players[username] = JSON.parse(newSnake);
});
```

Istället för att ersätta alla maskar läggs bara den nya masken till i den lokala maskarrayen. Den sparas med den nya användarens användarnamn som nyckel.

För uppdatering av maskens riktning, kollision, statusförändring och fränkopplingar, används samma funktioner på liknande sätt där meddelandets namn ändras beroende på vilken typ av uppdatering som inträffat. Samma gäller för datan som skickas till och från servern, samt hur klienten och servern bearbetar datan.

### 5.3.3 Long Pollings medelande struktur

I det här delkapitlet kommer vi att titta på samma funktion som för WebSockets, fast hur den ser ut i Long Polling lösningen. Precis som i WebSocket versionen av spelet, skickar även Long Polling versionen ut ett meddelande till servern som talar om att en ny användare anslutit sig.

```
$.post('./js/PHP/connect.PHP', { name: yourname }){
```

Med hjälp av jQuerys `.post()` funktion skapas en AJAX förfrågan från den nya klienten. Klienten skickar sitt användarnamn som en post-variabel till sidan `connect.PHP`. I `connect.PHP` skapar servern en ny mask åt klienten.

```
$username = $_POST['name'];
$newsnake = array("name" => $username, "direction" =>
"right", "nextDirection" => "right", "color" => "blue",
"alive" => true, "position" => array( array(14, 14),
array(13, 14), array(12, 14)), "poplast" => true);
$sql = "INSERT INTO players VALUES ('".$username."',
'".json_encode($newsnake)."')";
mysqli_query($link, $sql);
updatesnakes(sendSnakes());
```

PHP koden gör i princip samma sak som JavaScript-koden för WebSockets, i den mån att den skapar en ny mask och sedan skickar alla maskar till klienten. Så fort klienten anslutit sig till spelet kör Long Polling loopen igång.

```
var polling = $.post('./js/PHP/longpolling.PHP', { name:
yourname, sync: lastupdate }, function(data){
```

Användarnamnet tillsammans med tiden för den senaste uppdateringen skickas till `longpolling.PHP`. På servern jämförs konstant klientens och serverns tid för senaste uppdatering. Loopen bryts när servern har en nyare tid än klienten. Datat från servern returneras då till klienten.

```
polling.success(function() {
    var allsnakes = JSON.parse(returndata);

    game.players = allsnakes["snakes"];
    game.fruits = [allsnakes["apples"]];
    lastupdate = allsnakes["sync"];
});
```

`.success()` är en callback-funktion i jQuery `.post()` som automatiskt kör så fort AJAX förfrågan är genomförd. När `.success()` funktionen kallas för Long Polling loopen ersätts all data med data från servern.

## 6 Analys

I det här kapitlet analyseras data som samlats in från de olika testerna och enkäten. Den viktigaste aspekten hos testerna var att få fram data om latens/RTT för att kunna identifiera hur lång tid det tar för paket att åka från en klient till servern och tillbaka till klienten. Detta för att kunna se skillnader i hur snabbt WebSockets är i jämförelse med Long Polling. En analys av paketstorlekar och overhead gjordes också för att identifiera hur stor datatrafiken som skickas är. Data från enkäten visar vad användarna tyckte om applikationen och hur väl det fungerade, enkäten användes som en kontroll för att se om bra resultat på latens sidan hade någon påverkan på hur användare upplevde applikationen. Resultatet från enkäten finns i Appendix A.

### 6.1 Latens/Round Trip Time (RTT)

För att kunna mäta Round Trip Time för teknikerna skapades ett objekt i javaskript som mätte hur lång tid från att ett medelande skickats från en klient tills det fått ett svar från servern. Tiden mäts i millisekunder. All data från RTT mätningarna sparades sedan i en databas och sedan bearbetades av en PHP-applikation som presenterade resultatet. genom att mäta RTT kan man se hur mycket tid som går förlorad på att skicka paket mellan klient och server. Mätningarna av RTT delades upp i tre delar. Den första delen gick ut på att en klient spelade 30 rundor med Long Polling följt av 30 rundor med WebSockets. En runda motsvarade en spelomgång, d.v.s. samlade frukter tills masken krockade och dog. En runda varade i genomsnitt cirka 2 minuter. För den första delen kördes testerna över LAN. Samma test med en klient och 30 spelrundor Long Polling och 30 rundor med WebSockets kördes sedan över internet för att se hur stor effekt nätverkshastigheten skulle påverka resultaten. Den sista delen av testerna gick ut på att köra 30 rundor vardera med flera spelare, också över internet, detta för att testa hur applikationerna hanterar multiplanvändare. Alla testerna utfördes under kontrollerade former. Detta för att försäkra att ingenting som kunde sega ner applikationen hände på servern under tiden testerna ägde rum.

Eftersom olika applikationer, främst spel, har helt olika krav på vad som klassas som accepterad latens är det svårt att göra en helhetsbedömning av resultaten som talar om exakt vilken typ av realtidsapplikationer som Long Polling och websockets skulle klara av. Claypool & Claypool (2006) grupperade olika spel från olika genrer av spel, FPS (First Person Shooter, Rally, Sport, RTS (Real time strategy) och delade in dem i olika nivåer beroende på hur känsliga de olika spelen var för latens. Den nivån som krävde lägst latens låg på 100ms, medel nivån låg på 500ms och den högsta nivån på 1 sekund. En annan viktig aspekt av multiplayer är också antalet paket som nätverket klarar av att skicka. Som tidigare nämnts i bakgrundskapitlet hävdar Carl Gutwin et al. (2011) att i ett groupware så ligger max gränsen på antal paket som behöver skickas per sekund på ca 25 stycken paket. Deras resultat visar på att WebSockets klarar den gränsen med marginal men att Long Polling inte gör det. Detta betyder dock inte att Long Polling inte skulle klara av applikationer med mindre paket per sekund men som fortfarande har krav på låg latens.

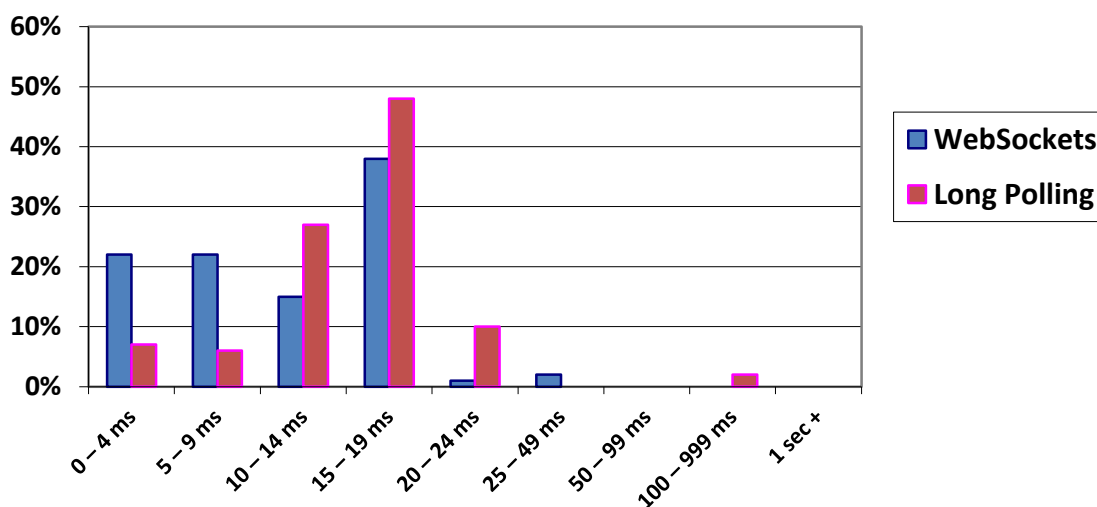
För alla tester användes Chrome v 18. Servern var en *Acer Aspire 7540G 17,3" WXGA* med 4 GB ram minne, en två kärnig processor av modellen *AMD Athlon II X2 M300* på 2 GHz. För online testerna var låg anslutningen till servern på hastigheten 10 Mbit/s för trafik både till och från servern. För testerna över LAN låg hastigheten på 100 Mbit/s för trafik både till och från servern. Klient datorn för testerna var en *MSI GT683DX-840* med 8GB ram och en *Intel Core i5* processor på 2.4 GHz För testerna med tre klienter användes även en *Macbook pro 13", early 2011*, med en 2,4 GHz Core i5 processor och 4GB ram samt en *Acer Aspire 5742G*

med 4Gb Ram och en processor på 2,67GHz. Testerna som kördes online kördes från högskolan i Skövde. Hastighet på högskolan i Skövdes internet mättes med hjälp av ett TP test till ca 30mbit/s för trafik till och från klienten. Servern som tidigare nämnt körde Ubuntu 11.10, båda PC klienterna körde Windows7 och Macbooken körde iOS.

### 6.1.1 LAN, 1 Klient

Från testerna som kördes över ett lokalt nätverk visade det sig att WebSockets lyckades hålla en genomsnittlig RTT på 11ms. För WebSockets låg den lägsta tiden på 1ms och den högsta på 56ms. Long Polling hade en genomsnittlig RTT på 15ms. För Long Polling låg den lägsta RTT-tiden också på 1ms men den högsta på 350ms

Graf 1

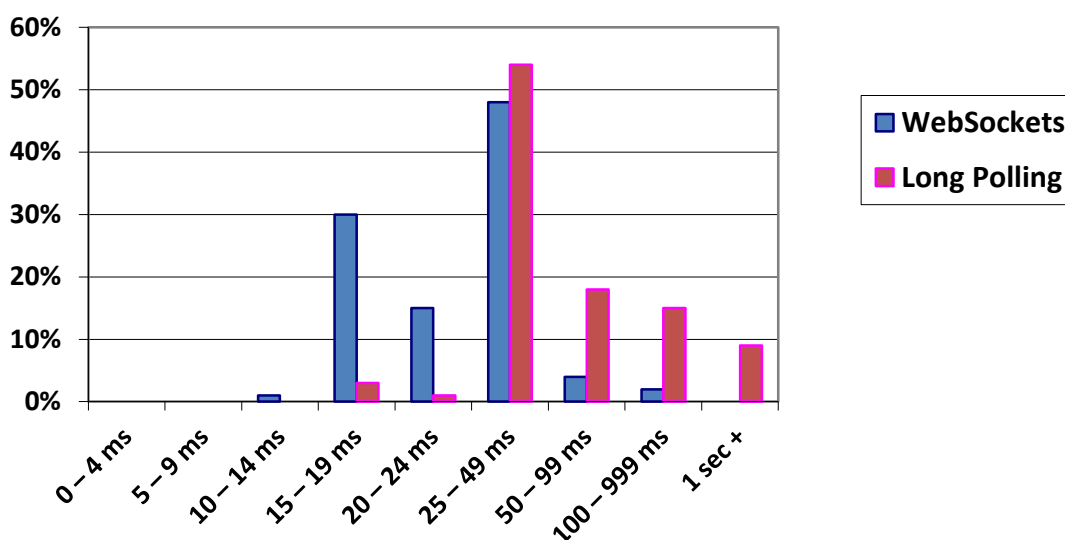


Graf 1 visar även att majoriteten av RTT tiden ligger mellan 0 och 19ms med högst antal mellan 15 – 19ms för båda teknikerna. Tittar man på de grupper Claypool & Claypool (2006) satt upp av olika typer av latens skulle både WebSockets och Long Polling klara av kraven för de spel med högre krav på låg latens där kravet låg på 100ms. Carl Gutwin et al. (2011) gjorde latens mätningar på WebSockets och deras resultat visade också att WebSockets höll en genomsnittlig latens på 11ms över LAN.

### 6.1.2 Internet, 1 Klient.

Samma test som för LAN kördes även över internet med en klient. WebSockets lägsta RTT tid var 10ms och den högsta låg på 970ms. Dock var det endast ett fåtal värden som hamnade över 100ms medans majoriteten låg mellan 15ms och 49ms. Över internet höll WebSockets en genomsnittlig RTT tid på 32ms. För Long Polling var det lägsta värdet 15ms medans det högsta låg på 4537ms. Long Polling höll en genomsnittlig RTT tid på 252ms.

### Graf 2

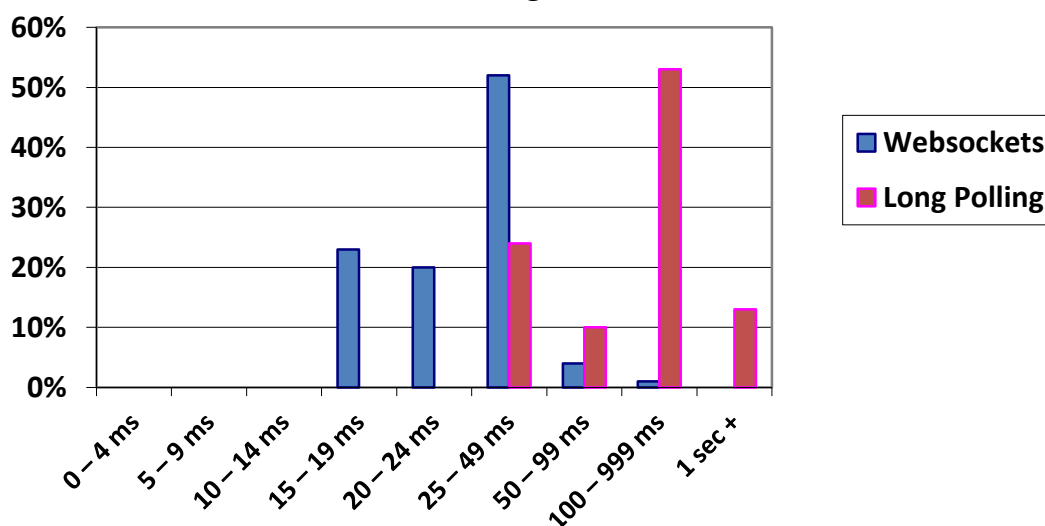


Graf 2 visar att majoriteten av RTT tiderna för WebSockets ligger mellan 15 och 50ms. Tittar man istället på Long Polling så ligger tiderna främst mellan 25 och 1000ms. Återigen till de riktlinjer Claypool & Claypool (2006) satt upp så skulle även här WebSockets klara av att köra de spel med krav på låg latens men inte Long Polling.

#### 6.1.3 Internet, 3 Klienter.

Tre användare fick spela spelet online för att testa hur väl Long Polling och WebSockets hanterade flera användare samtidigt. I det här testet höll WebSockets en genomsnittlig RTT på 31ms. För WebSockets låg den lägsta RTT-tiden på 16ms. Det högsta RTT värdet ligger på 535ms. För Long Polling låg den genomsnittliga RTT-tiden på 924ms. Det tog ibland så pass lång tid för Long Polling att svara att klienterna tappade anslutningen till servern.

### Graf 3



Graf 3 visar att majoriteten av RTT tider ligger mellan 15 och 50ms för WebSockets. Även om det finns RTT tider som ligger mellan 100 till 999ms så visar max värdet för WebSockets att den längsta RTT tiden ligger kring 500ms. Tittar man på de grupper Claypool & Claypool (2006) satt upp av olika typer av latens skulle WebSockets klara av kraven för de spel med

högre krav på låg latens där kravet låg på 100ms även när det körs över internet med flera klienter. Long Polling skulle inte möta den här gränsen då majoriteten av paketen skulle ha en längre RTT tid än 100ms.

## 6.2 Nätverkspaket och overhead

Paketstorlekar är intressant för mobila plattformar där antalet bytes som skickas bör hållas till ett minimum på grund av operatörernas max gränser på mobiltrafik. Genom att använda programmet Wireshark plockades nätverkspaket som skickades till och från servern upp och analyserades. WebSockets paket varierade i storlekar från ca 100 till 150 bytes. Där overheaden låg på 54 bytes per paket. Overhead är all den datan som finns med i nätverkspaketet som inte är en del av data avsedd för applikationen så som internet adresser och http information. För WebSockets var det största paketet som skickades var handskaningen mellan servern och klienten och den låg på ca 13kb. För Long Polling låg varje paket på ca 450 till 630 bytes. Long Polling hade en overhead från ca 300 till 600 bytes. Carl Gutwin et al. (2011) hävdar att en realtidsapplikation ska kunna skicka 25 paket per sekund detta skulle betyda att en sådan applikation som använder Long Polling skulle ligga på ca 13,5kb per sekund medans samma applikation med WebSockets skulle landa på ca 3,1kb per sekund. Detta betyder att den totala skillnaden mellan WebSockets och Long Polling skulle ligga på ca 10kb data per sekund. Storleken på paketen som skickas kan ha en betydande roll främst för mobila nätverk där hastigheten kan vara lägre och att mobil operatörer har en max gräns på hur mycket bandbredd kunderna får använda.

## 6.3 Enkät

Enkäten bestod av fem stycken frågor. Användarna som deltog fick inte veta vilken teknik som användes vid vardera tillfälle utan websocket spelet kallades test1 och Long Polling kallades test2. Efter att användarna spelat båda versionerna av spelet fick de gå till enkät sidan och svara på frågorna. Den första frågan för både Long Polling och WebSockets var hur väl de tyckte att spelet fungerade där de kunde välja ett värde från 1 som stod för *inte så bra* till 5 som stod för *jätte bra*. En annan fråga var om de tyckte applikationen var spelbar, här kunde användarna bara svara med *JA* eller *NEJ*. Sedan fanns det även en fråga som undrade om användarna märk någon skillnad mellan test1 och test2. Här kunde de välja från 1 som stod för *inte någon skillnad* till 5 som stod för *stor skillnad*. Totalt svarade 19 personer på enkäten. Majoriteten av folk som deltog har spelat flera online spel tidigare. Åldern på användarna varierade från 18 till 30. Nedan följer en sammanställning av frågorna ihop med svaren.

- Fråga 1 Hur väl tycker du test1 (WebSockets) spelet fungerade. 1 = *inte så bra*. 5 = *mycket bra*.
- Fråga 2 Var enligt dig test1 (WebSockets) spelbart. (*JA och NEJ fråga*).

Den genomsnittliga siffran för fråga 1 blev 4.2. Det vill säga att en majoritet av de användare som testade spelet tyckte att spelet fungerade bra. Resultatet från mätningarna på WebSockets för en klient online visar på att den genomsnittliga RTT tiden låg på 32ms. 32ms är betydligt lägre än vad relaterad forskning pekat på som max latens för olika spel. Pantel & Wolf (2002b) hävdar att latensen i ett bil spel inte bör överskrida 100ms. Christian Schaefer et al. (2002) testade skjutspellet XBlast och hävdar att en tolererad latens för det spelet ligger på 139ms. Tittar man på resultatet från fråga två svarade 89% JA det är spelbart.

- Fråga 3 Hur väl tycker du test2 (Long Polling) spelet fungerade. *1 = inte så bra. 5 = mycket bra.*
- Fråga 4 Var enligt dig test2 (Long Polling) spelbart. *(JA och NEJ fråga).*

För Long Polling låg den genomsnittliga siffran för fråga 3 på 2.1 vilket betyder att majoriteten av användarna inte tyckte spelet fungerade speciellt bra. Jämfört med resultatet från mätningarna så låg Long Pollings genomsnittliga RTT tid på 252ms vilket är över de gränser som den relaterade forskningen visat på är en tolererad latens. På fråga 4 svarade 21% att de ansåg spelet vara spelbart vilket tyder på att vissa användare fortfarande tyckte det gick att spela trots högre latens.

- Fråga 5 Märkte du någon skillnad mellan test1 (WebSockets) och test2 (Long Polling). *1 = inte så stor skillnad. 5 = mycket stor skillnad.*

På fråga fem låg det genomsnittliga resultatet på 4. Det vill säga att de flesta användarna märkte en ganska stor skillnad på de olika implementationerna precis som de tidigare svaren i enkäten visat.

## 7 Resultat

I detta kapitel kommer informationen från analysen ställas mot frågeställningen. Olika användningsområden för WebSockets och Long Polling kommer att undersökas samt vilken nytta detta arbete har och framtiden för teknikerna. Det kommer även diskuteras om studien och resultaten.

### 7.1 Resultatsammanfattning

Målet med studien var att undersöka om en applikation som använder WebSockets eller Long Polling kan möta användarens krav på uppdateringshastighet så en realtidwebbsapplikation kan realiseras. Olika forskningsartiklar, Tristan Henderson & Saleem Bhatti (2003), Pantel & Wolf (2002b), Christian Schaefer et al. (2002), har pekat på olika värden för hur hög latens som är acceptabel för olika applikationer, vilket gör det omöjligt att ge ett generellt svar som är sant för alla applikationer. Vad den här studien har gjort är att visa att det finns en möjlighet att öka hastigheten på nätverkskommunikation på webben. Den äldre tekniken, Long Polling, fick problem att hantera flera användare i testapplikationen och hade en genomsnittlig latens på 924ms. Carl Gutwin et al. (2011) visar i sin forskning att Long Polling inte klarar av att skicka nätverkspaket i den hastighet som krävs för mer avancerade realtidsapplikationer, detta ihop med den genomsnittliga latensen från testerna i den här studien visar på att tekniken inte är lämpad för de mer krävande realtids- webbapplikationerna. Resultatet av datan från användarna som testade applikationen och svarade på enkäten om Long Polling visade att de flesta användarna föredrog WebSockets applikationen före Long Polling och många ansåg även att Long Polling applikationen var ospelbar. Dock finns det fortfarande utrymme att använda Long Polling i applikationer med långsammare uppdateringar och utan krav på direkta svar mellan klienter och server. Claypool & Claypool (2006) delade in olika spel i olika nivåer beroende på hur känsliga spelen var för latens. Den högsta nivån låg på 1 sekund vilket Long Polling skulle klara av baserat på resultatet av den här studien. Spel som till exempel Alfapet, Memory och diverse kortspel skulle fortfarande kunna vara genomförbara med Long Polling.

Tittar man istället på WebSockets så var den genomsnittliga latensen oförändrad när applikationen kördes med en eller flera användare, WebSockets hade till och med 1ms mindre i genomsnittlig latens med flera användare i jämförelse med en. WebSockets genomsnittliga latens när applikationen kördes över internet låg kring 30ms vilket är betydligt lägre än vad tidigare forskning visat på skulle vara en rimlig max latens för olika spel, där den lägsta gränsen låg på ca 100ms (Tristan Henderson & Saleem Bhatti (2003), Pantel & Wolf (2002b), Claypool & Claypool (2006) och Christian Schaefer et al. (2002)). 89% av användarna som testade spelet tyckte att det var spelbart vilket också visar på att applikationen fungerade mycket bättre än Long Polling versionen som endast 21% ansåg vara spelbar.

En annan del av frågeställningen var att enligt Kuan-Ta Chen (2006) är UDP att föredra för att uppnå en så låg latens som möjligt i applikationer. Men eftersom WebSockets endast stödjer TCP, kommer det även undersökas om TCP kommer att räcka för att realisera en realtidsapplikation på webben. För den här typen av applikation visar analysen på att TCP räcker. Det förekom vissa fall där RTT tiden steg något för websocket implementationen men det gick inte att jämföra om det skulle gått att undvika med UDP då det protokollet inte finns tillgängligt för WebSockets. Även den genomsnittliga RTT-tiden på 30ms är enligt relaterad forskning inom ramen för vad som anses vara en rimlig latens (Tristan Henderson & Saleem

Bhatti (2003), Pantel & Wolf (2002b), Claypool & Claypool (2006) och Christian Schaefer et al. (2002)).

På grund av risken för att användarna sitter på en långsam anslutning och för att stödja WebSockets och Long Polling implementerades en Dead Reckoning teknik där klienten simulerar vart den tror att masken är på väg. Detta baserat på den senaste informationen klienten har om spelarna tills en ny uppdatering sker och användarens mask kan synkroniseras med serverns. Av användarstudien att döma så fungerade detta för WebSockets men Long Polling kom upp i så pass hög latens att den tekniken inte var tillräcklig för att täcka gapet mellan uppdateringarna.

## 7.2 Diskussion

Long Polling fungerade bra lokalt med en användare men när tekniken testades online med flera snabba uppdateringar och flera användare blev applikationen i min och en majoritet av användarnas mening helt ospelbar. Hur detta enbart beror på tiden det tar att skicka paket eller hur mycket tid det tar bara att hantera paketen på klient- och serversidan är svårt att avgöra av enbart dessa resultat. Det skulle behöva etableras en gräns för när Long Polling inte räcker till och WebSockets bör användas istället. Som jag ser det finns det inget negativt med att använda WebSockets över Long Polling för kommunikation förutom att WebSockets inte stöds av alla browsers. Men mer forskning behövs för att kunna sätta upp några tydliga riktlinjer för utvecklare om vilken teknik som bör användas för vad.

Testapplikationen, spelet, skulle fungera både för WebSockets och longpolling och inte ge någon av teknikerna någon fördel över den andra i testerna. Därför skrevs det väldigt generiskt och simpelt med få justeringar för att få det att fungera med båda teknikerna. Det skulle vara fördelaktigt att skriva applikationen fullt ut för WebSockets eller Long Polling och på så sätt kunna optimera applikationens kommunikationsstruktur för vald teknik. Detta skulle eventuellt kunna leda till ännu lägre resultat vad det gäller latens. Carl Gutwin et al. (2011) skriver i sin artikel att latensen varierar mycket på internet och att fler tester behöver genomföras för att ge en tydligare bild av WebSockets och Long Pollings specifika latens. Det här arbetet har visat att det helt klart finns en tydlig skillnad när det kommer till spel som uppdateras frekvent, körs över internet och tillåter flera användare. Jag anser att nästa steg är att vidare testa applikationer med ännu högre krav som liknar de vanliga program vi installerar på våra datorer och där testa om en webbapplikation med WebSockets kan mätas med den installerade programvaran.

Hemsidor blir mer och mer dynamiska, sociala medier för flera användare till samma hemsida och där interagerar de inte bara med hemsidan utan även med varandra. Av resultaten att döma skulle WebSockets kunna hjälpa till att förbättra kommunikationen mellan användare genom att öka hastigheten och etablera en mer pålitlig anslutning. WebSockets limiterar inte heller webbapplikationer till serverside språk som PHP och .net utan applikationen kan med hjälp av WebSockets även skicka och ta emot data från till exempel en applikation som kör i C# eller Java vilket också öppnar nya portar, inte bara för webbapplikationer.

Mobildatatrafik för smartphones är vanligt och med tanke på vinningarna i datatrafik av att använda WebSockets över Long Polling så finns det här utrymme att optimera webbapplikationer för smartphones. Eftersom Mobiltäckningen inte alltid är den bästa, och på grund av att mobiloperatörer sätter maxgränser för hur mycket mobildatatrafik kunderna

får använda. Detta kan leda till att minskad datatrafik är önskvärd och därför kan websocket vara ett bättre val för mobilwebbapplikationer med hög datatrafik.

I frågeställningen togs även webbens plattformsoberoende upp och att det bör undersökas hur väl webbapplikationen fungerar i olika webbläsare. Enligt Carl Gutwin et al. (2011) är just plattformsoberoendet en av webbens starka sidor. Eftersom WebSockets är en ny teknik stöds den inte av gamla webbläsare. Chrome v16 och Firefox 11 har stöd för det för tillfället aktuella websocket protokollet, men utvecklare kan med hjälp av olika bibliotek arbeta runt det här och använda till exempel flash och Long Polling för att simulera WebSockets i äldre browsers. Men om applikationen faller tillbaka på Long Polling kommer, enligt analysen, applikationen få en försämrad prestanda på nätverkstrafiken. Detta är någonting utvecklare måste vara medvetna om och antingen anpassa applikationen för- eller inte tillåta användare som använder en äldre browser. På grund av en icke kohesiv standard för de olika webbläsarna faller lite, enligt Carl Gutwin et al. (2011), påståendet om att applikationerna är plattformsoberoende när det istället blir ett nytt problem att anpassa applikationen för olika webbläsare. I problemställningen togs det även upp att applikationen borde testats i flera olika webbläsare. För att underlätta för de användare som skulle delta i testet valdes det bort och istället gjordes alla tester i Chrome v.18. Detta för att alla resultat skulle vara från samma webbläsare och på så sätt kunna garantera att olika webbläsare inte har någon effekt på resultaten i det här arbetet.

Användartesterna gjordes helt anonymt och över internet. All data finns presenterad i Appendix A ingen information om användarna sparades förutom det de svarade på enkäten. Det enda som går att härleda är användarens alla svar, det vill säga alla svar från samma användare är sparad i samma kolumn i databasen. All data som lagrades i databasen är också specifik för just det här arbetet och om någon obehörig skulle komma åt datan skulle det inte finnas några användar uppgifter eller personinformation lagrat i den. Enkäten låg online men länken delades endast ut till de användare som deltog i testerna och samma antal svarade som deltog i testerna. Detta kontrollerades för att undvika att obehöriga personer svarade på enkäten. För att få en spridning bland användarnas bakgrund och personlighet bad jag även användare som inte studerar programmering eller liknande att testa applikationen. Alla användare hade spelat masken tidigare, antingen på sina mobiltelefoner eller på en dator. Av de användare som deltog i testet var 18 killar och 1 tjej. Alla användare var bekanta och valdes beroende på om de var tillgängliga när testet genomfördes. Ingen speciell skillnad gjordes mellan de killar och de tjejer som deltog i testet.

Denna studie är svår att koppla till andra etiska perspektiv utöver forskningsetiken. Test spelet är helt fritt från blod och liknande då en mask dör försvinner den bara. Även all data insamlad från enkäten är helt anonym. Oavsett vilket genus eller bakgrund personen som använder testapplikationen har kan man anta att formen för masken är könsneutral. Om någon av de användare som testat spelet skulle haft problem med stötande innehåll hade de förmodligen hört av sig eller valt att inte delta. Oavsett har detta ingen påverkan på de mätresultat som presenterats i den här studien. Resultaten kan även anses som könsneutrala då mätningarna sker direkt mot applikationen och vilket inte kräver inblandning från användare, bortsett från enkäten.

### **7.3 Framtida arbete**

Analysen visar på att WebSockets klarar av att köra för en och få användare i den här testapplikationen. Här finns det utrymme att testa WebSockets på en större applikation med

flera användare som kör applikationen samtidigt och göra ett försök att pressa WebSockets till sin spets. Detta för att identifiera hur väl tekniken hanterar en större användarbas och hitta en gräns för var tekniken inte räcker till för att hantera pressen från applikationen och användarantalet. En del som utvecklare saknade i HTML5 standarden var möjligheten att streama video och ljud direkt i webbläsaren utan att använda några pluggins som flash eller java applets. WebSockets skulle kunna spela en stor roll i utvecklingen av dessa tekniker då låg latens och synkroniserade strömmar är en viktig del av video och ljud strömning. Ett utvecklat websocket API skulle även kunna hjälpa till vid synkronisering av filer över internet vilket är en del av cloud computing.

I det här arbetet testades teknikerna på en väldigt simpel applikation och framförallt WebSockets skulle behövas testas i en mer intensiv och krävande miljö för att verkligen kunna säkerställa vad tekniken är kapabel till att klara av. I en framtid där webbläsare och webbstandarder har blivit mer kraftfulla och tillåter utveckling av större program som till exempel Battlefield™, World of Warcraft™ och Crysis™ är det intressant hur väl WebSockets skulle förhålla sig till de kraven som ställs hos dessa programvaror. Vad utvecklare har pekat på är att WebSockets API:et är låst och endast ger tillgång till grundliga funktioner för nätverkskommunikation. För att öka prestandan skulle man eventuellt offra säkerhet för att ge utvecklare ett bredare socket API att arbeta med och på så sätt främja utvecklingen av mer komplexa mjukvaror.

För framtida arbeten inom realtidsapplikationer på webben skulle jag släppa Long Polling helt och fokusera arbetet på att optimera applikationer för WebSockets och testa hur långt man kan gå. Ett exempel kan vara att kombinera WebSockets med andra HTML5 tekniker som WebWorkers för att undersöka om en trådad JavaScript struktur med samtidig exekvering skulle kunna öka hastigheten på klient och server sidan för WebSockets och på så sätt få ner latensen ytterligare. Detta skulle även kunna underlätta för system med flera användare där till exempel chatt rum eller game lobbys skulle kunna delas upp på ett sätt så att servern skulle kunna spara processor kraft och lämna utrymme för snabbare exekvering. Det bör även undersökas om det finns någon säkerhets brist i WebSockets mot Long Polling för att ta reda på om någon av teknikerna är mer säker en den andra. Med säker menas att obehöriga personer kan komma åt datan som transporteras via protokollen. Detta kan vara aktuellt för webbapplikationer som behandlar känslig data som till exempel patientjournaler och bankuppgifter.

## Referenser

- Beigbeder, T., Coughlan, R., Lusher, C., Plunkett, J. (2004) The effects of loss and latency on user performance in unreal tournament 2003 *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. NetGames '04. New York, NY, USA, ACM. s. 144–151.
- Chen, K.-T., Huang, C.-Y., Huang, P. & Lei, C.-L. (2006) An empirical evaluation of TCP performance in online games. *Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*. ACE '06. New York, NY, USA, ACM.
- Claypool, M. & Claypool, K. (2006) Latency and player actions in online games. *Communications of the ACM*. 49 (11), s. 40.
- Conti, M. & Kumar, M. (2001) Quality of service in web services. *Proceedings of the 34th Annual Hawaii International Conference on System Sciences, 2001*. 3 January IEEE. s. 3550–3550.
- Cronin, E., Filstrup, B., Kurc, A.R. & Jamin, S. (2002) An efficient synchronization mechanism for mirrored game architectures. *Proceedings of the 1st workshop on Network and system support for games*. NetGames '02. New York, NY, USA, ACM. s. 67–73.
- DARPA (1981) *RFC 793 - Transmission Control Protocol*. 1981. Tillgänglig på Internet: <http://tools.ietf.org/html/rfc793> [Hämtad February 7, 2012].
- Fette, I. & Melnikov, A. (2011) *draft-ietf-hybi-thewebsocketprotocol-16 - The WebSocket Protocol*. 2011. Tillgänglig på Internet: <http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-16> [Hämtad February 14, 2012].
- Gutwin, C.A., Lippold, M. & Graham, T.C.N. (2011) Real-time groupware in the browser: testing the performance of web-based networking. *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. CSCW '11. New York, NY, USA, ACM. s. 167–176.
- Henderson, T. & Bhatti, S. (2003) Networked games: a QoS-sensitive application for QoS-insensitive users? *Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS: What have we learned, why do we care?* RIPQoS '03. New York, NY, USA, ACM. s. 141–147.
- Joyent, inc (2012) *node.js*. Tillgänglig på Internet: <http://nodejs.org/> [Hämtad April 18, 2012].
- LearnBoost.(2012) *Socket.IO*. Tillgänglig på Internet: <http://socket.io/> [Hämtad April 18, 2012].
- Lecollinet, G. & Lecollinet, F. (2012) *BrowserQuest*. Tillgänglig på Internet: <http://browserquest.mozilla.org/> [Hämtad June 6, 2012].
- Mark, C., Kajal, C. & Feissal, D. (2006) The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games. *In Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN)*. 19 January San Jose, California, USA.
- Moore, M. & Wilhelms, J. (1988) Collision Detection and Response for Computer Animation. *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '88. New York, NY, USA, ACM. s. 289–298.
- Morgenroth, J., Poegel, T., Heitz, R. & Wolf, L. (2011) Delay-tolerant networking in restricted networks. *Proceedings of the 6th ACM workshop on Challenged networks*. CHANTS '11. New York, NY, USA, ACM. s. 53–56.

- Pantel, L. & Wolf, L.C. (2002a) On the impact of delay on real-time multiplayer games. *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*. NOSSDAV '02. New York, NY, USA, ACM. s. 23–29.
- Pantel, L. & Wolf, L.C. (2002b) On the suitability of Dead Reckoning schemes for games. *Proceedings of the 1st workshop on Network and system support for games*. NetGames '02. New York, NY, USA, ACM. s. 79–84.
- Postel, J. (1980) *RFC 768 - User Datagram Protocol*. 1980. Tillgänglig på Internet: <http://tools.ietf.org/html/rfc768> [Hämtad February 7, 2012].
- Rovio Entertainment Ltd (2011) *Angry Birds Chrome*. 2011. Tillgänglig på Internet: <http://chrome.angrybirds.com/> [Hämtad February 2, 2012].
- Schaefer, C., Enderes, T., Ritter, H. & Zitterbart, M. (2002) Subjective quality assessment for multiplayer real-time games. *Proceedings of the 1st workshop on Network and system support for games*. NetGames '02. New York, NY, USA, ACM. s. 74–78.
- Sheldon, N., Girard, E., Borg, S., Claypool, M., *et al.* (2003) The effect of latency on user performance in Warcraft III. *Proceedings of the 2nd workshop on Network and system support for games*. NetGames '03. New York, NY, USA, ACM. s. 3–14.
- Smed, J., Kaukoranta, T. & Hakonen, H.(2002). Aspects of networking in multiplayer computer games. *The Electronic Library* 20-2-2002, 87-97.
- Taivalsaari, A. & Mikkonen, T. (2011) The Web as an Application Platform: The Saga Continues. *Proceedings of the 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. SEAA '11. Washington, DC, USA, IEEE Computer Society. s. 170–174.
- The jQuery Foundation (2012) *jQuery: JavaScript Library*. Tillgänglig på Internet: <http://jquery.com/> [Hämtad April 18, 2012].
- Verma, M.A. & McOwan, P.W. (2005). An adaptive methodology for synthesising mobile phone games using genetic algorithms i *Evolutionary Computation, 2005. The 2005 IEEE Congress on.*, ss. 864–871 Vol.1.
- w3c (2011) *The WebSocket API*. 2011. Tillgänglig på Internet: <http://dev.w3.org/html5/WebSockets/> [Hämtad February 16, 2012].
- Wessels, A., Purvis, M., Jackson, J. & Rahman, S. (2011) Remote Data Visualization through WebSockets. *2011 Eighth International Conference on Information Technology: New Generations (ITNG)*. 11 April IEEE. s. 1050–1051.
- ZeptoLab (2012) *Cut the Rope*. 2012. Tillgänglig på Internet: <http://www.cuttherope.ie/> [Hämtad February 2, 2012].

# Appendix A

Frågor på enkäten:

Hur väl tycker du **test1** spelet fungerade: 1 = *inte så bra*. 5 = *mycket bra*.

Var enligt dig **test1** spelbart:

Hur väl tycker du **test2** spelet fungerade: 1 = *inte så bra*. 5 = *mycket bra*.

Var enligt dig **test2** spelbart:

Märkte du någon skillnad mellan **test1** och **test2** : 1 = *inte så stor skillnad*. 5 = *mycket stor skillnad*.

\*\*test1 = WebSockets och test2 var Long Polling.

Resultat från användarankäten.

Fråga 1	Fråga 2	Fråga 3	Fråga 4	Fråga 5
5	JA	2	NEJ	4
4	JA	1	NEJ	5
4	JA	2	NEJ	3
4	JA	2	NEJ	5
4	JA	3	JA	4
3	NEJ	1	NEJ	3
4	JA	1	NEJ	5
4	JA	2	NEJ	4
5	JA	4	JA	2
4	JA	1	NEJ	4
5	JA	3	NEJ	4
4	JA	2	NEJ	4
5	JA	3	JA	4
4	JA	2	NEJ	4
4	JA	2	NEJ	5
5	JA	3	NEJ	4
3	NEJ	1	NEJ	3
4	JA	2	NEJ	4
5	JA	3	JA	5

# Appendix B

## Index.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>New simpler snake</title>
<script
src="http://46.239.102.194:8080/socket.io/socket.io.js"></script>
<script type="text/JavaScript" src="js/grafik.js"></script>
<script type="text/JavaScript" src="js/game.js"></script>
<script type="text/JavaScript" src="js/sockets.js"></script>
<script type="text/JavaScript" src="js/jquery-1.7.1.min.js"></script>
<script type="text/JavaScript">
function gameLoop(){
    game.uppdateGame();
    var gamerun = setTimeout("gameLoop()", 1);
}
$(document).ready(function(e) {

    gfx.isReady(56, 40, 16);
    socket.emit('addplayer', yourname);
    gameLoop();
});
</script>
<link href="css/crowstyle.css" rel="stylesheet" type="text/css">
</head>
<body>
    <div id="wrapper">
        <canvas id="canvas" width="480" height="560">

        </canvas>
        <div id="resultat"></div>
            <div class="version">v 0.0.7</div>
                <input id="derp" name="submit data" type="button"
onClick="submitdata();">
            </div>
    </body>
</html>
```

## grafik.js

```
gfx = function(){
  this.canvas = {};
  this.context = {};

  this.width = 0;
  this.height = 0;
  this.tilesize = 0;
  this.animation = 0;
  this.head_distance = 0;

  this.isReady = function(width, height, tilesize){
    this.canvas = document.getElementById("canvas");
    this.context = this.canvas.getContext("2d");

    this.width = width;
    this.height = height;
    this.tilesize = tilesize;
  }

  this.drawSnake = function(snakeArray, delta){
    for(object in snakeArray){
      if(snakeArray[object].alive){
        if(snakeArray[object].color != undefined || snakeArray[object] != null
        ){
          this.context.fillStyle = snakeArray[object].color;

          for(var i = 0; i < snakeArray[object].position.length;
          i++){

            var x = this.tilesize * snakeArray[object].position[i][0];
            var y = this.tilesize * snakeArray[object].position[i][1];

            this.context.fillRect(x, y, this.tilesize, this.tilesize);
          }}}

        }

        this.drawEatableObject = function(object){
          if(object.length != undefined){
            for(var i = 0; i < object.length; i++){
              this.context.fillStyle = "#0F58B3";
              var x = this.tilesize * object[i].xPos;
              var y = this.tilesize * object[i].yPos;
              this.context.fillRect(x, y, this.tilesize, this.tilesize);
            }
          }
        }
      }
    }
  }
}
```

```

}}

this.messageTimer = 0;
this.message = '';

this.setMessage = function(message){
this.messageTimer = 0;
this.message = message;
}
this.drawMessage = function(delta){

this.messageTimer+=delta;
this.context.fillStyle = "#FFF";
this.context.font = "28px Impact";
this.context.textAlign = "center";
this.context.textBaseline = 'top';
this.context.fillText(this.message, 240, 240);

if(this.messageTimer > 1){
this.messageTimer = 0;
this.message = '';
}
}
this.stats = "";
this.setStats = function(fps, ms){
this.stats = fps+" fps .:. "+ms+" ms";
}
this.drawStats = function(){

this.context.fillStyle = "#FFF";
this.context.font = "10px Arial";
this.context.textBaseline = 'top';
this.context.fillText(this.stats, 240, 500);

}
this.clearCanvas = function(){
this.context.clearRect(-1000, -1000, 1000000000, 1000000000);
}

}

```

### **game.js**

```

player = function(name, color, startX, startY){
this.name = name;
this.color = color;
this.alive = true;
this.poplast = true;

```

```

this.possition = [];
//head
this.possition.push([startX, startY]);
//body
this.possition.push([startX - 1, startY]);
//tail
this.possition.push([startX - 2, startY]);

this.direction = 'right';
this.nextDirection = 'right';

this.animation = 0;

this.setNewDirection = function(direction){
this.nextDirection = direction;
}

}

apple = function(X, Y){
this.xPos = X;
this.yPos = Y;
}

game = function(){

this.players = {};
this.fruits = [];
this.timeStamp = Date.now();
this.gamestate = 1;
this.gametics = 0;
this.countdown = 5;
this.framerate = 0;
this.netspeed = 0;

this.uppdatePossition = function(){
for(object in this.players){
if(this.players[object].alive){
var nextPosition = this.players[object].position[0].slice();
this.players[object].direction = this.players[object].nextDirection;

switch (this.players[object].direction) {
case 'left':
nextPosition[0] -= 1;
break;
case 'up':

```

```

nextPosition[1] -= 1;
break;
case 'right':
nextPosition[0] += 1;
break;
case 'down':
nextPosition[1] += 1;
break;
default:
throw('Invalid direction');
}
if(nextPosition[0] > 29){
nextPosition[0] = 0;
}else if(nextPosition[0] < 0){
nextPosition[0] = 29;
}

if(nextPosition[1] > 29){
nextPosition[1] = 0;
}else if(nextPosition[1] < 0){
nextPosition[1] = 29;
}

this.players[object].position.unshift(nextPosition);
if(this.players[object].poplast){
this.players[object].position.pop();
}else{
this.players[object].poplast = true;
}}}}

this.drawGFX = function(delta){
gfx.clearCanvas();
gfx.drawSnake(this.players, delta);
gfx.drawEatableObject(this.fruits);
gfx.drawMessage(delta);
gfx.drawStats();
}

this.bindEvents = function() {
var keysToDirections = {
37: 'left',
38: 'up',
39: 'right',
40: 'down'
};

$(document).keydown(function (event) {

```

```

var key = event.which;
var direction = keysToDirections[key];

if (direction) {
if(direction != game.players[yourname].nextDirection){
if(game.players[yourname].direction == 'left' && direction == 'right'){
//not ok
}else if(game.players[yourname].direction == 'right' && direction ==
'left'){
//not ok
}else if(game.players[yourname].direction == 'up' && direction ==
'down'){
//not ok
}else if(game.players[yourname].direction == 'down' && direction ==
'up'){
//not ok
}else{
game.players[yourname].nextDirection = direction;
event.preventDefault();
//sending newdirection with(new direction, name, X, Y)
socket.emit('newdirection', game.players[yourname].nextDirection,
yourname, game.players[yourname].position[0][0],
game.players[yourname].position[0][1]);
tjtm = new rtttimer(true);
}}
});
}
this.killPlayer = function(name){
this.players[name].alive = false;
if(name == yourname){
gfx.setMessage("You where killed");
presentresult();
}else{
gfx.setMessage("Player "+name+" was killed");
}
}
this.timer = 0;
this.sectimer = 0;
this.synkserver = 0;
this.uppdateGame = function(){
var now = Date.now();
this.framerate++;
var delta = (now - this.timestamp) / 1000;
if(delta > 1){
delta = 0;
}else{
this.timer+=delta;
this.sectimer+=delta;
}
}

```

```

if(this.gamestate == 1){
if(this.sectimer > 1){
this.countdown--;
if(this.countdown < 0){
this.gamestate = 0;
}else{
gfx.setMessage("Game starts in "+this.countdown+" seconds");
}}}
if(this.timer > 0.2){
this.gametics++;
this.uppdatePossition();
this.timer = 0;
}
if(this.sectimer > 1){
this.sectimer = 0;
netsynk.push(this.netspeed * 1000);
this.netspeed = Math.round((this.netspeed * 1000));
gfx.setStats(this.framerate, this.netspeed);
this.framerate = 0;
this.netspeed = 0;
this.synkserver++;
if(this.synkserver >= 10){
var snake = JSON.stringify(this.players[yourname].position);
this.synkserver = 0;
}}
this.bindEvents();
this.drawGFX(delta);
this.timeStamp = now;
}
}

```

### **socket.js**

```

var socket = io.connect('http://IPADRESS:8080');
var yourname = 'name' + Math.floor((Math.random()*100)+1);
var yournextdirr = 'right';

```

```

var game = new game();
var gfx = new gfx();
var synk_speed = 0;
var localSynk = {};
var netsynk = [];
var fps = [];
var sentPing = 0;
var reciping = 0;
var tajm = {};
var rtt = [];
function presentresult(){

```

```

var result = "";
for(var i = 0; i < netsynk.length; i++){
result+=netsynk[i]+"<br>";
}
$("#resultat").html(result);
}
function rtttimer(){
this.nottaken = true;
this.testtimer = Date.now();

this.stoptime = function(){
var newtime = (Date.now() - this.testtimer);
this.nottaken = false;
return newtime;
}}
socket.on('remote_players', function(data){
game.gamestate = 1;
game.countdown = 5;
game.gametics = 0;
game.timer = 0;
game.fruits = [];
var allsnakes = JSON.parse(data);
console.log(allsnakes);
game.players = allsnakes
});
socket.on('newSnake', function(username, newSnake){
var thenewone = JSON.parse(newSnake);
game.players[username] = thenewone;
});
socket.on('playerleft', function (name) {
delete game.players[name];
});
socket.on('newapple', function(data){
var newApples = JSON.parse(data);
game.fruits = newApples;
});
socket.on('apple_eaten', function(name){
game.players[name].poplast = false;
});
socket.on('deadsnake', function(name){
game.killPlayer(name);
});
function sendIdle(){
sentPing = 1;
reciPing = 0;
socket.emit('idle_k', yourname);
}

```

```

socket.on('idle_s', function (data){
  if(data == yourname){
    reciping = 1;
  }
});
socket.on('uppdatedirection', function(direction, name, x, y, snake,
timer) {
  if(name == yourname){
    if(tajm.nottaken){
      rtt.push(tajm.stoptime());
    }
    for(object in game.players){
      if(object == name){
        game.players[object].nextDirection = direction;
        if(game.players[object].position[0][0] != x){
          game.players[object].position = JSON.parse(snake);
        }
        if(game.players[object].position[0][1] != y){
          game.players[object].position = JSON.parse(snake);
        }
      }
    }
  });
  function submitdata(){
    $("#derp").remove();
    var senddata = $.post('./js/PHP/savedata.PHP', { data:
JSON.stringify(rtt) }, function(data){
  });
}

```

## Appendix C

### server.js

```
var app = require('express').createServer()
var io = require('socket.io').listen(app);
app.listen(8080);
app.get('/', function (req, res) {
  res.sendFile(__dirname + '/index.html');
});
player = function(name, color, startX, startY){
  this.name = name;
  this.color = color;
  this.alive = true;
  this.popleft = true;
  this.possition = [];
  //head
  this.possition.push([startX, startY]);
  //body
  this.possition.push([startX - 1, startY]);
  //tail
  this.possition.push([startX - 2, startY]);
  this.direction = 'right';
  this.nextDirection = 'right';

  this.animation = 0;
}

apple = function(X, Y){
  this.xPos = X;
  this.yPos = Y;
}

var gameSizeX = 30;
var gameSizeY = 30;
var gamestate = 1;

var gametics = 0;
var usernames = {};
var snakes = {};
var apples = [];
var colors = [];
var currentGameGrid = [];

colors.push("red");
colors.push("blue");
colors.push("yellow");
```

```

colors.push("green");
colors.push("orange");
colors.push("pink");
colors.push("purple");
colors.push("silver");

function updatePosition(){
for(object in snakes){

if(snakes[object].alive){
var nextPosition = snakes[object].position[0].slice();
snakes[object].direction = snakes[object].nextDirection;

switch (snakes[object].direction) {
case 'left':
nextPosition[0] -= 1;
break;
case 'up':
nextPosition[1] -= 1;
break;
case 'right':
nextPosition[0] += 1;
break;
case 'down':
nextPosition[1] += 1;
break;
default:
throw('Invalid direction');
}
if(nextPosition[0] > 29){
nextPosition[0] = 0;
}else if(nextPosition[0] < 0){
nextPosition[0] = 29;
}

if(nextPosition[1] > 29){
nextPosition[1] = 0;
}else if(nextPosition[1] < 0){
nextPosition[1] = 29;
}

snakes[object].position.unshift(nextPosition);
if(snakes[object].poplast){
snakes[object].position.pop();
}else{
snakes[object].poplast = true;
}}}}

```

```

function findCollision(){
for(object in snakes){
if(snakes[object].alive){
var headX = snakes[object].position[0][0];
var headY = snakes[object].position[0][1];
for(obj in snakes){
if(snakes[obj].alive){
for(var i = 1; i < snakes[obj].position.length; i++){
if(snakes[obj].position[i][0] == headX){
if(snakes[obj].position[i][1] == headY){
//snake is dead
snakes[object].alive = false;
killSnake(object);
}}}}
for(var i = 0; i < apples.length; i++){
if(apples[i].xPos == headX){
if(apples[i].yPos == headY){
apples.pop();
snakes[object].poplast = false;
io.sockets.emit('apple_eaten', object);}}}}}}

function createApple(){
var X = Math.floor((Math.random()*(gameSizeX - 1))+1);
var Y = Math.floor((Math.random()*(gameSizeY - 1))+1);

apples.push(new apple(X, Y));
var sendmsg = JSON.stringify(apples);
io.sockets.emit('newapple', sendmsg);
}

var timer = 0;
var timeStamp = 0;
var sectimer = 0;
var countdown = 5;

function serverloop(){
var now = Date.now();
var delta = (now - timeStamp) / 1000;

if(delta > 1){
delta = 0;
}else{
timer+=delta;
sectimer+=delta;
}
}

```

```

if(gamestate == 1){
if(sectimer > 1){
countdown--;
if(countdown < 0){
gamestate = 0;
}
sectimer = 0;
}}

if(timer > 0.2){
if(gamestate == 0){
findCollision();

if(apples.length == 0){
createApple();
}
}
updatePosition();
timer = 0;
}
if(colors.length == 0){
restoreCollors();
}
timeStamp = now;
gametics++;
}

function gameLoop(){
serverloop();
var gamerun = setTimeout(gameLoop, 1);
}
function restoreCollors(){
colors.push("red");
colors.push("blue");
colors.push("yellow");
colors.push("green");
colors.push("orange");
colors.push("pink");
colors.push("purple");
colors.push("silver");
}

function killSnake(name){
io.sockets.emit('deadsnake', name);
}
io.sockets.on('connection', function (socket) {
socket.on('newdirection', function (direction, name, x, y) {

```

```

var correction = false;
snakes[name].nextDirection = direction;
io.sockets.emit('updatedirection', direction, name,
snakes[name].position[0][0], snakes[name].position[0][1],
JSON.stringify(snakes[name].position), timer);
});
socket.on('addplayer', function(username){
socket.username = username;
usernames[username] = username;

console.log("user was sent all current snakes in the game "+username);
snakes[username] = new player(username, colors.pop(), 15, 15);

var activeplayers = JSON.stringify(snakes);
io.sockets.emit('remote_players', activeplayers);

var newSnake = JSON.stringify(snakes[username]);
//io.sockets.emit('newSnake', username, newSnake);
socket.broadcast.emit('newSnake', username, newSnake);

gamestate = 1;
sectimer = 0;
countdown = 5;
gametics = 0;
timer = 0;
apples = [];
});

socket.on('idle_k', function (name) {
io.sockets.emit('idle_s', name);
});
socket.on('synkserver', function(name, data) {
snakes[name].position = JSON.parse(data);
});
socket.on('disconnect', function(){
io.sockets.emit('updateusers', usernames);
delete usernames[socket.username];
delete snakes[socket.username];
socket.broadcast.emit('playerleft', socket.username);
});
});
gameLoop();

```

## Appendix D

### index.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Snake Bummer Remix</title>
<script type="text/JavaScript" src="js/grafik.js"></script>
<script type="text/JavaScript" src="js/game.js"></script>
<script type="text/JavaScript" src="js/long_polling.js"></script>
<script type="text/JavaScript" src="js/jquery-1.7.1.min.js"></script>
<script type="text/JavaScript">

function gameLoop(){
    game.uppdateGame();
    var gamerun = setTimeout("gameLoop()", 1);
}
$(document).ready(function(e) {
    connect();
    gfx.isReady(56, 40, 16);
    gameLoop();
});
window.onbeforeunload = function() {
    removeplayer(yourname);
};
</script>
<link href="css/crowstyle.css" rel="stylesheet" type="text/css">
</head>
<body>
    <div id="wrapper">
        <canvas id="canvas" width="480" height="560">

        </canvas>
        <div id="resultat"></div>
            <div class="version">v 0.0.7</div>
                <input id="derp" name="submit data" type="button"
onClick="submitdata();">
            </div>
    </body>
</html>
```

### grafik.js

```
gfx = function(){
this.canvas = {};
this.context = {};
```

```

this.width = 0;
this.height = 0;
this.tilesize = 0;
this.animation = 0;
this.head_distance = 0;

this.isReady = function(width, height, tilesize){
this.canvas = document.getElementById("canvas");
this.context = this.canvas.getContext("2d");

this.width = width;
this.height = height;
this.tilesize = tilesize;
}

this.drawSnake = function(snakeArray, delta){
for(object in snakeArray){
//player color
if(snakeArray[object].alive){
if(snakeArray[object].color != undefined || snakeArray[object] != null
){
this.context.fillStyle = snakeArray[object].color;

for(var i = 0; i < snakeArray[object].position.length;
i++){
var x = this.tilesize * snakeArray[object].position[i][0];
var y = this.tilesize * snakeArray[object].position[i][1];

this.context.fillRect(x, y, this.tilesize, this.tilesize);
}}}}

}

this.drawEatableObject = function(object){
for(var i = 0; i < object.length; i++){
this.context.fillStyle = "#0F58B3";
var x = this.tilesize * object[i].xPos;
var y = this.tilesize * object[i].yPos;
this.context.fillRect(x, y, this.tilesize, this.tilesize);
}}

this.messageTimer = 0;
this.message = '';

this.setMessage = function(message){
this.messageTimer = 0;

```

```

this.message = message;
}
this.drawMessage = function(delta){

this.messageTimer+=delta;
this.context.fillStyle = "#FFF";
this.context.font = "28px Impact";
this.context.textAlign = "center";
this.context.textBaseline = 'top';
this.context.fillText(this.message, 240, 240);

if(this.messageTimer > 1){
this.messageTimer = 0;
this.message = '';
}}
this.stats = "";
this.setStats = function(fps, ms){
this.stats = fps+" fps .:. "+ms+" ms";
}
this.drawStats = function(){
this.context.fillStyle = "#FFF";
this.context.font = "10px Arial";
this.context.textBaseline = 'top';
this.context.fillText(this.stats, 240, 500);
}
this.clearCanvas = function(){
this.context.clearRect(-1000, -1000, 1000000000, 1000000000);
}}

```

### **grafik.js**

```

player = function(name, color, startX, startY){
this.name = name;
this.color = color;
this.alive = true;
this.poplast = true;

this.possition = [];
//head
this.possition.push([startX, startY]);
//body
this.possition.push([startX - 1, startY]);
//tail
this.possition.push([startX - 2, startY]);

this.direction = 'right';
this.nextDirection = 'right';

```

```

this.animation = 0;
this.setNewDirection = function(direction){
this.nextDirection = direction;
}

}

apple = function(X, Y){
this.xPos = X;
this.yPos = Y;
}

game = function(){

this.players = {};
this.fruits = [];
this.timeStamp = Date.now();
this.gamestate = 1;
this.gametics = 0;
this.countdown = 5;
this.framerate = 0;
this.netspeed = 0;

this.updatePosition = function(){
for(object in this.players){
if(this.players[object].alive){
var nextPosition = this.players[object].position[0].slice();

this.players[object].direction = this.players[object].nextDirection;

if(this.players[object].direction != undefined){
switch (this.players[object].direction) {
case 'left':
nextPosition[0] -= 1;
break;
case 'up':
nextPosition[1] -= 1;
break;
case 'right':
nextPosition[0] += 1;
break;
case 'down':
nextPosition[1] += 1;
break;
default:
throw('Invalid direction');
break;
}
}
}
}
}

```

```

}
if(nextPosition[0] > 29){
nextPosition[0] = 0;
}else if(nextPosition[0] < 0){
nextPosition[0] = 29;
}

if(nextPosition[1] > 29){
nextPosition[1] = 0;
}else if(nextPosition[1] < 0){
nextPosition[1] = 29;
}

this.players[object].position.unshift(nextPosition);
if(this.players[object].poplast){
this.players[object].position.pop();
}else{
this.players[object].poplast = true;
}}}}

this.drawGFX = function(delta){
gfx.clearCanvas();
gfx.drawSnake(this.players, delta);
gfx.drawEatableObject(this.fruits);
gfx.drawMessage(delta);
gfx.drawStats();
}

this.bindEvents = function() {
var keysToDirections = {
37: 'left',
38: 'up',
39: 'right',
40: 'down'
};

$(document).keydown(function (event) {
var key = event.which;
var direction = keysToDirections[key];

if (direction) {
if(game.players[yourname] != undefined){
if(direction != game.players[yourname].nextDirection){
if(game.players[yourname].direction == 'left' && direction == 'right'){
//not ok
}else if(game.players[yourname].direction == 'right' && direction ==
'left'){

```

```

//not ok
}else if(game.players[yourname].direction == 'up' && direction ==
'down'){
//not ok
}else if(game.players[yourname].direction == 'down' && direction ==
'up'){
//not ok
}else{
game.players[yourname].nextDirection = direction;
event.preventDefault();
changedirection();
}}}}
});
}

```

```

this.killPlayer = function(name){
this.players[name].alive = false;
if(name == yourname){
gfx.setMessage("You where killed");
presentresult();
}else{
gfx.setMessage("Player "+name+" was killed");
}}

```

```

this.timer = 0;
this.sectimer = 0;
this.synkserver = 0;

```

```

this.uppdateGame = function(){
if(this.players != undefined){
var now = Date.now();
this.framerate++;
var delta = (now - this.timeStamp) / 1000;

```

```

if(delta > 1){
delta = 0;
}else{
this.timer+=delta;
this.sectimer+=delta;
}

```

```

if(this.gamestate == 1){
if(this.sectimer > 1){
this.countdown--;

```

```

if(this.countdown < 0){

```

```

this.gamestate = 0;
}else{
gfx.setMessage("Game starts in "+this.countdown+" seconds");
}}}

if(this.timer > 0.2){
this.gametics++;
this.uppdatePossition();
this.timer = 0;
}

if(reciPing == 1){
sentPing = 0;
}

if(sentPing == 1){
this.netspeed+=delta
}

if(this.sectimer > 1){
this.sectimer = 0;
netsynk.push(this.netspeed * 1000);
this.netspeed = Math.round((this.netspeed * 1000));
gfx.setStats(this.framerate, this.netspeed);
this.framerate = 0;
this.netspeed = 0;
this.synkserver++;
if(this.synkserver >= 10){
var snake = JSON.stringify(this.players[yourname].position);
this.synkserver = 0;
}}
this.bindEvents();
this.drawGFX(delta);
this.timeStamp = now;
}
longpolling_req();
}}

```

### **Long\_polling.js**

```

var yourname = 'name' + Math.floor((Math.random()*100)+1);
var $_GET = {};

document.location.search.replace(/\??(?:([\^=]+)=([\^&]*)&?)/g, function
() {
function decode(s) {
return decodeURIComponent(s.split("+").join(" "));
}

```

```

$_GET[decode(arguments[1])] = decode(arguments[2]);
});
var yourid = $_GET["id"];
var yourcounter = 1;
var yournextdirr = 'right';
var activereq = false;
var activesend = false;
var lastupdate = 0;

var game = new game();
var gfx = new gfx();
var synk_speed = 0;
var localSynk = {};
var netsynk = [];
var fps = [];

var sentPing = 0;
var reciPing = 0;

var tajm = new rtttimer();
tajm.stoptime();
var pingtime = [];

var rtt_send = [];
var rtt_get = [];
var rtt = [];

var de_bugger = false;

function removeplayer(username){
var removeing = $.post('./js/PHP/remove.PHP', { name: yourname, color:
game.players[yourname].color }, function(data){

});
}
function rtttimer(){
this.nottaken = true;
this.testtimer = Date.now();

this.stoptime = function(){
var newtime = Date.now();
var final = newtime - this.testtimer;
this.nottaken = false;
return final;
}}
function presentresult(){

```

```

var result = "";
for(var i = 0; i < netsynk.length; i++){
result+=netsynk[i]+"<br>";
}

$("#resultat").html(result);
}

function connect(){
var returndata = '';
var connecting = $.post('./js/PHP/connect.PHP', { name: yourname },
function(data){
sendIlde();
returndata = data;
});

connecting.success(function() {
reciPing = 1;
game.players = JSON.parse(returndata);
});
}
function changedirection(){
var direction = $.post('./js/PHP/direction.PHP', { name: yourname, id:
yourid, counter: yourid+""+yourcounter, direction:
game.players[yourname].nextDirection }, function(data){
/*if(!tajm.nottaken){
tajm = new rtttimer(true);
}*/
rtt_send[yourid+""+yourcounter] = Date.now();
//pingtime[yourid+""+yourcounter] = new rtttimer();
yourcounter++;
});
}

function longpolling_req(){
var allsnakes = '';
if(activereq){
return;
}else{
activereq = true;
var returndata = '';
var polling = $.post('./js/PHP/longpolling.PHP', { name: yourname,
sync: lastupdate }, function(data){
returndata = data;
});
}

polling.success(function() {
if(returndata != "pinged out"){

```

```

allsnakes = JSON.parse(returndata);
game.players = allsnakes["snakes"];
game.fruits = [allsnakes["apples"]];
lastupdate = allsnakes["sync"];
rtt_get[allsnakes[yourid]] = Date.now();
var ping = (rtt_get[allsnakes[yourid]]-rtt_send[allsnakes[yourid]]);
if(ping != null){
rtt.push(ping);
}}
activereq = false;
});

}}
function submitdata(){
$("#derp").remove();
var senddata = $.post('./js/PHP/savedata.PHP', { data:
JSON.stringify(rtt) }, function(data){

});
}

function sendIlde(){
sentPing = 1;
reciPing = 0;
}

```

## Appendix E

### Connect.PHP

```
<?PHP
include_once 'dbconnect.PHP';
include_once 'functions.PHP';

$sql = "SELECT * FROM colors ORDER BY ABS(id)";
$colors = mysqli_query($link, $sql);
$color = '';

while($row = mysqli_fetch_assoc($colors)){
    $color = $row['color'];
}

$sql = "DELETE FROM colors WHERE color = '".$color."'";
mysqli_query($link, $sql);

$username = $_POST['name'];
$newsnake = array("name" => $username, "direction" => "right",
"nextDirection" => "right", "color" => $color, "alive" => true,
"position" => array( array(14, 14), array(13, 14), array(12, 14)),
"poplast" => true);
$sql = "INSERT INTO players VALUES ('".$username."',
'".json_encode($newsnake)."'");
mysqli_query($link, $sql);

updatesnakes(sendSnakes());
echo json_encode(sendSnakes());
$link->close();
?>
```

### dbconnect.PHP

```
<?PHP
include_once 'dbinfo.PHP';
$link = mysqli_connect(DB_HOST, DB_USER, DB_PASSWORD, DB_DATABASE) or
die("server connect failed");
if(mysqli_connect_errno()) {
    printf("Det funkade inte att kontakta databasen: %s\n",
mysqli_connect_error());
    exit();
}
?>
```

### dbinfo.PHP

```
<?PHP define('DB_HOST', 'IPADRESS'); define('DB_USER', 'DBUSERNAME');
define('DB_PASSWORD', 'DBPASSWORD'); define('DB_DATABASE', 'DBNAME');
?>
```

### dbinfo.PHP

```
<?PHP
```

```

require 'dbconnect.PHP';
require 'functions.PHP';
$username = $_POST['name'];
$newdirection = $_POST['direction'];
$yourid = $_POST['id'];
$yourcounter = $_POST['counter'];
updatesnakes(sendSnakes());
updatecounter($yourid, $yourcounter);
$snakes = sendSnakes();
$snakes[$username]->direction = $newdirection;
$snakes[$username]->nextDirection = $newdirection;
storeUpdateName($username);
updatesnakedb($snakes);
?>

```

### **longpolling.PHP**

```

<?PHP
include 'dbconnect.PHP';
include 'functions.PHP';

$username = $_POST['name'];
$latestsync = $_POST['sync'];
$resync = false;
$now = time();

while(!$resync){

    $newtime = time();
    $serversynk = getLastupdate();
    if($serversynk != $latestsync){
        $jsonpackage["snakes"] = sendSnakes();
        $jsonpackage["apples"] = getApples();
        $jsonpackage["sync"] = $serversynk;
        $jsonpackage["playername"] = getPlayerName();
        $jsonpackage["a"] = getPlayerPing("a");
        $jsonpackage["b"] = getPlayerPing("b");
        $jsonpackage["c"] = getPlayerPing("c");
        $jsonpackage["d"] = getPlayerPing("d");
        echo json_encode($jsonpackage);
        $resync = true;
    }
    if($newtime > $now + 5){
        echo "pinged out";
        $resync = true;
    }
}
$link->close();
?>

```

## **remove.PHP**

```
<?PHP
include_once'dbconnect.PHP';
include_once'functions.PHP';

$username = $_POST['name'];
$color = $_POST['color'];

$sql = "DELETE FROM players WHERE name = '".$username.'";";
mysqli_query($link, $sql);

$sql = "INSERT INTO colors VALUES (NULL, '".$color.'')";
mysqli_query($link, $sql);
updatesnakes(sendSnakes());
?>
```