ARCADA

Utveckling av ett mjukvarubibliotek för dataöverföring från bildskärm till kamera

Max Gräsbeck

Examensarbete
Informations- och medieteknik
2015

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informations- och medieteknik
Identifikationsnummer:	4802
Författare:	Max Gräsbeck
Arbetets namn:	Utveckling av ett mjukvarubibliotek för dataöverföring från bildskärm till kamera
Handledare (Arcada):	Magnus Westerlund
Uppdragsgivare:	

Sammandrag:
Detta arbete beskriver DisplaySocket.js, ett JavaScript bibliotek som möjliggör överföring av data mellan två apparater med hjälp av bildskärm och kamera.
Teori för dataöverföring och tidigare arbete presenteras. Verktyg som använts för utvecklingen presenteras. Bibliotekets arkitektur och komponenter presenteras. Arbetet redogör för mätningsresultat.
Slutligen diskuteras bibliotekets samt teknikens framtid och på basis av diskussionen föreslås framtida arbete.

Nyckelord:	streckkod, QR-kod, javascript, dataöverföring, bibliotek, öppen källkod
Sidantal:	42
Språk:	Svenska
Datum för godkännande:	2.6.2015

DEGREE THESIS	
Arcada	
Degree Programme:	Information and Media Technology
Identification number:	4802
Author:	Max Gräsbeck
Title:	Development of a software library for data transmission from a display to a camera
Supervisor (Arcada):	Magnus Westerlund
Commissioned by:	

Abstract:
This thesis describes DisplaySocket.js, a JavaScript library that facilitates data transmission between two devices using a display and a camera.
Theory for data transmission and earlier work is presented. The tools utilized for development are presented. The architecture of the library and its components are presented. This thesis describes benchmarking results.
Finally, the future of the library and the technique are discussed and future work is suggested.

Keywords:	barcode, QR-code, javascript, data transmission, library, open source
Number of pages:	42
Language:	Swedish
Date of acceptance:	2.6.2015

INNEHÅLL

TABELLER

FIGURER

FÖRKORTNINGAR

API applikationsprogrammeringsgränssnitt (eng. application programming interface). 9, 18, 20, 28, 33

DS.js DisplaySocket.js. 8–10, 15–20, 22–25, 33, 37, 38

ES Ecmascript. 17

HCCB High Capacity Color Barcode. 9, 12

HRT High Resolution Time. 33

HTML HyperText Markup Language. 17

JS JavaScript. 8, 9, 14, 16, 17, 19, 25, 33

QR Quick Response. 9, 12, 14, 19, 20, 33, 35–37

WS WebSocket. 18, 20

1 INLEDNING

Detta arbete beskriver första publicerade versionen av DisplaySocket.js (DS.js), ett JavaScript (JS) bibliotek som använder kamera och skärm för överföring av data mellan apparater. Jag utvecklade biblioteket utgående från en prototyp som jag tidigare utvecklat. Användare kan importera DS.js i sin applikation, vilket möjliggör dataöverföring mellan slutanvändares apparater. DS.js lämpar sig för dataöverföring i sådana fall där traditionella överföringsmetoder, exempelvis över internet, inte är tillgängliga.

1.1 Bakgrund

Idén för överföring av data med kamera och skärm kom till när jag försökte föreställa mig en praktisk tredimensionell analogi för en tvådimensionell streckkod, eller datamatris. Figur 1 illustrerar ökning av mängden dimensioner. Det verkar logiskt att kalla den tredimensionella streckkoden för en *datakub* i brist på en bättre term. Eftersom vanliga bildskärmar endast har två dimensioner skulle det endast gå att visa en *skiva* av kuben i taget med en vanlig skärm, vilket i praktiken manifesterar sig som en video där varje ram innehåller en datamatris.

Figur 1. Från vänster till höger: En traditionell endimensionell streckkod, en datamatris och en datakub (konceptuell)

Fysiska datakuber har enligt Hamer (2005) exempelvis använts för att identifiera diamanter. Med en datakub menas i detta arbete egentligen en serie datamatriser. En serie datakuber skulle kunna kallas en datahyperkub.

I min undersökning för att få reda på om någon haft samma idé tidigare fann jag ett

blogginlägg av Nicholas (2011) som tangerade min idé. Undersökningen visade att han varken fortsatt med projektet eller gjort källkoden tillgänglig. Ett liknande projekt, Johnson (2012) och Johnson (2014), har därefter dykt upp men har inte öppen källkod och endast simplex kommunikation (se kap. 4.3.1). Före utvecklingen av DS.js utvecklades en prototyp i JS som med hjälp av biblioteken jsqrcode av Laszlo (2011) och qrcodejs av Sangmin (2014) överför godtyckliga mängder data via simplex kommunikation. Efter att utvecklingen av DS.js börjat hittade jag även ett liknande projekt av Langlotz & Bimber (2007), som ökar mängden dimensioner till fyra eller fem med hjälp av färg och intensitet. De har emellertid tydligen inte uppnått höga överföringshastigheter och använder endast simplexöverföring.

Även om inspirationen för DS.js är dataöverföring med en serie datamatriser borde biblioteket vara oberoende av metoden som används. Metoden kan lika väl bestå av text, streckkoder, datamatriser eller någon helt annan metod. Det finns många olika sorters streckkoder och datamatriser och jag antar att någon av dem lämpar sig bäst för dataöverföring, exempelvis med att ha hög kapacitet som High Capacity Color Barcode (HCCB) utvecklad av Microsoft (2015) eller för att tolkningen av dem är snabb som Quick Response (QR). I praktiken har endast metoden för överföring med QR-koder lyckats tills vidare.

1.2 Syfte och mål

Målet med detta arbete är att beskriva DS.js, ett JS bibliotek med öppen källkod. Arbetet beskriver hur DS.js möjliggör läsandet av data med en kamera från en skärm i en webbläsare.

Arbetet redogör för hur användare med hjälp av bibliotekets applikationsprogrammeringsgränssnitt (eng. application programming interface) (API) kan åstadkomma överföring och mottagning av data samt motivationen för API:ets design.

De olika komponenterna som uppgör biblioteket beskrivs i kap. 5. Komponenternas växelverkan beskrivs i kap. 2.5. Målet med detta är att ge läsaren en grundläggande förståelse

om hur DS.js åstadkommer dataöverföring.

Arbetet utreder de teoretiska gränserna för dataöverföring från maskin till maskin via utnyttjandet av bildskärm och kamera samt rapporterar för hur nära den teoretiska max-hastigheten DS.js i praktiken kommer.

1.3 Avgränsning

Arbetet går inte in på implementationsdetaljer av den bildanalys som möjliggör optisk teckenläsning, streckkodsläsning och datamatrisläsning.

Arbetet beskriver inte en optimal lösning för dataöverföring från bildskärm till kamera. Flera *genvägar* har tagits för att underlätta utvecklingen vilket uppenbarar sig i mängden funktioner som fattas i slutprodukten.

DS.js har ingen metod för att gissa vilken metod sändaren försöker använda och det ligger på användarens ansvar att se till att metoderna stämmer överens. En instans av biblioteket som försöker tolka text kommer inte att klara av att tolka en streckkod.

2 METODER FÖR VISUELL DATAÖVERFÖRING

Det finns en skillnad mellan data och information. Det finns också flera olika sätt att representera information med hjälp av data. Text är ett traditionellt sätt att representera data, men det är svårt för datorer att tolka text, medan streckkoder i relation är ett ganska nytt fenomen, men mycket lättare att läsas med maskin.

2.1 Vad är information

Information är något som har innehåll, ett system med tillstånd, vars betydelse beror på den kontext informationen behandlas i. Om tillståndet i ett system inte ändras på måfå kan systemet användas för att lagra information. Informationen kan ta sig från ett system

till ett annat. Om ett system rör på sig följer informationen systemet. (Israel & Perry, 1990)

Ett papper med text innehåller information. En beskrivning av varje atom som pappret uppstår av innehåller relativt mycket information, medan innehållet av texten kräver relativt lite information. Någon som inte kan tolka bokstäverna eller förstå språket kanske ändå får någon information tack vare den kontext pappret finns i, exempelvis en bussbiljett för en turist som inte kan språket. Däremot består en fullständig beskrivning av pappret till största delen endast av brus för någon som endast är intresserad av textens innehåll. Det finns en skillnad mellan data och information där information kan sägas vara nyttig data.

2.1.1 Representation av information i en dator

En dator kan lagra och bearbeta data, som i dagens läge oftast är elektriska signaler som representerar binära nummer. Binära nummer har två siffror, oftast representerade med 0 och 1 eller 0_2 och 1_2. En binär siffra kallas även för en bit. Åtta bitar kallas för en byte. Man använder hexadecimala nummer för att underlätta skrivandet av en byte, så att $240_{10} = 11110000_2 = F0_{16}$. (Bell et al., 2015)

2.1.2 Text

Man har kommit överens att vissa nummer representerar olika tecken. Tecknet 'A' representeras i ASCII av decimala numret 65_{10}, som i binär form blir 01000001_2. Strängen "Hello world!" skulle således kodas som 72 101 108 108 111 32 119 111 114 108 100 33 decimalt, eller 01001000 01100101 01101100 01101100 01101111 00100000 01110111 01101111 01110010 01101100 01100100 00100001 binärt. (Bell et al., 2015)

2.1.3 Streckkod

Symbolen till vänster i Figur 1 är en streckkod. Det finns flera olika streckkoder, som i princip fungerar på samma sätt. En streckkod består av en mängd parallella linjer och deras mellanrum. Dessa har varierande bredd. Linjerna representerar binära siffran 1 och deras mellanrum representerar binära siffran 0. Breda linjer och mellanrum representerar flera siffror så att t.ex. en tunn linje är 1 och en bred linje är 111.

Vanligen använder streckkoder någon form av kodning som hindrar linjer som är för breda. De har också vissa tekniker som underlättar läsande så som *tysta zoner* (eng. quiet zone) och *vaktmönster* (eng. guard pattern), fast dessa beror på formatet. (Galindo, 1999)

Med att ta en bild av en streckkod kan man analysera bilden och tolka den relativa bredden av mörka områden jämfört med ljusa områden.

2.1.4 Datamatris

Mellersta symbolen i Figur 1 är en datamatris. En datamatris fungerar i princip på samma sätt som en streckkod men ökar mängden dimensioner från en dimension till två. Vissa format lägger till färg, så som HCCB av Microsoft (2015) och *4D streckkoder* av Langlotz & Bimber (2007).

En datamatris innebär i detta arbete vilken som helst form av tvådimensionell sträckkod eller matrissymbologi och bör inte förväxlas med en Data Matrix av Microscan Systems eller andra matrissymbologiformat.

QR-kod

Ett format för datamatriser är QR-koder, illustrerade i Figur 4 och Figur 10, som är standardiserade av ISO/IEC (2000). De har *structured append mode* där upp till 16 QR-koder kan läsas i sekvens. Största QR-koden kan innehålla 2953 byte data. (ISO/IEC,

2000).

2.1.5 Bildskärm

Man kan få data ur en dator med hjälp av en utenhet. Av utenheter är bildskärmen mest relevant för detta arbete. På en bildskärm skapas bilder i regel med pixlar. Pixlarna består ofta av små ljus som är röda, gröna och blåa. Olika styrkor för de olika färgerna leder till en upplevelse av olika färger i en enskild pixel.

Denna representation av färger sker med 3 byte, vilket ger $3(2^8) = 16777216$ kombinationer för olika färger. För att rita ut en vit bokstav med svart bakgrund på en bildskärm skulle pixlar i samma mönster som bokstaven få värdet $FFFFFF_{16}$ medan bakgrunden skulle få värdet 000000_{16}.

Representation av text med en bildskärm innebär att en stor mängd data används för att visa relativt lite information i form av text. Mängden pixlar per bokstav kan variera drastiskt från en bildskärm till en annan beroende bland annat på operativsystem, resolution och textstorlek, men för att demonstrera konceptet antar vi en höjd och bredd på 16 pixlar. Bokstavens pixlar kan således representeras av en matris $P_{16,16}$ där varje pixel använder 3 byte, vilket leder till att man representerar en byte med $3(16^2) = 768$ byte. Detta liknar fenomenet som diskuterades i kap. 2.1. Man skulle kunna använda pixlarna till att representera 3 bokstäver var och skriva ut ett meddelande på 768 tecken inom P. Det skulle möjligen bli en intressant bild, men det skulle vara svårt att tolka meningen av bilden i vanliga fall. På samma sätt är en datamatris som ritas ut av en bildskärm ofta uppbyggd av flera pixlar per *modul*, dvs. de små rutorna som en datamatris består av, men de skulle kunna representeras av endast en pixel var.

2.1.6 Kamera

På samma sätt som en bildskärm fungerar som en utenhet för visuell information fungerar digitala kameror som en inenhet för visuell information. En digital kamera representerar data i stort sett på samma sätt som en bildskärm.

Det finns olika sätt för digitala kameror att få bilddata, men i huvudsak går det ut på att en matris av sensorer mottar ljus. Sensorernas data kan sedan skickas till en dator som kan använda det.

2.2 Läsning av text med ocrad.js

Optisk teckenläsning (eng. Optical Character Recognition (OCR)) är tekniken som används för att tolka bokstäver som finns i bilddata. Optisk teckenläsning är svår enligt Kwok (2015), och resultaten är ofta varierande.

Ocrad.js, en port av Ocrad, är ett JS bibliotek för optisk teckenläsning av Kwok (2015). Biblioteket testades men det gick inte att få en enda hel delsträng fullständigt tolkad. Licensen är GPL3 medan jsqrcode använder Apache 2. I fall att testet hade fungerat skulle vidare forskning om huruvida licenserna är kompatibla ha varit nödvändig.

2.3 Läsning av 1D-streckkoder

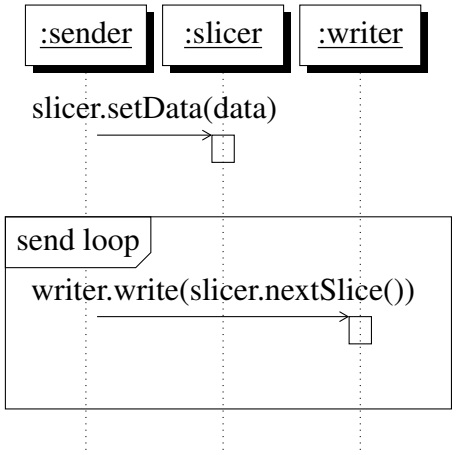
Tre bibliotek, JOB av Larsson (2015), unix-toolbox.js-exact-image av manuels (2015) och quaggaJS av Oberhofer (2015), testades för läsning av endimensionella streckkoder, men inget test lyckades läsa streckkoder i realtid.

2.4 Läsning av 2D-streckkoder med jsqrcode

För att tolka QR-koder används jsqrcode som Laszlo (2011) porterat från XZing biblioteket. En fil i porten modifieras så att man undviker anrop till console.log() funktionen och i stället rapporterar fel och statistik med ett returvärde.

2.5 Introduktion till processen

Användningsfallet är att användaren har en sträng som denna vill skicka från en apparat till en annan med DS.js. Strängen delas först upp i delar, varje del numreras enligt sin ordning och en checksumma räknas.

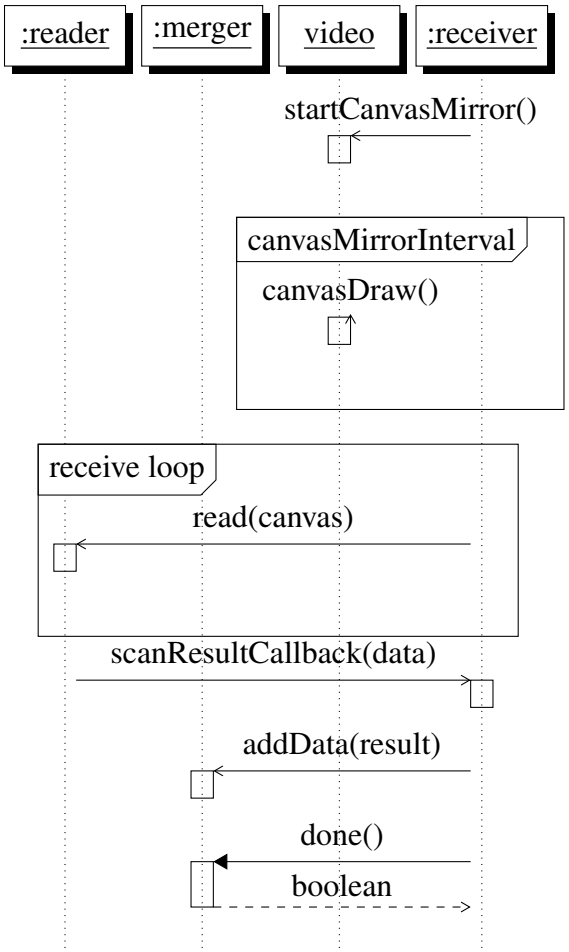


Figur 2. Sekvensdiagram som illustrerar processen för att sända data.

Första delen är en header-del som för tillfället inte innehåller annat än metadata. Första numret anger att det är frågan om en header-del, andra om hur många delar som kommer att följa (0 indexerat) och sista numret är en checksumma för hela nyttolasten.

Delens nummer, separerat av tecknet ’/’, följs av checksumman. Ett tecken läggs till, i detta fall ’$’, som separerar nyttolasten från metadata. I figur Figur 4 illustreras output från DS.js konfigurerad till en delsträngslängd på 10 tecken. Inputsträngen kommer från en fil med 10 byte av slumpmässig data som konverterats av demoapplikationen till en base64-kodad sträng.

I Figur 2 illustreras hur en instans av DS.js som skickar data, *sender*, först splittrar data och sedan i en obestämd tid visar nästa del. I Figur 3 illustreras hur en instans av DS.js som tar emot data, *receiver*, först startar videokomponenten som skriver till canvaselementet och sedan försöker tolka den bilddata som finns i canvaselementet. När instansen får tag i ett resultat matas det in i sammanfogaren (eng. merger). När denna lyckas med sammanfogningen är överföringen färdig.



Figur 3. Sekvensdiagram som illustrerar processen. Sender är en instans som skickar data, receiver är en instans som mottar data.

data:text/plain/base64,cJQ46cQoElPMUQ==

Figur 4. Ursprungliga strängen "data:text/plain;base64,cJQ46cQoElPMUQ==" delas upp i en header del och fyra nyttolastdelar.

3 VERKTYG

Det huvudsakliga programmeringsspråket i DS.js är JS och projektets uppgiftsautomatiserare är Grunt. Det finns också ett skript som underlättar skapandet av testcertifikat.

3.1 JavaScript/Ecmascript

JS är en dialekt av skriptspråket Ecmascript (ES). ES standarden har från och med 1997 publicerats av ECMA (2011) och stöds av så gott som alla webbläsare.

ES körs asynkront inom en värdmiljö, som förser programmet med input och output. En typisk värdmiljö är en webbläsares JavaScriptmotor. Det typiska användningsområdet för ES är manipulation av HyperText Markup Language (HTML) dokument.

ES är ett multiparadigmspråk som stöder imperativ, objektorienterad och funktionell stil. Språket är delvis inspirerat av programmeringsspråken Java, Self och Scheme. ES är prototyp-baserat, vilket innebär att man inte deklarerar klasser utan skapar kloner av objekt.

ES är dynamiskt typat. Det innebär att en variabel som först är ett nummer kan bli en sträng, en funktion eller vilken annan typ som helst.

ES har första klassens funktioner som kan vara anonyma. Detta innebär att funktioner är objekt och kan vara utan namn. Figur 5 illustrerar detta. På rad 9 skickas en anonym funktion som parameter till en annan funktion.

3.2 Grunt

Grunt är DS.js uppgiftsautomatiserare och sköter om konkatenering, förminskning och felupptäckning (eng. linting) samt vid behov servering och automatisk uppdatering av HTML-sidor. DS.js använder fem grunt plugins, grunt-contrib-concat, grunt-contrib-connect, grunt-contrib-jshint, grunt-contrib-uglify och grunt-contrib-watch för att automatisera uppgifter.

JSHint är ett verktyg som upptäcker fel och potentiella problem i JS-kod samt hjälper upprätthålla programmeringsstil inom ett projekt.

Grunt-contrib-connect används i utvecklingsstadiet som statisk webbserver. Servern kon-

figureras i Gruntfile.js till att använda sig av https protokollet. Det finns ett skript inkluderat i biblioteket som skapar certifikat som används av andra verktyg. De skapade certifikaten listas i projektets .gitignore fil, vilket leder till att de inte sparas av versionshanteringsprogrammet. LiveReload är inbyggt i grunt-contrib-watch. LiveReload laddar om en webbsida då specificerade filer ändras, vilket leder till ökad produktivitet eftersom mängden steg som behövs för att testa ny kod minskar till att man endast behöver spara en fil.

Konkatenering och minifiering

Eftersom det är lättare för användare har jag valt att man skall kunna lägga till biblioteket med endast en fil. Emellertid går det lättare att förstå ett program vars komponenter är i skilda filer. Projektet innehåller därför en distributionsfil som är resultatet av en konkatenering av projektets filer. Konkateneringen sköts av grunt-contrib-concat. Distributionsfilen finns i katalogen dist/. Den konkatenerade filens storlek kan minskas med att ta bort alla onödiga tecken från källkoden. Detta bör göras på ett sådant sätt att funktionaliteten inte ändras. Den resulterande filen sparas ofta med samma filnamn som ursprungsfilen men med en tilläggsfiländelse *.min*, så att om ursprungliga filnamnet är *displaysocket.js* så blir minifierade filnamnet *displaysocket.min.js*.

4 FÖRSTUDIE

Efter undersökningen som diskuterats i kap. 1.1 utvecklades en prototyp. Utvecklingen av prototypen hjälpte med att förstå problemområdet och fungerade som inspiration till DS.js. Ett bibliotek borde ha en API som är lätt att använda. För att öka sannolikheten att användare skulle förstå API:et togs inspiration från WebSocket (WS) API:et.

Som ett bibliotek kommer DS.js med externa beroenden färdigt inbakade i distributionsfilen.

Bibliotekets har två funktioner, skickande och mottagande av data. Beroende på tillgäng-

lig hårdvara kan antingen bägge funktioner användas samtidigt eller så kan en funktion åt gången användas.

4.1 Prototyp

För arbetet skapades en prototyp som, likt sin efterträdare, använder ett bibliotek, qrcodejs av Sangmin (2014), för att skapa QR-koder och ett annat bibliotek, jsqrcode av Laszlo (2011), för att läsa dem. Nedan beskrivs vilka begränsningar prototypen har och vilka resultat den gav.

Begränsningar

Prototypen använder sig endast av simplex dataöverföring, där en instans av prototypen skriver ut QR-koder i en oändlig loop medan den andra fångar dem tills den fått alla delar.

Resultat

Nyttan med utvecklingen av prototypen var att den visade att dataöverföring med en serie datamatriser fungerar i praktiken. Utvecklingen gav vissa antydanden om hur man skulle kunna utveckla metoden vidare. Prototypen fungerade som inspiration för DS.js projektets arkitektur.

I samband med utvecklingen av prototypen kom jag underfund med att det inte är nödvändigt att använda sig av QR-koder eller ens datamatriser för dataöverföring, vanliga streckkoder eller text skulle också kunna användas. Således beslöt jag mig för att skapa ett JS bibliotek som abstraherar metoden för att representera data på skärmen, så att man lättare kan hitta en optimal metod för dataöverföring med skärm och kamera.

4.2 Inspiration från WebSocket

WS är ett protokoll för fullduplex dataöverföring över en TCP socket. DS.js API är delvis inspirerat av WS API:et (och namnet), eftersom webbutvecklare kan antas vara bekanta med det. DS.js API är inte en fullständig kopia av WS API:et. På samma sätt som WS möjliggör dataöverföring t.ex. från klient till server tillåter DS.js dataöverföring från en apparat till en annan. Till skillnad från WS implementerar DS.js inte händelserna *open* eller *close*.

Överföringsmetod går att definieras, men eftersom andra metoder inte hittills fungerat är QR-kod metoden vald som standardöverföringsmetod.

```
var socket = new WebSocket('ws://echo.websocket.org');

// mottagning sker med antingen
socket.onmessage = function(event) {
    [ ... ]
}

// eller
socket.addEventListener("message", function(event) {
    [ ... ]
});
```

Figur 5. WebSocket exempel för mottagning av data

```
var socket = new WebSocket('ws://echo.websocket.org');
var data = "example data";
socket.send(data);
```

Figur 6. WebSocket exempel för skickande av data

För en användare som tidigare använt WS API:et borde DS.js API:et verka bekant. Både direkt definition av DS.js attribut eller bindandet av en lyssnare går att användas för mottagning av meddelanden i både WS (Figur 5) och DS.js (Figur 7). Skickande av data med DS.js (Figur 8) är på samma sätt inspirerat av WS (Figur 6). (W3C, 2011)

```
var videoElement = document.getElementById('video');
var displaySocket = new DisplaySocket({video: videoElement});

// mottagning sker med antingen
displaySocket.onmessage = function(event) {
    [ ... ]
}

// eller
displaySocket.addEventListener("message", function(event) {
    [ ... ]
});
```

Figur 7. DisplaySocket exempel för mottagande av data

```
var videoElement = document.getElementById('output');
var displaySocket = new DisplaySocket({output: outputElement});
var data = "example data";
displaySocket.send(data);
```

Figur 8. DisplaySocket exempel för skickande av data

4.3 Bibliotekets funktioner

I huvudsak består biblioteket av två funktioner, skickande och mottagande av data. Data skickas med att skriva ut det i ett område på skärmen specificerat av användaren. Olika sätt att representera data borde vara utbytbara, och en adapter för olika metoder behövs. Olika sätt att skriva ut data kan exempelvis vara vanlig text, streckkod eller datamatris.

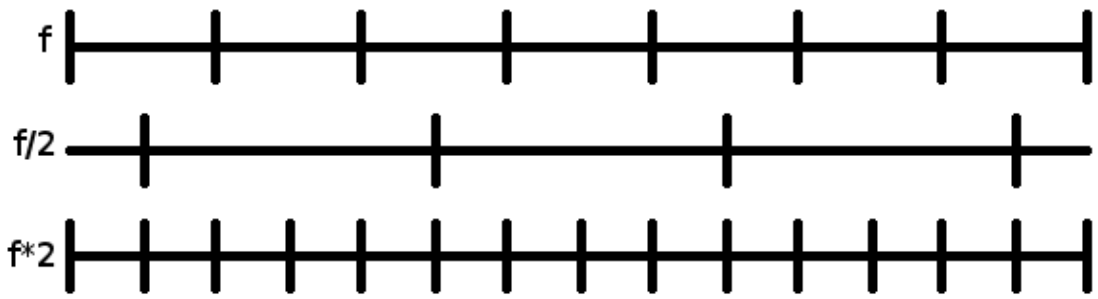
Andra funktionen går ut på att läsa data från en skärm. För detta måste man ha en kamera i apparaten (fast en videofil kan också användas). En bild i sänder läses och bearbetas. Det krävs att läsaren förstår de format som skrivaren har skrivit ut. Om skrivaren använder sig av t.ex. Code 128 streckkoder borde också läsaren försöka tolka bildens innehåll som det formatet, vilket kräver att biblioteket har en modul som implementerar tolkning av Code 128 streckkoder. Man skulle möjligen kunna pröva alla olika metoder i början av dataöverföringen för att finna en metod som ger önskad data och sedan fortsätta använda den metoden, men det är inte implementerat i praktiken, så användaren får se till att metoderna stämmer överens.

4.3.1 Simplexöverföring

Simplexöverföring innebär enligt Brown (2000) att data endast flödar i en riktning. Ett typiskt exempel av simplexöverföring i FM-radio. DS.js kan använda simplexkommunikation när ena parten inte har en kamera eller om man vill överföra data till flera parter samtidigt. Detta innebär att funktionen som sänder data spjälkar upp det den skall skicka i flera bitar och skriver ut data i den representationen som funktionen använder sig av. Samma spjälkning sker också i duplex-överföring, vilket ger upphov till utvecklandet av en skild komponent för spjälkningen.

4.3.2 Sändnings- och läsningsfrekvens

När någon läser data som skickas med simplex kommunikation gäller det att läsa minst två gånger så snabbt som den skickande parten visar sin data (liknar Nyquistfrekvens). För att kunna läsa signalen med frekvensen f bör man läsa med minst $2f$. Avläsningshastigheten $\frac{f}{2}$ missar således minst var annan del i signalen f (Figur 9).



Figur 9. Frekvenserna f, f/2 och 2f.

Om avläsningsfrekvensen är harmonisk med sändningsfrekvensen riskerar man att vissa meddelanden missas, eller att kameran tar bild precis när bilden byts ut, vilket leder till att en del av informationen förblir otillgänglig (Figur 10). Den skickande parten har i vanligt fall inget sätt att ta reda på den bästa frekvensen för bildutbyte. Man skulle emellertid kunna lägga till funktionalitet i gränssnittet som skulle tillåta ändring och adaptering av frekvensen. Då skulle slutanvändaren exempelvis med en slider kunna ändra frekvensen i den sändande apparaten. Tills vidare tillåter gränssnittet inte ändring av frekvensen, så

Figur 10. Bild tagen av QR-kod med webbkamera samtidigt som en QR-kod byts ut mot en annan

det är endast möjligt att ändra frekvensen genom att skapa en ny sändarinstans.

4.3.3 Halv- och fullduplex

Halvduplex innebär att data flödar i två riktningar men endast i en riktning i taget, exempelvis i en *Walkie-talkie*. Fullduplex innebär att data flöder i bägge riktning samtidigt, som vid vanliga telefonsamtal (Brown, 2000).

DS.js åstadkommer en form av halvduplex kommunikation på applikationsnivå med fullduplex på hårdvaronivå. Den skickande parten försöker avläsa den mottagande partens skärm, exempelvis när två mobiltelefoner med kameror på samma sida som skärmen kan *se* varandras bildskärmar. Den mottagande apparaten skickar ett kort meddelande om vilken del denna senast har tagit emot. Detta leder till en *sakta men säker* dataöverföring där sändning av en del kräver att sändaren tolkar mottagarens skärm och vice versa, innan nästa del kan skickas.

Problemet med tekniken som DS.js använder för sin duplex överföring är att data flödar i bägge riktningarna men åt användaren nyttig information flödar endast åt ett håll i taget. DS.js så gott som kräver att en ny instans skapas efter ett meddelande.

Äkta fullduplex förblir oimplementerat i DS.js tills vidare. Ett bättre sätt för äkta fulldu-

plex överföring skulle antagligen innebära två kontinuerliga simplexkanaler från bägge apparat. Dessa skulle sedan kunna skicka data oavbrutet och i sådant fall att de missar en del skulle de kunna meddela om det till den andra parten, som sedan skulle repetera den saknade delen. Nyttolasten kunde inkluderas och skickas kontinuerligt så att användaren både kunde sända och skicka meddelanden samtidigt, till skillnad från den nuvarande tekniken.

4.4 Hantering av externa beroenden

Antingen skulle ansvaret för att hantera bibliotek som DS.js behöver ligga hos användaren eller hos mig som upprätthållare av biblioteket. Jag har valt att själv inkludera bibliotek eftersom man på det sättet undviker problem som uppstår när man delar bibliotek med övrig mjukvara (Donald, 2003). Det är emellertid fullständigt möjligt för någon att experimentera med andra versioner av använda bibliotek eller att inkludera sådana bibliotek som inte ursprungligen varit med. Det skulle antagligen vara krångligt för användare att hantera beroenden på egen hand.

Exempelvis kräver inkludering av jsqrcode 17 filer med sidoeffekten att den läcker 3 onödvändiga globala variabler (Laszlo, 2011). Med att inkludera jsqrcode direkt i DS.js behöver användaren endast lägga till en fil och undviker med det samma onödvändiga globala variabler. Dessutom skulle det också med tiden kunna leda till att användaren behöver lägga till väldigt många bibliotek, varav vissa kan ha en tidskrävande tilläggningsprocess.

Distributionsfilens storlek ökar av att bibliotek inkluderas. Denna avvägning är godtagbar med tanke på att det gör utveckling snabbare för användaren.

5 BIBLIOTEKETS STRUKTUR

Varje komponent är definierad i en egen fil och har ett eget ansvarsområde. Komponenterna är inte tillgängliga för användaren direkt. Filerna finns i src/ katalogen. Filerna main.js

och defaults.js har en relativt imperativ struktur, fast de egentligen befinner sig innanför själva DisplaySocket objektet. De har hög koppling till DS.js. Filerna video.js, slicer.js, writer.js, reader.js och merger.js har en objektorienterad struktur och borde i teorin inte ha någon koppling till programmet. Filerna intro.js och outro.js finns till för att underlätta sättet på vilket bibliotekets distributionsfil konkateneras.

5.1 Intro.js och outro.js

Intro och outro filerna är en teknik som finns i flera andra JS bibliotek, exempelvis jQuery (2015) biblioteket. De finns till för att ge hela biblioteket egen *scope* så att tillägg av biblioteket endast skapar en global variabel. Intro filen påbörjar deklarationen av DS.js funktionen. Filen deklarerar också sådana variabler som senare inkluderade bibliotek av misstag gör globala.

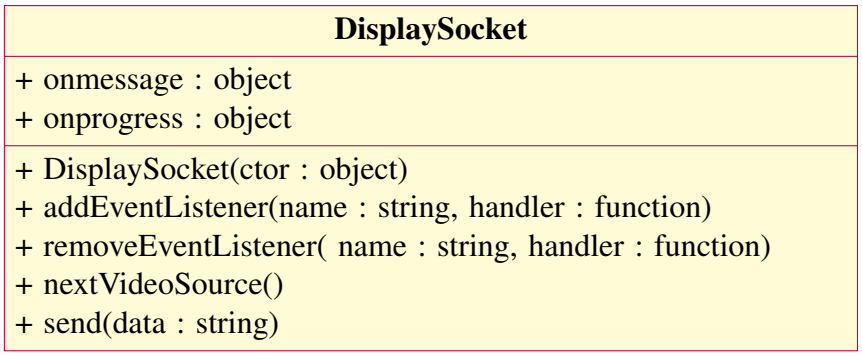
Outro filen returnerar gränssnittet och slutför deklarationen av DS.js funktionen.

5.2 Huvuddel - main.js

Bibliotekets huvudsakliga logik finns i huvuddelen, main.js. Även all sådan funktionalitet som jag inte ännu lyckats separera till en egen fil finns i huvuddelen tills vidare. Huvuddelen lägger till funktioner till gränssnittet som senare returneras i outro-filen.

5.2.1 Gränssnittet

En instans av DS.js har ett publikt gränssnitt, illustrerat i Figur 11, som definieras med en stil som liknar den som syns i Figur 12, som åstadkommer inkapsling. Det publika gränssnittet syns i Figur 11. Inspirationen för gränssnittet diskuteras i kap. 4.2.



Figur 11. DisplaySocket objektets publika gränssnitt

```
var A = function() {

    function B() {
        var privateValue = 0;
        return {
            doSomething : function() {
                return ++privateValue;
            }
        };
    }

    var b = new B();

    var iface = {
        publicFunction : function() {
            return b.doSomething();
        }
    };
    return iface;
};

var a = A();
a.publicFunction(); // 1
a.publicFunction(); // 2
a.b // undefined
```

Figur 12. Demonstration av hur man kan åstadkomma inkapsling

5.3 Standardvärden - defaults.js

Filen defaults.js hanterar konstruktorobjektet och definierar standardvärden som används av biblioteket.

Minimikrav

Biblioteket initialiseras med ett konstruktorobjekt som kräver åtminstone ett värde. Om man endast definierar *output*-egenskapen blir instansen en simplex sändare, medan definiering av endast *video*-egenskapen skapar en simplex mottagare. Detta illustreras i Figur 8 respektive Figur 7. Definition av bägge leder till att instansen beter sig som en duplex mottagare, varpå om man kallar på sändningsfunktionen ändras den till en duplex sändare. Definition av andra värden påverkar inte det grundläggande beteendet.

Delsträngslängd

Längden av nyttolasten som skickas kan konfigureras. Standardvärde är 250 tecken. Man skulle alternativt kunna definiera mängden delar man vill ha, men det skulle kunna leda till att man försöker skicka mera data på en gång än vad som ryms på bildskärmen.

Intervaller

Det finns tre huvudsakliga intervaller som definieras: basintervallen, läsningsintervallen och sändningsintervallen. Basintervallen används av de andra intervallerna om användaren inte specificerar deras värde skilt. Läsningsintervallen har i vanligt fall samma värde som basintervallen. Värdet för sändningsintervallen är fyra gånger basintervallen, eftersom detta tycks leda till mera stabil simplex kommunikation än om man gör sändningsintervallen dubbelt så lång som basintervallen, vilket i teorin borde räcka. Värdet för basintervallen är 30 millisekunder vilket betyder en frekvens på $33\frac{1}{3}$ Hz för läsning och $8\frac{1}{3}$ Hz för skrivning.

5.4 Video - video.js

Video
- canvasMirrorInterval : object - canvasBeingMirrored : boolean - context : object - currentVideoSource : number - videoSources : object - videoStream : object
+ Video(videoElement : object, method : string) - getSources(sources : object) - start(source : object) - videoSuccess(newVideoStream : object) - videoError(error : object + canvasDraw() + startCanvasMirror() + stopCanvasMirror() + capture() + changeSource()

Figur 13. Videokomponenten

Biblioteket innehåller en videokomponent, illustrerad i Figur 13, vars uppgift är att binda kamera output till en video-tag som användaren tidigare specificerat samt att flytta dess data till ett canvaselement från vilken olika läsare klarar av att tolka bilden. Videokomponenten tar som indata ett videoelement och ett canvaselement. Videokomponenten skriver data till canvaselementet i samma takt som läsaren läser från den.

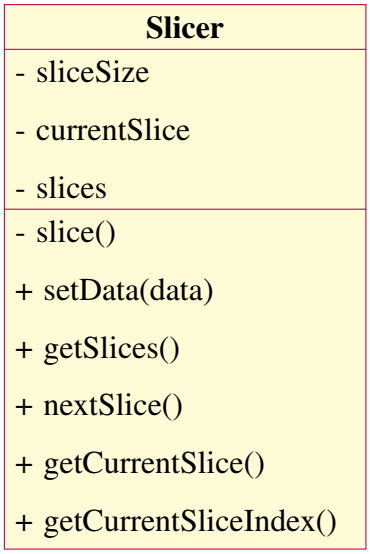
Komponenten tillåter byte av kamera, eller videokälla, om en apparat har flera videokällor.

Videokomponenten fungerar som en adapter för Media Stream API:et som är en del av WebRTC specifikationen av W3C (2015). Således borde det inte spela någon roll varifrån

videokällan härstammar, det är miljöns uppgift att hantera det.

Komponenten hanterar två uppgifter samtidigt vilket antagligen ökar komponentens komplexitet i relation till andra komponenter.

5.5 Delare - slicer.js



Figur 14. Delarkomponenten

Delaren delar data i mindre delar på sådant sätt att sammanfogaren kan lägga ihop dem oberoende av vilken ordning denna tar emot dem. Delaren mottar indata i form av en textsträng och delar den i flera mindre strängar.

I början av delsträngarna tilläggs ett nummer som representerar delsträngens position i ursprungliga strängen, samt ett annat nummer som fungerar som en checksumma av delsträngens nyttolast. Checksumman är nödvändig eftersom metoderna som används för läsning ibland ger fel resultat, även om t.ex. QR-koder har inbyggd felhantering.

En textsträng är inte det bästa sättet att representera data med tanke på informationstäthet, men är behändig i utvecklingsskedet.

Första delen skapad av delaren innehåller endast metadata om hur många delar delaren

har skapat samt en checksumma för hela nyttolastet.

5.6 Skrivare - writer.js

Writer
- chosenWriter : object - chosenWriterFunction : function
+ Writer(outputElement : object, method : string) + write(text)

Figur 15. Skrivarkomponenten

Skrivaren är en adapter för olika metoder att skriva ut data i det specificerade området. Skrivarens uppgift är att konfigurera det bibliotek som motsvarar valda metoden så att dess output är läsbart. Skrivaren väljer rätt metod på basis av vad användaren har valt. Skrivarens gränssnitt accepterar indata och skickar det vidare till den valda metodens funktion.

Biblioteket skickar delsträngar från delaren till skrivaren antingen periodvis eller när mottagaren meddelar att den tagit emot senaste utskrivna skivan. Skrivaren används också för att meddela vilken skiva som senast tagits emot med att skriva ut mottagna delens nummer.

5.7 Läsare - reader.js

Reader
- callbackFunction : function
+ Reader(callback : function) + read(canvas : object)

Figur 16. Läsarkomponenten

Läsaren, illustrerad i Figur 16, är en adapter för olika metoder att tolka bilden i canvas-elementet. Likt skrivaren väljer läsaren på basis av metoden som användaren valt vilket funktionsanrop som skall göras. Läsaren accepterar en callbackfunktion till vilken det tolkade datat skickas i form av en sträng.

5.8 Sammanfogare - merger.js

Merger
- slices : object - current : number - total : number - checksum : number - previous : number
+ addData(slice : string) + getPrevious() + getCurrent() + done() + merge() + getProgress()

Figur 17. Sammanfogarkomponenten

Sammanfogaren, illustrerad i Figur 17, har uppgiften att ta emot strängar som delaren skapat och foga ihop dem. Läsarens output matas till sammanfogaren, som lägger till delarna i en lista.

Sammanfogarens gränssnitt har funktioner för att lägga till delar, ta reda på om alla delar är mottagna, mata ut det sammanfogade nyttolasten samt en funktion som returnerar ett objekt med data som är relevant för en förloppsindikator.

Sammanfogaren spjälker upp mottagna strängar på basis av separationstecken, ’/’ och ’$’ och extraherar positionen för nyttolasten samt dess checksumma. Om checksumman inte

stämmer överens med nyttolasten förkastas delen. Om checksumman stämmer överens godkänns nyttolasten och läggs till i listan av accepterade delar.

I simplex överföring kan en mottagande instans missa vissa delar och därmed få delar i annan ordning än skickande parten skickat dem. En vanlig förloppsindikator skulle antagligen frustrera slutanvändare, både eftersom procentenheten för mottagna bitar kan proportionsmässigt minska och för att den totala mängden mottagna bitar i simplex överföring i slutet av överföringen har en tendens att förbli oläsliga tills kameran är rätt positionerad. För att lösa detta problem har demonstrationsapplikationen en osekventiell förloppsindikator. Denna syns i Figur 18. De svarta rutorna representerar en del som instansen har och de gråa en del som den inte har. Den gröna rutan representerar den senast lästa delen.

Figur 18. Demonstrationsapplikationens osekventiella förloppsindikator.

6 MÄTNINGSRESULTAT

Mätningsresultaten i detta kapitel finns till för att vägleda vidare utveckling och mätning men borde inte anses ge en fullständig bild av teknikens hastighet. Hastigheten beror i stort sett på hårdvarans kapacitet.

6.1 Förväntningar

Den teoretiska maximihastigheten estimeras av Nicholas (2011) ligga vid 43 KiB/s för QR-koder.

> 15 FPS x 2,953 bytes per frame = 43 KB/s.
>
> (Nicholas, 2011)

Min förväntning för effekten av delsträngslängd är att den metod som är *nära* vanliga icke-seriella överföringsmetoder borde vara snabbast, det vill säga mängden delar minskar hastigheten. Gällande basintervall och frekvensen som kan härledas från den, antar jag att en frekvens som är nära den frekvens som kameran tar bilder med borde ge den högsta hastigheten.

6.2 Metod

En bra överföringshastighet för en given apparat går att härledas heuristiskt genom att variera värdet för basintervall och delsträngslängd.

Endast QR-kodmetoden mäts, eftersom det är enda metoden som är fullständigt implementerad. Två testfiler skapas med slumpmässig data.

Mätningen utförs 50 gånger per värde som mäts när överförda filen är 1 KiB och 10 gånger för 10 KiB filen. Första mätningarna räknas inte med eftersom de påverkas för mycket av sådana faktorer som att kameran inte är rätt fokuserad eller positionerad.

High Resolution Time (HRT) är ett JS API definierat av W3C (2012) för mätning av tid. HRT anger tid med åtminstone en millisekunds noggrannhet. DS.js mäter tiden det tar från att första biten har blivit mottagen samt tiden då sista biten är mottagen med hjälp av HRT API:et. Mätningarna tar därmed inte i beaktande den tid det tar att tolka första QR-koden. Det minskar den uppmätta tiden lika mycket som det tar att tolka en QR-kod.

6.3 Instrument

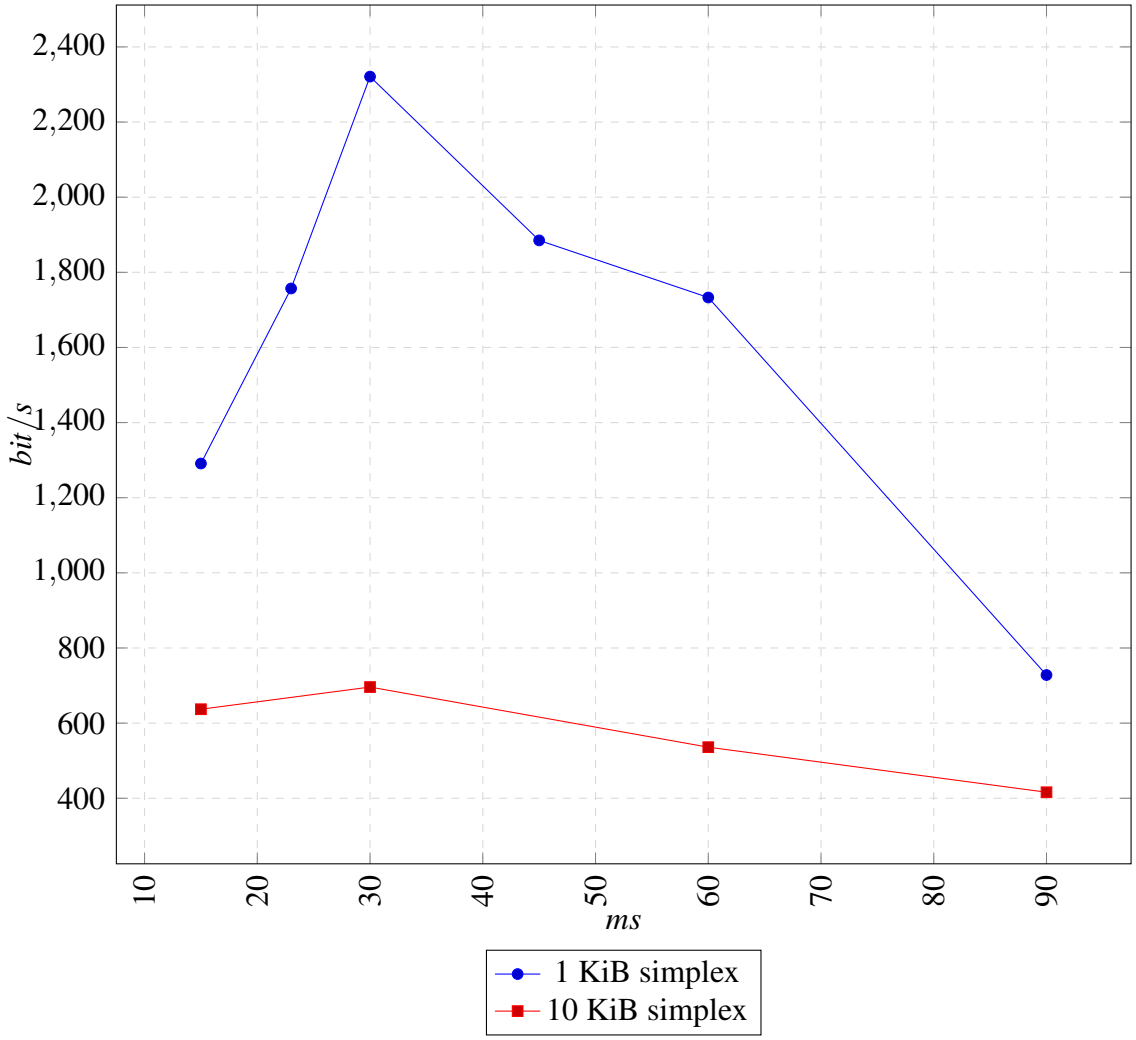
Webbkamera: *Z-Star Microelectronics Corp. Luxya WC-1200 USB 2.0 Webcam* med 640x480 resolution.

Telefon: *Nexus 5*.

6.4 Iakttagelser om simplexöverföring

Mätresultat

Simplexöverföringen är relativt snabb om man lyckas få tag i alla delar direkt i första passet. Största problemet är att man ofta blir utan en sista del, och då måste man vänta på att sändaren skickar delen som fattas på nytt. Detta kan, beroende på konfigurationen, ta en relativt lång tid. Ofta får man tag i ca. 80-90% av delarna på första passet, resterande 9-18% på andra passet och sista få delarna först efter flera pass. I bästa fall får man tag i alla delar direkt, hittills snabbaste uppmätta tiden för 1 KiB är 1545 ms med webbkamera, eller 5302 bit/s. I vanliga fall är hastigheten ca. 2322 bit/s, vilket framgår i Tabell 1.



Figur 19. Linjediagram över simplexöverföring av en 1 KiB fil och en 10 KiB fil med webbkamera. Delsträngslängd 150 tecken

Tabell 1. Resultat för simplexöverföring av en 1 KiB fil med webbkamera. Delsträngslängd 150 tecken.

Basintervall i ms	Genomsnittstid i ms	Hastighet i byte/s	bit/s
15	6344	161	1291
23	4660	219	1757
30	3528	290	2321
45	4344	235	1885
60	4726	216	1733
90	11247	91	728

Tabell 2. Resultat för simplexöverföring av en 10 KiB fil med webbkamera. Delsträngslängd 150 tecken.

Basintervall i ms	Genomsnittstid i ms	Hastighet i byte/s	bit/s
15	128437	79	637
30	117635	87	696
60	152723	67	536
90	196558	52	416

Ett inofficiellt rekord för överföringshastighet är 13.3 kbit/s där en 1 KiB fil överfördes på 602 ms från bildskärm till telefon. Liknande resultat var svåra att återskapa och det gick inte att få pålitliga resultat; vissa överföringar tog över 10 sekunder. Resultatet tyder på att högre överföringshastigheter är möjliga men högre stabilitet krävs av metoden.

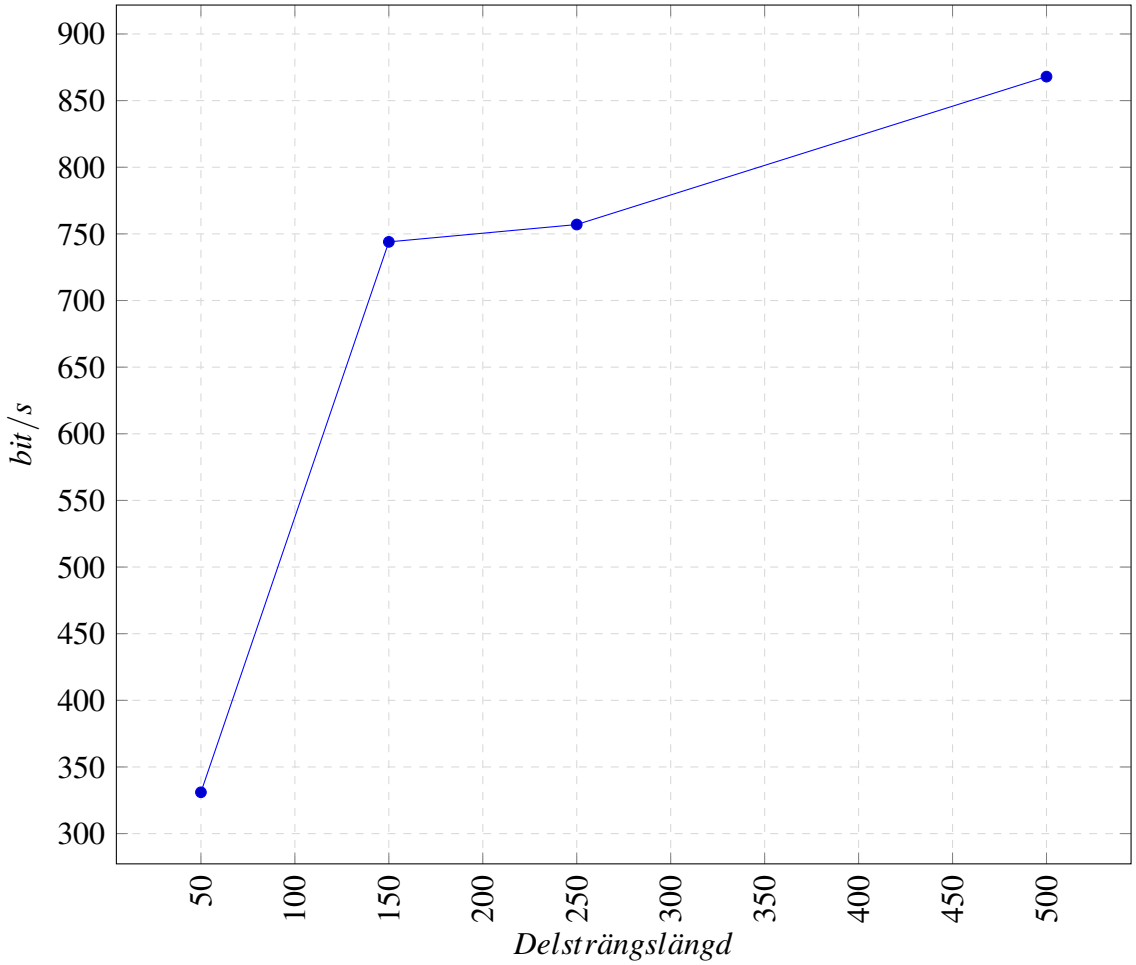
Övriga observationer

Det finns en betydlig skillnad i att låta kameran ligga stilla och i att hålla i den. Det verkar som att hålla i kameran ger bättre resultat. Detta beror antagligen på att den underliggande metoden för detektering av innehåll i QR-koden är deterministisk. Att röra på kameran hjälper emellertid inte alltid och det verkar som om vissa delar förblir svåra att läsa. Detta beror eventuellt på att data i tidigare delen leder till en QR-kod där stora områden är helt mörka. Exponeringskompensation i kameran kanske orsakar att följande kod blir oläslig. Det verkar inte finnas motsvarande problem med exponeringskompensation i duplexöverföring, eftersom en del som inte går att läsa läses på nytt tills läsningen lyckas.

Sändnings- och läsningsintervallet har mycket större betydelse än jag först hade förväntat mig. Eftersom ett pass tar kortare tid, tar det kortare tid före man på nytt har en möjlighet att överföra den sista delen, som ofta först mottas efter flera pass.

6.5 Iakttagelser om duplexöverföring

Duplexöverföring är mindre effektivt än förväntat. Duplexöverföringen är inte lika frustrerande som simplexöverföring kan vara, eftersom en *vanlig* förloppsindikator kan användas, som redan i början av överföringen vet hur många delar den skall ta emot.



Figur 20. Linjediagram över duplexöverföring av en 10 KiB fil från PC med webbkamera till telefon. 30 ms basintervall.

Jag utgår ifrån antagandet att endast ändring av delsträngslängden är nödvändig eftersom bilden byts ut direkt. Det finns inget behov att vänta på samma sätt som i simplexöverföring. Jag antar att en ändring av frekvensen påverkar hastigheten linjärt och därför inte är viktig för mätningen.

Resultatet visar att i duplexöverföring innebär mera data per QR-kod en högre överföringshastighet. Överföring blir omöjlig efter att en viss maximal storlek uppnås, som beror

på kamerans kvalitet.

Del	Genomsnittstid i ms	Hastighet i byte/s	bit/s
500	94295	108	868
250	108199	94	757
150	110053	93	744
50	247067	41	331

Tabell 3. Resultat för duplexöverföring av en 10 KiB fil från PC med webbkamera till telefon. 30 ms basintervall.

7 SLUTSATSER

DS.js kan utvecklas vidare. Flera funktioner bör ännu läggas till biblioteket före det är färdigt för att användas. Överföring från bildskärm till kamera är möjligt.

Vidareutveckling av biblioteket

Datakomprimering skulle antagligen öka överföringshastigheten. Det är inte nödvändigtvis bibliotekets uppgift att komprimera data, men det verkar som något som i varje fall kommer att implementeras.

En extra dimension i form av olika färger i enlighet med Langlotz & Bimber (2007) tidigare arbete skulle kunna öka överföringshastigheten. Detta skulle i teorin fungera utan att behöva vara specifikt någon enskild metod med att använda färgfilter.

Delaren och sammanfogaren kan förbättras så att header-delsträngen också skulle ha data i sig, så att om mängden data som skickas är tillräckligt liten skulle man inte behöva tolka mera än en kod. En tidigare och mindre komplicerad version av DS.js gjorde detta men hade nackdelen att varje delsträng innehöll hela strängens längd.

Metoden för duplexöverföring är tills vidare ineffektiv. En kombination av två kontinuerliga simplex kanaler från vardera apparaten på det sätt som diskuterades i kap. 4.3.3 skulle möjligtvis leda till högre kapacitet.

Med hjälp av en läsare av QR-koder eller andra tvådimensionella streckkoder som klarar

av structured append mode, som tas upp i kap. 2.1.4, skulle man kunna överföra godtyckliga mängder data med en metaserie. Mätningsresultaten anger att överföring av 1 kB är snabbare än överföring av 10 kB, vilket antagligen beror på fenomenet som beskrivs i kap. 6.4.

Tabell 4. Exempel på nuvarande metod som delaren använder för att dela data

Del	Data
1	ab
2	cd
3	ef
4	gh

Det största problemet med den nuvarande simplex metoden är att ca. 90% av alla delar tas emot redan i första passet och för de sista delarna krävs flera pass. Man skulle antagligen kunna förbättra simplexöverföringen betydligt om man inte skulle behöva alla delar med en gång. Man skulle kunna splittra data så att varje del innehåller hälften av data från den förra delen. I stället för att dela upp data i delar så som i Tabell 4 skulle man skapa överflödig data så som i Tabell 5.

Tabell 5. Förslag för ny metod att dela data

Del	Data
1	ab
2	bc
3	cd
4	de
5	ef
6	fg
7	gh
8	ha

Publicering av källkod

DS.js källkod är publicerad på Github. Som programvarulicens valdes MIT-licensen. (Gräsbeck, 2015)

Teknikens framtid

Dataöverföring från en bildskärm till en kamera är möjligt. Denna överföringsmetod kommer knappast att ersätta traditionella överföringsmetoder, men kan vara nyttig om hårdvara för traditionell överföring inte fungerar eller om traditionella metoder inte är tillgängliga eller behagliga.

KÄLLOR

Bell, Tim; Morgan, Jack & et.al. 2015, University of Canterbury. Tillgänglig: http://www.csfieldguide.org.nz/DataRepresentation.html, Hämtad: 6.5.2015.

Brown, Brian. 2000, *Part 14: Simplex, Half-Duplex and Full Duplex*. Tillgänglig: http://uva.ulb.ac.be/cit_courseware/datacomm/dc_014.htm, Hämtad: 11.4.2015.

Donald, James. 2003, *Improved Portability of Shared Libraries*, Princeton University, http://web.archive.org/web/20070926130800/ http://www.princeton.edu/~jdonald/research/shared_libraries/cs518_report.pdf , Hämtad: 27.4.2015.

ECMA. 2011, *Standard ECMA-262. ECMAScript Language Specification*, 5.1 uppl.. Tillgänglig: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, Hämtad: 10.2.2015.

Galindo, Enrique. 1999, *Barcodes - Cracking Barcodes*, Area 10 Mathematics and Technology Professional Development Center. Tillgänglig: http://www.indiana.edu/~atmat/units/barcodes/bar_r1.htm, Hämtad: 15.5.2015.

Gräsbeck, Max. 2015, *maxg0/displaysocket.js*. Tillgänglig: https://github.com/maxg0/displaysocket.js, Hämtad: 18.5.15.

Hamer, Mick. 2005, *3D barcodes to identify stolen valuables*. Tillgänglig: http://www.newscientist.com/article/dn7756-3d-barcodes-to-identify-stolen-valuables.html, Hämtad: 11.4.2015.

ISO/IEC. 2000, *18004*. Tillgänglig: http://www.swisseduc.ch/informatik/theoretische_informatik/qr_codes/docs/qr_standard.pdf, Hämtad: 8.5.2015.

Israel, David & Perry, John. 1990, *What is Information?* Tillgänglig: http://www.albany.edu/acc/courses/inf723spring2008/whatisinfo.pdf, Hämtad: 6.5.2015.

Johnson, Brad. 2012, *Wireless File-Transmission: Using QR Code Stream (Week 10 Presentation)*. Tillgänglig: https://www.youtube.com/watch?v=1JggOlwM_ps, Hämtad: 11.4.2015.

Johnson, Brad. 2014, *Jumping the Gap – Data Transmission Over An Air Gap*. Tillgänglig: http://volumelabs.net/jumping-the-gap-data-transmission-over-an-air-gap/, Hämtad: 4.2.2015.

jQuery. 2015, jQuery Foundation. Tillgänglig: https://github.com/jquery/jquery, Hämtad: 26.4.2015.

Kwok, Kevin. 2015, *antimatter15/ocrad.js*, GitHub. Tillgänglig: https://github.com/antimatter15/ocrad.js, Hämtad: 15.5.2015.

Langlotz, Tobias & Bimber, Oliver. 2007, *Unsynchronized 4D Barcodes*. Tillgänglig: http://www.jku.at/cg/content/e60566/e155460/e156518/U4DB.pdf, Hämtad: 12.5.2015.

Larsson, Eddie. 2015, *EddieLa/JOB*, GitHub. Tillgänglig: https://github.com/EddieLa/JOB, Hämtad: 15.5.2015.

Laszlo, Lazar. 2011, *LazarSoft/jsqrcode*, Github. Tillgänglig: https://github.com/LazarSoft/jsqrcode/, Hämtad: 13.4.2015.

manuels. 2015, *manuels/unix-toolbox.js-exact-image*, GitHub. Tillgänglig: manuels/unix-toolbox.js-exact-imag://github.com/manuels/unix-toolbox.js-exact-image/, Hämtad: 15.5.2015.

Microsoft. 2015, *About High Capacity Color Barcode Technology*, Microsoft. Tillgänglig: http://research.microsoft.com/en-us/projects/hccb/about.aspx, Hämtad: 6.5.2015.

Nicholas, Stephen. 2011, *QuickeR: Using video QR codes to transfer data*. Tillgänglig: http://stephendnicholas.com/archives/310, Hämtad: 4.2.2015.

Oberhofer, Christoph. 2015, *serratus/quaggaJS*, GitHub. Tillgänglig: https://github.com/serratus/quaggaJS, Hämtad: 15.5.2015.

Sangmin, Shim. 2014, *davidshimjs/qrcodejs*, Github. Tillgänglig: https://github.com/davidshimjs/qrcodejs, Hämtad: 12.5.2015.

W3C. 2011, *The WebSocket API*, W3C. Tillgänglig: http://www.w3.org/TR/2011/WD-websockets-20110929/, Hämtad: 28.4.2015.

W3C. 2012, *High Resolution Time*, W3C. Tillgänglig: http://www.w3.org/TR/hr-time/, Hämtad: 11.05.2015.

W3C. 2015, *WebRTC 1.0: Real-time Communication Between Browsers*. Tillgänglig: http://www.w3.org/TR/webrtc/, Hämtad: 13.5.2015.