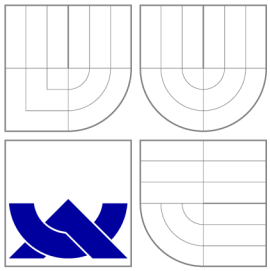
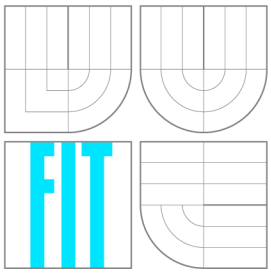


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PŘENOS A ZOBRAZENÍ VIDEA V PROHLÍŽEČI PRO ZÁZNAM PRACOVNÍ PLOCHY

PROCESSING AND STREAMING VIDEO IN BROWSER FOR SCREEN RECORDING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. LUKÁŠ SVAČINA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAL ŠPANĚL, Ph.D.

BRNO 2016

Abstrakt

Cílem práce je navrhnout a realizovat unikátní službu kombinující nejnovější technologie na poli webových aplikací. Služba umožní záznam pracovní plochy s následnou P2P distribucí mezi sledujícími uživateli a to jen za pomocí webového prohlížeče. Základem práce je analýza dostupných moderních a připravovaných technologií, které dovolují prakticky realizovat takovou službu. To zahrnuje získání přístupu k pracovní ploše, její záznam, zpracování získaných dat a jejich přenos sítí s následným přehráním pozorovateli. Videa jsou upravena pro přehrávání po částech s možností posuvu. Pro zajištění škálovatelnosti řešení je implementována distribuovaná architektura výměny částí záznamu mezi uzly formou hybridní P2P VoD sítě. Přehrávač tak umožňuje získání potřebných částí od aktuálně sledujících uživatelů a to včetně plynulého přehrávání videa během probíhajících přenosů s možností libovolného přesunu ve videu. Vzhledem k velmi rané či experimentální podpoře některých použitých API je vždy zpracován patřičný rozbor dostupné podpory prohlížečů.

Abstract

Aim of the thesis is to design and implement unique service based on the newest technologies in web apps field which allows screen recording followed by P2P distribution between participating users using web browser only. Thesis deals with the analysis of modern and coming technologies which allow practical implementation of such a service. It involves obtaining access to the screen source data, its recording, transforming and transmission over the network followed by playing on the other side. Recorded videos are adapted for part by part use in a player with seeking capability. Distributed architecture for data exchange between peers using peer-to-peer connection based on hybrid P2P VoD network provides scalability of the solution. The player allows obtaining the necessary parts of the videos from the current watchers with smooth video playback experience during ongoing transmissions whilst allowing arbitrary video shifting. In cosideration of early stages of development or experimental support for some of the APIs needed for this work, research into browsers' support with discussion on realistic applicability nowadays is always performed.

Klíčová slova

sdílení plochy, záznam plochy, Video on Demand, P2P, JavaScript, getUserMedia(), WebM, MPEG4, asm.js, Node.js, WebRTC, RTCPeerConnection, RTCDataChannel, Media Source Extensions, HTTP Streaming

Keywords

screen sharing, screen recording, Video on Demand, P2P, JavaScript, getUserMedia(), WebM, MPEG4, asm.js, Node.js, WebRTC, RTCPeerConnection, RTCDataChannel, Media Source Extensions, HTTP Streaming

Citace

Lukáš Svačina: Přenos a zobrazení videa v prohlížeči
pro záznam pracovní plochy, diplomová práce, Brno, FIT VUT v Brně, 2016

Přenos a zobrazení videa v prohlížeči pro záznam pracovní plochy

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Michala Španěla, Ph.D. a uvedl jsem veškeré literární prameny a publikace, ze kterých jsem čerpal informace.

.....
Lukáš Svačina
27. července 2016

Poděkování

Rád bych poděkoval panu Ing. Michalu Španělovi, Ph.D. za nespočet rad při tvorbě této práce a za trpělivé vedení, který mi poskytl.

Obsah

1 Úvod **3**

2 Technologie pro přenos a záznam videa **5**
2.1 World Wide Web konsorcium 5
2.2 JavaScript 6
2.3 Záznam videa 10
2.4 Zpracování videa 20
2.5 Přenos videa 22
2.6 Přehrávání videa 32

3 Návrh řešení pro záznam, přenos a přehrávání videa **38**
3.1 Případy užití 38
3.2 Záznam videa 39
3.3 Zpracování videa 42
3.4 Přenos videa 44
3.5 Přehrávání videa 45
3.6 Souhrn 47

4 Implementace **49**
4.1 Infrastruktura 49
4.2 Záznam videa 51
4.3 Zpracování videa 55
4.4 Distribuce videa 60
4.5 Přehrávání videa 71
4.6 Poskytování videa 74
4.7 Video-on-Demand služby 76
4.8 Spojení uzlů 79

5 Výsledky **81**
5.1 Volba datového toku 81
5.2 Srovnání hashovacích funkcí 82
5.3 Vektor stahovaných částí 83
5.4 Počet stahovacích slotů 86
5.5 Video on Demand služby 89
5.6 Kvalita přenosového média 90
5.7 Flash crowd 92
5.8 Shrnutí experimentů 94

6 Závěr **96**

A Navazující rozšíření **100**
A.1 Záznam videa 100
A.2 Zpracování videa 100
A.3 Přenos videa 100
A.4 Přehrávání videa 101
A.5 Testování 101

B Znázornění přenosů **102**

C Instalace **104**
C.1 Prerekvizity 104
C.2 Virtual hosts 104
C.3 Příprava aplikace 105
C.4 Spuštění aplikace 105

Kapitola 1

Úvod

V posledních letech lze pozorovat trend přesunu čím dál více nativních aplikací do prostředí webových služeb fungujících na principu SaaS[1] a orientaci na cloudový přístup. Toto je umožněno především rychlým vývojem v oblasti webových standardů a jejich implementacemi výrobci webových prohlížečů.

Aplikace webového prohlížeče tak již nabízí vývojářům mnohem více než pouhé vykreslení statické či dynamické webové stránky. Je možné přistupovat k lokálním souborům uživatele, pracovat s jeho lokací, zobrazovat push notifikace, přenášet video či zvuk z kamery v reálném čase a mnoho dalšího.

Webové prohlížeče se tak stávají centrální aplikací osobních počítačů s tendencí nahradit aplikace nativní. Příkladem budiž webová implementace kancelářského balíku Office.[2]

Stále však existují aplikace, které jsou dominantou nativních implementací nebo doplňků prohlížečů.[3] Jedním takovým typem jsou aplikace pro vzdálené sdílení obrazovky či ovládání vzdálené plochy.

Jedno z nových API, která se postupně teprve začínají implementovat do prohlížečů, umožňuje přístup právě k obrazu pracovní plochy hostitelského systému. Tato data lze ve webové aplikaci získat, zpracovat a případně kamkoliv odeslat.

Za opravdu revoluční lze považovat API umožňující peer-to-peer komunikace mezi prohlížeči bez asistence webového serveru. Toto umožňuje nové cesty rozložení zátěže při streamingu videa, přímý přenos dat mezi více prohlížeči nebo třeba realizaci P2P CDN.[4]

Vhodným využitím těchto moderních možností prohlížečů bylo možné vytvořit unikátní webovou službu umožňující pohodlné sdílení a nahrávání videí pracovních ploch uživatelů bez nutnosti instalovat nativní aplikace. Vysoké nároky na centrální server poskytující nahraná videa byly výrazně omezeny implementací hybridní P2P sítě, kdy si jednotlivé uzly právě sledující video vyměňují jeho části přímo mezi sebou.

Pro takovou distribuci však bylo nejen nutné navrhnout P2P síť pracující v rámci webových prohlížečů, ale sestavit celou infrastrukturu takové služby včetně úpravy nahraných videí pro následnou P2P distribuci. Aby bylo možné video plynule přehrávat během probíhajících přenosů, bylo nutné navrhnout funkční VoD[5] rozšíření P2P sítě. Ta tedy musí reagovat

[1] Software as a Service – hostování aplikace provozovatelem služby, kdy zákazníci k aplikaci přistupují přes Internet.

[2] Ať již v podání firmy Microsoft (Office Live) nebo Google (Google Docs).

[3] Mezi nejrozšířenější patří Adobe Flash či Java Applet. Oba jsou však v pokročilé fázi ústupu ve prospěch standardu HTML5.

[4] Content Distribution Network – globálně distribuovaná síť proxy serverů v mnoha datových centrech.

[5] Video on Demand je služba umožňující uživateli volbu co a kdy chce sledovat. Zároveň je běžné pozastavení sledování či libovolný posuv v čase videa.

na požadavky uživatelů pro posouvání ve sledovaném videu a prioritně vyměňovat díly potřebné pro okamžité sledování videa.

Aplikace výrazně šetří přenosové pásmo serveru a umožňuje rychlejší stažení videí současným stahováním od více uživatelů.

Rozsáhlé možnosti využití takové aplikace vystihují motivaci této diplomové práce. Od jednoduchého poskytování vzdálené podpory pomocí nahrávání video návodů přes pořádání webcastů[6] až po záznam videí z her. Současně však také kladou nemalé požadavky na reálný provoz podobné služby a nutnost orientace práce i na škálovatelnost řešení.

Tato diplomová práce plynule navazuje na vypracovaný semestrální projekt. Kapitola Technologie pro přenos a záznam videa a Návrh řešení pro záznam, přenos a přehrávání videa tak s menšími úpravami vycházejí právě z něj. Následuje kapitola Implementace, která provází čtenáře implementačními detaily v podobném sledu jako kapitoly předchozí. Kap. Výsledky popisuje uskutečněné experimenty, jejich závěry a možnosti případného zlepšení. Poslední kap. Závěr pak shrnuje celou práci z hlediska jejich přínosů a možných navazujících prací.

[6] Kombinace seminářů a webového prostředí za účelem online vzdělání či předávání informací.

Kapitola 2

Technologie pro přenos a záznam videa

V této kapitole je uveden popis technologií umožňující realizaci aplikace. První podkapitola JavaScript se věnuje hlavnímu jazyku, na kterém je většina použitých technologií založena. Jedná se o moderní jazyk `JavaScript` v jeho aktuální verzi `ECMAScript 6`. Následuje rozbor metod záznamu videa v části Záznam videa a popis jeho zpracování v Zpracování videa. Dále je uvedena problematika přenosu získaného videa v sekci Přenos videa a také metody jeho přehrávání v poslední kapitole Přehrávání videa.

2.1 World Wide Web konsorcium

`W3C` konsorcium je hlavní mezinárodní organizací spravující standardy pro **World Wide Web**. Kromě samotných členů konsorcia zasahuje do vývoje standardu také veřejnost.

Vzhledem k nutnosti orientace v dokumentech `W3C` je vhodné zmínit jejich fáze (*Maturity Levels*) v souladu s `W3C` specifikací:[1]

- **Working Draft** je pracovní návrh publikován `W3C` pro posouzení komunitou včetně veřejnosti.
- **Candidate Recommendation** je dokument, který splňuje technické požadavky pracovní skupiny a již obdržel posouzení komunitou. Očekává se, že dokument bude přijat jako konečné doporučení. Dokument je určen k implementaci a v připomínkovém řízení se tak objevují implementační záležitosti.
- **Proposed Recommendation** je schválený dokument `W3C` vedoucí osobou zajišťující dostatečnou kvalitu pro to se stát finálním `W3C` doporučením.
- **W3C Recommendation** je specifikace nebo sada směrnic či požadavků schválená členy `W3C` včetně vedoucí osoby. `W3C` doporučuje široké rozšíření takového dokumentu jako standardu pro web. Později mohou být vydány navazující revize či celé nové edice vydaných doporučení.

Pracovní skupiny mohou vydávat tzv. „Editor's drafts“, které nemají zařazení mezi oficiální fáze a **nemusí** tak být nutně dále zpracovány a schváleny skupinou `W3C`. Často se spolu s výsledkem vedených diskuzí jedná o předchůdce *Working Draft*.

[1] https://www.w3.org/2015/Process-20150901/

2.2 JavaScript

JavaScript je multiplatformní netypový interpretovaný objektově orientovaný jazyk. Objektově orientované programování v tomto jazyce je založeno na prototypové dědičnosti.

Pracuje s objektovým modelem dokumentu (DOM) a jeho modifikací dovoluje překonat původně statickou povahu webových aplikací. Tento jazyk tedy zajistil rozšíření dynamických webových služeb a postupem času se zasloužil o standardizaci a vzájemně kompatibilní implementace napříč různými webovými prohlížeči.

V současné době webové prohlížeče umožňují prostřednictvím tohoto jazyka využívat různá API[2] standardu HTML5, díky kterým lze vytvářet bohaté multimediální aplikace. Některá taková API jsou základními stavebními prvky této práce.

Využití jazyka

Interpretované skripty v tomto jazyce jsou vkládány přímo do webových stránek (případně jako připojitelný externí soubor) a jsou vykonávány na straně klienta ve webovém prohlížeči.

Současně lze však taktéž použít jazyk JavaScript mimo prostředí webu. Lze pomocí něj programovat například programy pro běhové prostředí NodeJS, čímž se jazyk rozšířil z původně klientské strany i na stranu serverovou.

Mezi další příklady využití lze zařadit skriptování v NoSQL databázovém systému MongoDB nebo použití ve známém grafickém prostředí GNOME 3.

ECMAScript 6

Specifikace jazyka ECMAScript 6 nebo také ECMAScript 2015 vydána v **červnu 2015** [7] významně rozšiřuje možnosti tohoto jazyka. Přidává podporu pro lepší udržitelnost rozsáhlých projektů díky podpoře pro moduly a definici tříd obdobně jako v jiných objektově založených jazycích.

Ukázka kódu 2.1: Ukázka použití třídy

```
1  class Car {
2    constructor(color, fuel) {
3      this.color = color;
4      this.fuel = fuel;
5      this.pos = [0, 0];
6    }
7    move(x, y) {
8      this.pos = [x, y];
9    }
10 }
11
12 var c = new Car("red", "diesel");
13 c.move(10, 20);
```

JavaScript je prototypově orientovaný jazyk, který v předchozí specifikaci jazyka syntakticky pracoval pouze s objekty a používání přístupů známých z třídně založených jazyků bylo neprůhledné a komplikované (viz kniha [20], kapitola 3). Kód 2.1 ukazuje novou zjednodušenou syntaxi definice a použití třídy. Syntaxe dovoluje taktéž používat dědičnost jako v jiných třídně založených jazycích. Současně lze stále využívat prototypových vlastností jazyka.

[2] V textu bude často použita zkratka API, která značí rozhraní pro programování aplikací, tedy sbírku funkcí či tříd, které může programátor při tvorbě aplikací využívat.

Ukázka kódu 2.2: Ukázka typovaných polí

```
this.buffer    = new ArrayBuffer(24);
this.id        = new Uint32Array (this.buffer,  0,  1);
this.username  = new Uint8Array  (this.buffer,  4, 16);
this.amount    = new Float32Array(this.buffer, 20,  1);
```

Nově lze využívat typovaných polí, což umožňuje skriptovacím jádrům (například `Google V8`) vykonávat nízkoúrovňové optimalizace a výrazně tak zvyšovat rychlost aplikací (viz knihovna `asm.js` 2.4).

Nativní je nyní také práce s binárními daty či podpora návrhového vzoru `Promise` na úrovni samotného jazyka (viz kapitola 3 knihy [15]). Ten je cestou k odstranění častého neduhu asynchronního programování v tomto jazyce, kdy dochází k mnohaúrovňovému zanořování `callback` funkcí.

Ukázka kódu 2.3: Využití řetězení Promise objektu

```
getJSON("/api/article/1").then(function(article) {
  return getJSON(article.commentAPI);
}).then(function(comments) {
  console.log(comments);
}).catch(function(error) { /
  console.log("First or second request has failed");
});
```

Specifikace `ECMAScript 6` jazyk výrazně rozšiřuje a zavádí do něj mnoho nových konstrukcí. Toto však klade značné nároky na vývojáře webových prohlížečů, aby specifikaci do prohlížečů rychle implementovali. Nejnovější verze dvou nejrozšířenějších prohlížečů[3] již notnou část specifikace implementují. Stále však chybí podpora pro některé důležité části a především podpora starších prohlížečů.

Babel

`Babel` se řadí do rodiny kompilátorů nad jazykem `JavaScript`. Slouží k transformaci kódu dle nejnovějších specifikací (nyní `ES6`) do kódu dle specifikace starší (typicky `ES5`). Tím je umožněno programátorovi využívat ještě neimplementované části nových specifikací jazyka.

Ačkoliv některé prohlížeče již samotný `ES6` z velké části podporují, pro některé zásadní funkce je nutné `Babel` (či obdobný kompilátor) použít. Momentálně se jedná především o podporu modulů a vyhodnocování jejich závislostí. Žádný ze současných prohlížečů toto nemá implementováno.

Ukázka kódu 2.4: Transformovaný kód 2.1 pomocí Babel

```
"use strict";

var _createClass = function () { function defineProperties(target, props) {
    for (var i~= 0; i~< props.length; i++) { var descriptor = props[i];
    descriptor.enumerable = descriptor.enumerable || false; descriptor.
    configurable = true; if ("value" in descriptor) descriptor.writable =
    true; Object.defineProperty(target, descriptor.key, descriptor); } }
    return function (Constructor, protoProps, staticProps) { if (protoProps)
    defineProperties(Constructor.prototype, protoProps); if (staticProps)
    defineProperties(Constructor, staticProps); return Constructor; }; }();

```

[3] V době psaní práce se jedná o Chrome 50 a Firefox 46.

```
5  function _classCallCheck(instance, Constructor) { if (!(instance instanceof
       Constructor)) { throw new TypeError("Cannot call a~class as a~function");
        } }
6
7  var Car = function () {
8    function Car(color, fuel) {
9      _classCallCheck(this, Car);
10
11     this.color = color;
12     this.fuel = fuel;
13     this.pos = [0, 0];
14   }
15
16   _createClass(Car, [{
17     key: "move",
18     value: function move(x, y) {
19       this.pos = [x, y];
20     }
21   }]);
22
23   return Car;
24 }();
25
26 var c = new Car("red", "diesel");
27 c.move(10, 20);
```

WebIDL

`JavaScript` pracuje s `HTML` skrz `DOM`. To je hierarchická struktura obsahující veškeré elementy webové stránky včetně jejich atributů a přístupových funkcí. `DOM` tedy umožňuje jazyku `JavaScript` přistupovat a manipulovat s HTML objekty. Pro specifikaci rozhraní těchto objektů se používá jazyk `WebIDL`[4] a určuje tak vlastnosti a metody, které `HTML` objekt zpřístupňuje ve skriptech.

Vývojáři webových prohlížečů implementují jednotlivá `HTML5 API` v souladu s `WebIDL` specifikacemi, což poskytuje programátorům přehled v rozhraních přístupných z jazyka `JavaScript`.

Je nutné rozlišovat dva typy atributů:

Kontextové atributy jsou definovány jako atributy jednotlivých `HTML` elementů v souladu se specifikací `DTD`.[5] Lze je jednoduše nastavit uvedením přímo v `HTML` kódu dané značky (viz kód 2.5). Kromě toho je možné kontextové atributy číst a zapisovat v rámci skriptů, protože jsou implicitně mapovány na `IDL` atributy (viz kód 2.6). Tyto mapované atributy se také nazývají *reflektované kontextové atributy* (viz kniha [16], kapitola 3).

Ukázka kódu 2.5: Kontextové atributy video elementu.

```
1 <video src="video/Cars.webm" autoplay preload="auto"></video>
```

Ukázka kódu 2.6: Reflektované kontextové atributy video elementu.

```
1 interface HTMLMediaElement : HTMLElement {
```

[4] Web interface definition language, viz https://www.w3.org/TR/WebIDL/.

[5] `Document Type Definition` definuje popis struktury `SGML` dokumentů: například validní názvy elementů či hodnoty jejich atributů.

```
2     attribute DOMString src;
3     attribute DOMString crossOrigin;
4     attribute DOMString preload;
5     attribute boolean autoplay;
6     attribute boolean loop;
7     attribute boolean controls;
8     attribute boolean defaultMuted;
9   };
10  interface HTMLVideoElement : HTMLMediaElement {
11    attribute unsigned long width;
12    attribute unsigned long height;
13    attribute DOMString poster;
14  };
```

IDL atributy na rozdíl od atributů kontextových nemohou být nastaveny v `HTML` kódu (jako kód 2.5), ale lze k nim přistupovat pouze v rámci skriptů. Často jsou určeny pouze ke čtení a reprezentují aktuální stav objektu (viz kód 2.7). Avšak například atribut `currentTime` lze i editovat a tím posouvat aktuální pozici ve videu.

Ukázka kódu 2.7: Některé IDL atributy video elementu.

```
1   interface HTMLMediaElement : HTMLElement {
2     // error state
3     readonly attribute MediaError error;
4
5     // network state
6     readonly attribute DOMString currentSrc;
7     readonly attribute unsigned short networkState;
8     ...
9
10    // ready state
11    readonly attribute unsigned short readyState;
12    readonly attribute boolean seeking;
13    ...
14
15    // playback state
16    attribute double currentTime;
17  };
18  interface HTMLVideoElement : HTMLMediaElement {
19    readonly attribute unsigned long videoWidth;
20    readonly attribute unsigned long videoHeight;
21  };
```

`WebIDL` kromě definice atributů (ať již `IDL` nebo reflektovaných kontextových atributů) uvádí rovněž deklarace metod nad danými objekty, které lze v prostředí skriptu využívat. V případě `video` elementu by se tak jednalo o metody `load()`, `play()` nebo například `canPlayType()` pro detekci implementovaných kodeků.

Při využívání nových či experimentálních `HTML5 API` je `WebIDL` popis v rámci `HTML5 API` specifikace **často jedinou dostupnou dokumentací**. Nutno podotknout, že taková dokumentace slouží především vývojářům prohlížečů a konkrétní implementace i samotná rozhraní se mohou lišit.

2.3 Záznam videa

Získání video streamu v sandboxu webového prohlížeče bývalo možné pouze pomocí využití rozšíření třetích stran. Adobe Flash umožňuje získání přístupu k uživatelově webové kameře, ne však k streamu obsahu pracovní plochy. Obdobně jsou na tom Java applety, ty však podporu pro záznam pracovní plochy přidávají.

Prohlížeče, a to nejen mobilní, upouštějí od podpory technologie Flash[6] stejně jako technologie Java appletů.[7]

Současně prohlížeče rychlým tempem implementují různá API standardu HTML5, která nahrazují nutnost používání těchto zmíněných rozšíření. V současnosti zde však vzniká propast, protože mnoho funkcionality zavržených rozšíření není dostatečně podporováno webovými prohlížeči nebo je podporují pouze v experimentálních vývojových větvích.[8]

HTML Media Capture

HTML Media Capture je prvním pokusem o standardizaci záznamu médií v rámci prohlížeče. Sémanticky se jedná o rozšíření HTML značky pro výběr souborů o nové hodnoty atributu accept (viz [14]). Tím lze pro daný input vyžádat spuštění rozhraní prohlížeče pro záznam videa či audia. Tento záznam je pak odeslán klasicky formulářem jako soubor.

Ukázka kódu 2.8: HTML Media Capture přístup k médiu.

```
1 <input type="file" accept="video/*;capture=camcorder">
2 <input type="file" accept="audio/*;capture=microphone">
```

Tento přístup podporují veškeré moderní mobilní platformy, avšak žádné platformy desktopové. Možné využití je tedy pouze pro jednoduché vyvolání rozhraní pro záznam videa či pořízení fotografie na mobilním zařízení (viz obr. 2.1). Nelze takto zachytávat pracovní plochu ani prostředí mobilního zařízení.

Formát záznamu se liší dle mobilní platformy. Apple iOS 9 vytváří **mov** kontejner, kdežto Google Android 6 vytváří kontejner **mp4**. Obě platformy pak v kontejnerech používají video kodek H264 - MPEG-4 AVC s audio kodekem MPEG AAC Audio.

MediaStream API

Získat MediaStream[9] lze v moderních prohlížečích pomocí volání API getUserMedia(), které umožňuje specifikovat vlastnosti požadovaného streamu a koncovému uživateli zobrazit uživatelské rozhraní s volbou vhodného zařízení.

Při volání getUserMedia() prohlížeč zobrazí typicky modální dialog, který informuje uživatele o důsledcích povolení přístupu k danému zařízení. Některé prohlížeče taky po celou dobu povoleného přístupu aplikace k streamu zobrazují kontrolní panel always-on-top (viz obr. 2.2).

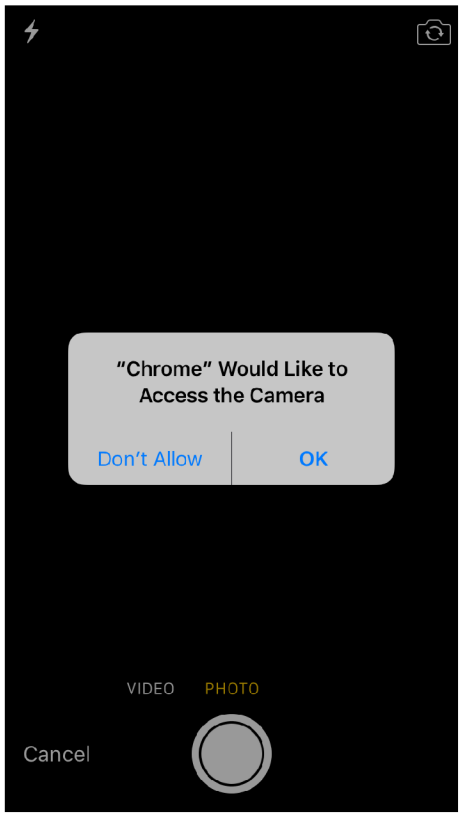
Tato metoda byla později formalizována do širší skupiny různých API pro zachytávání, nahrávání a streamování dat pod názvem MediaStream API (viz kniha [13], kapitola

[6] Google Chrome 42 ve výchozím nastavení zobrazí Flash plugin pouze na vyžádání. To je například pro internetovou reklamu nepřijatelné.

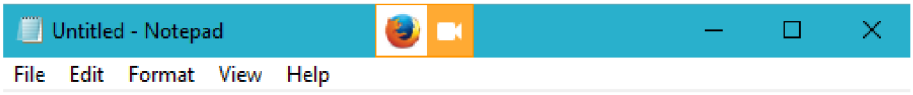
[7] Google Chrome 45 ve výchozím nastavení nepodporuje NPAPI rozhraní nutné pro běh Java appletů.

[8] Například podpora pro souběžný záznam videa i audia v případě prohlížeče Chrome nebo podpora pro hardwarově akcelerované přehrávání video kodeků.

[9] Objekt reprezentující proud dat z fyzických či virtuálních zařízení mající povahu tyto data poskytovat: webkamery, mikrofony, ale například i proud dat zachytávající pracovní plochu stanice. Aby nedošlo k nedorozumění, pojem media stream není dále překládán.



Obrázek 2.1: Vyvolání rozhraní fotoaparátu na iOS 9.3



Obrázek 2.2: Informování uživatele o aktivním přístupu k webové kameře.

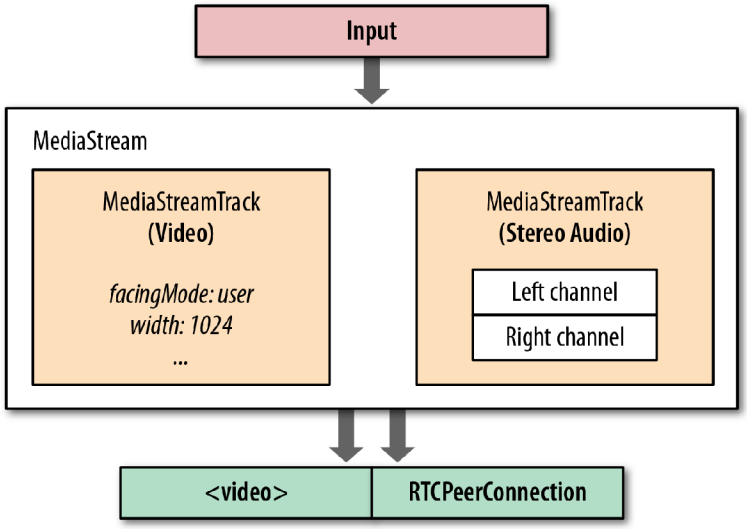
2). V současné době je jediným a hlavním vstupním bodem pro získání nízkoúrovňového přístupu k fyzickým či virtuálním vstupním zařízením v rámci skriptů interpretovaných webovými prohlížeči.

Samotná definice metody `getUserMedia()` byla v rámci `W3C` specifikace změněna.[10] Toto způsobilo nekompatibilní dvě rozhraní, z nichž starší používá jmenný prostor `Navigator UserMedia` a novější `MediaDevices`. Starší rozhraní dále využívá `callback` funkcí předávaných pomocí parametrů metody, kdežto novější již plně využívá moderního návrhového vzoru `Promise` (viz kód 2.3).

`MediaStream` může dále obsahovat více instancí `MediaStreamTrack` pro každý zachycený stream (viz obr. 2.3). Lze tak pomocí `MediaStream API` získat **synchronizovanou** audio a video stopu v rámci jednoho `MediaStream` objektu (viz kniha [5], kapitola 18).

Zjednodušeně je tedy `getUserMedia()` jednoduché `API` pro získání streamů (audio, video, obrazovka, ...) z platformy pod běžícím prohlížečem. Médium je automaticky optimalizováno, kódováno a zpětně dekódováno pomocí `WebRTC` audio a video komponent.

[10] Viz sekce *10.2 MediaDevices Interface Extensions* dokumentu *W3C Editor's Draft 22 February 2016* http://w3c.github.io/mediacapture-main/archives/20160222/getusermedia.html



Obrázek 2.3: MediaStream – jeho obsah a další využití. Zdroj: [5], kapitola 18.

Stream constraints

Získávaný `MediaStream` lze detailně specifikovat pomocí tzv. constraints. Omezení jsou definovány pomocí `MediaStreamConstraints` obsahující jednotlivé `MediaTrackConstraints` – pro video a pro audio. Ty již definují požadované vlastnosti streamu konkrétní stopy. `Constraints` lze aplikovat voláním metody `applyConstraints` rozhraní `MediaStreamTrack` nebo specifikací `MediaStreamConstraints` při volání `getUserMedia()` (viz část 2.3).

V kódu 2.9 je uveden výtah `WebIDL` definic ze specifikace rozhraní `MediaStream`.[11] Znázorňuje tak zmiňované závislosti a ukázkový výčet konkrétních constraints. Sekvence `advanced` slouží k specifikaci množiny constraints, ze které se vybere první bezpodmínečně vyhovující položka. Pokud taková neexistuje, hledá se mimo sekvenci `advanced`.

Ukázka kódu 2.9: WebIDL specifikace a závislosti constraints.

```
partial interface MediaDevices {
  Promise<MediaStream>  getUserMedia (MediaStreamConstraints constraints);
};
dictionary MediaStreamConstraints {
             (boolean or MediaTrackConstraints) video = false;
             (boolean or MediaTrackConstraints) audio = false;
};
dictionary MediaTrackConstraints : MediaTrackConstraintSet {
             sequence<MediaTrackConstraintSet> advanced;
};
dictionary MediaTrackConstraintSet {
             ConstrainLong      width;
             ConstrainLong      height;
             ConstrainDouble    aspectRatio;
             ConstrainDouble    frameRate;
             ConstrainDOMString facingMode;
        ...
};
```

[11] V době psaní práce se jednalo o nejnovější `W3C Editor's Draft` z 22. února 2016. Zdroj: http://w3c.github.io/mediacapture-main/archives/20160222/getusermedia.html

Praktický příklad využití constraints pro získání video streamu s požadovanými vlastnostmi je uveden v kódu 2.10. Opět však závisí na jednotlivých prohlížečích, které z uvedených constraints implementují a zohlední.

Ukázka kódu 2.10: Praktické využití constraints.

```
navigator.mediaDevices.getUserMedia({video: {
  width: {min: 640, ideal: 1280, max: 1920},
  height: {min: 480, ideal: 720, max: 1080},
  frameRate: 30,
  facingMode: "user"
}});
```

Sdílení plochy

Konsorcium `W3C` pracuje na standardu `Screen Capture`, jehož cílem je přivést sdílení vykreslených webových stránek a celých obrazovek do prostředí webu. Standard i implementace již prošla několika koly, což způsobilo značné nekonzistence mezi specifikací a její implementací webovými prohlížeči.

Aktuálně veřejně publikovaná verze standardu[12] zavádí novou metodu `getOutputMedia()` po boku metody `getUserMedia()`. Příklad užití je uveden v kódu 2.11. Constraints zde dovolují specifikovat typ zařízení `OutputCaptureSurfaceType` následovně:

- **monitor**: fyzický displej nebo kolekci displejů,
- **window**: obsah okna displaye nebo okno konkrétní aplikace,
- **application**: obsah okna konkrétní aplikace nebo celá kolekce takových oken,
- **browser**: obsah okna prohlížeče.

Ukázka kódu 2.11: Metoda getOutputMedia() dle W3C WD 10. února 2015.

```
navigator.mediaDevices.getOutputMedia({ video: true })
  .then(stream => {
    // we have a~stream, attach it to a~feedback video element
    videoElement.srcObject = stream;
  }, error => {
    console.log("Unable to acquire screen capture", error);
  });
```

Nejnovější verze standardu[13] mění název hlavní metody na `getDisplayMedia()` a taky se více zaměřuje na bezpečnost a důsledky celé funkcionality.

Žádný ze současných prohlížečů zatím neimplementuje sdílení plochy tak, jak definují uvedené standardy. Dále jsou samostatně popsány přístupy ke sdílení obrazovky dvou nejrozšířenějších webových prohlížečů, které sdílení plochy, ačkoliv ne podle standardu, podporují.

[12] `W3C First Public Working Draft` z 10. února 2015. Zdroj: http://www.w3.org/TR/2015/WD-screen-capture-20150210/

[13] W3C Editor's Draft z 15. ledna 2016. Zdroj: http://w3c.github.io/mediacapture-screen-share/

Google Chrome

V dřívějších verzích `Google Chrome` (verze 26–34) experimentoval s podporou pro sdílení obrazovky rozšířením sady constraints (viz část 2.3) o nové položky (viz kód 2.12) umožňující získat stream obrazovky. Tento nestandardní přístup byl dále podmíněn využitím zabezpečeného `HTTPS` protokolu a aktivací experimentální funkcionality v rozšířeném nastavení prohlížeče.[14]

Ukázka kódu 2.12: Google Chrome získání přístupu k záznamu obrazovky.

```
var constraints = {
    video: {
        mandatory: {
            chromeMediaSource: 'screen'
        }
    }
};
getUserMedia(constraints, callback);
```

Vzhledem k možnostem zneužití této experimentální funkcionality sociálním inženýrstvím byla možnost její aktivace s verzí `Chrome 36` odstraněna. Aktivaci lze nyní vykonat pouze nastavením parametru `--enable-usermedia-screen-capturing` při spuštění aplikace.[15]

Od verze 34 je oficiálně doporučováno používat `chooseDesktopMedia API` podporované v rámci rozšíření prohlížeče.

Mozilla Firefox

`Mozilla Firefox` stejně jako `Google Chrome` zatím nepodporuje `Screen Capture API` (viz část 2.3). Od verze `Firefox 33` lze však využívat nestandardní rozšíření `API` `getUserMedia()` k záznamu obrazovky obdobně jako v dřívějších verzích prohlížeče `Chrome` (viz část 2.3). Pomocí specifických hodnot constrains (viz kód 2.13) lze získat přístup k jednotlivým oknům (hodnota window), konkrétním aplikacím (hodnota application) nebo celým obrazovkám (hodnota screen). Opět je nutné komunikovat pomocí zabezpečeného protokolu.

Ukázka kódu 2.13: Mozilla Firefox získání přístupu k záznamu obrazovky.

```
var constraints = {
  video: {
    mozMediaSource: 'screen',
    mediaSource: 'screen'
  }
};
getUserMedia(constraints, callback);
```

Stejně jako `Chrome`, také `Firefox` zatím z bezpečnostních důvodů neumožňuje používat toto `API` přímo. U `Chrome` byla nutná instalace rozšíření nebo spouštění prohlížeče se speciálním parametrem. `Firefox` vyžaduje po uživateli vykonat následující kroky:

[14] Na adrese `chrome://flags/#enable-usermedia-screen-capture` položka *Enable screen capture support*.

[15] Oznámení a vysvětlení v diskuzi k issue #347641: https://bugs.chromium.org/p/chromium/issues/detail?id=347641.

- v `about:config` povolit `media.getusermedia.screensharing.enabled`,
- přidat doménové jméno webové aplikaci do seznamu `media.getusermedia.screensharing.allowed_domains`

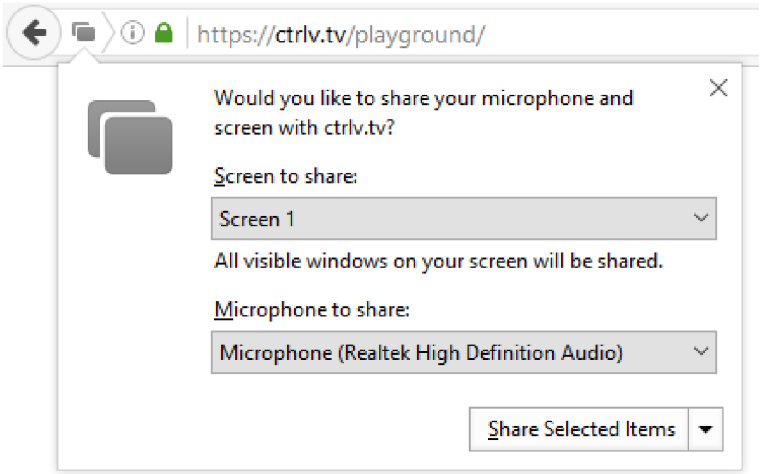
Nebo lze také vytvořit rozšíření, který tyto dva kroky vykoná automaticky.

Sdílení audia

Záznam obrazovky kombinovaný se záznamem audio stopy lze využít pro kompletní nahrávání videonávodů či screencastů. V definici constraints lze položkou audio definovat požadavek na prohlížeč na dodání video stopy doplněné o audio stopu.

V současné době prohlížeč `Google Chrome 49` takové constrains neumí splnit a vrátí chybový stav. Naopak `Mozila Firefox 45` zobrazí v uživatelském rozhraní volby obrazovky ke sdílení taky volbu audio vstupu, který do výsledného kontejneru přiloží a sesynchronizuje (viz obr. 2.4).

V `Chrome` je tedy pro přidání záznamu audia nutné vyžádat od prohlížeče oddělený audio stream, nahrávat ho po celou dobu záznamu obrazovky a nakonec obě části ručně sesynchronizovat a spojit. Toho lze docílit odesláním ke zpracování serveru nebo experimentálním multiplexingem obou streamů na straně klientské (viz část 2.4).



Obrázek 2.4: Mozilla Firefox – vyvolání UI dialogu výběru obrazovky/okna ke sdílení včetně audio vstupu.

Vzhledem k ranému vývoji v oblasti popisované touto kapitolou bylo velmi komplikované najít jakékoliv relevantní zdroje. Často jsem tak musel experimentovat s nedokumentovanými vlastnostmi prohlížečů, pročítat diskuzní skupiny vývojářů a zkoušet, které ukázky kódu, z těch pár dostupných, ještě nejsou zastaralé a fungují. Nebyla nalezena žádná knižní literatura s informacemi o této oblasti. Tuto část teoretické kapitoly tedy považuji za užitečný a přehledný zdroj pro nadcházející zájemce z této oblasti.

Záznam plochy

Jak získat datový proud bylo popsáno v kapitole 2.3. Pro využití tohoto proudu jinak než k *realtime* přehrání je nutné získat jeho záznam. To s sebou nese nutnost proud multimediálních dat vhodně zkomprimovat použitým kodekem a zakódovat do některého z multimediálních kontejnerů.

Pro záznam videa v prostředí webu stále vládne technologie `Adobe Flash`. Postupně však, stejně jako v případě přehrávání videa, tato technologie ustupuje do pozadí na úkor moderních `HTML5 API` umožňujících postupně tuto technologii zcela nahradit.

Whammy

Základním přístupem k záznamu proudu dat pomocí `HTML5` je získávání jednotlivých snímků z proudu dat v čase a jejich vykreslení do elementu `<canvas>`, který je určen pro práci s grafikou pomocí `HTML5 Canvas API`. Jedna z metod, které `API` nabízí, je získání vykresleného snímku v požadovaném formátu. Během záznamu tak stačí dostatečně rychle překreslovat tento element a získávat jednotlivé snímky. Ty lze pak buď jednotlivě odesílat na serveru nebo je lze spojit do výsledného videa na klientské straně a odeslat dále ke zpracování již hotové video.

Takový enkodér je implementován v knihovně `Whammy.js`, kde z jednotlivých `WebP` snímků získaných nativně z pomocí `HTML5 Canvas API` generuje `WebM` video. Více k jednotlivým formátům je uvedeno v části 2.6.

Ukázka kódu 2.14: Duplikace proudu dat do elementu `<canvas>`.

```
1        var video = document.getElementById("video");
2        var canvas = document.getElementById("canvas");
3        video.srcObject = stream;
4        video.addEventListener("play", paintFrame, false);
5        function paintFrame() {
6            canvas.getContext("2d").drawImage(video, 0, 0, 200, 200);
7            requestAnimationFrame(paintFrame);
8        }
```

V kódu 2.14 lze vidět ukázku překreslení jednotlivých snímků z videostreamu do elementu `<canvas>`. První překreslení spustí obsluha události `play` a dále se funkce rekurzivně volá s využitím volání `requestAnimationFrame`. To zajistí překreslení tak, aby nedocházelo k narušení uživatelského dojmu z plynulého přehrávání a současně aby nebyly zbytečně vytěžovány prostředky hostitelského systému (což se dříve typicky dělo pomocí volání `setTimeout()` s velmi nízkým či nulovým intervalem).

Tento způsob optimalizovaného vykreslování voláním `requestAnimationFrame` elementu `HTMLCanvasElement` však není vhodný pro záznam videa, kde je kladen důraz na kvalitu a stabilní zpoždění mezi jednotlivými snímky. Zmíněná metoda navíc například v případě přesunu prohlížeče do pozadí, z optimalizačních důvodů, snímkovací frekvenci snižuje či úplně pozastavuje.[16]

Pro další zpracování jednotlivých snímků lze využít metody `toDataURL()` elementu `HTMLCanvasElement`, který obsah plátna `canvas` zakóduje do zvoleného formátu a převede do formátu `Base64`. **Tento převod s sebou nese výkonnostní režii a především nárůst přenášených dat** (přibližně o 33 %).[17]

[16] Více informací o této funkci je uvedeno v HTML standardu pracovní skupiny `WHATWG`. Zdroj: https://html.spec.whatwg.org/multipage/webappapis.html#animation-frames

[17] Více v knize [9].

Od verze Mozilla Firefox 25 a aktuálně taky od nejnovější verze prohlížeče Google Chrome 50 je již podporována metoda toBlob() stejného elementu HTMLCanvasElement, která vytváří objekt Blob reprezentující získaný obraz plátna v binární podobě a tedy **bez zmiňované režie**.[18]

Media Recorder API

Mezi další z nových JavaScript API založených na raných fázích specifikací patři Media Recorder API (dříve MediaStream Recording API). API definuje nový MediaRecorder objekt umožňující záznam připojeného proudu dat získaného existující funkcí getUserMedia (viz část 2.3).

Prohlížeč poskytující toto API se nativně postará o záznam, komprimaci a zakódování do multimediálního kontejneru. Výsledkem je tak opět binární objekt Blob, který lze dále poskytnout ke stažení či odeslat na server k dalšímu zpracování.

Pro zpětně kompatibilní využití objektů Blob prohlížeče implementují v API metodu volání createObjectURL(Blob), která pro daný binární zdroj vygenerují dočasnou (po dobu existence dokumentu či do odvolání pomocí revokeObjectURL(Blob)) **speciální URL adresu**, kterou lze použít kdekoliv, kde lze z principu URL adresy používat (zdroj odkazů, obrázků, videí, stylů apod.)[19]

Ačkoliv rozpracovaný standard hovoří o možnosti volby kódování včetně dotazu API na podporované formáty, v současnosti je podpora pouze viz tabulka 2.1.

V Chrome zatím není možné zvolit požadovaný bitrate videa a do verze 49 ani nahrávat audio.[20] Není přítomna žádná podpora pro nahrávání ve formátu MPEG4 (viz sekce 2.6). Firefox implementuje nastavení mimeType i bitsPerSeconds jako podmínky předávané objektu MediaRecorder(stream, options). Objekt options může definovat:

- **mimeType** volba určující kontejner do kterého chceme záznam exportovat (zatím jen video/webm),
- **audioBitsPerSecond** zvolený bitrate pro audio složku nahrávaného média (výchozí je adaptivní bitrate),
- **videoBitsPerSecond** zvolený bitrate pro video složku nahrávaného média (výchozí je 2,5 Mbps),
- **bitsPerSecond** slouží pro společné nastavení dvou předchozích vlastností (při souběžné definici zastupuje chybějící složku).

Schéma 2.5 znázorňuje proces záznamu streamu s možností následného odeslání na serverovou stranu (pomocí XMLHttpRequest verze 2, která přidává podporu pro binární přenosy) nebo vygenerování unikátního dočasného URL.

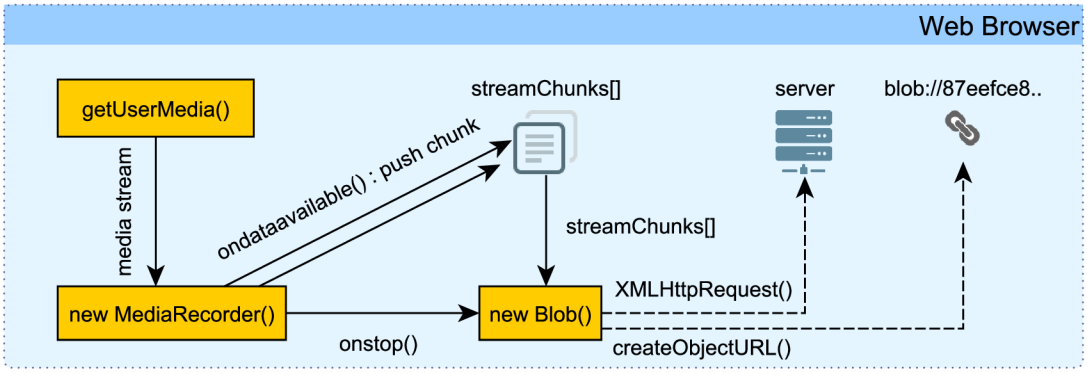
[18] Popis metody viz specifikace HTML5. Zdroj: https://www.w3.org/TR/html5/scripting-1.html#dom-canvas-toblob

[19] Metody jsou definovány v nedokončené specifikaci File API. Zdroj: https://w3c.github.io/FileAPI/#dfn-createObjectURL

[20] Viz WebRTC diskuze https://groups.google.com/forum/?#!msg/discuss-webrtc/n11m846oV4I/0b3ycjmjCAAJ

	Chrome 47	Chrome 49	Firefox 30+
Kontejner	webm	webm	webm
Video kodek	VP8	VP8/VP9	VP8
Audio kodek	**žádný**	Opus 48 kHz	Vorbis 44.1 kHz

Tabulka 2.1: Podporované formáty pro záznam.



Obrázek 2.5: Záznam získaného proudu dat pomocí MediaRecorder API a následné zpracování.

Bezpečnostní důsledky

Umožnit webovému prohlížeči odesílat snímky obrazovky uživatele vede k mnoha otázkám z hlediska bezpečnosti. Tyto otázky se objevují postupně a jsou analyzovány a zapracovávány v rámci procesu vývoje `W3C` standardu `Screen Capture`.

Informovanost uživatele Je nutné, aby `UI` prohlížeče jasně informovalo uživatele o tom, že dané webové službě uděluje patřičná oprávnění a to včetně informace o trvanlivosti takového rozhodnutí (viz část 2.3). Současně je potřeba po celou dobu přístupu k zařízení (kamera, obrazovka, ...) uživatele o informovat (nespoléhat pouze např. na rozsvícení diody webové kamery).[21]

Integrita dat Méně zřejmý důsledek spočívá v možnosti úniku informací postranním kanálem (obraz), se kterým standardní bezpečnostní ochrany prohlížečů nepočítaly.

Ochrana `CSRF`[22] spočívá ve vynucení autorizace požadavku jednorázově vygenerovaným klíčem, který je odeslán prohlížeči ve zdrojovém kódu stránky (skrytý prvek formuláře, část url, ...). Tento klíč je současně uložen na serveru pro následnou autorizaci před vykonáním požadavku. Po použití je vhodné klíč pro další akci přegenerovat. Proti úniku klíče například z vnořeného rámce (element `iframe`) se standardně uplatní **same-origin policy**[23] a prohlížeč nedovolí žádná data z `iframe` přečíst.

Útočník však může skrytě (za naprosto legitimní aplikací) v rámci své webové stránky

[21] Vychází z kapitoly 6, W3C Editor's Draft z 15. ledna 2016. Zdroj: http://w3c.github.io/mediacapture-screen-share/

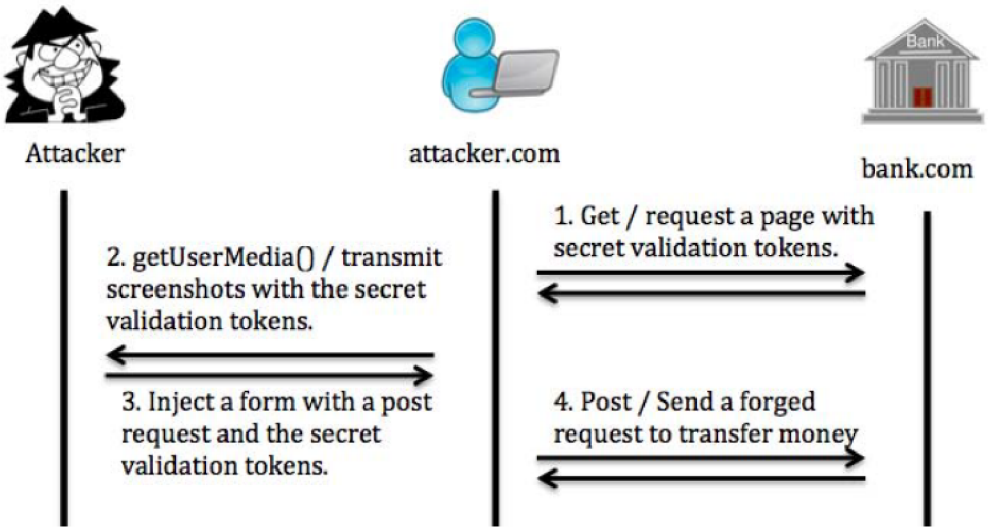
[22] Cross Site Request Forgery – útok na bázi problému zmateného zástupce, kdy útočník donutí prohlížeč odeslat autorizované požadavky díky autentizaci pomocí cookies.

[23] Povolení přistupovat pouze k datům mající stejný původ. Původ je definován jako kombinace URI schématu, hostname a portu. Viz https://www.w3.org/Security/wiki/Same_Origin_Policy

zobrazovat tento `iframe` a důvěryhodné informace (vygenerovaný klíč) tak postranním kanálem získat. K tomu lze využít například pseudoprotokol `view-source`,[24] který by v `iframe` zobrazil přímo vygenerovaný klíč pro ochranu před `CSRF`. Ze zachyceného obrazu či videa lze pak klíč například pomocí `OCR` automatizovaně zpracovat a ochranu tak prolomit. Nutno podotknout, že **únik informací obrazem může být pro oko oběti nepostřehnutelný**. A pokud k odhalení přece dojde, stejně je již pozdě, protože informace unikla.

Příklad útoku je znázorněn ilustrací 2.6:

1. Útočníkova stránka `attacker.com` načte zabezpečenou stránku `bank.com` pomocí `CSRF` ochrany (tajným klíčem). V modelovém případě by útočníkova stránka legitimně nabízela službu sdílení plochy.
2. Snímek obrazovky se zachyceným klíčem (např. pomocí zobrazení zdrojového kódu) je odeslán útočníkovi díky `Screen Capture API`.
3. Útočník připraví formulář se zlomyslnou akcí doplněný o zachycený klíč.
4. Tento formulář je odeslán z `attacker.com` během sdílení obrazovky. Stránka `bank.com` tento požadavek považuje za autorizovaný a zpracuje jej.



Obrázek 2.6: Útok na CSRF ochranu. Zdroj: [24].

Trvanlivá autorizace Jednou udělená trvanlivá oprávnění pro zvolenou třetí stranu by neměla dovolit automatické odeslání audio/video streamů. Uživatel může důvěřovat doméně `https://example.com` pro kterou udělí trvanlivé oprávnění přístupu ke streamu. Útočník pak může donutit uživatele (sociální inženýrství, přesměrování, ...) otevřít adresu automaticky zahajující přenos, např. `https://example.com/?callto=attacker` . Vzhledem k ustavené důvěře bude automaticky odeslán důvěryhodný obsah (video z kamery, záznam obrazovky, ...) útočníkovi skrze důvěryhodný web.

[24] Google Chrome a Mozilla Firefox již nedovolují zobrazit zdrojový kód v iframe. Viz [6] a [2]. Stále však existují metody jak data odcizit. Například pomocí otevření `popup` okna, kde již `view-source` fungovat musí.

Ostatní hrozby Mezi další informace, které mohou být zneužity patří automatické doplňování formulářů, včetně kreditních karet, historie prohlížeče, jeho nastavení apod. S větším nadhledem již nelze ani při použití vrstvy `SSL` zajistit v prohlížeči důvěrnost dat, protože mohou být pomocí `Screen Capture API` odchytnuty v dešifrované podobě.

Další bezpečnostní důsledky `Screen Capture API` včetně návrhu jejich řešení jsou uvedeny v práci *Attacks Exploiting the HTML5 Screen Sharing API* [24].

2.4 Zpracování videa

Metody uvedené v předchozí kapitole popisují způsoby jak získat a zaznamenat video proud dat na straně prohlížeče. Jsou však založeny na datech ve formě, jakou poskytuje implementace prohlížeče.

Nelze tak zvolit formát ani například oblast výřezu videa. Volba formátu záznamu je důležitá z důvodu kompatibility při následném přehrávání videa (viz část 2.6). Dále jsou popsány metody zpracování získaného videa.

Server side

Tradiční přístup je odeslat získané video na server k dalšímu zpracování. Ten vykoná potřebné operace typu konverze videa do více formátů nebo jeho ořez.

Běžně používaný open source nástroj `ffmpeg` umožňuje veškeré konverze audio i videoformátů přítomných v prostředí webu (viz kniha [17], kapitola 1). Základní konverzi získaného videa ve formátu `webm` (s kodekem `VP8`) do formátu `mp4` (s kodekem `h264`) lze docílit příkazem:

```
ffmpeg -i input.webm -strict -2 output.mp4
```

Vzhledem k rozdílným kodekům použitých v obou kontejnerech je nutné překódování videa, což je výkonově náročná operace.

Veškeré tyto operace musí výkonově pokrýt server. Z hlediska provozu takové služby by mohlo být výhodnější vykonávat zpracování videa na straně klientské.

Client side

V kapitole 2.3 bylo nastíněno jak zpracovávat zachytávaný proud dat **snímek po snímku** kopií do elementu `canvas`. Nad tímto elementem lze volat `JavaScript API`:

```
HTMLCanvasElement.getContext(contextType, contextAttributes)
```

Toto `API` poskytuje bohaté možnosti grafických transformací, kreslení či exportu. Kontext plátna `canvas` může být typu `2D` nebo `WebGL`. `WebGL` kontext umožňuje vykonávat hardwarově akcelerované operace pomocí programovatelných shaderů grafických karet s využitím `OpenGL ES 3.0` (více viz práce [23]).

Ačkoliv lze metodou překreslování snímků ze streamu do elementu `canvas` video tedy prakticky jakkoliv modifikovat (kniha [17], kapitola 5), získat záznam takto upraveného videa je o poznání komplikovanější.

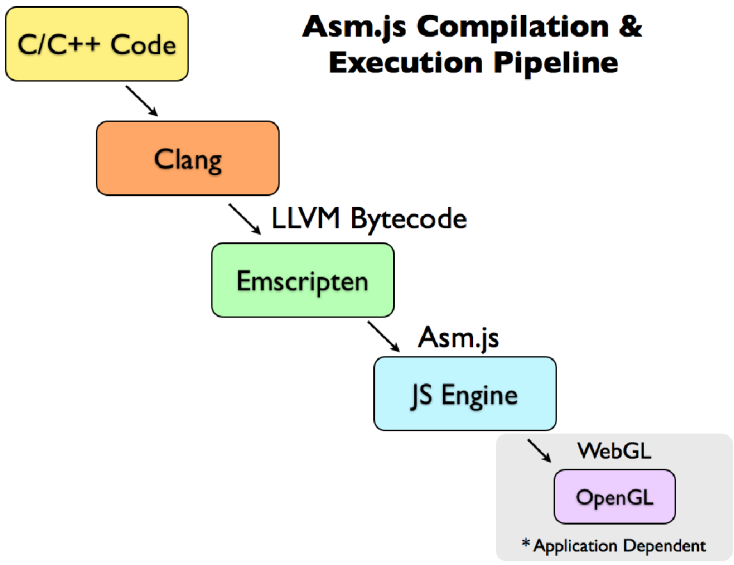
Jednou z variant je možnost získat export zpracovávaného videa technikou knihovny `Whammy.js` (viz část 2.3) snímek po snímku s omezeným výkonem. Druhou variantou je použití experimentálního `CanvasCaptureMediaStream API`, které má dle specifikace umožnit **získání vlastních** media streamů například i z elementu `canvas` voláním:[25]

```
HTMLCanvasElement.captureStream(optional double frameRate)
```

Takový stream lze dále nahrávat stejně jako je popsáno v sekci 2.3. Toto `API` je však podporované zatím **pouze v prohlížeči Firefox 41** a to s nutností modifikovat nastavení prohlížeče.[26]

asm.js

Kompromis mezi popsanými přístupy spočívá ve spouštění nativních serverových nástrojů na straně klientské v jazyce `JavaScript`. K tomuto účelu lze využít `source-to-source`[27] kompilátor `Emscripten`, který je určen pro kompilaci standardních spustitelných aplikací a knihoven v jazyce `C/C++`[28] do podmnožiny jazyka `JavaScript` jménem `asm.js` (viz schéma 2.7). Takto zkompilované aplikace lze poté spouštět přímo v prostředí webových prohlížečů.



Obrázek 2.7: Průběh kompilace nativních aplikací do JS. Zdroj: http://ejohn.org/blog/asmjs-javascript-compile-target/.

`Asm.js` je vysoce optimalizovaná nízkoúrovňová **podmnožina** jazyka `JavaScript`. Je zpětně kompatibilní a nevyžaduje pro běh nový ani upravený virtuální stroj. Ačkoliv lze v této podmnožině programovat i ručně, `asm.js` je primárně určen jako efektivní cílový jazyk pro kompilátory (kniha [3], kapitola 5). Díky omezené sadě konstrukcí, vynucené

[25] Viz rozpracovaná specifikace https://w3c.github.io/mediacapture-fromelement/#idl-def-CanvasCaptureMediaStream

[26] Ve výchozím nastavení deaktivováno. Pro aktivaci nutno nastavit direktivu `canvas.capturestream.enabled` na `true`.

[27] Tradiční kompilátory vykonávají překlad z jazyka s vyšší abstrakcí do jazyka s abstrakci nižší. Source-to-source kompilátory naopak překládají mezi jazyky s podobnou úrovní abstrakce.

[28] Obecně jakýkoliv jazyk, který lze zkompilovat do LLVM. Původně se jednalo pouze o `C/C++`, dnes již i mnoho dalších.

typovosti a optimalizované práci s pamětí (paměť je implementována jako typované pole) mohou tradiční just-in-time JS kompilátory vykonávat ahead-of-time kompilaci, což vede k výraznému zrychlení zpracování.[29]

Ukázky kódu 2.15 a 2.16 znázorňují převod z jazyka C++ do asm.js. První řádek JS kódu napovídá interpretu, že se jedná o asm.js kód a lze na něm vykonat patřičné optimalizace. Bitová operace OR s číslem 0 se využívá pro vynucení konverze operandů do 32b datového typu signed integer. Současně lze vidět využití zmíněné implementace paměti jako typového pole MEM8 a simulaci práce s ukazateli.

Ukázka kódu 2.15: Vstup (C++).

```
1  size_t strlen(char *ptr) {
2    char *curr = ptr;
3    while (*curr != 0) {
4      curr++;
5    }
6    return (curr ? ptr);
7  }
```

Ukázka kódu 2.16: Výstup (asm.js).

```
1  "use asm";
2  function strlen(ptr) {
3    ptr = ptr|0;
4    var curr = 0;
5    curr = ptr;
6    while (MEM8[curr]|0 != 0) {
7      curr = (curr + 1)|0;
8    }
9    return (curr ? ptr)|0;
10 }
```

Mezi prohlížeče podporující AOT kompilaci asm.js kódu patří v současné době pouze Mozilla Firefox 22 a vyšší. V některých případech se výkon jejich optimalizačního modulu OdinMonkey blíží **polovině** výkonu nativních aplikací před kompilací.[30] Současně lze však díky zmíněné zpětné kompatibilitě i bez AOT kompilace zkompilované programy spouštět jako klasický JavaScript ve většine prohlížečů. I tam lze díky optimalizacím na úrovni JS pozorovat zvýšení výkonu.[31]

Existuje mnoho programů, které fungují ve webovém prostředí právě díky asm.js. Patří mezi ně například SQLite, gnuplot, vim, Unreal Engine 4, Unity, Dosbox, Lua VM a další.

2.5 Přenos videa

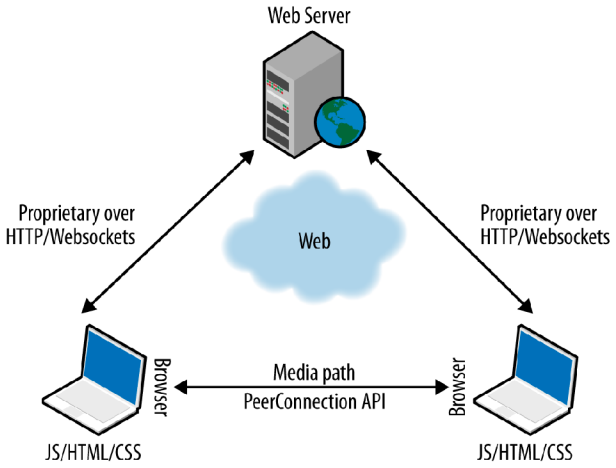
V prostředí webu existuje více technologií umožňující přenos textových i binárních dat (obr. 2.8). Běžně používaná je **client-server** architektura, na jejímž principu fungují technologie XHR (XMLHttpRequest) či WS (WebSocket). Pro přenos videa může být však vhodná taktéž architektura client-client neboli peer-to-peer. Až donedávna nebylo možné takový přenos v prostředí webu uskutečnit (alespoň ne bez použití rozšíření typu Flash apod.). Nyní již lze využít technologie stejnojmenné pracovní skupiny WebRTC[32] i pro přenos dat **mezi samotnými prohlížeči**.

[29] Specifikace asm.js je v době psaní práce v nedokončeném stádiu Working Draft. Zdroj: http://asmjs.org/spec/latest/

[30] Představení AOT kompilace viz https://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/

[31] Například Chrome 28 zvyšuje výkon asm.js ačkoliv přímo AOT kompilaci nepodporuje. Zdroj: http://blog.chromium.org/2013/05/chrome-28-beta-more-immersive-web.html

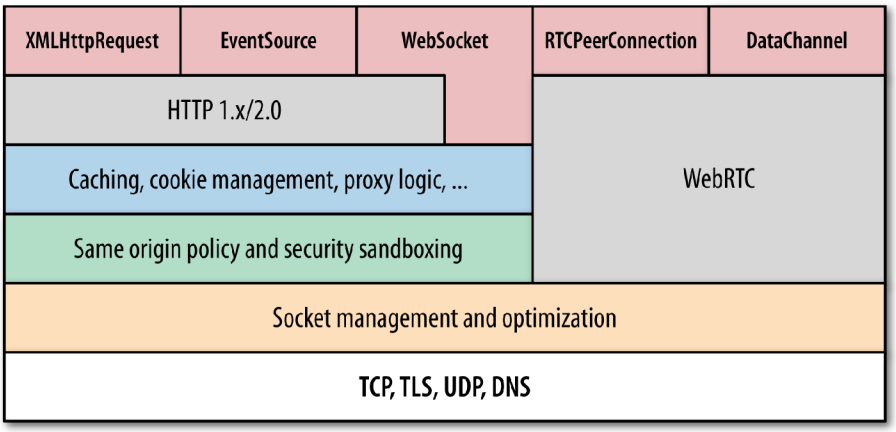
[32] Ve spolupráci se skupinou RTCWEB zajišťující specifikaci protokolů, datových formátů a ostatních požadavků peer-to-peer přenosů.



Obrázek 2.8: Možnosti přenosu videa v rámci webové aplikace. Zdroj [12].

Client-server architektura

Pro komunikaci se serverem se stále využívá protokol `HTTP`, nejčastěji ve verzi 1.1.[33] Mezi hlavní nevýhodu tohoto protokolu pro oboustrannou komunikaci se serverem patří absence plně duplexního režimu přenosu dat. Dlouhou dobu se tak využívaly různé techniky získávání dat ze serveru na principu pollingu či long-pollingu.[34] O úspěchu těchto technik svědčí například jejich dlouhodobé používání pro doručování notifikací v síti Facebook.

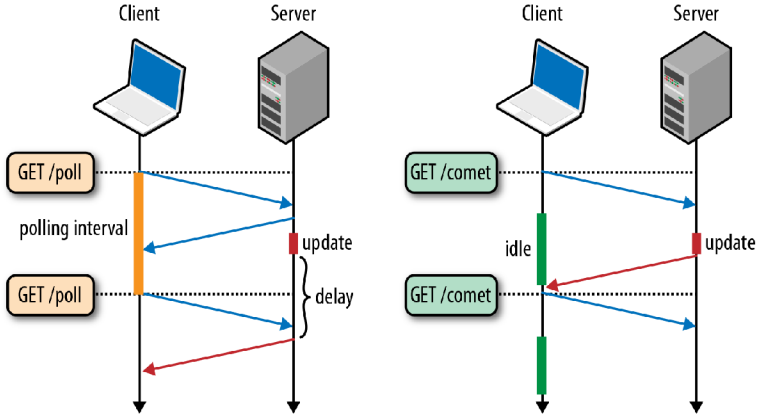


Obrázek 2.9: Vysokoúrovňové API, protokoly a služby webových prohlížečů. Zdroj [5].

XMLHttpRequest (XHR) je `API` prohlížečů umožňující síťový přenos dat pomocí jazyka `JavaScript`. Sestavuje se klasický `HTTP` požadavek s možností přenášet textové i binární data (prohlížeč zajistí automatické (de)kódování). Díky použití `JS` lze zpracovávat požadavky asynchronně bez potřeby znovunačtení webové stránky.

[33] Tato verze přinesla zvýšení výkonu díky persistentním keepalive HTTP spojením, kdy se eliminuje zbytečné vícenásobné otevírání a zavírání TCP spojení. Další snížení přenosové režie přináší nadcházející verze protokolu HTTP 2.

[34] První zmíněná spočívá v opakovaném dotazování serveru zdali má připravena nějaká data k doručení klientovi. Druhá minimalizuje zpoždění umělým prodlužováním otevřeného TCP spojení, dokud se data k odeslání neobjeví. Viz schéma 2.10.



Obrázek 2.10: Zpoždění techniky polling (vlevo) a long-polling (vpravo). Zdroj [5].

Jedná se o klíčovou technologii moderních webových aplikací využívající `AJAX`. **Bohužel však nepodporuje streamování dat, jelikož odesílaný zdroj dat musí být před samotným odesláním vždy kompletní.**

WebSocket protokol[35] umožňuje oboustrannou **plně duplexní** textovou i binární komunikaci nad `HTTP` protokolem použitím sdíleného `TCP` socketu. Díky `WS` již není třeba využívat techniky jako je `long-polling` nad `XHR`. Je však nutná podpora `WS` protokolu ze strany serveru.

Typicky se pro `WS` používá `NodeJS` server, ale podporu lze najít ve většině známých serverů formou modulů. Současně `WS` komunikuje také nad `TLS`, což zajišťuje důvěryhodnost komunikace. Podporovány jsou všechny současné prohlížeče.[36]

Specifikace definuje dvě nová `URI` schémata `wss` a `ws`. Pro připojení se tak používá řetězec formátu `ws://10.0.0.1:8181` (případně s prefixem `wss` pro `TLS` variantu). Inicializace WS komunikace (kódy 2.17 a 2.18) je založena na jednoduchém `HTTP` požadavku využívající `Connection: Upgrade` a další hlavičky **zavedené v HTTP/1.1**. Jejich úplný popis včetně vysvětlení procesu `handshake` `WS` protokolu je uveden v literatuře [11].

Ukázka kódu 2.17: HTTP WebSocket požadavek.

```
1 GET ws://localhost:8181/ HTTP/1.1
2 Origin: http://localhost:8181
3 Host: localhost:8181
4 Sec-WebSocket-Key: zy6Dy9mSAIM7GJZNf9rI1A==
5 Upgrade: websocket
6 Connection: Upgrade
7 Sec-WebSocket-Version: 13
```

Ukázka kódu 2.18: HTTP WebSocket odpověď.

```
1 HTTP/1.1 101 Switching Protocols
2 Connection: Upgrade
3 Sec-WebSocket-Accept: EDJa7WCAQQzMCYNJM42Syuo9SqQ=
4 Upgrade: websocket
```

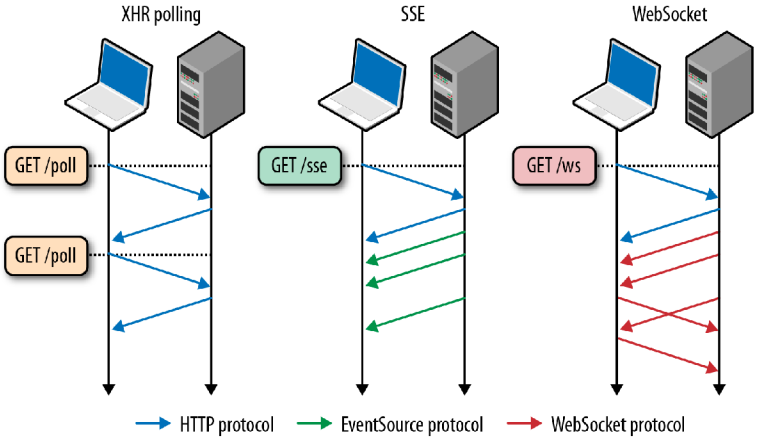
[35] Protokol standardizován `IETF` jako `RFC 6455`. `API` přístupné v prohlížečích standardizuje dále `W3C`.
[36] Viz http://caniuse.com/#search=websockets.

Inicializace komunikace probíhá pomocí `HTTP` protokolu na standardních portech 80 resp. 443 taky z důvodu jejich prostupností firewally po cestě. Mezilehlé prvky však nemusí rozumět `WS` protokolu a mohou způsobovat různá selhání komunikace (modifikace obsahu `HTTP` požadavků či odpovědí, buffering rámců, ...). Doporučuje se tedy před samotnou inicializací spojení ustavit `end-to-end` zabezpečený tunel, například pomocí použití `WSS` místo `WS`. **Tím se většina možných komplikací elegantně vyřeší, protože mezilehlé prvky nemohou do komunikace zasahovat** (viz strana 301 knihy [5]).

Po inicializaci lze komunikovat protokolem `WS`, který již samotný protokolu `HTTP` podobný není, na standardních portech 80 resp. 443 nebo na libovolných jiných. Opět však platí výhoda komunikace přes standardní porty kvůli firewallům a proxy serverům po cestě. Např. zmiňovaný `NodeJS` server umožňuje na stejném portu provozovat současnou obsluhu protokolů `HTTP` a `WS`.

Srovnání `XHR` je optimalizováno pro komunikaci typu požadavek-odpověď (obr. 2.11) pomocí úplných `HTTP` požadavků a odpovědí obsahující mnoho režijních dat (např. cookies) v každé zprávě. `XHR` tedy ani nepodporuje streamování dat, vždy se odesílá kompletní zásobník dat.

Na rozdíl od `WS` však `XHR` velmi **dobře podporuje** kódování, cachování nebo komprimaci přenášených dat (např. `gzip` pro textová data). V případě `WS` je na samotné aplikaci, aby implementovala veškeré tyto optimalizace a další funkcionalitu, kterou `XHR` automaticky dědí z klasické `HTTP` komunikace (např. autentizace, přesměrování, držení stavu pomocí cookies[37] apod.). `WS` však nabízí rychlou plně duplexní komunikaci bez často zbytečné režie a nutnosti pollingu.



Obrázek 2.11: Srovnání komunikace XHR, WS a pro úplnost taky ES protokolu. Zdroj [5].

Peer-to-peer architektura

`WebRTC` je kolekce standardů, protokolů a několika `JS API`, které dohromady umožňují `peer-to-peer` audio, video a data přenosy **mezi prohlížeči**. Prohlížeče tedy vystupují v `peer-to-peer` architektuře jako jednotlivé koncové uzly, které jsou spojeny mezi sebou bez účasti centrálního prvku (serveru). Centrální prvek se využívá pouze pro inicializaci ko-

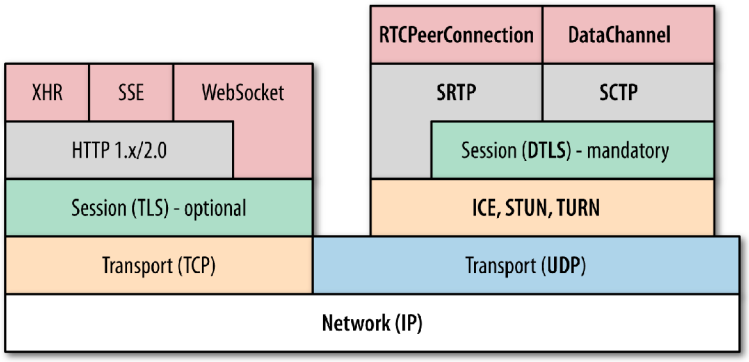
[37] V případě `WebSocket` komunikace se odesílají cookies pouze v rámci HTTP `handshake` operace a dále již ne.

munikace, dohody parametrů spojení (`signaling`) a jako rezervní (`TURN`) prvek poskytující data klientům, které se technicky nepodaří propojit přímo.[38]

Není nutné využívat jakýchkoliv doplňků třetích stran, vše potřebné je vývojářům webových systémů zpřístupněno pouze v několika `JS API`. Podporovány jsou prohlížeče `Firefox`, `Chrome` a `Opera`. Prohlížeče `Safari` a `IE` je nutné obsluhovat tradiční architekturou `client-server`.

Mezi hlavní dostupná `API` patří:

- `MediaStream`: zajišťuje získání audio/video streamů a je popsáno v části 2.3,
- `RTCPeerConnection`: zajišťuje spojení mezi uzly a umožňuje přenos audio/video dat (část 2.5),
- `RTCDataChannel`: umožňuje **přenos libovolných dat** aplikace (část 2.5).



Obrázek 2.12: `WebRTC` hiearchie protokolů. `DataChannel` je označení v jádře `Gecko`, jiné označení je `RTCDataChannel`. Zdroj [5].

Na schématu 2.12 lze vidět, že `client-server` technologie používají spolehlivou transportní vrstvu TCP. V případě `peer-to-peer` komunikace však `WebRTC` využívá **nespolehlivý přenos pomocí UDP**. `UDP` na rozdíl od `TCP` nezajišťuje úvodní handshake, negarantuje doručení, pořadí doručení paketů ani ochranu proti vícenásobnému doručení. Zajišťuje však datovou integritu paketu pomocí kontrolního součtu. **Režie UDP je minimální** a jedná se tedy o vhodný protokol pro přenos realtime audia/videa, kdy je vhodnější tolerovat výpadek paketů nežli čekat na jejich správné doručení.[39].

Pokud by bylo použito `TCP`, pokaždé když by se některý z paketů ztratil, `TCP` by začalo všechny následující ukládat do vyrovnávací paměti, čekat na znovu odeslání ztraceného paketu a až poté by doručilo pakety ve správném pořadí aplikaci. Tady vzniká zmíněná nevhodná latence. `UDP` tedy:

- nezaručuje doručení – žádné potvrzování paketů, znovu odesílání ani časovače,
- nezaručuje pořadí doručení – žádná sekvenční čísla paketů ani přeuspořádání,
- žádné informace o stavu – chybí oznámení o ustavení spojení i reprezentace stavu automaty,

[38] Typicky dva uzly kdy jsou oba za komplikovanou síťovou infrastrukturou, např. symetrickými `NATy`.

[39] Unikátní vlastnosti lidského mozku ukazují, že jsme velmi dobří ve vyplňování mezer, ale velmi citliví na detekovaná zpoždění ([5], strana 315)

- žádné řízení zahlcení – chybí data od klientů či sítě pro řízení zahlcení.

Dále je na ilustraci vidět potřeba implementace TURN, STUN a ICE pro ustavení a správu peer-to-peer komunikace nad UDP se schopností najít spojení mezi klienty i v neideálních síťových topologiích zahrnující proxy servery a NATy.

WebRTC dále vyžaduje šifrovaný přenos UDP paketů, což zajišťuje rozšíření UDP protokolem DTLS, který požadované zabezpečení zajišťuje díky tomu, že je založen na TLS.[40]

SRTP a SCTP jsou pak aplikační protokoly pro multiplexing streamů, zajištění řízení zahlcení, spolehlivého přenosu a dalších služeb nad UDP. Lze tedy pozorovat, že se WebRTC snaží **zajistit služby TCP protokolu nad protokolem UDP** pomocí kombinace různých jiných protokolů na UDP založených.

Multimediální komunikace v reálném čase vyžaduje mnohem více než přenos datagramů po síti. Je nutné adaptivně uzpůsobovat bitrate přenášeného streamu, vykonávat synchronizaci zpoždění, vylepšovat kvalitu snímků a mnoho dalšího. O všechny tyto operace se WebRTC stará a přináší tak kompletní multimediální podporu na platformu webu. Vzhledem k zaměření práce, které nespočívá v *realtime* komunikaci, jsou podrobnosti těchto částí WebRTC mimo rozsah této práce.[41]

Ustavení peer-to-peer spojení

Narozdíl od ustavení XHR nebo WS komunikace se známým serverem je nutné u P2P spojení najít cestu mezi spojovanými uzly. To zahrnuje použití technik pro tzv. NAT traversal, protože situace, kdy jsou oba uzly vzájemně viditelné v rámci sítě Internet, jsou spíše výjimkou.[42] Pro ustavení P2P komunikace je tak třeba zajistit:

1. oznámení uzlu o úmyslu P2P připojení, ať začne naslouchat příchozím spojením,
2. identifikovat směrovatelné cesty mezi uzly a tuto informaci oznámit všem uzlům,
3. dohodnout parametry přenosu, kodeky, protokoly, kódování apod.

WebRTC samostatně řeší bod 2. pomocí vestavěné podpory ICE protokolu. Bod 1. a 3. musí zajistit webová aplikace implementací signalizace.

Signalizace

Z důvodu kompatibility s existujícími systémy signalizace (XMPP, SIP, aj.) nespecifikuje WebRTC žádný konkrétní mechanismus ani protokol, který musí vývojář pro signalizaci použít. Signalizaci lze implementovat přímo v rámci aplikace použitím technik XHR nebo WebSockets (viz sekce 2.5).

Pro popis parametrů peer-to-peer spojení WebRTC používá SDP.[43] Ten specifikuje vlastnosti spojení, typ přenášeného média, kodeky a jejich nastavení, informace o přenosovém pásmu a další metadata (viz kód 2.19).

[40] DTLS tak nabízí podobné bezpečnostní záruky. Rozdíl je například v handshake mezi dvěmi WebRTC klienty, kdy se spoléhá na self-signed certifikáty a je tak potlačen řetěz důvěry a nelze takto autentizovat jednotlivé strany. [12]

[41] Velmi podrobně se technologií WebRTC zabývá mnohokrát citovaná publikace High Performance Browser Networking [5]

[42] Oba uzly by musely disponovat veřejnou IP adresou a nesměly by se nacházet za NATy bez nastaveného předávání portů.

[43] Session Description Protocol

Ukázka kódu 2.19: Ukázka SDP offer. Vysvětlení jednotlivých částí viz [8].

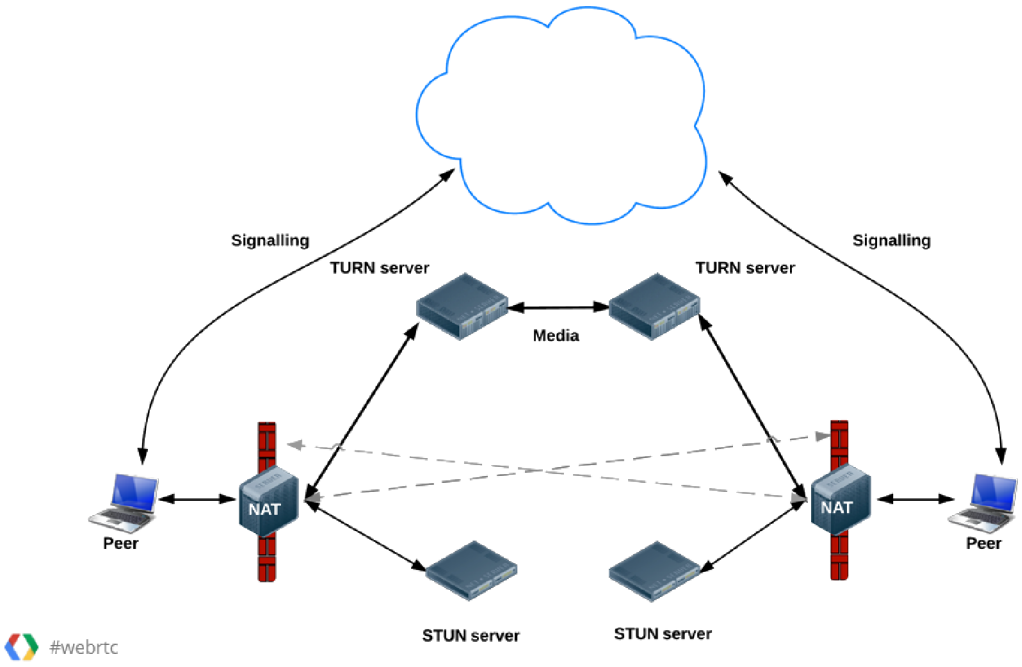
```
1 ...
2 m=audio 1 RTP/SAVPF 111 ...
3 a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
4 a=candidate:1862263974 1 udp 2113937151 192.168.1.73 60834 typ host ...
5 a=mid:audio
6 a=rtpmap:111 opus/48000/2
7 a=fmtp:111 minptime=10
8 ...
```

JSEP je protokol, který abstrahuje nutnost přímé práce s SDP na volání několika metod objektu RTCPeerConnection přímo pomocí jazyka JavaScript.

Vzájemná výměna SDP popisu přes signalizační kanál je prvním krokem k P2P komunikaci. **Dále je nutné ustavit přímé spojení mezi uzly.**

RTCPeerConnection

Jedním z prostředků, který komplikuje přímé spojení uzlů je NAT. Ačkoliv oddaluje vyčerpání IPv4 adres směrováním více uzlů za jedinou adresu, v praxi komplikuje vzdáleně iniciovanou komunikaci s takovými uzly.[44] Nemáme žádnou garanci přímého propojení uzlů v případě přítomnosti NAT prvků v síti. Existuje však několik metod a postupů, jejichž cílem je ustavení spojení i za takových podmínek. Viz schéma 2.13.



Obrázek 2.13: Komunikace uzlů s přítomností NAT. Použití TURN v případě symetrického NAT. Zdroj [4].

STUN je protokol umožňující získání externí IP adresy a komunikačního portu. Typicky se jedná o veřejně přístupný server na síti Internet, který požadavkům vrací jejich zdrojovou

[44] A skutečnost, že princip NAT byl různě implementován na mnoha síťových prvcích dříve, než byl standardizován, situaci komplikuje ještě více.

IP a port. Toto slouží zařízením za NATy ke zjištění skutečné směrovatelné IP adresy. STUN server je jednoduchý a levný na provoz.

Lze tak obejít restrikce těchto třech nejčastějších typů NATů: Full, Address-restricted a Port-restricted cone. Pro Symmetric NAT je nutné použít techniku odlišnou. Podrobný popis jejich odlišností viz [22], kapitola 6.

Postup připojení mezi uzly je pak následující:

1. Uzel A kontaktuje STUN server, aby zjistil, že jeho interní dvojice IPiNodeA:PORTiNodeA je mapována na IPeNodeA:PORTeNodeA.

2. Uzel B obdobně zjistí mapování IPiNodeB:PORTiNodeB na IPeNodeB:PORTeNodeB.

3. Uzly si tyto informace vymění pomocí signalizačního kanálu (client-server-client).

4. NAT po cestě k uzlu A nyní příchozí pakety na IPeNodeA:PORTeNodeA namapuje zpět na IPiNodeA:PORTiNodeA. Obdobně pro uzel B.[45]

5. Uzly mohou zahájit přímou komunikaci dále bez asistence serveru.

Toto hledání nachází server reflexive kandidáty. ICE používá také peer reflexive kandidáty, což zjednodušeně označuje získání externí IP adresu z pohledu jiného uzlu namísto STUN serveru.[46]

TURN je protokol používaný v případě, kdy nelze uzly přímo propojit a je nutné přistoupit na variantu přeposílání (relay) aplikačních dat skrz veřejně dostupný směrovatelný server. Nejedná se již tedy o peer-to-peer komunikaci a provoz takového serveru je drahý, protože skrze něj tečou veškerá data.

Jak již bylo zmíněno, nikdy nemáme garanci propojení všech uzlů peer-to-peer sítě a je tedy nutné pro spolehlivou komunikaci takovýto server provozovat a snažit se minimalizovat potřebu jeho použití. Typickou situací, kdy nezbývá než TURN použít, je přítomnost symetrického NAT.[47] Jedná se tedy o záložní řešení připojení pro pokrytí spojení všech uzlů, které jsou schopné odesílat pakety do sítě Internet.

The Interactive Connectivity Establishment (ICE) je protokol zajišťující automatické nalezení **optimální komunikační cesty** mezi dvěma uzly. Volitelně mu lze dodat adresy STUN a TURN serverů pro optimální rozhodování. WebRTC nabízí ICE agenta, kterého obsahuje každé RTCPeerConnection spojení. Je zodpovědný za získávání kandidátů připojení (dvojic IP adres a portů), vykonávání kontrol spojení a jejich udržování (keepalive).

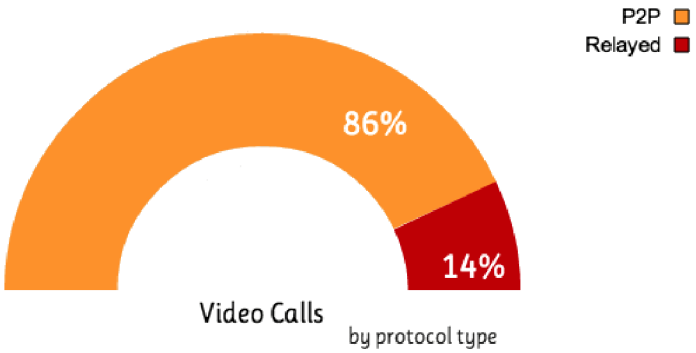
Po výměně SDP popisu sezení zahájí ICE agent hledání možných kandidátů. Jakmile jsou nalezeni, aplikace je o tomto stavu informována (asynchronně pomocí callback funkce) a je možné přegenerovat SDP nabídku a vyměnit ji přes signalizační kanál (tentokrát s již nalezenými kandidáty připojení) s druhým uzlem.

Po přijetí kandidátů druhým uzlem je zahájena druhá funkce ICE agenta – kontrola spojení. Dle priorit zkouší druhý uzel navázat spojení s uzlem prvním pomocí STUN protokolu. Agent zašle požadavek STUN binding request a očekává úspěšnou odpověď STUN

[45] Předpokládáme použití Full cone NAT, nicméně situace by byla obdobná i v případě ostatních typů vyjma Symmetric NAT.

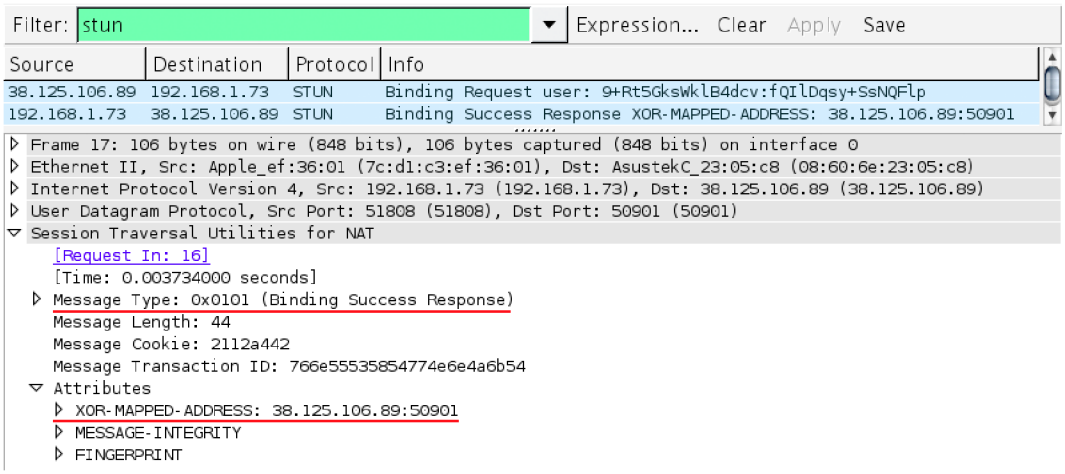
[46] Více informací je uvedeno v knize [18].

[47] Ten na rozdíl od ostatních typů NATů pro každé odchozí spojení namapuje náhodný port. To znamená, že otevřené spojení k STUN serveru použije port A, který je otevřen exkluzivně pro adresu STUN serveru. Při pokusu o připojení na jinou adresu (uzlu) není tento port A použit znovu, ale je přiřazen náhodný port B. Uzel nikdy nezjistí port na kterém je skrz NAT dostupný pro budoucí komunikaci a nezbývá než použít TURN. Podobně se chovají i méně restriktivní typy NATů v případě kolize odchozích portů vícero zařízení.



Obrázek 2.14: Většina P2P přenosů pomocí WebRTC teče mimo TURN servery. Zdroj http://webrtcstats.com.

`response` (obr. 2.15). Pokud připojení selže, zkusí se kandidát další. Pokud neselže, je otevřena komunikační cesta mezi uzly.[48] Zde však součinnost agenta nekončí, protože po celou dobu otevřeného spojení zajišťuje jeho udržování pomocí periodických `STUN` požadavků.

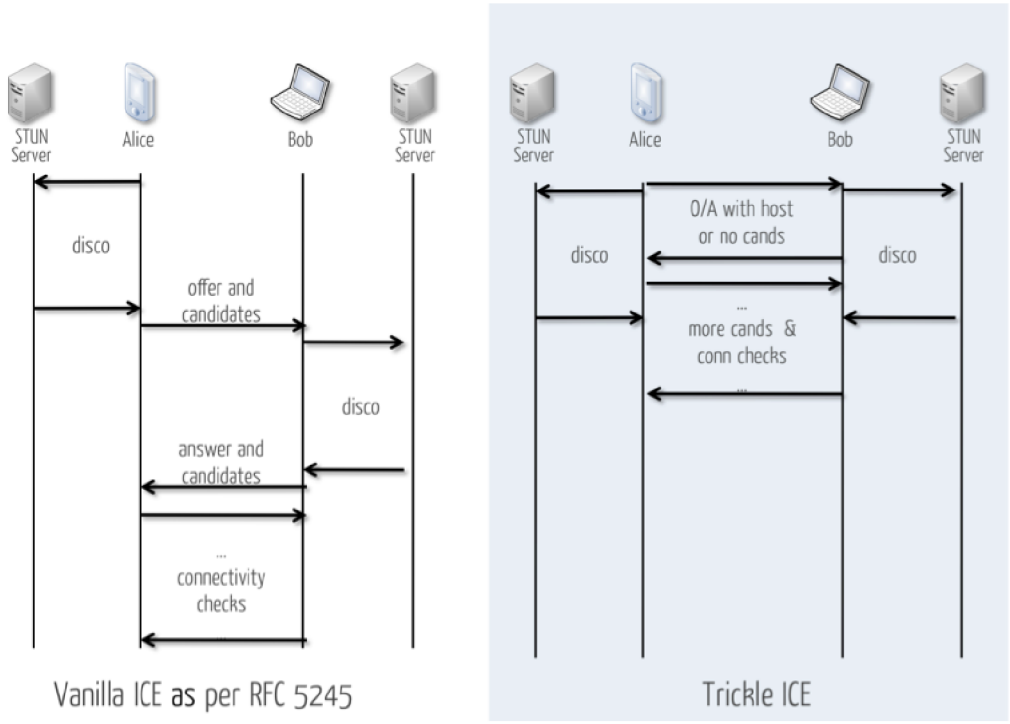


Obrázek 2.15: Wireshark záznam kontroly `P2P` spojení pomocí `STUN` požadavku a odpovědi. Zdroj [5].

Trickle ICE je rozšíření umožňující inkrementální získávání kandidátů a vykonávání kontrol spojení. Důvodem je zpoždění způsobené komplikovaným ustavováním připojení. Princip spočívá v odeslání `SDP` popisu signalizačním kanálem co nejdříve je to možné.[49] (tzn. bez čekání na `ICE`) Poté je možné inkrementálně odesílat získané kandidáty ihned po jejich získání. Druhá strana tak může dříve započít kontroly spojení a ustavit připojení. Kdykoliv se objeví vhodnější kandidát, je opět vyměněn přes signalizační kanál a spojení je adekvátně upraveno.

Tento princip s sebou nese vyšší režii metadat přenášených signalizačním kanálem (`client-server-client`), ale má za následek rychlejší ustavení `P2P` spojení (obr. 2.16).

[48] Dle funkčního použitého kandidáta – tedy buď přímo, skrze `NAT` díky `STUN` nebo pomocí `TURN` serveru.
[49] V prvním SDP offer se pak objeví místo IP 0.0.0.0 značící použití Trickle ICE.



Obrázek 2.16: Srovnání ICE a rozšíření Trickle ICE. Zdroj https://webrtchacks.com/trickle-ice/.

RTCDataChannel

Z hlediska této práce je **nejzajímavější** `RTCDataChannel API` pracující nad protokolem `SCTP` umožňující výměnu libovolných aplikačních dat. `API` je navrhnuto s cílem nabídnout vývojářům univerzální prostředek pro přenos obecných (nejen audio/video) dat pomocí oboustranné `peer-to-peer` komunikace. Jedná se tedy o zobecnění datových přenosů mezi uzly v prostředí webových aplikací.

Jak již bylo zmíněno v úvodní části podkapitoly 2.5, `WebRTC` komunikuje výhradně nad `UDP` z výkonnostních důvodů. U přenosů obecných dat však můžeme potřebovat spíše **spolehlivé datové přenosy** služeb protokolu `TCP`.

Standardizace dospěla k řešení použitím protokolu `SCTP` (Stream Control Transmission Protocol) nad `DTLS` nad `UDP`. Ve spojení s `ICE` toto řešení poskytuje robustní `peer-to-peer` spojení nad téměř libovolnou síťovou topologií, důvěryhodnost, autentizaci a integritou chráněné přenosy. `SCTP` umožňuje pro každé spojení přes `RTCDataChannel` definovat:

- spolehlivé, nespolehlivé nebo částečně spolehlivé doručování zpráv,
- zajištění nebo nezajištění doručování zpráv ve správném pořadí.

Kombinace nespolehlivého doručování zpráv bez zajištění správného pořadí je pouze jedna kombinace a právě ta odpovídá sémantice protokolu `UDP`. Částečná spolehlivost doručování umožňuje specifikovat maximální počet pokusů o znovu doručení či nastavení patřičných časových limitů.

`DTLS` je navržen tak, aby byl co nejvíce podobný protokolu `TLS`, který ovšem pracuje nad `TCP` a ne `UDP`. V protokolu je tedy znovu implementováno `TLS handshake`, fragmentaci, znovu odesílání ztracených paketů, sekvenční číslování apod.

	TCP	UDP	SCTP
Spolehlivost	spolehlivý	nespolehlivý	nastavitelné
Doručování	v pořadí	mimo pořadí	nastavitelné
Přenos	bajty	zprávy	zprávy
Řízení toku	ano	ne	ano
Řízení zahlcení	ano	ne	ano

Tabulka 2.2: Srovnání protokolů TCP, UDP a SCTP

WebRTC automaticky generuje self-signed certifikáty pro každý uzel a proto je autentizace přenechána na samotné aplikaci. Musí být použity blokové šifry, protože šifry proudové implicitně závisí na in-order doručování dat. Navíc se každý DTLS záznam musí vlézt do jediného síťového paketu kvůli absenci fragmentace (TLS záznam může mít až 16 kB).

SCTP je transportní protokol podobný TCP a UDP, který může pracovat přímo nad IP protokolem. Avšak v případě WebRTC, které vyžaduje šifrování komunikace, je SCTP tunelováno nad zabezpečeným DTLS tunelem, který samotný pracuje nad UDP.[50] Jedním z důvodů je to, že routery a NAT zařízení jednoduše často neumí zpracovat čisté SCTP nad IP korektně. Nejjednodušší řešení je pak komunikaci šifrovat jako to dělá právě WebRTC pomocí DTLS. Srovnání protokolů je uvedeno v tabulce 2.2.

Pro srovnání s client-server architekturou má RTCDataChannel nejblíže WebSocket protokolu. API RTCDataChannel je nadmnožinou WS API. Kromě zmíněné možnosti konfigurace spolehlivosti a doručitelnosti přidává RTCDataChannel taky multiplexování vícero proudů dat jedním kanálem.

Ani jedno řešení nenabízí automatickou kompresi přenášených dat. Pro WS však existuje rozšíření protokolu, avšak ne pro WebRTC. Vhodnou konfigurací parametrů lze dosáhnout spolehlivosti a doručitelnosti **stejné jako v případě použití** WebSocket**, ale nad architekturou** peer-to-peer.

2.6 Přehrávání videa

S příchodem HTML5 byl představen element <video>, který měl za cíl nahradit dosavadní řešení vkládání multimediálního obsahu do webových aplikací pomocí elementů <object> či <embed> využíváním technologií jako je Adobe Flash nebo Java. Specifikace neurčuje, které formáty musí být prohlížeče schopny přehrát. Vzniká tím prostor pro nekompatibility a případné **nutnosti překódování** videí do více formátů.

Specifikace dovoluje definovat více zdrojů video/audio elementu. Prohlížeče pak sekvenčně procházejí tento seznam a první formát, který dokážou přehrát vyberou (viz kapitola 2 knihy [16]). Ukázka kódu 2.20 představuje komplexní příklad připojení videa do webové aplikace. Je použit volitelný atribut type, který zajistí, že prohlížeč nemusí zbytečně stahovat data, aby zjistil, že video neumí přehrát.

Ukázka kódu 2.20: Video element s více zdroji a volitelnou specifikací typu.

```
1 <video poster="cats.png" controls>
2   <source src="cats.ogv" type='video/ogg; codecs="theora, vorbis"'>
```

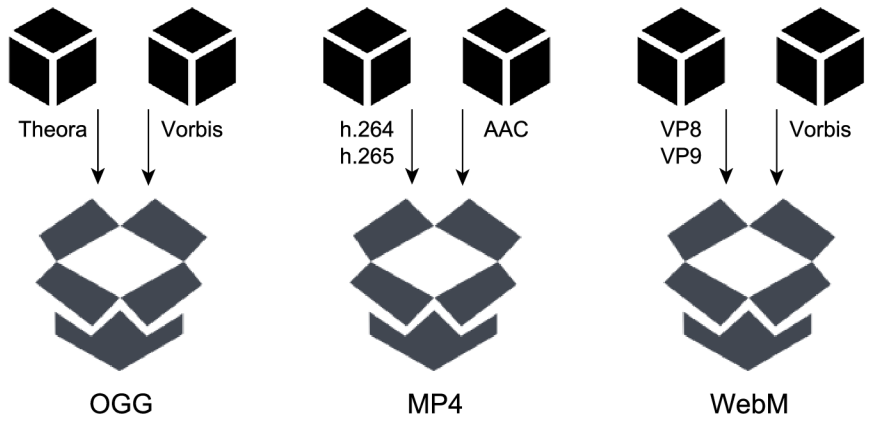
[50] SCTP pak pro potřeby WebRTC například zbytečně znovu přenáší čísla portů.

```
3    <source src="cats.mp4" type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'>
4    <source src="cats.webm" type='video/webm; codecs="vp8, vorbis"'>
5  </video>
```

Kromě tohoto jednoduchého přístupu, který postačuje základním požadavkům na přehrávání multimédií, existuje bohaté `JS API`, které dovoluje element `<video>` kompletně ovládat. To zahrnuje spouštění, pozastavování, přetáčení zdroje, ovládání stahování, přístup k metadatům, detekci podporovaných kodeků, ovládání bufferování, připojování titulků a mnoho dalšího.

Kontejnery a kodeky

V současnosti jsou na poli kodeků pro webové video **dva hlavní formáty**, které soupeří o to stát se základním formátem, kdy se na podporu jejich kodeků mohou vývojáři spolehnout. Jedná se o kodek `H.264` formátu `MPEG4` a kodek `VP8` formátu `WebM`, resp. jejich nejnovější (zatím špatně podporované) varianty `H.265` a `VP9` dle tab. 2.3.[51] Často se uvádí jako třetí zástupce kodek `Theora` formátu `OGG`, ovšem tento lze považovat za **překonaný** předchozími zmíněnými (dle kap. 1 knihy [16]) a není pro potřeby této práce dále uvažován.



Obrázek 2.17: Vztah hlavních formátů a kodeků v prostředí webu.

MPEG4 je kontejner, obvykle s příponou `.mp4` a typicky obsahuje `H.264` nebo `H.265` zakódované video stopy a `ACC (Advanced Audio Codec)` audio stopy. Je založen na starším `.mov` formátu (`Apple`) a je jedním z nejčastěji používaným formátem videokamerami, mobily či tablety.

WebM kontejner založený na kontejneru `Matroska` používá příponu `.webm` a jedná se o **licenčními poplatky nezatížený otevřený formát** speciálně vyvinutý pro `HTML5` video. Vývoj sponzoruje společnost `Google`. Obsahuje video stopy zakódované pomocí `VP8/VP9` kodeků a audio stopy pomocí kodeků `Vorbis` nebo `Opus`. Tento kontejner nativně podporuje mnoho moderních prohlížečů kromě `IE`[52] a `Safari`.

[51] Ačkoliv teoreticky by to mohlo být možné, vzájemné kombinace kodeků a formátů přípustné nejsou, což situaci zjednodušuje.

[52] A taky kromě prohlížeče `Edge` do nejnovější verze 14.

Prohlížeč	H.265 (MP4)	VP9 (WebM)
Android Browser	-	4.4
Chromium	-	r172738
Google Chrome	-	29
Internet Explorer	-	-
Microsoft Edge	-	preview builds
Mozilla Firefox	-	28
Opera	-	16
Safari	-	-

Tabulka 2.3: Podpora nejnovějších video kodeků v prohlížečích. Zdroj: https://en.wikipedia.org/wiki/HTML5_video, duben 2016.

H.264/H.265 kodeky vyvinuté s vidinou existence jediného kodeku pro vše. Od telefonů s nízkým výkonem a vysokou citlivostí na odběr baterie po stolní počítače s výkonem vysokým. Toho je dosaženo kódováním s využitím rozličných profilů, které pokrývají různé případy užití. Velmi častá je implementace dekodéru ve speciálním čipu v `HW`, čímž je dosaženo dostatečné rychlosti dekódování s nízkým odběrem i na jinak nevýkonných zařízeních.

VP8/9 kodeky mají za cíl nahradit `H.264/H.265` podobnou kvalitou při podobné velikosti, ale bez nutnosti odvádět licenční poplatky za jejich užití. `Safari` kodek nepodporuje a podporovat nechce, za což může taky výrazně nižší `HW` podpora nežli v případě `H.264/H.265`. Nicméně mnoho předních výrobců oznámili přidání `HW` podpory pro (de)kódování těchto kodeků.

Vývoj a podpora

Informace v této sekci vycházejí z knih [16] a [17]. Současně taky z mnoha oznámení a zápisů vývojářů prohlížečů v případech, kdy knižní literatura nebyla dostatečně aktuální.

V `HTML5` specifikaci byl prvně jako základní formát doporučován `OGG`. `Apple` vydal první prohlížeč s podporou `<video>` elementu a rozhodl se podporovat pouze `H.264` kritizujíce kodek `Theora` za horší kvalitu a chybějící podporu na mobilních zařízeních.[53] Podobně postupovaly další společnosti.

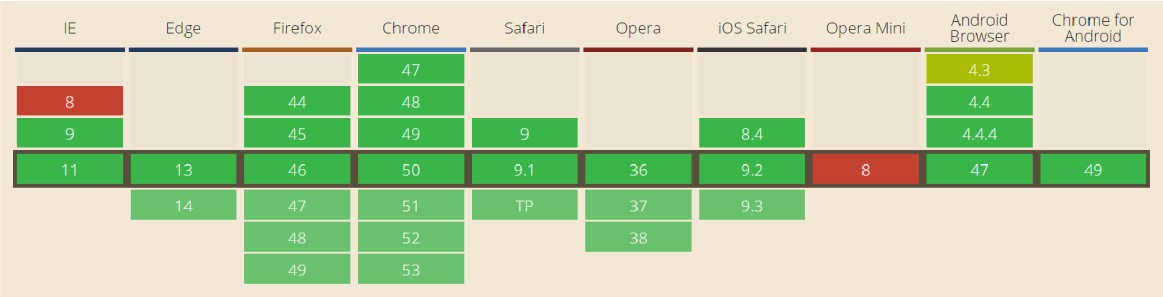
`H.264` byl sice schválen jako standard společně organizacemi `ITU` a `ISO/IEC`, ale **vyžaduje licenční poplatky, což je v rozporu se specifikací** `HTML5`, aby mohl být přijat jako základní kodek. Později, 26. srpna 2010, bylo oznámeno, že internetové video zakódované pomocí `H.264`, které je poskytováno zdarma, je a vždy bude zdarma pro koncové uživatele. To však nestačí, protože komerční užití a HW produkty stále musí poplatky odvádět. Přesto dlouho dobu neexistovala adekvátní náhrada a formát se tedy stával více a více populární.[54]

Změna přišla až 19. května 2010, kdy `Google` oznámil zahájeni projektu `WebM` s cílem překonat dosavadní možnosti videa v prostředí webu bez nutnosti odvádět jakékoliv

[53] Zdroj tohoto příběhu dostupný v archívu `Apple Insider` programu: http://www.appleinsider.com/articles/09/07/06/ogg_theora_h_264_and_the_html_5_browser_squabble.html

[54] `YouTube` dlouhou dobu kombinaci `MP4 H.264/AAC` používal ve spojení s `Adobe Flash` pro poskytování video obsahu.

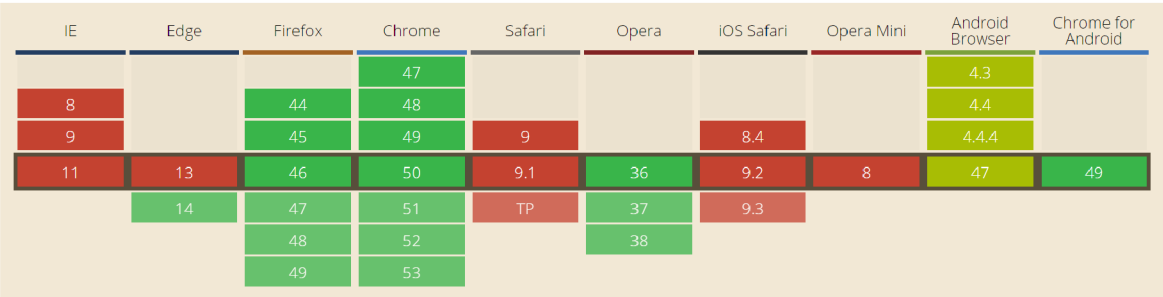
licenční poplatky.[55] Následovalo oznámení Mozilly a Opery, že nebudou podporovat MP4 H.264/AAC kvůli licenčním poplatkům a kvůli rozporu s myšlenkou otevřenému webu.[56]



Obrázek 2.18: MPEG4 video formát – podpora v současných prohlížečích (úplná 78 %, částečná 11 %). Procentuální pokrytí vychází z dat StatCounter GlobalStats pro duben 2016. Zdroj: http://caniuse.com/

Později však, 18. března 2012, Mozilla oznámila podporu pro H.264 ve svém prohlížeči Firefox na mobilních zařízeních kvůli rozšířené HW podpoře. Následně, 20. února 2013, doimplementovala podporu pro dekódování H.264 také na Windows 7 a novějších (FF 21 a novější). Obdobně pak později pro Linux (od verze FF 26), kdy je nutná přítomnost vhodných gstreamer rozšíření v systému. Ve všech případech však funkčnost závisí na externích dekódovacích knihovnách a **žádný dekodér není přítomný ve zdrojovém kódu prohlížeče**.[57]

Ke všemu, 30. října 2013, Cisco oznámilo vydání binárních i zdrojových souborů H.264 video kodeku pod jménem OpenH264 pod zjednodušenou BSD licencí. Zároveň se zavázalo **odvádět veškeré poplatky** při použití předkompilovaných binárních souborů společností Cisco.[58] Na to zareagovala Mozilla oznámením, že by této nabídky využili v dalších verzích prohlížeče Firefox tam, kde kodeky pro dekódování H.264 chybí.[59] Momentálně je OpenH264 využíváno pro WebRTC, ale ne pro <video> tag, protože zatím OpenH264 nepodporuje high profil často používaný pro streaming videa.[60]



Obrázek 2.19: WebM video formát – podpora v současných prohlížečích (úplná 66 %, částečná 5 %). Procentuální pokrytí vychází z dat StatCounter GlobalStats pro duben 2016. Zdroj: http://caniuse.com/

[55] Vydáno pod BSD open-source licencí.

[56] Viz http://shaver.off.net/diary/2010/01/23/html5-video-and-codecs/

[57] Viz oznámení https://hacks.mozilla.org/2013/01/firefox-development-highlights-h-264-mp3-support-on-windows-scoped-stylesheets-more/

[58] Veškerý SW, který však používá Cisco zdrojové kódy namísto binárních souborů, stále poplatky odvádět musí sám.

[59] Viz https://blog.mozilla.org/blog/2013/10/30/video-interoperability-on-the-web-gets-a-boost-from-ciscos-h-264-codec/

[60] Viz https://andreasgal.com/2014/10/14/openh264-now-in-firefox/

V současné době je již tedy situace odlišná a prohlížeče, které formát MPEG4 zamítly, přišly s řešením v podobě externích zásuvných modulů. V tabulkách 2.18 a 2.19 lze vidět, že z hlediska podpory v prohlížečích je MPEG4 vítěz nad WebM. A to především kvůli chybějící podpory WebM ze strany Safari a starších prohlížečů MS. Nutno však podotknout, že podpora ze strany prohlížečů Firefox a Opera je závislá na dostupnosti HW či SW třetí strany (viz kapitola 6 knihy [1]) a stále je nutné odvádět poplatky za poskytování komerčního obsahu. V praxi se tak často videa distribuují v obou formátech pro zajištění úplné kompatibility.

Komplikovanou situaci s licencováním MPEG4 se právě snaží vyřešit Google s jeho novým (a tedy méně podporovaným) formátem WebM. Důkaz, že se mu to daří, budiž použití tohoto formátu v souboru technologií WebRTC či službách jako jsou Wikipedia, Skype nebo YouTube. YouTube se zavázal překódovat celé své portfolio videí do WebM a používat MPEG4 pouze pro nepodporované prohlížeče.[61]

Vytvoření streamů dat

Z hlediska této práce je zajímavá možnost programové změny zdroje videa elementu <video> na vygenerovaný odkaz z libovolně poskládaných binárních dat (popsáno v části 2.3). Pro takové skládání existuje Media Source Extensions API,[62] které je určeno pro generování media streamů pro použití ve <video>, resp. <audio> elementech. Tyto elementy tedy nejsou omezeny pouze na dotázání existujících video souborů uložených na serverů a jejich následné přehrání.

Na straně klienta je možné implementovat **adaptivní streaming**,[63] preferování kvality dat, libovolné přesouvání v rámci streamu, vkládání reklam apod.

Media Source Extensions API klade, dle specifikace, za cíl:

- Dovolit JS konstruovat media streamy **nezávisle** na způsobu jejich získání (či jejich částí).
- Definovat modely rozdělování a načítání médií s jejich současnou minimalizací potřeby parsování v JS.
- Nezáviset na **žádném** formátu či kodeku a využívat cache browseru co nejvíce.
- Specifikovat požadavky na formát zpracovávaných streamů bajtů dat.

Odesílání streamů dat

Skládání částí videí lze docílit například pomocí zmíněného API v předchozí podsekci. Pro získání těchto částí lze využít několik technik.

Tradiční metodou doručování audia a videa přes Internet je RTP streaming použitím RTP/RTSP protokolů. Tato metoda je však standardně určena pro sledování médií v reálném čase, kdy data nejsou u klienta cachována, nelze se ve streamu posouvat, má horší prostupnost firewally a NATy a je vyžadována infrastruktura speciálních streaming serverů. Navíc podpora tohoto přístupu ze strany prohlížečů spočívá v **nutnosti použití externích doplňků**.

[61] Viz kniha [10] a https://www.youtube.com/watch?v=poHqoBKiSeY

[62] Specifikace ve fázi Candidate Recommendation z 12. listopadu 2015. Zdroj: https://www.w3.org/TR/media-source/.

[63] Technika doručování videa, typicky přes protokol HTTP, kdy je kvalita zobrazovaných dat dynamicky uzpůsobována rychlosti a kvalitě přenosového média (či výkonu procesoru apod.).

V rámci webových aplikací je **mnohem rozšířenější** metoda `HTTP Streaming` s využitím `HTTP Progressive Download`. `HTML5` standardně pracuje se servírováním video dat pomocí `HTTP` serveru. Progresivní stahování spočívá v doručování přesného rozsahu dat ze strany serveru. Ten musí podporovat `HTTP 1.1 Byte Range` požadavky.

Nejdříve prohlížeč načte několik počátečních bajtů dat, které obsahují metadata a poté se postupně dotazuje serveru na potřebné rozsahy. Je možné přesunout video libovolně do ještě nenačtených částí, protože prohlížeč odešle další požadavek s jiným bajtovým rozsahem (a případně zruší požadavek předchozí). Standardně jsou jednou načtená data uložena do `cache` prohlížeče[64] a další případné přesouvání v rámci videa negeneruje redundantní požadavky. Ukázka 2.21 využívá jednoduché nastavení `HTTP` hlavičky `Range` pro získání první části videa. Komunikace mezi prohlížečem a serverem je pak zachycena v ukázkách 2.22 a 2.23.[65]

Ukázka kódu 2.21: Progresivní stahování a následné skládání streamu.

```
var sourceBuffer = initSB(); // mimo rozsah příkladu
var xhr = new XMLHttpRequest();
xhr.setRequestHeader('Range', 'bytes=0-500');
xhr.responseType = 'blob';
xhr.open('GET', 'upload.mp4', true);
xhr.onload = (e) => {
  var part = new Uint8Array(e.target.result);
  sourceBuffer.append(part);
};
xhr.send();
```

Ukázka kódu 2.22: Range požadavek.

```
GET /upload.mp4 HTTP/1.1
Host: ctrlv.tv
Connection: keep-alive
Range: bytes=0-500

...
```

Ukázka kódu 2.23: Range odpověď.

```
HTTP/1.1 206 Partial Content
Accept-Ranges: bytes
Content-Length: 91027
Content-Range: bytes 0-500/91027
Content-Type: video/mp4

...
```

Různé společnosti si s postupem času nad `HTTP` implementovaly odlišné metody streamování dat v dobách, kdy to nebylo s pomocí čistého `HTML5` a `JS` možné. Mezi takové patří `Apple HTTP Live Streaming (HLS)`, `Microsoft Smooth Streaming` či `Adobe HTTP Dynamic Streaming`.

Základní princip těchto technologií je stejný – na serveru jsou přístupná média zakódovaná s různými datovými toky či rozlišeními, klient je o těchto verzích informován souborem s metadaty a musí zajistit vhodnou volbu dle aktuálních podmínek (rychlost a kvalita připojení, vytížení procesoru, ...). Jediný prohlížeč, který nativně podporuje některou z těchto technologií je `Safari` od společnosti `Apple` (viz kniha [16]).

V současné době je však jasně viditelná orientace na implementaci `HTTP` streamingu pomocí stále více podporovaných `JS API` na úkor těchto proprietárních technologií.[66] Jako příklad lze uvést server `YouTube`, jehož obsah je nabízen výhradně pomocí `HTTP Progressive download` a nevyužívá tedy `RTP` (kniha [16], kap. 2.3.3).

[64] Což může být z hlediska vyžadování zabezpečení dat technologiemi typu `DRM` problém.
[65] Jsou uvedeny pouze hlavičky, které se týkají nastavení bajtového rozsahu.
[66] Výhodou proprietárních technologií je možnost lepšího zabezpečení `DRM` šifrováním.

Kapitola 3

Návrh řešení pro záznam, přenos a přehrávání videa

Cílem této práce je vytvořit unikátní službu kombinující nejnovější technologie na poli webových aplikací. Jak napovídá obsah i rozsah kapitoly teorie, existuje mnoho řešení různých dílčích podproblémů. Nyní se pokusím z každé části vybrat nejvhodnější řešení a to s ohledem na celkovou realizaci a praktické využití služby.

Z povahy použitých technologií nebude prioritou co nejvyšší podpora ve všech možných prohlížečích, ale současně nebudou vybrány technologie podporou významně omezující použitelnost aplikace.

Část Případy užití uvádí příklady aplikací řešeného systému, pro který je nutné postupně navrhnout jak video zaznamenat (sekce Záznam videa), zpracovat (sekce Zpracování videa) a přenést (sekce Přenos videa). Až poté se lze zabývat samotným přehráváním nahraného videa v podkapitole Přehrávání videa. Kompletní souhrn uvedený v poslední části Souhrn slouží jako vstupní bod do části implementace.

3.1 Případy užití

Hlavním záměrem je nabídnout pohodlné a rychlé nahrání dění na obrazovce uživatele s vygenerováním krátkého odkazu pro sdílení nahrávky. **Nebude tedy realizován přenos v reálném čase**, ale záznam, persistence a škálovatelná distribuce nahrávek.[1]

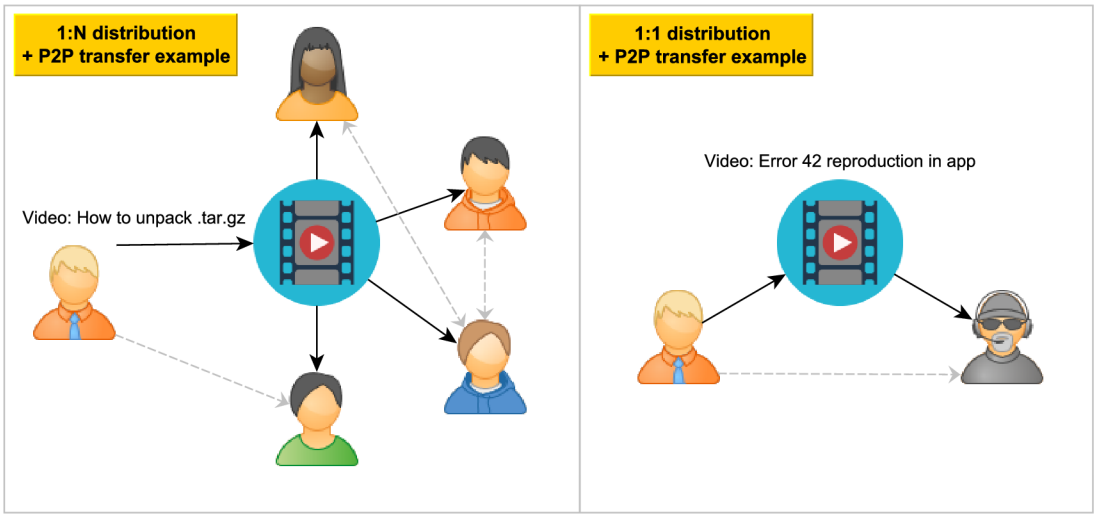
Typickým případem užití může být nahrání návodu typu `webcast`, kdy přednášející zaznamená postup řešení úkolu, např. programovací úlohy a umístí ho online. Poté odkaz pošle účastníkům nebo veřejně nasdílí.

Kromě nahrávek určených jako `1:N` lze očekávat časté využívání služby pro `1:1` nahrávky jako rychlé dobarvení kontextu komunikace videem tam, kde je použití tradičních technologií jako `Skype` nebo `TeamViewer` zdlouhavé či zbytečné (viz schéma 3.1).

V neposlední řadě jsou velice moderní záznamy dění během hraní počítačových her. Jako příklad lze uvést populární službu `twitch.tv`.

Nároky na datové úložiště a konektivitu serveru jako centrálního prvku pro distribuci nahrávek by byly pochopitelně velmi vysoké. Proto se v práci snažím najít cestu škálovatelnosti distribuce nahrávek pomocí `peer-to-peer` komunikace.

[1] Což samozřejmě neznamená, že by technologie přenosu v reálném čase nebyla z hlediska práce zajímavá. To zajisté je, ale nejedná se o takovou technologickou výzvu s nutností využívat nejnovějších `API` prohlížečů. Současně již služba umožňující sdílení obrazovky v reálném čase existuje: https://talky.io/.



Obrázek 3.1: Ukázka příkladu užití s přenosovými cestami.

Ideálním případem užití, kdy se naplno projeví výhody tohoto přístupu, může být případ školení, kdy všichni účastníci na lokální síti `LAN` přistoupí ke stejné nahrávce připravené přednášející osobou. **Části videa se začnou distribuovat mezi účastníky na úrovni `LAN` sítě, což zajistí vyšší rychlost přenosu a minimální nároky na centrální prvek služby.**

Z hlediska uživatele je důležité zajistit:

- rychlé a co nejjednodušší spuštění nahrávání videa kdekoliv potřebuje,
- průběžné nahrávání a zpracování i za cenu případu, kdy uživatel nedokončí záznam,
- jednoduchý, krátký a snadno zapamatovatelný odkaz, který není nutné opisovat,
- alespoň tak rychlé stahování videa při přehrávání, ať lze nerušeně sledovat,
- umožnit pozastavení videa či libovolný posuv bez zbytečného čekání na stažení nezajímavých úseků.

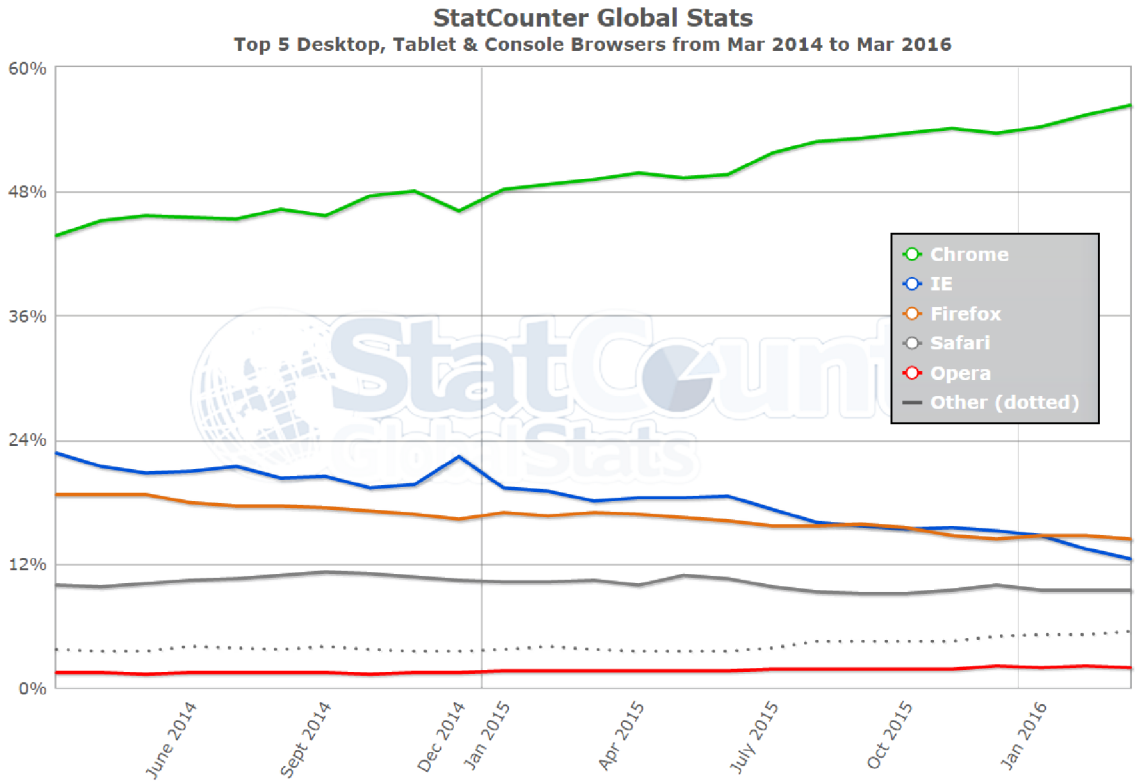
Nyní je nutné navrhnout metody záznamu videa, jeho zpracování, přenosu a přehrávání. Při tom je nutné zvolit formát videí a navrhnout architekturu celé aplikace.

3.2 Záznam videa

V práci se zaměříme na záznam videa obrazovky uživatele. Případná rozšíření o záznam zvuku či videa z kamery budou diskutována v závěru.

Z dostupných metod popsaných v teorii v kapitole 2.3 bude pro získání přístupu k streamu pracovní plochy uživatele použito moderní `MediaStream API`. Pro sdílení plochy je v případě prohlížeče `Google Chrome` a `Opera` **nutné implementovat rozšíření** (viz část 2.3). Pro prohlížeč `Mozilla Firefox` je implementace obdobného rozšíření vhodné, protože je pro uživatele snazší než po něm vyžadovat změny v nastavení prohlížeče. Rozšíření

toto vykoná automaticky. **Jiné prohlížeče podporu pro sdílení plochy nenabízejí.**[2] Srovnání je uvedeno v tabulce 3.1.



Obrázek 3.2: Vývoj podílu prohlížečů na trhu v posledních letech. Zdroj: http://statcounter.com

Tímto se množina úplně podporovaných prohlížečů zužuje na tři, z toho dva jsou nejrozšířenější (pokrývající téměř 70 % uživatelů, viz graf 3.2). To však neznamená, že další technologie budou vybrány bez ohledu na podporu ostatními prohlížeči. Záznam videa jen jedna z částí práce (důležitější je schopnost video získat a přehrát) a podpora může být kdykoliv přidána i ostatními výrobci.

Pro samotný záznam získaného streamu bude z popisovaných metod v kapitole teorie (2.3) implementována varianta s využitím `Media Recorder API`. Techniky využívající vlastní `JS` enkodéry vykazují nevyhovující výkon i komprimaci.[3] Dalším důvodem je, z hlediska práce, dostatečná podpora `Media Recorder API` pro záznam pracovní plochy (podporovány jsou `Chrome`, `Opera` i `Firefox`).

Pro případ, kdy by bylo třeba nahrávat video z externího zdroje (např. webová kamera), což podporuje mnohem více prohlížečů, a překonat omezení enkodérů v `JS`, lze použít realtime přenos nad `RTP`. Prohlížeč uživatele by šlo připojit ke speciálně nakonfigurovanému umělému uživateli, který bude data přijímat a ukládat na server. Jedná se však o komplikovaný *workaround*, který nemá žádný přínos pro sdílení plochy (protože oba prohlížeče,

[2]Takové prohlížeče mohou v rámci rozšíření podporovat alespoň sdílení videa z externích zdrojů typu webová kamera. Toto je však mimo hlavní záměr práce.

[3]Ve skutečnosti často žádnou ani nevykonávají. Viz http://antimatter15.com/wp/2012/08/whammy-a-real-time-javascript-webm-encoder/. Problémy s výkonem z principu techniky kódování jednotlivých snímků (cca 10 fps) viz http://ericbidelman.tumblr.com/post/31486670538/creating-webm-video-from-getusermedia

	Firefox	Chrome
Nutné rozšíření	ne	ano
Nutná konfigurace prohlížeče	ano (lze rozšířením)	ne
Vyžaduje `HTTPS`	ano	ne
Sdílení vybrané obrazovky	ano	ano
Sdílení vybraného okna	ano	ano
Sdílení záložky prohlížeče	ne	ano
Sdílení vybrané aplikace	ano	ne
Současný záznam audia	ano	ne
Oznámení během sdílení	pruh nahoře, vždy	pruh dole, uživatel může skrýt

Tabulka 3.1: Srovnání sdílení plochy dvěmi podporovanými prohlížeči.

které umí sdílet plochu, umí tyto proudy dat i nahrávat) a proto není dále uvažován.[4]

Pro přístup k datům streamů v `JS` bude vždy využito postupů, které nevykonávají překódování dat do `base64` kódování (to dělá např. `readAsDataURL()`), ale využívají moderní typovaná pole v `JS` (například `Uint8Array` pro reprezentaci bajtů pro síťový přenos) nebo využití volání `URL.createObjectURL()` (viz kapitola 2.3).

Z hlediska volby formátu pro záznam streamu je volba jednoduchá, protože **je podporován pouze** `WebM` s kodekem `VP8`, případně `VP9` v nejnovějších verzích prohlížeče `Chrome` (viz tabulka 2.1).

Srovnání záznamu obrazovky

Pro vyhodnocení praktické použitelnosti `MediaRecorder API` v kombinaci se záznamem obrazovky byla vykonána sada testů srovnávající oba prohlížeče nad několika scénami. Naměřené hodnoty jsou uvedeny v tabulce 3.2. Záznam i následné přehrávání probíhalo vždy v rámci testovaného prohlížeče.

Scéna A je volně dostupná[5] ukázka z filmu, scéna B taktéž, ale z kresleného seriálu a scéna C zachycuje práci uživatele na počítači. Všechny vzorky mají délku jedné minuty, rozlišení 1920x1200 a neobsahují zvukovou stopu. Pro každý test bylo otevřeno samostatné okno prohlížeče pro eliminaci odchylek v měření paměťové náročnosti.

Z naměřených hodnot vyplývá několik důležitých poznatků pro realizaci projektu. V první řadě `FF` **na každém testovaném vzorku selhal v bezchybnosti záznamu** (přenos v rámci prohlížeče jen do video elementu chyby neobsahoval, takže jde pravděpodobně o špatnou implementaci `MediaRecorder API`). Po určité době jsou ve videích přítomny artefakty, problikávání videa a skoky v aktuálním rozlišení videa. Sám `FF` dokáže taková videa přehrát pouze do prvního takového problému. Přehrávač `VLC` umožňuje tyto chyby přeskočit a v přehrávání pokračovat. Podrobnější průzkum tohoto problému spolu s možnými řešeními bude realizován během implementace. `Chrome` v obou variantách zaznamenal video bezchybně.

Z hlediska kvality pořízeného záznamu vítězí kodek `VP9`, který však při záznamu klade na systém jednoznačně **nejvyšší nároky** a slabší `HW` sestavy tak budou mít s tímto kodekem problém nahrávat.

[4] A z podstaty `RTP` nad `UDP`, kdy dochází ke změnám kvality a výpadkům paketů, přináší nekvalitní výstup.

[5] Zdroj: http://www.h264info.com/clips.html

	Firefox 46 VP8	Chrome 50 VP8	Chrome 50 VP9
Scéna A velikost	13,8 MB	55,2 MB	20,5 MB
Scéna A kvalita	2	3	1
Scéna A náročnost záznamu	233 MB / 21 %	114 MB / 19 %	165 MB / 37 %
Scéna A náročnost přehrání	224 MB / 4.3 %	43 MB / 5.1 %	55 MB / 6.6 %
Scéna A bezchybný záznam	ne	ano	ano
Scéna B velikost	17,2 MB	51,8 MB	21,3 MB
Scéna B kvalita	2	3	1
Scéna B náročnost záznamu	212 MB / 17 %	103 MB / 15 %	163 MB / 21 %
Scéna B náročnost přehrání	225 MB / 5,1 %	48,4 MB / 3,9 %	54 MB / 4,7 %
Scéna B bezchybný záznam	ne	ano	ano
Scéna C velikost	17,7 MB	46,9 MB	22,9 MB
Scéna C kvalita	3	2	1
Scéna C náročnost záznamu	217 MB / 19 %	107 MB / 14 %	163 MB / 13 %
Scéna C náročnost přehrání	228 MB / 5,2 %	64 MB / 4,7 %	56 MB / 4,5 %
Scéna C bezchybný záznam	ne	ano	ano

Tabulka 3.2: Srovnání záznamu obrazovky v podporovaných prohlížečích. Kvalita reprezentuje subjektivní dojem autora relativně vůči testovaným vzorkům na stupnici 1–3 od nejlepšího po nejhorší v přehrávači VLC. Využití systémových zdrojů je výsledkem průměru třech měření. Použitý procesor: `Intel Core i7-6700HQ CPU@2.6Ghz 4x2 threads`

Pokud jako prioritní považujeme velikost záznamu kvůli jeho následné distribuci, jednoznačně vede komprimace `VP8` přítomna ve `FF`. To je však vykoupeno mnohem vyšší paměťovou náročností při přehrávání.

Jako doporučení lze tedy považovat využívání nejvyšší dostupné verze kodeku `VP9` pro přehrávání ale i pro záznam, pokud je systém dostatečně výkonný. V opačném případě se lze spokojit s nižší kvalitou a vyšší výslednou velikostí při použití staršího kodeku `VP8`.

V případě `FF`, který dosahuje skvělé výsledné velikosti s přijatelnou kvalitou, nelze vyvozovat doporučení, protože je v takové formě nepoužitelný. Připadá v úvahu experimentovat s rozlišením, umělou volbou nižšího datového toku nebo postupným uvolňováním prostředků průběžným odesíláním částí nahrávky.

3.3 Zpracování videa

Z hlediska zpracování získaného videa si lze u vyvíjené aplikace představit mnoho případů užití. Patří mezi ně především překódování do jiných formátů pro pokrytí širšího spektra prohlížečů, různé ořezy, výřezy nepovedených částí, zakreslování poznámek do videí, přidání autorství ve formě vodoznaku atd. V teoretické části 2.4 jsou rozebrány možnosti jak tyto úpravy vykonat. Obdobně jako u záznamu **vylučujeme** metody získání upraveného videa snímek po snímku s následným kódováním v rámci `JS`.

Další metoda, která by transformace videa[6] ideálně zvládala (a to včetně `HW` akcelerace díky technologii `WebGL`) závisí na podpoře `CanvasCaptureMediaStream API`, které je v době psaní práce podporováno pouze prohlížečem `Firefox` ve verzi 41 a především je ve výchozím

[6] ne však překódování do jiného formátu, dokud prohlížeče více formátu v `Media Recorder API` nebudou podporovat

	Chrome 50	Firefox 45	nativně
Rychlost zpracování	7 min 44 s	6 min 10 s	21 s
Špičkové využití paměti	455 MB	371 MB	59 MB
Zhoršené UX aplikace	ne	ne	ne

Tabulka 3.3: Srovnání výkonu `ffmpeg` pomocí `asm.js` s nativní implementací. Hodnoty jsou průměry ze tří měření. Použitý procesor: `Intel Core i7-6700HQ CPU@2.6Ghz 4x2 threads`.

nastavení deaktivováno.[7] Tato metoda proto v rámci této práce realizována nebude a je ponechána jako velmi zajímavé rozšíření do budoucna.

Nabízí se možnost použít nástroj `FFmpeg` a to buď klasicky na straně serveru za cenu výkonnostní režie spojené s překódováváním či jiným úpravy a nebo experimentálně na straně prohlížečů uživatelů. K tomu je nutné zkompilovat nástroj `FFmpeg` do `asm.js` pomocí kompilátoru `Emscripten` (viz teorie 2.4).

Analýza řešení za pomocí FFmpeg v JS

Stejně jako mnoho jiných programů, i `FFmpeg` již byl zkompilován do jazyka `JavaScript`. Pro analýzu tohoto řešení tak lze navázat na práci autorů projektu `Videoconverter.js`.[8]

Projekt nabízí dvě zkompilované verze nástroje `FFmpeg`:

- `ffmpeg.js` s velikostí 24,1 MB (pro přenos lze využít `HTTP gzip` kompresi a přenášet pouze 6,1 MB),
- `ffmpeg-all-codecs.js` s velikostí 27,5 MB (`gzip` verze opět pouze 6,9 MB).

Jedná se o verzi 2.2.1 z roku 2014. Pokud by se řešení ukázalo smysluplné, bylo by vhodné jejich práci zopakovat a zkompilovat novější verzi tohoto nástroje.[9]

Jako testovací video bylo zvoleno volně dostupné video `bigbuckbunny.webm` v rozlišení 640x360[10] s použitým kodekem `VP8`. Volání `FFmpeg` je vždy vykonáváno v odděleném vlákně prohlížeče a to následovně:[11]

```
ffmpeg -i bigbuckbunny.webm -strict -2 -c:v libx264 -threads 1 output.mp4
```

Testuje se tedy reálný případ užití a to konverze zachyceného videa ve formátu `WebM` do více dostupného `MPEG4`. Aby nebyl prohlížeč znevýhodněn tím, že nevyužívá více než 1 vlákno, je jejich počet přepínačem omezen. V testu je volán úplně stejný příkaz na stejné verzi `FFmpeg` s použitím stejného kodeku. Výsledky měření jsou uvedeny v tabulce 3.3.

Ačkoliv z hlediska funkcionality vše pracuje bezchybně a opravdu lze v jednotlivých vláknech prohlížeče pomocí `JS` spouštět nativní příkazy, **rozdíl výkonu v případě zkompilovaného nástroje `FFmpeg` je obrovský**.

[7] Nastavení by opět mohl aktivovat doplněk instalovaný uživatelem obdobně jako v případě nahrávání obrazovky

[8] Ze zvědavosti chtěli zkusit, zdali je něco takového možného a svoji práci uveřejnili jako knihovnu `videoconverter.js` bez jakýchkoliv testů výkonu. Zdroj: http://bgrins.github.io/videoconverter.js/

[9] `GitHub` repozitář projektu oznamuje více než dva roky od posledního vydání a tudíž se již knihovna pravděpodobně nevyvíjí. Jsou však od autorů k dispozici instrukce jak kompilaci zopakovat.

[10] Zdroj videa například: http://www.quirksmode.org/html5/videos/big_buck_bunny.webm

[11] Za pomocí technologie `Web Workers`. Viz https://html.spec.whatwg.org/multipage/#toc-workers

Prvotní načtení souboru `ffmpeg.js` a jeho následné uložení do cache prohlížeče není z praktického hlediska práce problém. Výkonová propast již však problém je a ačkoliv není konverze na pozadí v odděleném vlákně z hlediska uživatelského dojmu nijak znatelná, reálné využití může spočívat pouze ve velmi jednoduchých transformací videí s nízkým rozlišení.

Pro potřeby zpracování videa v rámci této práce bude nakonec využito **serverové varianty nástroje** `FFmpeg` s vidinou budoucí optimalizace po implementaci zmíněného `CanvasCaptureMediaStream API`.

3.4 Přenos videa

Tak jako v teoretické části, i zde volbu technologií rozdělíme dle architektury komunikace. Obě architektury se doplňují a pokusím se k nim tak uvést předpokládané případy užití.

Client-server architektura

Pro textové i binární přenosy mezi klienty a serverem bude výhradně využíváno technologie `WebSockets` kvůli minimalizaci zpoždění a režii. Odpadá tak nutnost používat techniky jako je `long-polling` (viz teorie 2.5) apod.

Pro zajištění průchodnosti proxy serverů a ostatních mezilehlých prvků bude komunikace probíhat výhradně přes zabezpečenou variantu `WSS` (vysvětlení viz teorie 2.5).[12] Binární data (typicky zaznamenané video) budou rozdělena na menší zprávy, které se budou postupně odesílat. To sníží čekání po dokončení nahrávání a zároveň minimalizuje `head-of-line` blokování (viz kapitola 17 knihy [16]).

`WebSockets` budou použity taky pro ruční implementaci signalizačního kanálu `P2P` komunikace. Důvodem je zbytečná komplikace řešení použitím tradičních protokolů jako je `SIP` nebo `Jingle`, které nabízejí mnoho, pro tuto práci, zbytečné funkcionality.

Vzhledem k tomu, že `client-server` komunikace není hlavním záměrem této práce, **bude pro usnadnění vývoje použita knihovna** `socket.io`.[13] Ta nabízí abstrakci jednotlivých spojení na úroveň místností s podporou `broadcast` zpráv, streamování binárních dat apod.

V případech, kdy bude výhodné využít `cache` prohlížeče (či po cestě) nebo nativní komprimaci přenášených dat `HTTP` protokolem, bude použito místo `WS` technologie `XHR`.

Peer-to-peer architektura

Distribuce částí videí mezí uzly bude zajištěna nad datovým kanálem ustaveným pomocí různých `API` prohlížeče. Pro spojení uzlů se jedná o `RTCPeerConnection API`, které zajistí komunikaci napříč sítí s využitím `STUN` a `TURN` serveru. Dále `RTCDataChannel API` zajistí výměnu datových paketů pomocí protokolu `SCTP`, který lze s výhodou nastavit do režimu **spolehlivého doručování bez zajištění pořadí** (viz část 2.5), což v architektuře `peer-to-peer` nezpůsobuje komplikace. Tím lze očekávat vyšší rychlost přenosu.

Vzhledem k povaze distribuovaných dat (videa uložená na centrálním serveru), **není nutné konfigurovat** `TURN` **server** pro přeposílání dat mezi uživatele v případě nesměrovatelného spojení mezi uzly. Pokud se spojení nepodaří ustavit, stáhnou se potřebné části

[12] Ostatně jako veškerá komunikace v rámci této práce. Práce s potenciálně citlivými daty uživatele vyžaduje adekvátní zajištění důvěryhodnosti.

[13] Serverová část knihovny pracuje nad platformou `Node.JS`. Viz http://socket.io/

z centrálního serveru pomocí WS úplně stejně jako by se stáhly v případě, kdy nebude online žádný uzel nabízející potřebná data.

Po uskutečnění přímého spojení mezi uzly může přebrat funkci signalizačního kanálu a ještě tak snížit nároky na centrální server. Signalizace přes WS by se tak použila pouze než se naváže přímé spojení mezi uzly.

Navržená architektura se tak podobá P2P CDN (content-delivery network) síti pro distribuci videa, jejíž největší výhodou je **minimální cena doručení pro distributory obsahu**. Současně se taková architektura škáluje dle popularity obsahu a ideálně se tak vypořádává se situacemi, kdy je škálovatelnost potřeba. A to tedy při nárazovém zájmu o určitý obsah. A v případě, kdy nelze nebo není výhodné P2P architekturu využít, není dotčena dostupnost obsahu, akorát je obsah distribuován s vyššími provozními náklady.

Sekvenční diagram 3.3 zjednodušeně zachycuje typickou situaci, kdy uživatel A odešle nahrávku na server a chvíli nechá prohlížeč otevřený. Postupně se připojují pozorovatelé B, C a D. Uzel A později ukončí svoji činnost nečekaně ztrátou spojení, kdežto uzel B před zavřením prohlížeče odešle serveru informaci o ukončení spojení, čímž jeho adresu přestane server dále nabízet. Poslední uzel znázorňuje situaci, kdy existuje (jediný) peer se žádanými daty, ale nelze s ním navázat přímé spojení. Proto se data stáhnou ze serveru (obdoba TURN). Pro přehlednost je záměrně vynecháno dělení přenášených dat do více částí.

Peer-to-peer přenosy pro optimalizaci přenosového pásma serveru bude možné zatím využít pouze v prohlížečích dle tabulky 3.4. Ostatním prohlížečům jednoduše server vždy vrátí v seznamu uzlů pouze sám sebe a nepodporovaný prohlížeč tak data stáhne některou z client-server metod.

3.5 Přehrávání videa

Prohlížeče umožňují HTTP streaming videa pomocí progresivního stahování velice dobře a transparentně (více viz teorie 2.6). Zpočátku je nutné stáhnout několik prvních bajtů metadat jako je trvání videa, rozlišení či použité kodeky. Poté až interakce uživatele s video přehrávačem iniciuje další požadavky na další části dat videa.[14]

Tento mechanismus ovšem transparentně pracuje nad architekturou client - server. Pro tuto práci se budeme snažit realizovat obdobné řešení, **ovšem nad architekturou kombinovanou s P2P přenosy.** Samotné přehrávání videa bude zajišťovat stále element <video> a patřičná JS API. Avšak získávání jednotlivých částí videa, pořadí jejich stahování s ohledem na sekvenční charakter přehrávání videa a zároveň nesekvenční charakter P2P sítí, skládání a přehrávání neúplného streamu, kontrola integrity a další dílčí části je nutné implementovat ručně.

Skládání jednotlivých částí videí bude zajišťovat Media Source Extensions API, které podporuje nadmnožina prohlížečů podporujících potřebné WebRTC.[15]

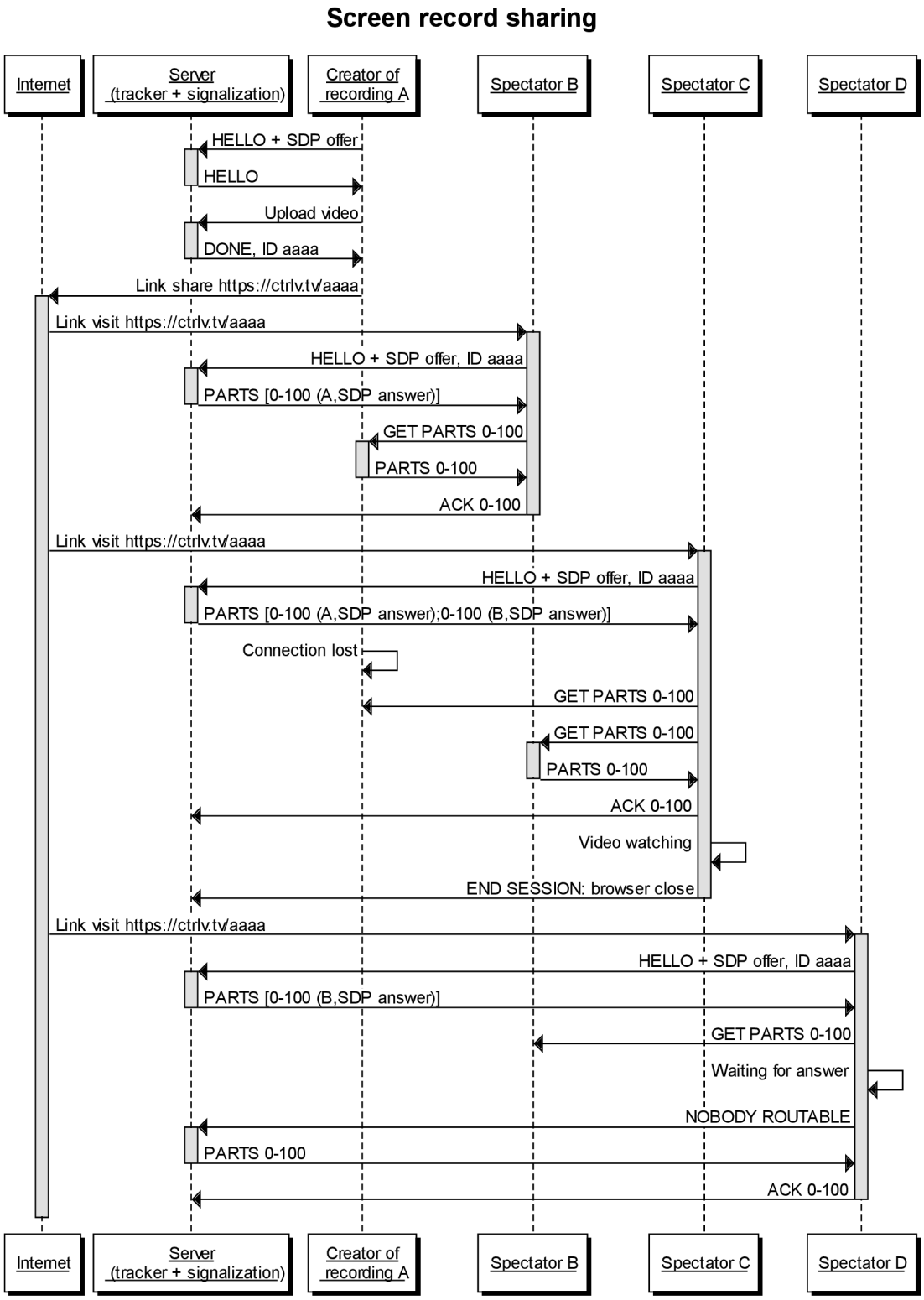
Kontejnery a kodeky

Vzhledem k tomu, že všechny prohlížeče podporující kodek VP8 v novějších verzích podporují také kodek VP9, bude nahrávání probíhat vždy v co nejvyšší možné verzi kodeku (zatím jen Chrome 49 umí VP9, ale toto se pravděpodobně velice brzo změní).

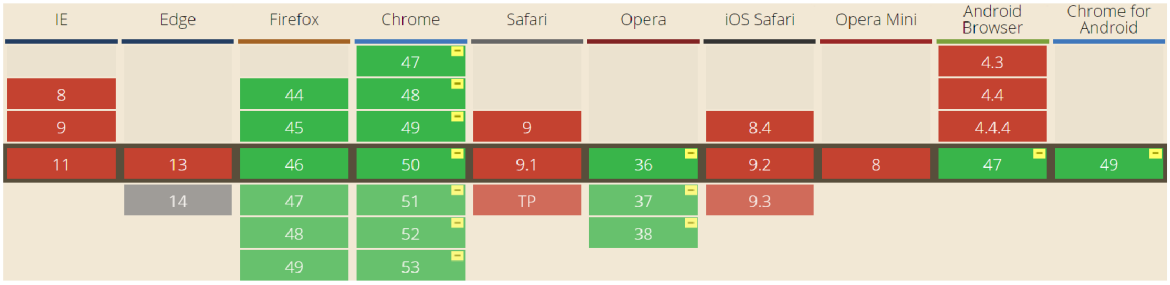
Mezi prohlížeče, které formát WebM nepodporují patří IE, Edge kromě nejnovější verze 14 a veškeré Safari. Pro tyto bude nutné překódovat (na straně serveru, viz část 3.3) videa

[14] Toto je často standardní chování, které lze ovládat atributem metadata elementu <video>.

[15] Viz srovnání http://caniuse.com/#search=MSE a http://caniuse.com/#search=webrtc.



Obrázek 3.3: Návrh komunikace mezi uzly P2P spojení.



Obrázek 3.4: WebRTC – podpora v současných prohlížečích (celkem 57 %). Procentuální pokrytí vychází z dat `StatCounter GlobalStats` pro duben 2016. Zdroj: http://caniuse.com/

do `MPEG4`. Současně však tyto prohlížeče nejsou zajímavé z pohledu `WebRTC`. **Toto má důležitý důsledek v tom, že není nutné distribuovat více verzí stejného videa** (`WebM` i `MPEG4`) a tím dále mělnit potenciální uzly nabízející jen svoji jednu verzi. Všechny prohlížeče, které podporují `WebRTC` nutně podporují současně i `WebM` a tudíž by `MPEG4` verze nebyly komu distribuovat, protože takové uzly se nemají do `P2P` sítě jak připojit.

Bylo by však možné videa nahrávat ve formátu `WebM`, na serveru je převádět do `MPEG4` a v rámci aplikace kompletně používat pouze `MPEG4`. Z hlediska `P2P` distribuce by více uzlů nepřibylo, ale odpadala by práce s přehráváním dvou typů formátů. Z hlediska podpory lze `MPEG4/H.264` přehrát téměř všude, ale nelze jej použít pro komerční účely a je nutné u některých prohlížečů spoléhat na externí dekodéry (viz část 2.6). V neprospěch volby formátu `MPEG4` dále hraje to, že ve srovnání podpory kodeků nové generace (tedy `VP9` a `H.265`) jednoznačně vede `VP9` (viz tabulka 2.3).

3.6 Souhrn

Z hlediska podpory prohlížečů lze v současné době realizovat jednotlivé části následovně:

- Záznam obrazovky: `Firefox` + `Chrome` + `Opera` **s nutností instalace jednoduchých rozšíření**. Použít pro záznam `VP9` kde výkon dovolí, jinak `VP8`.
- Zpracování videa: `Firefox` nejnovější (verze 41, standardně **deaktivováno**), ostatní techniky z hlediska výkonu nepraktické.
- Přenos videa `WebSockets`: všechny prohlížeče.
- Přenos videa `P2P`: `Firefox` + `Chrome` + `Opera` + `Android Chrome`.[16]
- Přehrávání videa `WebM (VP8/9)`: `Firefox` + `Chrome` + `Opera` + `Android Browser` + `Android Chrome`.
- Přehrávání videa `MPEG4 (H.264)`: všechny prohlížeče (kodek `H.265` však žádné).

Aplikace tedy bude schopna získat záznam obrazovky ze dvou nejpoužívanějších prohlížečů. Optimalizovaně ho mezi nimi distribuovat pomocí `P2P` sítě navíc včetně `Opery` a `Android Chrome`. A přehrát video otevřením odkazu dokáže kterýkoliv prohlížeč díky záložnímu překódování do `MPEG4`.

Pokud vycházíme z uváděných statistik prohlížečů zdroje `StatCounter`, optimalizovaná distribuce se týká zhruba 62 % uživatelů. Pokud dále uvažujeme zhruba 80% šanci ustavení

[16] `Android Browser` neumí, pro tuto práci, stěžejní část `WebRTC` – `RTCDataChannel`. `Android Chrome` ano.

přímého P2P spojení, lze až 50 % datových přenosů serverů podobné služby ušetřit. To však záleží na popularitě obsahu, setrvání uzlů na otevřené webové stránce a mnoho dalších aspektech, které budou diskutovány po implementaci práce.

Lze si povšimnout, že ačkoliv MPEG4 dominuje podporou, pro novější technologie není používán. Pro potřeby této aplikace tak jednoznačně vyhrává mladý formát WebM.

Bohužel, pro přístup k obrazovce uživatele, v současné době nezbývá, než vyžadovat instalaci jednoduchého rozšíření, které zpřístupňuje pokročilé API prohlížečů vývojářům. To se však může kdykoliv změnit. V případě prohlížeče Mozilla Firefox existuje naděje na zařazení služby mezi povolené ve výchozím nastavení prohlížeče přímo výrobcem.

Taktéž je škoda, že CanvasCaptureMediaStream API, otevírající široké možnosti záznamu videa libovolného <canvas> plátna, je zatím prakticky nepodporované. V budoucnu se bude jistě jednat o široce využívaný prostředek pro úpravy videí na klientské straně.[17]

[17] Ve spojení s možností programovat shadery grafické karty přímo v prohlížeči díky technologie WebGL by se jednalo o velice mocnou kombinaci technologií.

Kapitola 4

Implementace

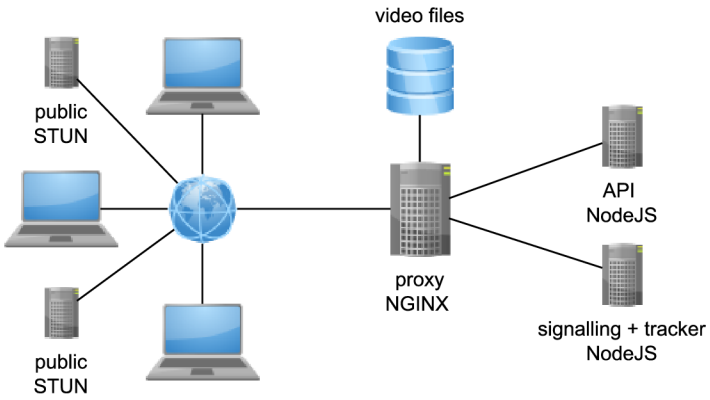
Na základě analýzy v kapitole Návrh řešení pro záznam, přenos a přehrávání videa byla implementována komplexní aplikace umožňující záznam, přehrávání a škálovatelný přenos videa pracovní plochy. V sekci Záznam videa je uvedeno, jak získat přístup k video streamu pracovní plochy implementací rozšíření pro nejrozšířenější prohlížeče. Dále v sekci Zpracování videa následuje rozbor problémů, kterými trpí získaná videa s ohledem na zpětné přehrávání.

Hlavní část implementace však začíná sekcí Distribuce videa, kde je krok po kroku uvedena vlastní implementace **peer-to-peer hybridní sítě** pro distribuci videozáznamů. Následuje jejich přehrávání popsané v sekci Přehrávání videa a zpětná distribuce dále v části Poskytování videa.

Poté je implementace postupně rozšířena o prioritní zpracování pro zajištění `VoD` služeb, což shrnuje sekce Video-on-Demand služby. Kapitolu uzavírají podrobnosti datového spojení uzlů sítě v části Spojení uzlů.

4.1 Infrastruktura

Aplikace se skládá ze dvou nezávisle běžících `NodeJS` serverů, `NGINX` proxy serveru, `STUN` serveru a jednotlivých klientským stanic interpretujících sadu skriptů pro webové prohlížeče (viz obr. 4.1).



Obrázek 4.1: Implementovaná infrastruktura aplikace.

Pro velmi rychlé odbavování požadavků na statické soubory je nakonfigurován výkonný NGINX HTTP server, který požadavky na uložené videosoubory odbavuje bez potřeby dalšího backend zpracování. Nativní podpora pro HTTP Range požadavky splňuje podmínky pro progresivní stahování (viz část 2.6) videozáznamů. V případě nedostatečného výkonu jednoho serveru (typicky I/O operace) lze NGINX server efektivně **horizontálně škálovat** nastavením proxy módu s vyvažováním zátěže.[1]

Dále je implementováno jednoduché API rozhraní pro nahrávání video částí, jejich spojování, generování metadat apod. Toto API je implementováno v NodeJS v kombinaci s Express frameworkem jako HTTP API server. Volba technologie NodeJS zakládá na vysokém podílu I/O operací, kdy lze velmi výhodně využít asynchronnost programování v NodeJS a dosáhnout tak velmi vysokého výkonu.

Druhý NodeJS server slouží jako WebSocket server založený na real-time engine socket.io zjednodušující oboustrannou komunikaci mezi prohlížečem a serverem. Slouží jako **vlastní implementace signalizačního serveru** pro WebRTC P2P komunikaci. Kromě funkce signalizace pro navázání komunikace taky zastává roli trackeru v implementované P2P síti. Odbavuje tak dotazy na dostupnost částí videosouborů a rozděluje práci uzlům na základě definovaných strategií.

Ačkoliv by oba servery mohly být implementovány jedinou NodeJS aplikací, z důvodů odlišné funkcionality a budoucí škálovatelnosti jsou provozovány odděleně.

Klientské stanice pomocí webových prohlížečů interpretují aplikaci, která zajišťuje jádro implementované funkcionality. Kromě záznamu a odesílání videa z klientské stanice zajišťuje především decentralizovanou řídící logiku pro distribuované šíření částí videosouborů během přehrávání.

STUN servery jsou nezbytné pro úspěšné navazování komunikace v rámci sítě Internet. Pomáhají ustavit komunikaci mezi uzly používajícími NAT (jak viz část 2.5). Jsou použity veřejně dostupné servery firmy Google s možností budoucího provozu serveru vlastních.

ECMAScript 6

Z použitých technologií vyplývá, že je programový kód celé aplikace v jazyce JavaScript. S výhodou je využíváno sdílení stejných částí kódu mezi klientskou a serverovou stranou (například datové struktury). Implementace s vysokou mírou využívá nejnovějšího standardu ES6 pro přehlednější a pohodlnější programování.

Na straně NodeJS není z hlediska podpory u většiny konstrukcí nutné využívat předkompilaci do starší verze JS.[2]

Pro webové prohlížeče je však zdrojový kód kompilován do nižší verze ES5 pro zajištění podpory napříč různými výrobci a verzemi prohlížečů.[3] Je použit kompilátor Babel, jehož popis je uveden v teoretické části (viz 2.2).

Nejen kvůli nutnosti s každou změnou zdrojového kódu vykonat kompilaci pomocí Babelu, bylo nutné použít automatizační nástroj. Je zvolen nástroj Gulp v konfiguraci s automatickým sledováním zdrojových souborů. V případě potřeby vykoná posloupnost (stream) nakonfigurovaných kroků jako je spojení menších souborů do jednoho balíku

[1] Například dle prefixu IP adresy nebo geografické polohy.

[2] Výjimkou je například nepodporovaný nový způsob využívání modulů a je nutné používat starý CommonJS formát.

[3] Toto na serverové straně odpadá – stačí aktualizovat NodeJS na nejnovější verzi na jednom místě.

(`bundle`)[4] nebo minifikaci.[5] Nakonec je zahrnuta právě kompilace `ES6` do `ES5` pomocí kompilátoru `Babel` (více viz 2.2).

Stejně lze automatizovat spouštění testů. Sekvenci naprogramovaných úkonů lze buď spustit manuálně pomocí `gulp taskname` nebo lze definovat `gulp.watch` nad potřebnými soubory. Pak je celá sekvence úkonů spouštěna automaticky při změně některého ze zdrojových souborů.

Asynchronní programování

Pro maximální výkon aplikace je ve velké míře využíváno vlastností **událostmi řízeného modelu souběžného vykonávání kódu** v `JS`. Veškeré operace, které by mohly blokovat vykonávání vlákna,[6] obdrží `callback`, který je volán po ukončení čekání na danou operaci. Mezitím smyčka událostí (`event-loop`) zpracovává další požadavky v pořadí.

Aby nedocházelo k zanořování `callbacků`, je všude využíván návrhový vzor `Promise` (viz část 2.2). Téměř nikde v kódu aplikace tak nelze pozorovat klasické volání `return` z funkcí a následné použití navrácené hodnoty. Většina sekvenčního volání funkcí je od jejich dokončení časově posunuta.

Takové přerušení běhu skriptu a čekání na asynchronní událost s sebou však nese problém zamrznutí aplikace v případě ztráty události či neodchycení chyby (viz kód 4.1). Na několika místech v aplikaci tak lze pozorovat uměle přidané časovače, které po vypršení vyvolají chybový `callback` a vykonávání pokračuje. Typicky situace nastává při ztrátě očekávaného paketu[7] nebo spojení.

Ukázka kódu 4.1: Časovač jistící asynchronní volání

```
1 return new Promise((resolve, reject) => {
2   signalling.connect()
3     .then(() => resolve('signalling-done')) // zachycen korektni stav
4     .catch(() => reject('signalling-error')); // zachycen chybovy stav
5   setTimeout(() => {
6     reject('signalling-timeout'); // dlouho se nic nedeje, nezachyceno
7   }, Config.signalling.timeout);
8 });
```

Finální sekvence zpracování

Celá aplikace tvoří komplexní systém s mnoha logicky oddělenými bloky, které spolu asynchronně spolupracují. Ačkoliv nelze takto komplexní strukturu jednoduše shrnout do sekvenčního diagramu, účelem schématu 4.2 je tyto bloky znázornit v abstraktním sekvenčním pojetí. Jednotlivé sekce této kapitoly dále aplikaci popisují v podobném sledu.

4.2 Záznam videa

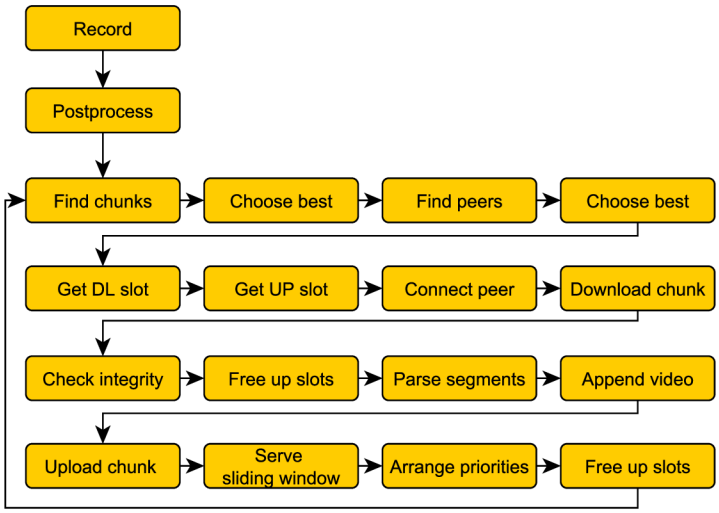
V návaznosti na návrh řešení záznamu videa (viz část 3.2) jsou naprogramovány dvě rozšíření webových prohlížečů. Ty umožňují této aplikaci přístup k záznamu pracovní plochy uživatele. V jiných prohlížečích nežli Firefox, Opera a Chrome nelze záznam získat.

[4] Protože nejrozšířenější prohlížeče paralelně zpracovávají maximálně 6 požadavků (viz kniha [21]).

[5] Zmenšení zdrojových kódů pomocí vynechávání bílých znaků a nahrazovaní opakujících se bloků bez dopadu na funkčnost. Není nutné kód deminifikovat.

[6] což je v případě jediného vlákna pro vykonávání JS skriptu kritické

[7] Především v režimu `SCTP unreliable`, ale i při náhlém tvrdém odpojení uzlu apod.



Obrázek 4.2: Abstraktní pohled na aplikaci z hlediska sekvence samostatných bloků.

Chrome Extensions

Od verze 34 je oficiálně doporučováno používat `chooseDesktopMedia API` podporované v rámci rozšíření prohlížeče. Rozšíření jsou programována taktéž v jazyce `JavaScript` a instalují se pomocí `Chrome Web Store`. Rozšíření umožňují vývojářům využívání jinak nepřístupných `Chrome API` (tzv. `chrome.* APIs`).

Například tak lze vyvolat (kód 4.2) stejný dialog volby sdílení obrazovky jako v dřívějších verzích prohlížeče. V tomto dialogu (obr. 4.3) může uživatel zvolit kterou obrazovku, případně které konkrétní okno chce sdílet. Tato volba je pro každé volání neměnná. Volání vrací unikátní stream identifikátor, jehož znalost umožňuje přistupovat k danému streamu pomocí speciální constraints položky `chromeMediaSourceId`.

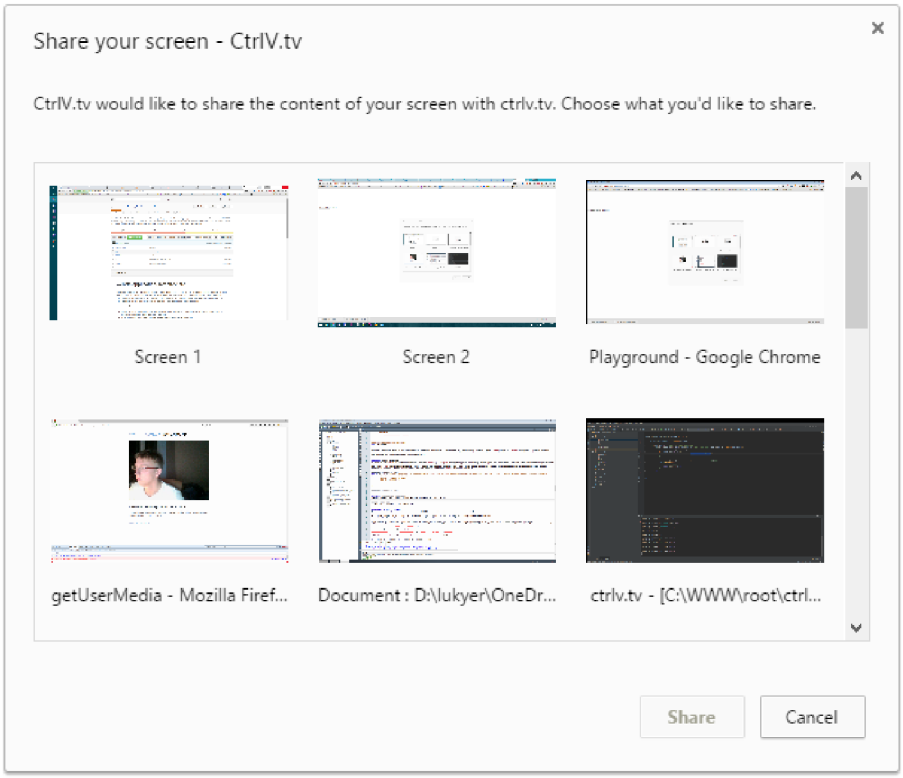
Ukázka kódu 4.2: API chooseDesktopMedia volané v rámci rozšíření.

```
chrome.desktopCapture.chooseDesktopMedia(
        ['screen', 'window'], null,
        function (streamid) {
        ...
    }
);
```

Komunikace s rozšířením Rozšíření tvoří dvě části:

- **Content script** je `JavaScript` vykonávaný v rámci kontextu stránky, do které je načten. Může číst a manipulovat s `DOM` aplikace, komunikovat s druhou částí rozšíření, ale samostatně nemůže volat jinak nepřístupné `Chrome Extension APIs`.
- **Background script** je hlavní část rozšíření s přístupem ke speciálním `Extension APIs`. Na rozdíl od `content script` nemůže nijak zasahovat do samotné aplikace.

Pro komunikaci mezi těmito dvěma části, stejně tak jako pro komunikaci mezi webovou aplikací a rozšířením samotným, lze využít `postMessage API`. Toto `API` slouží k mezi doménové (`cross-origin`) komunikaci, která jinak není z bezpečnostních důvodů povolena (viz část 2.3). Funguje na principu zasílání zpráv a obsluhou jejich přijetí.



Obrázek 4.3: Google Chrome – vyvolání UI dialogu výběru obrazovky/okna ke sdílení.

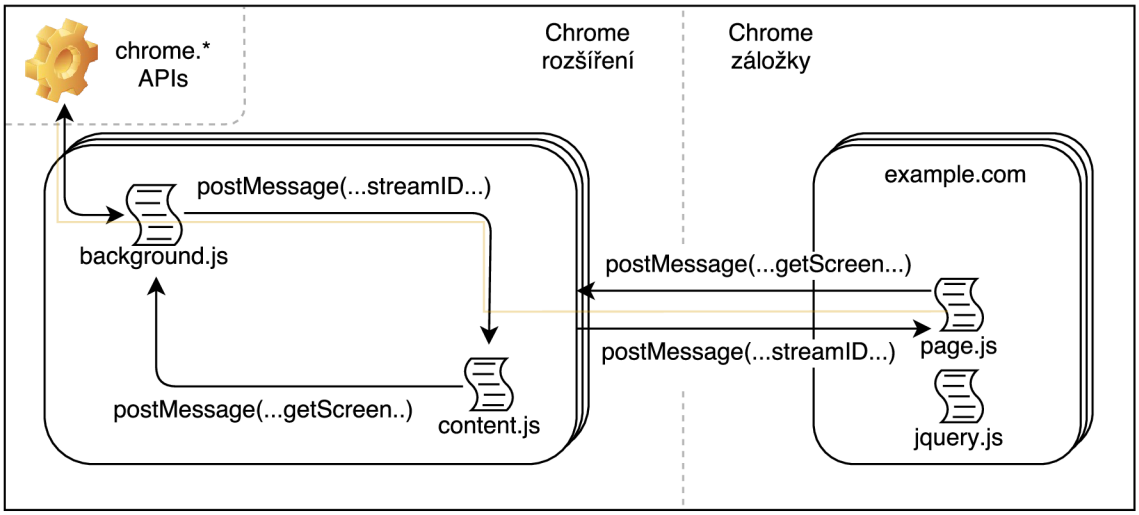
Příklad mezi doménové komunikace je uveden v kódu 4.3. Kód současně demonstruje nutnou obezřetnost při ověřování původu zpráv, protože se jedná o komunikaci mezi jakýmikoliv webovými aplikacemi a zprávy tak může zasílat kdokoliv. Použitím tohoto `API` programátor **úmyslně porušuje** standardní `sandbox` prohlížeče a přejímá bezpečnostní rizika s tím spojená.[8] I zdánlivě bezpečné předávání zpráv do skriptem vytvořeného okna (řádek 3) musí znovu definovat povolenou `URL`, protože vytvořené okno může svoji adresu (uživatelem nebo útočníkem) změnit.

Ukázka kódu 4.3: Cross origin komunikace s využitím `postMessage API`.

```
1  /* domena1.com:8080 */
2  var popup = window.open("domena2.com", ...);
3  popup.postMessage("Tajny klic je 12345", "domena2.com");
4  window.addEventListener("message", function(event) {
5    if (event.origin !== "http://domena2.com") return;
6    // Komunikace ustavena!
7  }, false);
8
9
10 /* domena2.com */
11 window.addEventListener("message", function(event) {
12   if (event.origin !== "http://domena1.com:8080") return;
13   event.source.postMessage("Tajny klic obdrzen", event.origin);
14 }, false);
```

[8] Podrobné bezpečnostní důsledky a patřičná oprávnění jsou uvedeny ve specifikaci https://html.spec.whatwg.org/multipage/comms.html#security-postmsg.

Diagram 4.4 ilustruje popisovanou komunikaci mezi aplikací, rozšířením a běžně nepřístupným `JavaScript API` prohlížeče `Chrome`. Oranžová čára značí cestu získaných dat (dále přístupný stream identifikátor) od `API` až do skriptu webové aplikace.



Obrázek 4.4: Google Chrome – vyvolání UI dialogu výběru obrazovky/okna ke sdílení.

Firefox extensions

Rozšíření se, podobně jako pro Chrome, programují v jazyce `JavaScript` a umožňují programátorům přístup ke speciálním `API`. Jedno z takových (`sdk/preferences/service`) lze využít pro automatické nastavení prohlížeče uživatele.[9] Není nutné využívat `background script` běžící na pozadí, ani `content script` běžící v kontextu konkrétní webové aplikace. Postačí využití definice skriptu vykonávaného při povolení rozšíření či restartu prohlížeče (`main script`).

Pro potřebu schválení rozšíření je nutné dodržovat tzv. `AMO` politiky.[10] Pro případ úpravy konfigurace prohlížeče je například nutné zajistit vrácení nastavení do původního stavu při deaktivaci rozšíření.

Instalace rozšíření

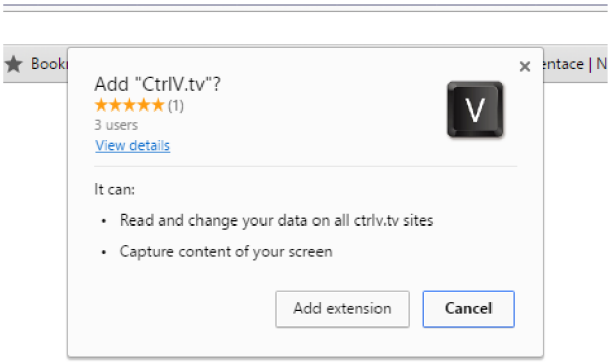
Po implementaci, tvorbě výsledného balíčku, odeslání ke schválení a následném podpisu rozšíření výrobcem prohlížeče, je možné rozšíření instalovat do prohlížečů uživatelů klasicky, jak jsou zvyklí přes obchody s rozšířeními.

Z hlediska uživatelského komfortu je však implementována detekce, zdali je již rozšíření dostupné a pokud ne, je uživatel při interakci s aplikací (tedy když chce začít nahrávat video) odkázán na instalaci s vysvětlením (obr. 4.5).[11]

[9] Podrobnosti včetně ukázky použití uveden v oficiální dokumentaci: https://developer.mozilla.org/en-US/Add-ons/SDK/Low-Level_APIs/preferences_service

[10] Viz https://developer.mozilla.org/en-US/Add-ons/AMO/Policy

[11] Je obecně doporučovaný fakt vyžadovat oprávnění/interakci uživatele až potom, co on sám interakci započal. Pro úplnost – `Chrome` vyžaduje kliknutí uživatele před instalací, `Firefox` ho nevyžaduje, ale zobrazuje varování v obou případech.



Obrázek 4.5: Google Chrome – vyvolání UI dialogu instalce rozšíření přímo na stránce.

Parametry záznamu

Získaný záznam má variabilní parametry dle rozlišení obrazovky či vybraného okna. Maximální povolené rozlišení je zvoleno jako 1920x1200 při 30 FPS s případným přepočtem z rozlišení vyššího. Záznam je kódován do formátu `WebM` s video kodekem `VP8` (volba viz část 3.5) a případně audio kodekem `Vorbis` (jen `Firefox`).

Datový tok videa je specifikován konfigurační direktivou na výchozí hodnotu 2 Mb/s. Dle specifikace[12] se však jedná **pouze o doporučení** pro cílový enkodér, který může s hodnotou naložit dle uvážení. Pro prohlížeče je v praxi tato hodnota velmi orientační a průměrný variabilní datový tok specifikovaný tímto parametrem mnohdy není dosažen nebo je překročen. Proto je od uživatele získáváno raději video s vyšším datovým tokem s možností jeho snížení při překódování na straně serveru.

`MediaRecorder API` dovoluje specifikovat `timeslice` parametr definující přibližnou délku videa, po které bude aplikaci dosud zaznamenaná část zpřístupněna (`chunk`). Pro optimální využití paměti `RAM` uživatele (jednou odeslaná data jsou z paměti uvolněna) a využití přenosového pásma již během záznamu je výchozí hodnota nastavena na 1 s.

4.3 Zpracování videa

Původní představa byla, že zpracování videa na straně serveru[13] bude sloužit pro dodatečné funkce (ořez, vodoznak, ...) nebo překódování videa z formátu `WebM` do `MP4` pro zpřístupnění videí uživatelům IE/Edge/Safari.

Bohužel se však během práce s nahranými videi objevilo mnoho komplikací způsobujících problémy s jejich přehráváním. V mnoha případech byly poškozeny nejen záznamy z prohlížeče Firefox (viz část 3.2), ale také prohlížeč Chrome neuměl po sobě často video přehrát.

Ukázalo se, že je velký rozdíl mezi přehráváním záznamů přímo v prohlížeči (odkaz na video) a načtením videa v rámci webu po částech pomocí `MSE`. V prvním případě se prohlížeče s chybnými záznamy často vyrovnaly a chybné úseky přeskočily s trhnutím obrazu, vykreslením artefaktů nebo vkládáním jednobarevných snímků. V druhém případě však video často nešlo vůbec přehrát nebo se zaseklo a přes určité části nešlo přehrát. Obdobně dopadalo přehrávání i v jiných přehrávačích, včetně `VLC`.

[12] https://w3c.github.io/mediacapture-record/MediaRecorder.html
[13] klientská strana byla zavržena, viz návrh 3.3

Existuje několik otevřených bugů napříč jednotlivými verzemi prohlížečů[14] poukazující na to, že rané implementace `MediaRecorder`u obsahují chyby a přehrávače se pouze více či méně dokáží z těchto chyb zotavovat.

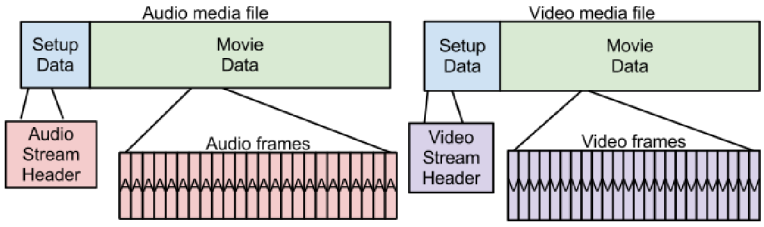
V rámci implementace jsem experimentoval s postupy k opravě videozáznamů tak, ať je lze jednou zpracovat při nahrávání a pak distribuovat k uživatelům pokud možno bezchybné.

Jako inspirace sloužilo video `test.webm` v uveřejněné ukázce spojování částí `WebM` videa za pomoci `Media Source API`.[15] Toto video šlo bezchybně přehrát všude, šlo v něm posouvat a především – bylo možné jej připojovat do přehrávaného bufferu po částech v libovolném pořadí voláním `SourceBuffer.appendBuffer(part)`.

Chybějící elementy

Formát `WebM` je založený na kontejneru `Matroška`. Analýzou vnitřní struktury[16] nahrávaných souborů bylo zjištěno, že se jedná o typ „Live Streaming“ určený pro online streaming videa. Typ je specifický v tom, že může mít teoreticky neomezenou délku.

Některé elementy však nejsou u tohoto typu dostupné. Jedná se především o `Meta Seek`, `Cues`, `Chapters` a `Attachments`. Implementace `MediaRecorder API` tedy do záznamů nepřidává některé potřebné elementy (`Cues` a `SeekHead`) pro zpětné přehrávání a veškeré clustery umisťuje do jediného segmentu (element `Segment`). Lze se domnívat, že byl použit tento typ kontejneru z důvodu **předem neznámé délky** nahrávání s možností odesílání částí záznamu již v jeho průběhu.[17]



Obrázek 4.6: Požadavek na formát kontejneru pro Video on Demand. Zdroj: http://wiki.webmproject.org/adaptive-streaming/webm-vod-baseline-format

Referenční fungující video obsahuje navíc `Cues` a `SeekHead` elementy, což umožňuje korektní posuv ve videu, jeho opakované přehrávání či bezchybné pozastavování. Současně přítomnost těchto elementů eliminuje problémy samovolného zastavování videa či dokonce potlačuje některé artefakty. `MSE` tak zjevně není připraveno na přehrávání `Live Streaming` typu kontejneru videí. Současně však tento formát prohlížeče produkují v rámci záznamu pomocí `MediaRecorder API`.

Je tedy nutné nahrávaná videa zpracovat a **dopočítat chybějící elementy**. Ve zdrojových kódech knihovny `libwebm`[18] se nachází nástroj sample_muxer[19] po jehož kompilaci

[14] Například https://bugs.chromium.org/p/chromium/issues/detail?id=606000 nebo https://bugzilla.mozilla.org/show_bug.cgi?id=969290.

[15] Odkaz: http://html5-demos.appspot.com/static/media-source.html

[16] Data jsou uložena ve formátu `EBML`, což je binární rozšíření `XML`. Většina nástrojů pro práci s `WebM` tak pracuje s formátem `XML`. V implementované `JS` aplikaci je však nativním formátem `JSON`.

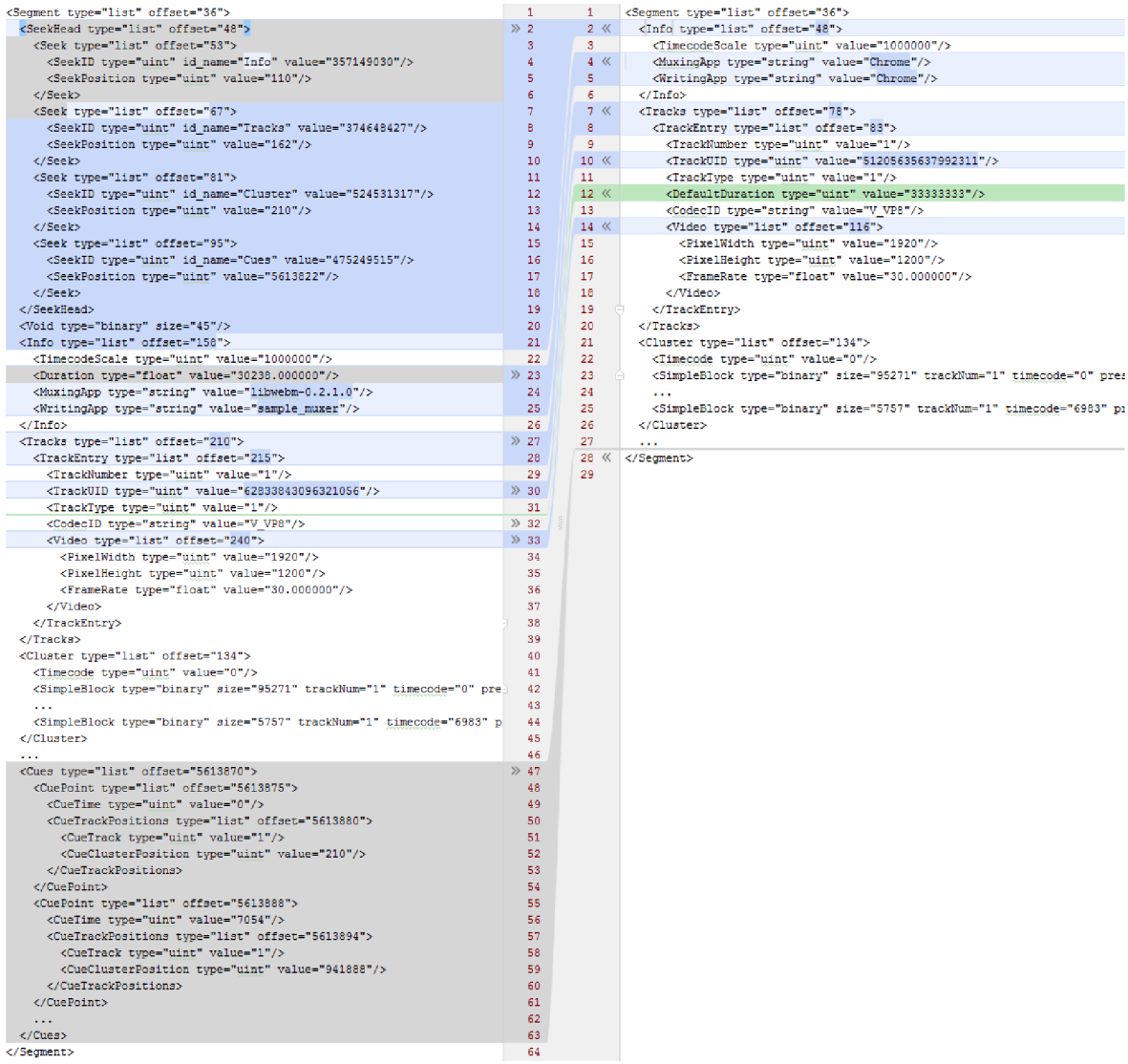
[17] Požadavek na formát kontejneru pro `Video on Demand` však nahraná videa splňují (obr. 4.6)

[18] K dispozici zde: https://github.com/webmproject/libwebm

[19] A kromě něj i několik dalších užitečných nástrojů pro práci s `WebM` jako například webm_dump pro zde uvedené výstupy struktur kontejnerů.

lze voláním sample_muxer -i recorded.webm -o fixed.webm opravit vstupní videosoubor tak, **aby obsahoval korektně vložené chybějící elementy** `Cues` a `Meta Seek` (obr. 4.7).

Zpracování je rychlé, protože nedochází k překódování vstupního videa. Kromě nápovědy příkazu však nebyla nalezena žádná dokumentace ani ukázky použití. Nástroj nabízí několik přepínačů, jejich studium je však podpořeno pouze přístupnými zdrojovými kódy. Nutno podotknout, že nástroje jako `ffmpeg` nedokáží korektně zarovnat `clustery` a vygenerovat `Cues` tak, aby byly v kombinaci s `MSE` použitelné.



Obrázek 4.7: Kontejner nahraného videa s chybějícími elementy (vpravo) a opraveného videa (vlevo).

Počet clusterů

Opravené video s přidanými a korektně zarovnanými `Cues` je dostatečné pro **sekvenční načítání videa** do `MSE` bufferu část po části. Nezáleží přitom na velikosti jednotlivých částí, stačí dodržet podmínku sekvenčního přidávání. Toto je dostatečné pro většinu typických

aplikací adaptivního streamingu pomocí `MSE`.[20]

V implementované aplikaci však chceme využít **nesekvenčního charakteru** stahování v rámci implementované `P2P` sítě. Současně chceme umožnit uživateli libovolný posuv ve videu s okamžitou odezvou a vynecháním stahování nepotřebných částí.

Bylo vypozorováno, že je pro takovou implementaci nutné video vhodně **rozčlenit na clustery**, které mohou být přehrávány `samostatně` (obsahují na začátku klíčový snímek). Čím více clusterů, tím větší granularita posuvu ve videích a také více elementárních jednotek pro následnou distribuci. S počtem klíčových snímků však roste režie videí a klesá účinnost komprimace.

Další problém byl zpozorován při záznamu videa v prohlížeči `Firefox`. Vytvořený videosoubor někdy neumí nástroje z knihovny `libwebm` zpracovat a končí chybovou hláškou „Could not add frame“. Později bylo zjištěno,[21] že problém nastává v případě záznamu videa včetně audio stopy (jen `Firefox`, `Chrome` ani neumožňuje). **Nelze tak vygenerovat `Cues` pokud záznam obsahuje audio**.

Překódování videa

Jednotlivé prohlížeče clustery generují libovolně a nebylo vypozorováno na čem to závisí.[22] Ani specifikace parametru `MediaRecorder.start(timeslice)` neovlivňuje tvorbu clusterů, ale pouze velikost binárních objektů částí videa pro přenos sítí. Počet clusterů tedy **nelze přímo ovlivnit** a není výjimkou ani objemná nahrávka obsahující jen několik clusterů (ukázka 4.4).

Ukázka kódu 4.4: Testovaná scéna scene1-ch-vp8-long.webm v `Chrome 50`.

```
{
  "type": "video/webm; codecs=\"vp8\"",
  "live": true,
  "init": { "offset": 0, "size": 134},
  "media": [
    { "offset": 134, "size": 17924379, "timecode": 0.000000 },
    { "offset": 17924513, "size": 40051292, "timecode": 30.008000 },
    { "offset": 57975805, "size": 25087176, "timecode": 60.008000 }
  ]
}
```

Takový soubor je pro tuto aplikaci naprosto nevhodný, protože je nutné stáhnout přibližně 30 s videa najednou[23] a připojit do bufferu až jakmile je stažen celý segment. Taktéž posuv v takovém videu není uživatelsky přívětivý.

Dosud bylo možné video zpracovat velmi rychle. Pro změnu velikosti a počtu clusterů je již nutné video překódovat, což je časově náročná operace. Lze však rovnou normalizovat datový tok videí, snížit jejich velikost (použitím kvalitnějšího kodéru) či například video ořezat. Dalším důvodem pro překódování je problematický záznamu audia v prohlížeči `Firefox` (viz část 4.3), který překódování eliminuje.

Jako první krok po spojení uložených částí nahraného videa tedy následuje volání nástroje `ffmpeg`[24], který zajistí překódování videa s vhodným počtem clusterů pro účely aplikace:

[20] a zmiňovaná veřejná ukázka funguje právě takto

[21] a nahlášeno autorům `libwebm` k prozkoumání

[22] Aktuální verze `Chrome 51` generuje clustery již relativně stabilně co 5–15 s. Empiricky zjištěno.

[23] Později je proto zavedena vyšší abstrakci nad binárními daty pro `P2P` přenos.

[24] Nabízí se možnost překódování clusterů přímo nástrojem `sample_muxer`. Ačkoliv volbu nastavení délky clusteru nástroj nabízí, z hlediska struktury kontejneru clustery opravdu vytvoří, nepodařilo se mi získat

```
ffmpeg -i inFile -g 60 -cpu-used 8 -threads 8 -c:v libvpx -b:v 1M outFile
```

Parametr `-g` určuje skupinu snímků, které tvoří cluster. Zde se tedy vygeneruje cluster pro každých 60 snímků, což odpovídá každým dvěma vteřinám při 30 FPS. Variabilní datový tok 1 Mbit je zvolen pro aplikaci jako optimální (viz část 5).

Takto vytvořené video je dále zpracováno zmíněným nástrojem `sample_muxer` (generace chybějících `Cues` elementů).

Přeskládání kontejneru

Specifikace doporučují umístění bloků `Cues` ze začátku videosouborů pro snazší streaming (jsou získány hned v rámci metadat). Pro jejich přesun je použit doporučovaný nástroj autorů `Matrošky` – `mkclean`.

Ačkoliv nástroj `mkclean` avizuje opravu poškozených či špatně zarovnaných videosouborů včetně generování potřebných bloků podobně jako nástroj sample_muxer, ani tento si bohužel nedokáže poradit se záznamy z `MediaRecorder API`. Mnoho chyb na výstupu a nedokončený export videa byly důvody k ohlášení problému tvůrcům a setrvání u kombinace nástroje sample_muxer pro opravu záznamu a `mkclean` pro přesunutí vygenerovaných `Cues` na začátek video souboru.

Finální sekvence zpracování

Z popisu je zřejmé, že najít **funkční kombinaci správných nástrojů** v kombinaci s různými bugy prohlížečů či samotných nástrojů bylo velice komplikované. Nicméně finální nalezená kombinace zaručuje kvalitní výstup za všech okolností s ideální přípravou pro následnou `P2P` distribuci. Záznamy jsou děleny na krátké samostatně distribuovatelné clustery a **lze je připojovat do přehrávače v libovolném pořadí**. Lze se ve videu libovolně posouvat bez nutnosti stahovat přeskočené části a tím je umožněno stahovat části videa až jakmile jsou opravdu aktuálně potřebné pro přehrávání.

Finální implementovaná sekvence použitých nástrojů je uvedena na obrázku 4.8.

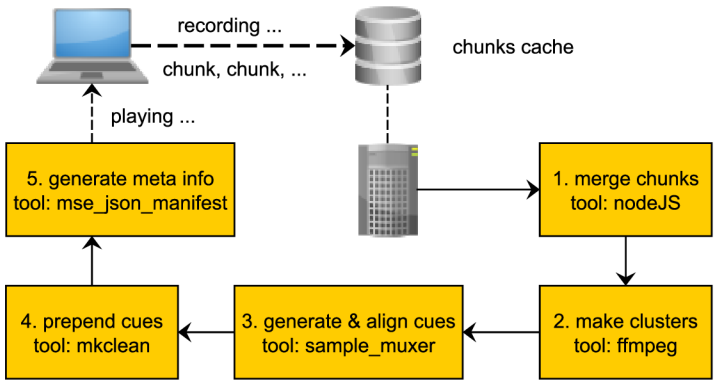
Ukázka kódu 4.5: Ukázka analýzy zpracovaného videa.

```
{
  "type": "video/webm; codecs=\"vp8\"",
  "duration": 85769.000000,
  "startDate": 2016-07-17T12:56:33Z,
  "init": { "offset": 0, "size": 258},
  "media": [
    { "offset": 258, "size": 131997, "timecode": 0.000000 },
    { "offset": 132255, "size": 70913, "timecode": 1.048000 },
    { "offset": 203168, "size": 56683, "timecode": 2.096000 },
    ...
    { "offset": 82167656, "size": 896748, "timecode": 84.937000 }
  ]
}
```

Takto připravené video je možné po clusterech distribuovat a v libovolném pořadí načítat do `MSE` bufferu. Předtím je však nutné připojit metadata videa. Ukázkový výstup nástroje mse_json_manifest[25] uvedený v kódu 4.5 analyzuje rozsah metadat (0–258 B), délku

použitelný výstup. Nástroj nejspíše nevytváří potřebné klíčové snímky (čemuž napovídá i velmi rychlé zpracování kolem 0,3 s v případě testovacího videa) a tudíž je pro naší aplikaci nezajímavý.

[25] Zdroj: https://github.com/acolwell/mse-tools, licence neuvedena.



Obrázek 4.8: Sekvence zpracování uživatelova videa pro následné zpětné přehrání.

trvání a jednotlivé clustery (např. první má rozsah 258–132254 B, druhý 132255–203167 B apod.).[26] Tyto rozsahy lze jednoduše stahovat pomocí `HTTP Range` požadavků (viz část 2.6) nebo distribuovat `P2P`.

4.4 Distribuce videa

Stěžejní částí práce byla implementace **hybridní distribuované** `P2P VoD`[27] **sítě** mezi uživateli sledujícími dříve zaznamenaná videa (kap. 4.2). Tato síť umožňuje vysoké škálovaní při provozu reálné služby.[28]

Kromě standardních výzev v implementaci `P2P` sítě, jako je problematika distribuce dat, strategie výběru uzlů či částí souborů, zajištění integrity nebo vypořádaní se s heterogenními prostředky jednotlivých uzlů, přidává `VoD` služba[29] **nutnost včasného doručování potřebných video clusterů tak, aby nebyl uživatel při přehrávání přerušován**.

To s sebou nese nutnost volby vhodných strategií umožňující vyhodnocení priorit v daném okamžiku. Vzhledem k libovolnému požadavku uživatele na přesun ve videu, je nutné řešit situace, kdy aktuálně připojené uzly potřebné části videa zrovna nemají či se jednoduše náhle odpojily od sítě, protože již video dosledovaly (peer churn). Zároveň je vhodné držet nízký signalizační overhead, aby byl systém dobře škálovatelný a vypořádal se s nárazovým zájmem o některé soubory (flash crowds).

Následují sekce se zabývají jednotlivými výzvami, které implementace této `P2P` sítě přinesly.

Client-server architektura (TURN)

Základní `client-server` `HTML5` přehrávač videa sekvenčně stahuje potřebné části videa dle rozsahů bajtů jednotlivých clusterů (získaných z metadat dle části 4.3) a ty připojuje do `SourceBuffer` bufferu `MediaSource` objektu zvoleného `<video>` elementu.

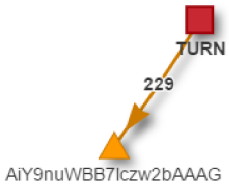
[26] Na ukázce výstupu si lze povšimnout nevalidního `JSON` formátu, kdy je chyba v položce `startDate` a hodnota není uzavřena do uvozovek. To způsobuje problémy `JSON` parseru v `JS` a je nutné tento výstup nejdříve opravit regulárním výrazem. Oprava chyby byla zaslána autorovi nástroje.

[27] `Video on Demand`, tedy streaming videa řízený samotným uživatelem – výběr co chce kdy sledovat s možností přetáčení, pozastavení apod.

[28] S výrazným snížením nákladů vynaložených na konektivitu serveru.

[29] narozdíl od tradičních `P2P` sítí jako je `BitTorrent` nebo `Napster`

Server poskytující tyto části videí je dále označován za TURN server[30] a je použit vždy jako poslední záložní řešení v případě, že **nelze potřebnou část získat** P2P. Má k dispozici vždy veškerá data všech přenášených videí.



Obrázek 4.9: Ukázka úplného stažení videa z TURN serveru, když není k dispozici žádný P2P uzel.

TURN distribuuje metadata (např. kontrolní součty chunků), chybějící části videí v P2P síti a v případě potřeby (žádný uzel) i celá videa (obr. 4.9). Dále je potřeba tohoto serveru při použití prohlížeče, který není schopen participovat v P2P síti. **V následujících sekcích bude odkazem na TURN server myšlen právě takovýto server.** Jedná se o již zmiňovaný (viz část 4.1) nginx proxy server pro velmi rychlou distribuci videí. Jeho dostupnost je stěžejní pro inicializaci P2P přenosu. V analogii se sítí BitTorrent se jedná o neustále dostupný seed konektivitou dostupný pro všechny peery, ovšem s minimální prioritou, protože je jeho přenosové pásmo drahé.

Neustále dostupný seed přináší do P2P sítě několik zajímavých poznatků a umožňuje jednodušší realizaci VoD, protože z něj lze v nejhorším případě[31] vždy potřebná data rychle stáhnout. Jako protiklad lze uvést situaci, kdy BitTorrent při odpojení uzlu s, byť jen několika, unikátními bajty stahovaného souboru znemožní jeho dostahování všem.

Zapouzdření clusterů videa

Clustery jsou různě veliké shluky videodat tvořící samostatně připojitelné jednotky do SourceBufferu. Pro datový přenos je však vhodné pracovat se stejně velikými shluky dat nezávisle na velikosti zapouzdřených clusterů. Implementace tak pracuje s **abstraktní obálkou nad celým videem** (a tedy nejen nad clustery) dělící binární data do tzv. **chunků**. Toto slovo pro jasný popis dále používám v anglickém znění.

Velikost chunku je ve výchozím nastavení stanovena na **32 kB**. Volba této velikosti ovlivňuje počet kontrolních součtů v metasouboru (a tím jeho velikost) a nepřímo ovlivňuje komunikační režii. Větší chunky snižují režii, ale jsou náchylnější k chybám během přenosu, což nese nutnost častějšího znovu odeslání v případě problematického spojení mezi uzly (více viz část 4.8). Menší chunky a tedy jejich větší počet zase nese vyšší paměťové nároky uzlů i signalizačního serveru. Protokol BitTorrent doporučuje držet počet chunků torrentu mezi 1000–1500. S předpokládanou průměrnou velikostí nahrávky kolem 50 MB se zvolená velikost 32 kB blíží doporučovanému rozsahu.[32]

V textu budou tedy nadále zmiňovány dvě datové jednotky – chunk a cluster.[33] Je-

[30] Název pochází z terminologie ICE metodologie značící Relay server pro vždy spolehlivé (ale drahé) zprostředkování dat mezi uzly neumožňující přímé propojení. V našem případě se tak **nejedná** o aplikaci protokolu TURN.

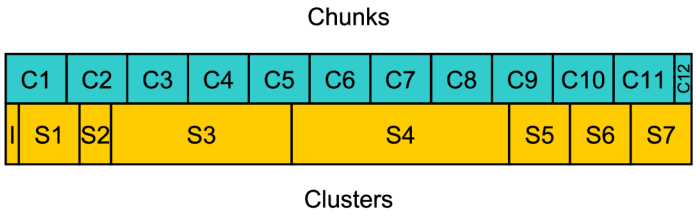
[31] typicky čekání uživatele na načtení další části videa

[32] Velikost chunku je jedna z položek metadat a je tedy možné dynamicky rekonfigurovat uzly dle skutečné velikosti videa. To však zahrnuje dynamickou konfiguraci mnoha dalších konfigurovatelných konstant jako například hlídacích časovačů vypršení apod. To je však mimo rámec této práce.

[33] Ve zdrojovém kódu aplikace jsou clustery výhradně pojmenovány jako „video parts“. Důvodem je vývoj aplikace, kdy kvůli mnoha zmiňovaným komplikacím s převody videa nebylo zřejmé, zdali bude ve výsledku

jich vztah je znázorněn schématem 4.10. Za povšimnutí stojí, že jeden cluster může být rozprostřen přes několik chunků a naopak jeden chunk může zapouzdřovat více clusterů.

Nejmenší stavební jednotkou videa je tedy cluster a nejmenší vyměnitelnou jednotkou po síti je chunk. Jejich vztah je M:N.



Obrázek 4.10: Ukázka zapouzdření video clusterů do chunků. Velikost chunku je 32 kB. I je init část videa.

Metadata videa

Kromě samotného video souboru, jehož chunky jsou libovolně distribuovány v `P2P` síti, je potřeba zavést **doplňující množinu informací** vázající se ke každému videu v síti. V terminologii sítě `BitTorrent` se jedná o `.torrent` soubor, který je distribuován odděleným kanálem. V našem případě se jedná o vygenerovaný `JSON` soubor, který vrací `TURN` server pro všechna nabízená videa. Tento soubor však není třeba nijak distribuovat, protože `TURN` server (`tracker`) používáme jediný.

Soubor obsahuje metadata potřebná pro `P2P` výměnu videodat. Jedná se o:

- informace o videu, jako je použitý kodek, délka trvání, inicializační oblast videa a jednotlivé clustery s jejich bajtovými rozsahy a časovými známkami (viz kód 4.5),
- velikost souboru,
- velikost chunku a pole obsahující `SHA-1` otisk pro každý jednotlivý chunk (více viz část 4.5).

Tyto metadata jsou tedy distribuována zvlášť, vždy formou `client-server` komunikace a tedy z přenosového pásma `TURN` serveru. To zajišťuje jednak jejich okamžité a spolehlivé získání, možnost bezproblémového cachování (odbavuje klasický HTTP server) po cestě a především je možné důvěřovat vypočteným `hash` otiskům chunků (distribuují se výhradně přes `HTTPS`).

Jako další typ metadat lze považovat několik prvních bajtů samotného video souboru. Jedná se tak o metadata kontejneru Matroška obsahující elementy `Header`, `Meta Seek`, `Cues` a další. Tyto metadata jsou označována jako `INIT` video souboru. Ačkoliv by bylo možné je distribuovat `P2P`,[34] kvůli požadavku `MSE` na připojení `INIT` dat jako prvních dat vůbec jsou stahována **prioritně** z `TURN` serveru. Postup tedy vypadá následovně:

1. stáhni meta soubor požadovaného videa z `TURN` serveru,
2. inicializuj potřebné datové a vyhledávací struktury,

základní stavební jednotkou videa opravdu `WebM` cluster. Nakonec tomu tak opravdu je.

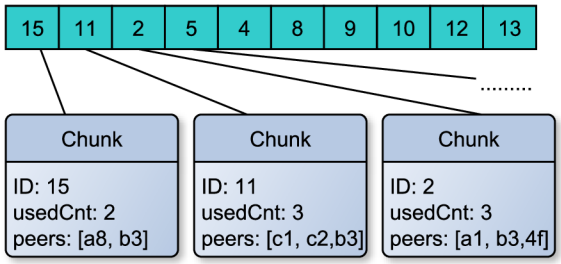
[34] ve skutečnosti tomu tak je, první chunk je redundantně obsahuje

3. z metadat zjisti rozsah bajtů `INIT` sekce videa,
4. `HTTP Range` požadavkem stáhni přesně `INIT` sekci z `TURN` serveru a připoj ji do přehrávače (`MSE`),
5. spusť `P2P` logiku, začni stahovat chunky v libovolném pořadí a **začni připojovat kompletní clustery v libovolném pořadí**.

Vektor stahovaných částí

Každý uzel nejdříve požádá signalizační server o vektor chunků, které má začít stahovat. Lze si jej představit jako uspořádanou množinu pořadových čísel jednotlivých chunků.

Ve skutečnosti je však tato datová struktura složitější, protože obsahuje pro každou položku informaci o tom, kolik uzlů daných chunk momentálně nabízí a taky jejich identifikátory (obr. 4.11). Vektor je seřazen dle zvolené strategie vždy tak, ať jsou nejdůležitější prvky umístěny vlevo a nejméně důležité vpravo.



Obrázek 4.11: Vektor částí ke stažení vrácený signalizačním serverem. Prioritní zleva.

K tomu, aby server mohl tento vektor poskytovat, musí vědět **v každém okamžiku o všech uzlech sítě, kterými chunky disponují**. Proto uzly ihned po stažení chunku tuto informaci oznamují serveru. Taky je důležité, aby v případě výpadku uzlu, server co nejdříve uzel ze svých datových struktur odstranil.

Velikost vektoru si určuje uzel a v implementaci je definována v konfiguračním souboru jako $bufferChunks = 2 * downloadSlots$. Počet komunikačních slotů včetně vztahu k velikosti vektoru jsou vysvětleny dále. Podstatné je, že je vektor v každém okamžiku pro každý uzel **unikátní**, protože vynechává chunky, které již daný uzel úspěšně potvrdil za stáhnuté.

Strategie volby chunku

Uzel se postupně, vždy když je to nezbytné, dotazuje serveru na další část práce k vykonání (vektor chunků). Ačkoliv by to mohl udělat pouze jednou při začátku stahování s maximální délkou vektoru,[35] bylo by takové řešení velice **nepružné ke změnám v síti**, které z principu `VoD` služby nastávají velmi často. Než by tak začal uzel stahovat chunky ke konci získaného vektoru, priority v síti by mohly být úplně jiné a stejně tak uzly ve vektoru (viz obr. 4.11) již odpojené a případné nové uzly neobjevené.

[35] rovnu počtu chunků videa

Aplikace tak vždy dotazuje pouze nezbytnou část práce, kterou aktuálně začne zpracovávat (`bufferSize`). Až se začne tento vektor blížit dokončení, je server opět dotázán na další část práce reflektující aktuální situaci v síti.

Které chunky se v tomto vektoru objeví a v jakém pořadí je **rozhodující pro délku životnosti** videa v síti. Je implementován systém zásuvných strategií pro jednoduchou změnu a experimentování s dopadem strategií na kvalitu sítě. Tyto strategie mohou být dynamicky střídány za chodu uzlu (této vlastnosti s výhodou využívá strategie volby uzlu, viz dále).[36]

Standardem v P2P sítích je preferovat ke stažení „vzácné“ části souborů, ať se rozložení počtu disponujících uzlů blíží rovnoměrnému. Důležitým důvodem je také to, že odpojení uzlu s unikátní částí dat znamená znehodnocení souboru pro všechny ostatní uzly. To v našem případě sice nehrozí (viz část 4.4), ale **této strategie se držíme taktéž** (ukázka 4.6).

Pro maximalizaci rychlosti přenosů v P2P sítích je nutné, aby bylo stále co vyměňovat. Toto se nazývá **požadavek diverzity**.[37] Zvolená strategie, vzhledem k neustálému reflektování stavu sítě a všech stažených chunků uzlu, požadavek diverzity splňuje.

Ukázka kódu 4.6: Strategie volby dle „vzácnosti“ chunku.

```
1 class RarestChunk extends Chunk {
2   static compare(chunkA, chunkB) {  // porovnavaci funkce pro sort()
3     return chunkA.usedCnt - chunkB.usedCnt; // menekrat pouzito jde drive
4   }
5 }
```

V případě, že již server nemůže poskytnout dostatek prvků vektoru (žádné další části již nikdo nemá), **doplní vektor chybějícími chunky stahovaného videosouboru**.[38] Tato implementace zajišťuje **univerzální programový kód** a úplnou kontrolu serveru nad uzly. Z principu je tak vyřešeno několik případů užití jako například přidělení práce pro první připojený uzel do P2P sítě.

Schéma 4.12 znázorňuje podrobněji význam prvků vektoru. Fixní délka 10 prvků obsahuje jednotlivé chunky (úplná struktura viz obr. 4.11) ke stažení od nejdůležitějších (95) po nejméně důležité (17). Zelená značí prvky, které se začnou obratem stahovat, šedá ty, které již uzel zpracovává (ale ještě nedokončil) a fialová další prvky v pořadí, které se už do limitu 10 prvků vektoru nevlezly. Modrá ohraničuje prvky, které se nacházejí v P2P síti (a tedy alespoň jeden uzel jejich stažení nabízí) a oranžová doplněné prvky, které ještě nikdo v síti nemá a budou staženy z TURN serveru (viz předchozí odstavec). Proč se do vektoru dostanou šedé prvky bude vysvětleno dále (viz část 4.4).

Obrázek 4.12: Vektor s opakovanými (šedá), novými (zelená) a zatím nevybranými (fialová) prvky. Nabízí P2P síť (modrá) nebo až TURN server (oranžová).

[36] Kromě strategie `RarestChunk` je k dispozici strategie `RandomChunk` nebo `LatestChunk` preferující chunky videa odzadu (což je protiklad `VoD` služby a lze pěkně testovat důsledky kombinace `VoD` přístupů, viz dále).

[37] Více o formálních aspektech P2P sítí je uvedeno v literatuře [19].

[38] Opět s výjimkou těch, které již dotazující uzel stáhl a pochopitelně s výjimkou těch, které se již ve vektoru nacházejí.

Priorita uspořádání vektoru

Rozlišujeme dvě úrovně priorit. První a důležitější je priorita zavedená dále v sekci 4.7 rozhodující o okamžitém stažení potřebného clusteru pro pokračování v přehrávání videa (sekvenční postup stahování během sledování videa). Druhá, méně důležitá úroveň je tvořena uspořádáním stahovaného vektoru. Server vždy řadí vektor od nejdůležitějších prvků po ty nejméně důležité. V případě `RarestChunk` strategy je tak nejdůležitější stáhnout ty chunky, které má **nejméně uzlů**.

Z pohledu uzlu je vektor zpracováván sekvenčně zleva doprava a tudíž jsou priority přirozeně uděleny dřívějším prvkům. Jedná se především o dřívější zabrání komunikačního slotu kdy již pro další prvky v pořadí nemusí být žádný volný. Dále jsou pak standardně chunky do všech front řazeny metodou `FIFO` a budou tedy přirozeně dříve obslouženy.

Řídící smyčka

Základní algoritmus získávání a odbavování úkolových vektorů je uveden v ukázce 4.7. Jedná se o velice zjednodušený sekvenční přepis asynchronního odbavovaní položek **hlavní řídící fronty Q**. Tato fronta obsahuje postupně všechny stahované chunky, jejich aktuální stav (stahováno, rezervováno, dokončeno apod.) a příznak extra priority dle části 4.7. Kromě těchto dvou rozšíření je struktura shodná se strukturou získávaného vektoru (viz obr. 4.11).

V návaznosti na postup 4.4 následuje:

1. `extendQueue()`: volání rozšíření fronty pomocí vstupního bodu algoritmu,
2. `getChunkVector()`: je získán (první) vektor seřazených chunků ke stažení,
3. `appendQueue()`: jednotlivé prvky vektoru jsou označeny příznakem `NEW` a přidány do řídící fronty Q, **pokud tam již nejsou**[39]
4. `processQueue()`: průchod frontou Q se stažením nových chunků nebo těch, které selhaly,
5. `downloaded()`: asynchronní dokončení stahování chunku změní příznak na `DOWNLOADED` a pokud je počet stažených prvků v řídící frontě větší než limitní hodnota (viz dále), znovu opakuje krok 1.

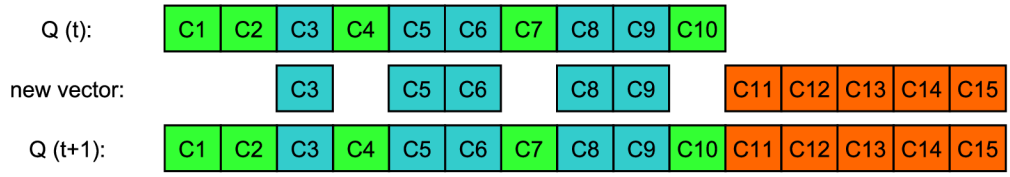
Ukázka kódu 4.7: Pseudokód kontrolní smyčky (control loop).

```
1  Q = []; waiting = false;
2
3  appendQueue(vec) {
4    for (chunk in vec) {
5      if (chunk in Q) continue;
6      Q.status = NEW;
7      Q.push(chunk);
8    }
9  }
10 processQueue() {
11   for(chunk in Q) {
12     if (chunk.status == NEW || chunk.status == FAILED) download(chunk);
13   }
14 }
```

[39] Tímto dochází k přeskakování chunků z vektoru, které jsou znázorněny šedou barvou na obrázku 4.12.

```
15 download(chunk) {
16   getFromTurnOrP2P(chunk);
17   downloaded(chunk);
18 }
19 extendQueue() {
20   waiting = true;
21   vec = getChunkVector(RarestChunk, config.bufferChunks);
22   appendQueue(vec);
23   processQueue();
24   waiting = false;
25 }
26 updateStatus(chunk, newStatus) {
27   Q.get(chunk).status = newStatus;
28   if (!waiting &&
29     Q.count(DOWNLOADED) >= Q.length—config.extendThreshold*config.
          bufferChunks) {
30     extendQueue();
31   }
32 }
33 async downloaded(chunk) {
34   updateStatus(chunk, DOWNLOADED);
35 }
36 extendQueue();
```

Hodnota, při které se začne stahovat další vektor chunků dynamicky závisí na **aktuální délce fronty Q** a konfiguračních direktivách `bufferChunk` (udává kolik prvků vektoru vyžadujeme, zvoleno 20) a `extendThreshold` (udává poměr kolik těchto prvků musí být úspěšně staženo, aby se začaly stahovat vektory další, zvoleno 1/2). Výchozí hodnoty v dané rovnici znamenají prodlužování fronty pokud ve frontě Q zbývá 10 nebo méně nestažených (tedy klidně rezervovaných) prvků.



Obrázek 4.13: Fronta Q před a po připojení získaného vektoru. Modrá znázorňuje přesahující prvky, oranžová prvky efektivně přidané.

Server na takový požadavek odpoví novým vektorem obsahující dalších `bufferChunk` (= 20) ještě nestažených chunků. Je však nutné počítat s tím, že až $extendThreshold * bufferChunk$ $(1/2 * 20 = 10)$ chunků může být již rozpracováno, ale **server o této informaci neví** (režie spojená s opakovaným odesíláním prvků vektorů by byla v nejhorším případě stejná jako komunikační režie v případě oznamování počátku stahování chunku).[40]

[40] Lze namítnout, že server přece ví, které prvku vektoru odeslal kterému uzlu ke zpracování. Ve skutečnosti tuto informaci nikde neukládá z důvodu prostorové složitosti. Nicméně možné by to bylo.

$$Ef_{min} = extendThreshold * bufferChunks \quad (4.1)$$
$$Ef_{max} = bufferChunks \quad (4.2)$$
$$Rs_{min} = \lceil chunk/Ef_{max} \rceil \quad (4.3)$$
$$Rs_{max} = \lceil chunks/Ef_{min} \rceil \quad (4.4)$$

V nejlepším případě tak bude efektivní využití vektoru rovno hodnotě Ef_{max} s potřebou vyžádání si Rs_{min} vektorů a v nejhorším pak Ef_{min} s vyžádáním Rs_{max} vektorů. Vliv na to, kolik zbytečných chunků v intervalu $< Ef_{min}; Ef_{max} >$ bude doručeno, strategie tvorby vektoru, kdy již chunky, které mohou kolidovat, nejsou znova vybrány. Typicky je to proto, že již nejsou tak „vzácné“ (`RarestStrategy`) nebo jiná heuristika ovlivní položky výsledného vektoru.

Race conditions

Na mnoha místech v kódu aplikace, kde může docházet k asynchronnímu volání, je nutné **implementovat ochrany vícenásobného volání** formou sdílených proměnných, jako je vidět v pseudokódu 4.7 u proměnné `waiting`. Pokud by nestřežila kritickou sekci získávání vektoru ze serveru, nastávala by situace vícenásobného volání tohoto kódu v čase mezi voláním a asynchronní odpovědí serveru pro každé asynchronní dokončené stahování.

V popsaném případě by docházelo pouze k plýtvání přenosovou kapacitou (stejné prvky se zahodí), ale na jiných místech v aplikaci by tyto race conditions způsobovaly problémy. Často je tak implementováno podobné vyloučení na kritické sekci spolu s uložením callbacků jednotlivých čekajících `Promise` objektů a jejich pozdější ruční vyvolání po obdržení asynchronní události.

Konstrukci modeluje příklad 4.8, kde by bez použití zmíněné ochrany cyklus vyvolal mnoho paralelních spojení namísto jediného. V praxi jsou často ještě doplněny časovače zajišťující maximální garantované čekání a eliminující deadlocky.

Ukázka kódu 4.8: Ochrana před race condition v kombinaci se vzorem `Promise`.

```
1  connecting = []; // [peerId]
2  resolves = [];   // [cbk]
3  rejectes = [];   // [cbk]
4
5  connect(peerId) {
6    return new Promise((resolve, reject) => {
7      if (connecting[peerId]) {
8        resolves.push(resolve);
9        rejectes.push(reject);
10       return;
11     }
12     exchangeHello(peerId).then(() => {
13       resolves.forEach((cbk) => cbk());
14     }).catch(() => {
15       rejectes.forEach((cbk) => cbk());
16     });
17   });
18 }
19
20 for (let i~= 0; i~< 100; i++) {
21   connect('peertest').then(() => {
```

```
22      console.log('Connected! Downloading...');
23      download(i); // download chunk with id
24    });
25  }
```

Je důležité poznamenat, že jazyk `JavaScript` není náchylný ke klasickým race condition problémům, protože má model souběžnosti založen na `event-loop` přístupu (více viz část 4.1). Proto lze pro vyloučení na KS použít obyčejnou proměnnou.

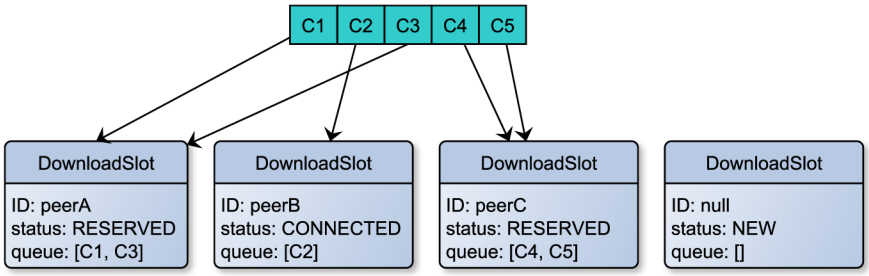
Stahovací sloty

Bez omezení počtu současných spojení by asynchronní povaha aplikace způsobila okamžité připojování uzlu ke všem možným protějškům a došlo by k zahlcení přenosového pásma uživatele. Současně by také hrozilo vyčerpání paměti apod.

Rozumná P2P síť omezuje počet otevíraných spojení a nedovolí zahlcení síťových prvků a jejich zásobníků. Běžné `BitTorrent` klienty standardně otevírají maximálně 50 spojení na torrent s globálním limitem 200 spojení. Vzhledem k experimentálnímu charakteru takových přenosů v prohlížeči byl v aplikaci zvolen limit 10 stahovacích slotů s možností jednoduché změny konfigurace v rámci experimentovaní.[41]

Počet prvků získávaného vektoru je však vždy vyšší a je tedy zřejmé, že není možné pro každý získávaný chunk alokovat vlastní stahovací slot. Vzhledem k režii spojené s otevřením P2P spojení by to ani nedávalo smysl. Kromě datové struktury řídící fronty Q aplikace **udržuje množinu otevřených stahovacích slotů** s maximální mohutností danou konfigurací aplikace.

Každý chunk je mapován vztahem **N:1** na volné sloty (obr. 4.14). Jeden chunk je vždy obsluhován pouze jediným slotem. Každý slot je dále jednoznačně přiřazen k jednomu uzlu sítě (**1:1**). Pochopitelně není možné libovolné mapování, protože různé uzly disponují různými chunky.



Obrázek 4.14: Mapování chunk:slot (N:1) a slot:peer (1:1).

Plánovač stahování tak vždy musí rozhodnout:

- od koho chunk, který je na řadě, začít stahovat,
- kterému slotu ho přidat do fronty ke stažení,
- případně který slot vynuceně odpojit a uvolnit tak místo pro jiné připojení.

[41] Z hlediska teoretické přenosové rychlosti se tak jedná o $speed = downloadSlots * averageSpeed$, což je v případě asymetrické linky 1,0/0,25 Mbit ve zvoleném nastavení rychlost 320 kBps, což odpovídá 2,5 Mbps. Vzhledem k tomu, že chceme distribuovat video s průměrným datovým tokem 1 Mbps, je tato rychlost postačující s dostatečnou rezervou.

Všechna tato rozhodnutí řídí další dynamické strategie.

Strategie výběru uzlu

Jak již bylo řečeno, zpracování fronty Q probíhá metodou `FIFO`. Prochází se všechny nové nebo problémové chunky a začínají se stahovat (viz alg. 4.7).

To v první řadě znamená vybrat z množiny uzlů, které chunk nabízejí, **nejvhodnějšího kandidáta**. Pro optimální využití pásma se nabízí strategie výběru vždy ještě nepřipojeného uzlu (`NewPeer`). Dochází tak k rychlému a opakovanému obsazování všech stahovacích slotů. Současně je u více otevřených spojení větší šance, že další potřebný chunk bude někdo mít. V případě zaplnění všech slotů by však brzo docházelo k situacím, kdy je nutné některé sloty zase uvolnit.

Proto používáme **kombinaci dvou protichůdných strategií**:

```
strategy = (noSlot || extraPriority) ? Strategies.ReusePeer : Strategies.NewPeer
```

Pokud jsou všechny sloty obsazeny, preferujeme již připojené uzly, pokud ne, tak uzly nové. Dále může být dynamické rozhodování ovlivněno požadavkem prioritního zpracování, kdy není čas na ustavování nového připojení (více dále v části 4.7).

Tato strategie dobře využívá volné sloty a zároveň zbytečně nevynucuje odpojování existujících připojení. V případě výběru z více uzlů odpovídající strategii (výběr z více „stejně dobrých“) zavádíme váhování, kdy jsou sloty s kratšími frontami preferovány před sloty s frontami delšími. **Preferují se tak méně zahlcené sloty.**[42]

Stažení chunku

Máme tedy uspořádanou množinu preferovaných uzlů poskytujících stahovaný chunk. Pokud je množina prázdná, typicky to znamená, že žádný uzel chunk nenabízí. Případně byli kandidáti z jiných důvodů odstranění (blacklist, prioritní stahování apod.). Každopádně to znamená stáhnutí chunku z `TURN` serveru namísto z `P2P` sítě (popsáno v části 4.4).

Pokud je množina neprázdná, **vyjmeme první prvek** (nejlepší kandidát) **a zahájíme stahování**. To spočívá ve:

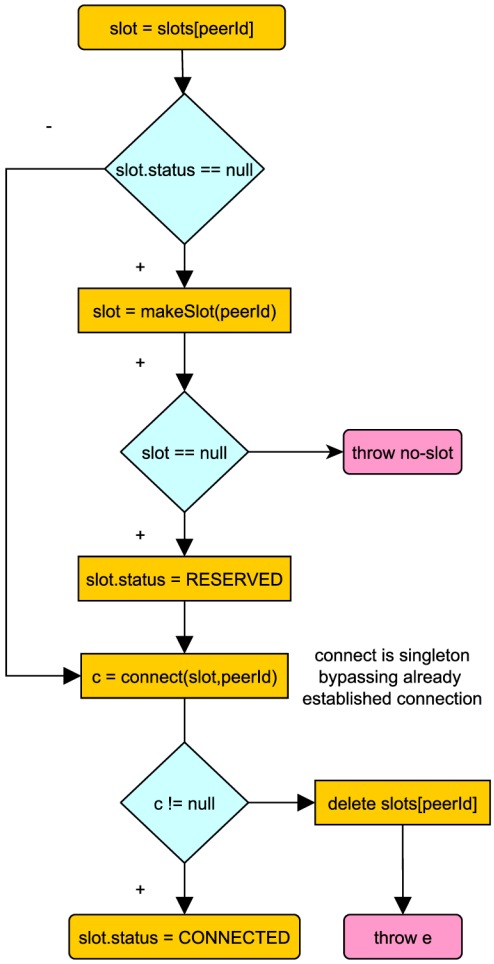
1. výběru volného slotu pro stahování, pokud již není vybrán,
2. ustavení `P2P` spojení s vybraným uzlem, pokud již není ustaveno,
3. uvolnění nejméně důležitého slotu, pokud je potřeba,
4. zařazení stahovaného chunku do fronty odpovídajícího slotu pro stahování,
5. spuštění hlídacího časovače pro garantované dokončení přenosu.

Body 1 a 2 znázorňuje vývojový diagram 4.15. Pro připojení je nejdříve nutné zarezervovat slot. Poté je možné se pokusit o připojení k uzlu. Selhání musí korektně uvolnit rezervace pro další žádosti. Selhat připojení může také na časovači (což je nejčastější varianta – dva uzly se nepodařilo v časovém limitu vzájemně spojit).

Pokud nedošlo k chybovému stavu (např. nedostupnost volného slotu), **chunk je zařazen do interní fronty slotu.**[43] Tato fronta je postupně odbavována v rámci životního

[42] Vzhledem k tomu, že je u každého slotu vedena informace o tom, kolik přenosů již uskutečnil a kolik z toho neúspěšně, lze tyto heuristiky značně vylepšovat.

[43] Nejedná se o řídící frontu Q obsahující postupně všechny chunky, ale o interní frontu každého slotu.



Obrázek 4.15: Posloupnost kroků při ustavování spojení s uzlem.

cyklus slotu (viz dále). Současně je spuštěn další časovač, který hlídá dobu prostoje ve frontě v případě prioritního stahování. Více v části 4.7.

Pokud však k chybovému stavu došlo, selže přenos chunku a aplikace se v rámci zotavení zachová následovně:

- Pokud není nalezen volný slot, znamená to, vzhledem ke kombinované strategii (viz část 4.4), že **existuje uzel disponující tímto chunkem, ale žádný z námi připojených uzlů jej nemá.** Limituje nás tedy počet slotů, takže zkusíme některý zavřít (dle strategie, viz dále). Aktuální uzel **je navrácen** mezi kandidáty disponující stahovaným chunkem a stav položky ve frontě Q je označen jako `FAILED`.

- Pokud dojde k jiné chybě, například vypršení časovače připojení, položka je ve frontě taktéž označena jako `FAILED`, ale uzel zpět mezi kandidáty navrácen není. Pravděpodobně s tímto uzlem neumíme komunikovat a není tak nutné to zkoušet znova.

Řídicí fronta Q v dalším kole odbavování svých prvků začne znova stahovat `FAILED` prvky od kandidátů, kteří jsou další v pořadí (viz pseudokód 4.7). Pokud byl uvolněn některý slot z důvodu nedostupnosti slotu v předchozím kole, je vysoká šance, že nyní již

stahování proběhne úspěšně. Pokud by však byl uzel neustále ve frontě předbíhán, vyhladovění stejně nehrozí, protože v nejhorším případě se nutně stahovací slot uvolní po stažení $chunksCount - downloadSlots + 1$ chunků.[44]

Uvolnění stahovacího slotu

Vzhledem k použité strategii, která se snaží využít veškeré stahovací sloty, je možné, že dojde ke zmiňované situaci nedostatku slotů. Současně však implementace bere v potaz to, že se `P2P` stahování v takové situaci používá v neprioritním režimu a tudíž **nejsou sloty uvolňovány agresivně**.

Zároveň je nutné držet co nejvíce spojení otevřených po celou dobu stahování, pro případ, že přijde na řadu chunk, který některý z připojených uzlu bude mít, třebaže se dlouho nic od takového uzlu nepřenášelo.[45]

Procedura recyklace slotů tak vždy uvolní pouze **první vyhovující**. Za vyhovující se považuje slot dle kódu 4.9. Tedy takový, který právě nestahuje, není v procesu rezervace (dle obr. 4.15) a jeho fronta je momentálně prázdná, tudíž jen „vyčkává“ na přidělení práce.

Ukázka kódu 4.9: Výběr slotu pro stahování k uvolnění.

```
1 const candidates = this._slots.filter((slot) => {
2   return slot.isQueueEmpty() && !slot.isDownloading() && !slot.isReserved();
3 });
```

Typicky však bude kandidátů na uvolnění více a je nutné opět vybrat nejvhodnějšího pomocí strategie. Použitá `WeakestSlot` strategie obsahuje ohodnocující funkci na základě poměru neúspěšných přenosů slotu:

$$score = \frac{slot.failureCnt}{slot.failureCnt + slot.successCnt}$$

Slot s vyšším ohodnocení a tedy nejspíše s horším spojením s mapovaným uzlem vybere k uvolnění.

Životní cyklus stahovacího slotu

V návaznosti na zařazení stahovaného chunku do fronty některého ze slotů (viz bod 4 seznamu 4.4) je spuštěn obdobný asynchronní životní cyklus jako v případě obsluhy hlavní řídící smyčky (viz část 4.4). S každým vložením chunku do fronty je spuštěna její obsluha. Vždy je obsluhován pouze jeden přenos zároveň (kritická sekce). Případná chyba vybublá až do hlavní řídící smyčky (odstavec 4.4) a chunk bude přeplánován k jinému uzlu.[46]

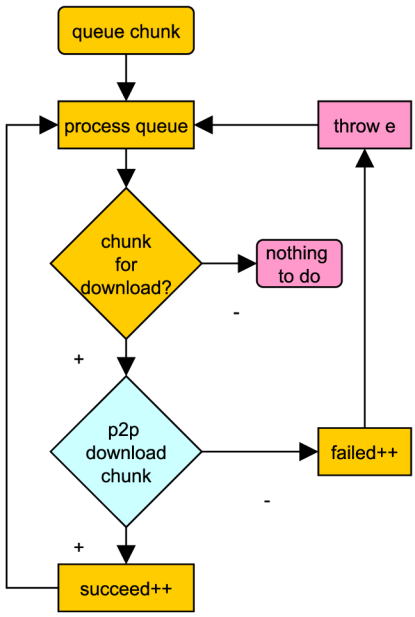
4.5 Přehrávání videa

Získaný chunk dat je základním předpokladem pro následné ověření integrity, dohledání obsažených video segmentů a jejich připojení do sledovaného videa.

[44] V případě videa tvořeného 100 chunky a stahovaného 10 sloty se tak nutně po stažení 91 chunků nějaký slot uvolní.

[45] spojení se drží „kdyby náhodou“

[46] nebo v případě, že neexistuje alternativní uzel bude stažen z `TURN` serveru



Obrázek 4.16: Životní cyklus každého stahovacího slotu.

Integrita dat

Po stažení chunku následuje nutnost ověření, že nebyly data během přenosu či úmyslně změněny. Je vypočítán `SHA-1` otisk chunku a porovnán s očekávaným otiskem distribuovaným **v rámci metadat** (viz část 4.4). Metadata jsou důvěryhodná, protože je poskytuje server přes zabezpečené spojení `HTTPS`. Volba hashovací funkce je inspirována protokolem `BitTorrent` a její vhodnost v prostředí webových prohlížečů ověřena v rámci experimentování (viz část 5.2).

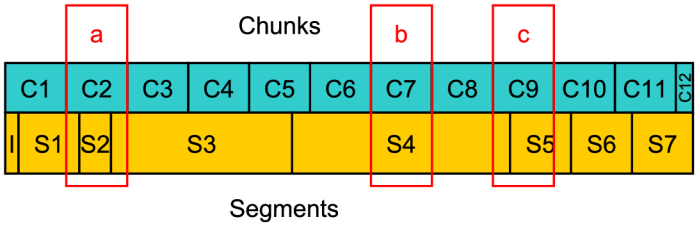
Pokud otisk nesedí, je uzel přidán na `blacklist` a již od něho nebude dále nic stahováno.[47]

Zpracování chunku

Následuje připojení staženého chunku do bufferu přehrávaného videa. Jak bylo řečeno dříve, chunky jsou pouze abstrakce nad clustery daného videa. Připojování částí do bufferu videa tak nutně probíhá **po clusterech**. Zpracování chunku pro použití ve videu tak obnáší:

- dle bajtových rozsahů dohledat úplné clustery (obr. 4.17a) a připojit je,
- případné zleva a zprava přebývající bajty spojit rekurzivně se svými sousedy (clustery mohou ležet přes více chunků, obr. 4.17b) a dohledat dokončené clustery,
- v případě zatím chybějících okolních chunků uložit části levých a pravých clusterů pro pozdější spojení,
- korektně z paměti po bajtech uvolňovat jednotlivé už připojené clustery (jinak zabírají až 2x tolik paměti).

[47] Vzhledem ke zvolenému spolehlivému přenosu `SCTP`, který zaručuje na nižší vrstvě přenos nepoškozených dat, se lze domnívat, že otisk nesedí s nejvyšší pravděpodobností úmyslně.



Obrázek 4.17: Tři různé situace při mapování chunků na clustery.

Vždy se tedy před připojením další části videa čeká na **úplné stažení** nějakého clusteru, který může být rozprostřen přes několik chunků a naopak může zabírat jen velmi malou část chunku jediného.

Základní situaci znázorňuje obr. 4.17c, kdy v jednom chunku končí i začíná cluster. V případě předchozího doručení `C10` se tak nyní spojí cluster `S5`, připojí do videa a uvolní z paměti. Pokud však ještě není možné udělat to samé s clusterem `S4`, je uvolněna jen část `C9` odpovídající clusteru `S5`.

V případě chybějícího průběžného chunku (nezačíná v něm ani nekončí žádný cluster) musí být v datové struktuře uloženi veškeří sousedi po cestě. Po jeho přijetí se dohledává zleva[48] počátek nejbližšího clusteru a ten se, za předpokladu úplnosti, teprve připojí do videa. Na příkladě 4.17 by to znamenalo situaci `b` kdy chybí chunk `C7`, ale již přišly `C5`, `C6`, `C8` a `C9`. V té době je v paměti držena část `C5` (netřeba celá pokud již přišlo `C4`, `C3` a `C2`), `C6`, `C8` a opět část `C9` (netřeba pokud již přišlo `C10`). S příchodem `C7` je zleva hledán nejbližší cluster, což je cluster `S4` začínající v `C5`. Spustí se spojovací algoritmus, který, nyní již úplný, cluster spojí, **připojí do videa a odstraní z paměti**.

Pro další popis rekurzivních algoritmů zajišťující paměťově efektivní mapování chunků na clustery odkazuji čtenáře do zdrojového kódu aplikace (modul `Player`).

Připojení clusteru

Získané clustery jsou připojovány do `SourceBuffer` objektu vázaného na konkrétní `<video>` element. Vzhledem k tomu, že lze s bufferem pracovat pouze pokud nemá nastavený atribut `updating` a rozhraní nenabízí žádnou jednoduše použitelnou metodu pro sledování změny této hodnoty, je nutné použít techniku jednorázové události uvedenou v kódu 4.10.

Obdobně lze postupovat při uzavírání bufferu voláním `MediaSource.endOfStream()`.[49]

Ukázka kódu 4.10: Konstrukce bezpečného manipulace s objektem SourceBuffer.

```
1 const safeAppend = () => {
2   if (this._sourceBuffer.updating) {
3     const self = this; // pro pristup z~closure
4     this._sourceBuffer.addEventListener('updateend', function evt() {
5       self._sourceBuffer.removeEventListener('updateend', evt);
```

[48] Z důvodu předpokladu pravděpodobnějšího doručování chunků v sekvenci pro případ prioritního stahování – tam je rychlost zpracování klíčová.

[49] Nutno podotknout, že ačkoliv se tato metoda ukázala jako jednoznačně nejspolehlivější, i přesto občas toto volání selže kvůli údajnému nastavenému stavu `updating`. Kód by však díky použité technice měl vždy vyloučit přístup do kritické sekce. Neukončení streamu se projeví zaseknutím videa zhruba vteřinu před koncem. Možná se jedná o bug prohlížeče `Chrome 51`.

```
6         safeAppend();
7       });
8     } else { ... }
9   };
```

4.6 Poskytování videa

Po úspěšném stažení a připojení chunku videa následuje jeho zpřístupnění ostatním uzlům sítě. To dovoluje získaná data distribuovat dříve, než je celé stahování dokončeno a tím zvyšovat životnost videa.

Ideální by byla možnost přímého přístupu k jednotlivým bajtům připojených clusterů videa. `MediaSource API` bohužel nic takového nenabízí a jednou připojené clustery jsou tak již plně v režii prohlížeče, který si obstarává jejich persistenci a načítání do paměti v případě potřeby. Vzhledem k charakteru `P2P` přenosu by však tato metoda pravděpodobně stejně nebyla vyhovující.

Prohlížeče nabízejí několik různých prostředků pro trvanlivé uložení dat. Základní metody jako `cookies` nebo `LocalStorage` mají omezení typicky na 5 MB. Ostatní způsoby vyžadují uživatelskou interakci při ukládání nebo získávání souborů z úložiště, což bylo shledáno prozatím jako nepřijatelné.

Aplikace tak zatím implementuje jediné úložiště – `MemoryStorage`. Každý stažený chunk je tak držen v paměti prohlížeče. Po zpřístupnění chunku v tomto úložišti je signalizačním kanálem oznámeno serveru, že uzel úspěšně stáhl daný chunk a server může uzel zahrnout do svých plánovacích strategií.

Odesílací sloty

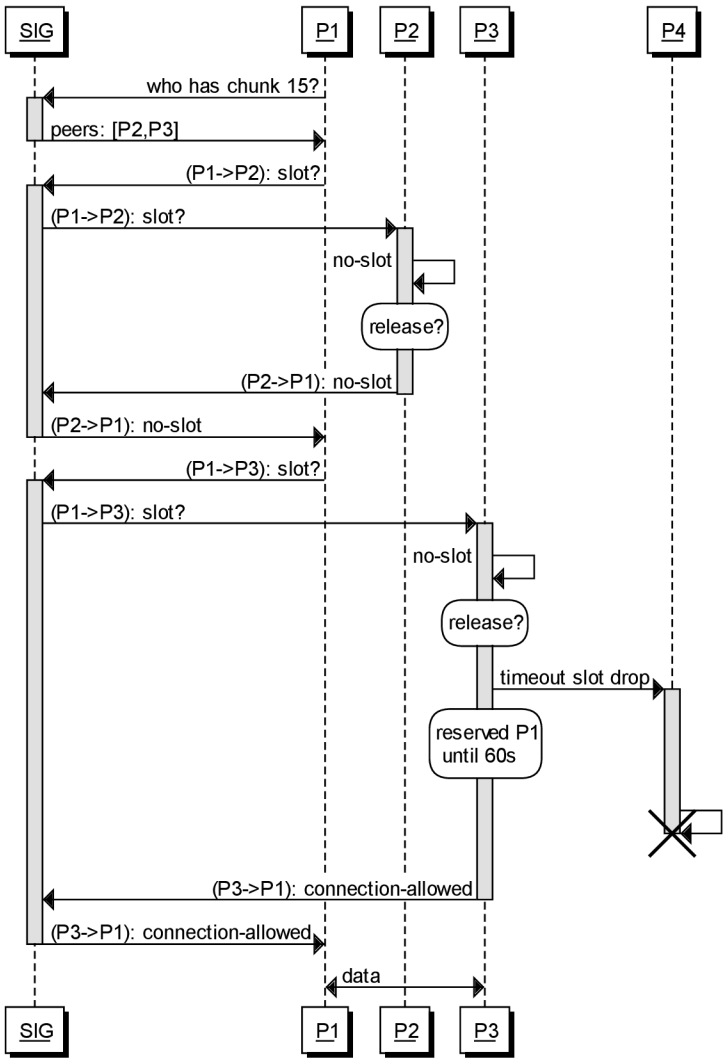
Obdobně jako řídící smyčka stahování je implementována obdobná **smyčka odesílání**. Před každým spojením dvou uzlů proběhne v rámci `handshake` domluva, zdali uzel disponuje volným slotem k odesílání dat. Počet těchto slotů je určen konfigurací s výchozí hodnotou 1/4 počtu slotu stahovacích (inspirace v asymetrii ADSL).

Odesílací sloty, na rozdíl od slotů stahovacích, **netvoří žádné fronty** a požadavky na spojení od protistran jednoduše zamítají. Protistrany pak zkouší jiné uzly popřípadě `TURN` server.

Na diagramu 4.18 lze vidět průběh pokusu o stažení chunku `15` uzlem `P1` z pohledu odesílací smyčky. Ještě není ustaveno přímé spojení mezi `P1` a žádným dalším uzlem a proto je veškerá komunikace přeposílána skrz signalizační kanál `SIG`. Uzel `P2` nemá žádné volné sloty, ale pokusí se nějaký uvolnit (viz strategie dále). Nepodařilo se mu to a tudíž musí `P1` zkusit další uzel v pořadí (`P3`). Ten má také všechny sloty využity,[50] ale jeden z nich (`P4`) přesáhl hranici nečinnosti a proto může být uvolněn. Je udělena **časově omezená rezervace** (pro případ, že by nešlo navázat přímé spojení mezi uzly nebo by uzel ztratil zájem a odpojil se) slotu uzlu `P3` pro uzel `P1`. Uzel `P1` navazuje úspěšně, tentokrát již přímou, komunikaci a startuje přenos potřebných dat.

V rámci uvolnění odesílacího slotu opět uvolňujeme **pouze jeden** (maximalizace počtu spojení) a to takový, který je připojený nebo rezervovaný (rezervace jsou časově omezeny) a časový otisk poslední interakce přesáhl nakonfigurovaný limit.

[50] Opět se uplatňuje pravidlo využití dostupných slotů na maximum a udržování otevřených spojení kdyby protistrana potřebovala ještě nějaký chunk dostahovat později.



Obrázek 4.18: Sekvenční diagram získání upload slotu.

Typicky však bude kandidátů na uvolnění více a je nutné opět vybrat nejvhodnějšího pomocí strategie. Použitá `IdlestSlot` strategie obsahuje porovnávací funkci:

```
compare(slotA, slotB){ return slotB.timestamp − slotA.timestamp }
```

Ta vychází z časového otisku kdy byly slotem odeslány poslední data. **Preferuje tak k uvolnění sloty déle nečinné.**

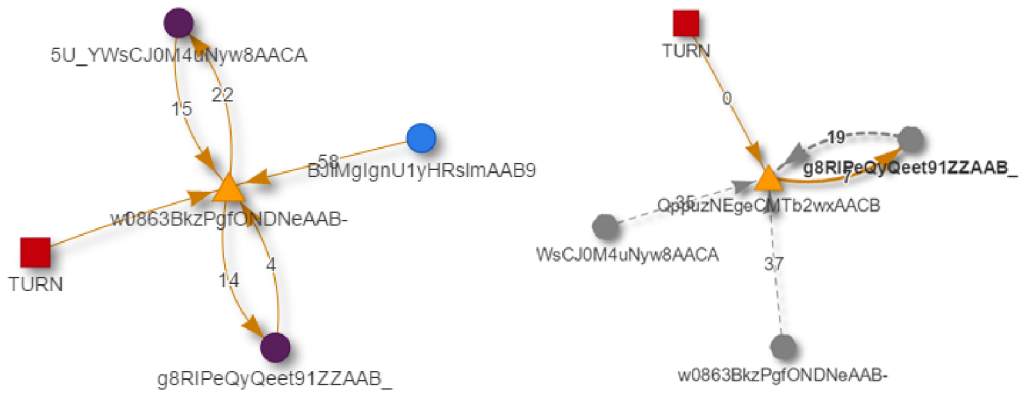
Zajímavá situace nastává, kdy část videa odesílá jediný uzel, ale má obsazeny všechny sloty a nemůže je uvolnit. Uzly by pak opakovaně pro každý chunk bombardovaly tohoto odesílatele a zahlcovaly signalizační kanál. Je tak nutné udělit **dočasný časový ban** při každém obdržení zprávy `no-slot`. Princip je stejný jako v případě ostatních banů – požadavek na spojení je rovnou odmítnut již na straně klienta a aplikace se s nastalou situací musí vypořádat jinak.

Uzavírání spojení

Uzavření přímého komunikačního spojení mezi uzly může přijít **prakticky kdykoliv** a aplikace se musí vypořádat s mnoha různými případy. Jedná se tak o odpojení uzlu během připojovaní, během jakékoliv chyby spojení, během náhlého odpojení uzlu, v případě vypršení jakéhokoliv časovače a v neposlední řadě také v případě dokončení stahování (graceful disconnect).

Uzel, který úspěšně stáhne celé video, co nejdříve zašle všem uzlům, od kterých stahoval (tedy zabíral jejich odesílací sloty), informaci, že nemusí při uvolňování slotů čekat na překročení komunikačního prahu, ale mohou ho uvolni okamžitě. Jedná se o optimalizaci pro eliminaci zbytečného blokování uzlů.

To se však týkalo pouze uvolnění slotu, což nutně **neznamená uvolnění datového spojení** mezi uzly. To totiž může klidně pokračovat v případě, kdy uzel `A` právě dostahoval od uzlu `B`, ale ten ještě na oplátku stahuje od uzlu `A`. Datové spojení mezi uzly je vždy jedno a je nutné jej odpojit pouze až je jisté, **že nebude potřebné v ani jednom směru.** Smyčky ilustrují vykreslené grafy z aplikace 4.19.[51] Vpravo lze vidět popsanou situaci, kdy nelze uzavřít datový kanál mezi uzly i po dokončeném stahování.



Obrázek 4.19: Vznik výměnných smyček mezi uzly. Trojúhelník značí aktuální uzel, kruhy další uzly v síti a obdélník „záchranný“ server. Vlevo je probíhající přenos ze všech uzlů. Vpravo již dokončené stahování s pokračováním v odesílání uzlu napravo.

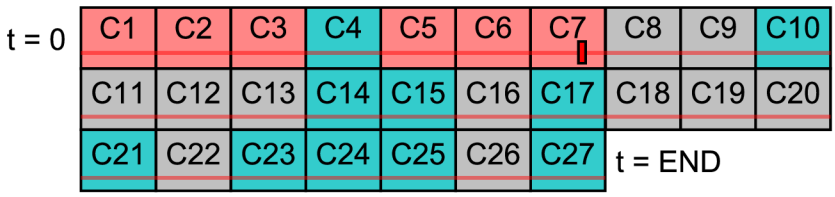
4.7 Video-on-Demand služby

Dosud jsme se zabývali problematikou efektivního stažení chunků vybraného videa od různých uzlů v síti. Logická `overlay` síť byla především zaměřena na zajištění robustnosti sítě. Samotné spuštění přehrávání videa uživateli bylo až na druhém místě. Z hlediska realizace služby je však důležitější **uživatelská použitelnost** řešení.

Je tedy nutné spustit přehrávání videa **dříve** i za cenu „zbytečného“ stažení dat z `TURN` serveru. Stejně tak je nutné **udržovat plynulost** videa a dokonce dovolit uživateli přeskočit kamkoliv na časové ose přehrávače. Při současné implementaci by v případě chybějících právě požadovaných clusterů v okolí aktuální pozice ve videu nastalo čekání až data „náhodou“ dorazí.

Diagram 4.20 znázorňuje chunky videa stažené dosud popisovanou strategií `RarestFirst` (modrá barva) v kombinaci s prioritní strategií zajišťující `VoD` služby (červená barva). S po-

[51] Pro vykreslování topologie sítě je využívána knihovna `vis.js` pod `Apache 2.0` a `MIT` licencí.



Obrázek 4.20: Kombinace prioritní (červená) a `RarestFirst` (modrá) strategie. Zatím nestažené chunky jsou šedé.

hybem aktuální pozice ve videu se stahují chunky prioritní strategií, pokud již nebyly dříve staženy strategií `RarestFirst`. Aktuální pozice na obrázku by znamenala okamžité stažení chunků `C9`, `C10` a možná i dalších **v závislosti na velikosti časového okna** (viz dále).

Nejjednodušší implementací by bylo sekvenčně stahovat chunky rovnou z `TURN` serveru a přitom alespoň něco stahovat strategií `RarestFirst` z `P2P` sítě.

Implementována byla vylepšená verze kombinující tento přístup s včasným pokusem o stažení z `P2P` sítě. Pokud se nepodaří v časovém limitu chunk stáhnout, přechází se okamžitě na stažení z `TURN` serveru.

Navržená síťová struktura je klasifikována jako **hybridní overlay síť kombinující prvky `mesh-based` metodologie a `client-server` architektury.** Z existujících `P2P` sítí se podobá síti `PONDER` využívající „záchranný“ `TURN` server také. Více o dělení `P2P` `VoD` sítích je uvedeno v knize [19].

Tradiční `P2P` sítě jako je `BitTorrent` nakládají s každou částí dat ekvivalentně. Priority jsou typicky určeny akorát strategií `Tit-for-Tat`, která určuje komu bude uzel data odesílat. Preferují se tak uzly, které na oplátku zasílají data zpět (v naší implementaci obdoba grafů 4.19). Samotná data však mají rozložení váhy ekvivalentní. V `VoD` systémech jako je tento mají nejvyšší prioritu chunky **nejblíže aktuální pozici přehrávaného videa.**

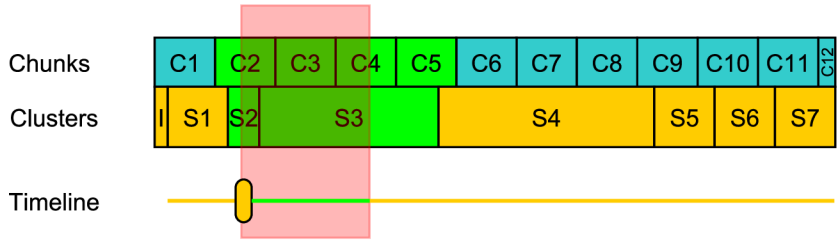
Klouzavé okno

Obrázek 4.21 znázorňuje implementovanou techniku klouzavého okna (červený obdélník) při přehrávání či posuvu v rámci videa. S každou změnou aktuální pozice ve videu se z intervalu `<teď, teď + velikost okna>` **dohledají potřebné clustery** (`S2` a `S3`), které jsou pro přehrání následujících několika vteřin nezbytné. Pro tyto video clustery se dopočítá **minimální množina chunků** z kterých je lze získat (`C2`, `C3`, `C4`, `C5`).

Velikost okna je zásadní pro zachování plynulosti přehrávaného videa, ale může zbytečně stahovat části z `TURN` serveru. Proto je v konfiguraci uvedena výchozí hodnota `5 s` s možností pozdějšího experimentování nebo rozšíření o dynamické přizpůsobování jeho velikosti podle stavu sítě.[52]

Vzhledem k velmi častému volání (při každé změně pozice ve videu, tedy i při přehrávání) funkce pro hledání množiny minimálního pokrytí chunků je nutná její vysoká optimalizace. Byla proto implementována datová struktura pro **časově rozsahové hledání** nad množinou clusterů a chunků. Tato struktura je podobná hashovací tabulce se sekvenčním dohledání konkrétního časového záznamu a je inicializována při získání metadat.

[52] Dále je z obrázku zřejmé, že v případě zbytečně dlouhých clusterů (viz 4.3) by při každém posuvu bylo nutné prioritně stáhnout mnoho chunků najednou místo plynulého stahování, které lépe reflektuje aktuální situaci `P2P` sítě.



Obrázek 4.21: Technika klouzavého okna pro prioritní stahování.

Pro další popis optimalizovaných algoritmů dohledávání pokrytí pro zvolené časové rozsahy odkazuji čtenáře na zdrojový kód aplikace (modul `Player`).

Prioritní zpracování

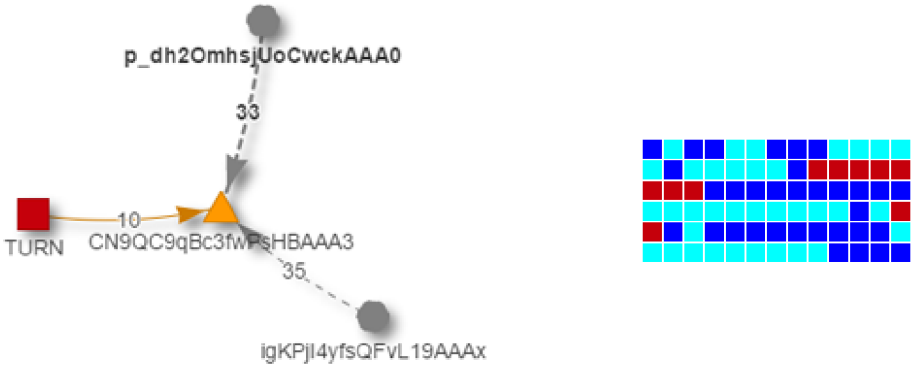
Pro každý **ještě nestažený** chunk ze získané množiny se popořadě:

- Zjistí, zdali se již nachází v **řídící frontě Q**. Pokud ne, je dotázán signalizační server na množinu uzlů chunkem disponujících. Chunk je označen za prioritní a zařazen do čela fronty Q.
- Pokud ano a ještě není zpracováván (NEW nebo FAILED), je přeznačen za prioritní a taktéž přesunut do čela fronty Q.
- Pokud ano a již je přiřazen do fronty nějakého slotu, je přeznačen za prioritní a v této **interní frontě** přesunut do čela.
- Pokud ano a již se nenachází v žádné frontě žádného slotu, znamená to, že je právě stahován a tudíž se nechá stahování dokončit. Spustí se však nový, striktnější časovač hlídající maximální dobu čekání na dokončení stahování. Pokud je překročena, přenos je předčasně ukončen a chunk stažen odjinud (`TURN`).

Označení chunku za prioritní způsobí napříč celou popsanou implementací několik odlišností v chování:

- Chunky jsou do front vždy řazeny do čela místo nakonec.
- Strategie výběru uzlu je vždy vynucena na `ReusePeer`, protože není čas na připojování k novému uzlu (viz část 4.4). Pokud takový uzel k dispozici není, následující bod jiné kandidáty vyřadí.
- Množina kandidátů, od kterých lze chunk stahovat, je po seřazení patřičnou strategií uměle omezena na **kandidáta jediného** a to podmíněně již připojeného. Pokud se nepodaří stáhnout chunk v časovém limitu od něj, bude se stahovat rovnou z `TURNu`, i přestože by bylo možné zkusit další kandidáty. Situaci znázorňuje červenou barvou obrázek 4.22.

- Je zaveden časovač hlídající maximální čas prostoje chunku ve frontě.[53] Pokud tak slot ještě stahuje předchozí chunk a přijde nový chunk prioritní, čeká se na dokončení stahování chunku předchozího **maximálně po tuto dobu.** Pokud se předchozí chunk odbavit nestihne, tento časovač prioritní chunk z fronty čekání odstraní a stáhne ho jinak (`TURN`).
- Ostatní časovače mají striktnější hodnoty (viz konfigurační soubor).

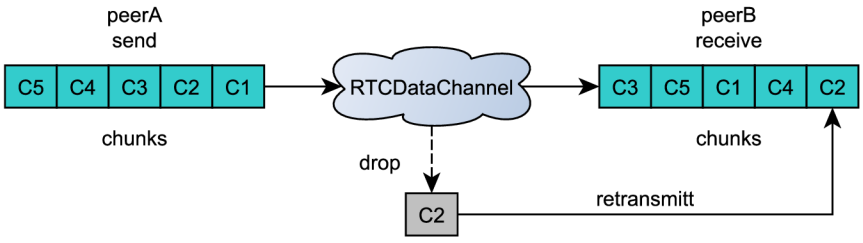


Obrázek 4.22: Stažení videa od dostupných uzlů (modrá) s výpomocí TURN serveru pro prioritní stažení (červená).

4.8 Spojení uzlů

Pro P2P komunikaci uzlů je použito `RTCPeerConnection API` s výměnou kandidátů signalizačním kanálem pomocí inkrementálního přístupu `Trickle ICE` (více kap. 8 knihy [5]).

Pro výměnu datových zpráv využíváme `RTCDataChannel API`, který umožňuje zvolit parametry `SCTP` protokolu, aby se blížil více protokolům `TCP` nebo `UDP` (více v teorii viz 2.5). V implementaci je využíván **částečně spolehlivý přenos** se specifikací maximálního času časovače, který znovu odesílá pakety v případě selhání. Není nutné, aby se transportní vrstva snažila doručovat pakety, pokud už je aplikační vrstva neočekává. Hodnota časovače je tedy nastavena shodně na hodnotu `maxPacketLifeTime = max(connectionTimeout, transferTimeout)`.



Obrázek 4.23: Částečně spolehlivý negarantovaný přenos mezi uzly.

Vzhledem k tomu, že jsme zvolili velikost chunku `32 kB`, není vhodné použít úplně nespolehlivý přenos dat (podobný `UDP`). Vzhledem k reálnému přenosu zhruba 1100 bajtů

[53] Výchozí hodnota je `1 s`, ale opět se jedná o další konfiguraci určenou k experimentování.

dat jedním paketem dojde k fragmentaci na $\lceil \frac{32*1024}{1100} \rceil = 30$ paketů. Pokud vezmeme v úvahu 1% ztrátu paketu, pravděpodobnost selhání přenosu nespolehlivým kanálem je přibližně $P = 1 - (P_{delivery})^{count} = 1 - 0,99^{30} = 26$ %.

Druhým parametrem přenosového kanálu je to, zdali garantuje zachování pořadí přenášených dat. V rámci optimalizace je zvolen režim, který **pořadí přenášených dat negarantuje.**[54] Pak je však nutné každou binární zprávu (chunk) **rozšířit o identifikaci:**

$$chunkId = ((data[0] << 8)|data[1])$$

Každý, standardně `32 kB` chunk, je tak rozšířen o další `2 B` s identifikaci umožňující příjem dat mimo pořadí odesílání. Rozsah `2 B` vystačí na $(2^8)^2 = 65536$ chunků, což s danou velikostí každého chunku umožňuje adresovat až $chunkSize * chunkCount = 32 * 65536 =$ 2 GB velké videa. Tato adresace přináší zanedbatelnou režii.[55]

Specifikace `WebRTC` zmiňuje podporu pro definici přenosových priorit, které by mohly implementaci zjednodušit. Tato podpora je však volitelná a žádný webový prohlížeč ji zatím nijak neimplementuje.

Pro procházení přes různá `NAT` zařízení po cestě mezi uzly využíváme veřejných `STUN` serverů společnosti Google a to včetně některých provozovaných na portu 80 z důvodu prostupnosti firewally. Pro spolehlivé ustavení spojení mezi jakoukoliv dvojicí uzlů lze použít speciální `TURN Relay` server. Jak bylo zmíněno na začátku kapitoly (4.4), `TURN` server **v tomto smyslu nepoužíváme**, protože provoz takového serveru je na přenosové pásmo více náročný jako stahování chunků klasickou metodou `client-server`.[56]

[54] Tím se aplikace stává odolná proti problému `Head of Line` blokování. Více viz kapitola 17 knihy [16].

[55] Vezmeme-li v úvahu průměrně `50 MB` velké video, režie extra přenášených dat tvoří `3200 B`.

[56] `Relay` server, za předpokladu vypnutého cachování, musí stáhnout data uzlu A a odeslat je uzlu B. Průtok dat je tak oproti `client-server` řešení, kde jsou videa stejně uložena, teoreticky dvojnásobný.

Kapitola 5

Výsledky

V rámci testování jsem experimentoval s různými parametry, které aplikace dovoluje konfigurovat. Většina parametrů byla popsána v rámci kapitoly Implementace s uvedením výchozích nastavených hodnoty. Objasnění volby těchto hodnot je uvedeno v této kapitole.

Každá sekce představuje jednu ucelenou sadu vykonaných testů které následují pořadí implementace.

V případě uvedení pouze grafů bez zdrojových dat tak činím z prostorových důvodů. Zdrojová data včetně samotných grafů jsou uvedena na přiloženém DVD (soubor `grafy.xlsx`).

5.1 Volba datového toku

Během implementace bylo zjištěno, že bude nutné videa v rámci zpracování **překódovat** (viz část 4.3). Vzhledem k tomu, že se jedná o zásadní úzké hrdlo aplikace, bylo otestováno několik kombinací použitého kodeku a výsledného datového toku.

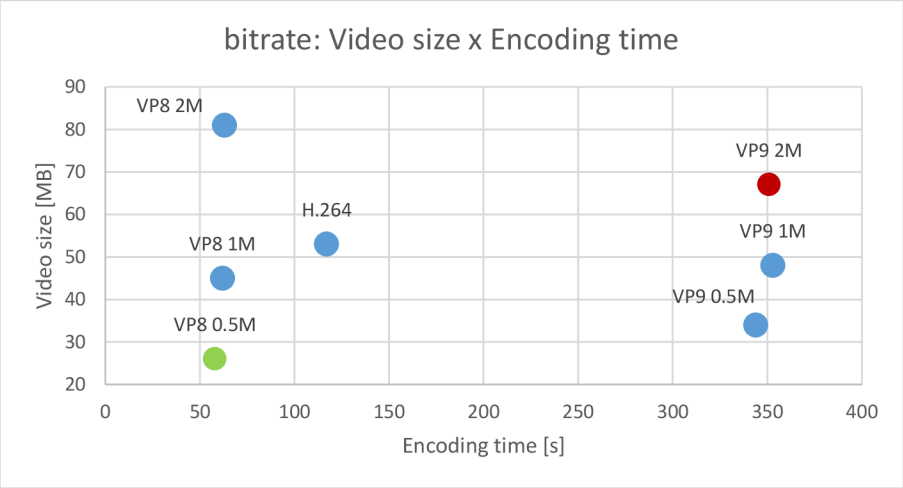
Prioritou byla nejen rychlost zpracování, ale i výsledná velikost souboru a subjektivní kvalita výsledného videa.

- **Hardware**: procesor `Intel Xeon E5-2630` (8 jader), 4096 MB RAM
- **Software**: `Debian 8`, `OpenVZ` kontejner, `ffmpeg` GIT kompilace: `N-81025-g25ca74d`
- **Příkaz pro VP8**: `ffmpeg -i original.webm -g 60 -cpu-used 16 -threads 16 -c:v libvpx -b:v 1M output.webm`
- **Příkaz pro VP9**: `ffmpeg -i original.webm -g 60 -cpu-used 8 -threads 16 -c:v libvpx-vp9 -b:v 1M output.webm`[1]
- **Vstupní video**: soubor `record1.webm`, záznam různorodé práce uživatele na PC, rozlišení 1920x1200, 30 FPS, délka 7:11 min, velikost 58,4 MB, zdrojový kodek `VP8` s variabilním požadovaným `bitrate` 2 Mbps, enkodér `Chrome 51`.[2]

Graf 5.1 shrnuje naměřené hodnoty. Jednoznačně lze vyloučit použití nového kodeku `VP9`, protože i v nejnovější verzi nástroje `ffmpeg` trvá kódování **téměř 5x pomaleji** než v případě kodeku `VP8`. Z hlediska velikosti výsledného videa je na tom `VP8` ve většině

[1] kodek neumožňuje nastavit vyšší hodnotu parametru `cpu-used`

[2] Jak bylo řečeno v kapitole 4.2, jedná se pouze o doporučení. Výsledný datový tok je ve skutečnosti cca 1,1 Mbps.



Obrázek 5.1: Srovnání kombinací kodeků a datových toků s výslednou velikostí a časem překódování. Čím nižší hodnoty, tím lepší. Zelený je výsledek nejlepší, červený nejhorší.

-g	**čas zpracování**	**velikost videa**
30	1 m 1 s	52 MB
60	1 m 2 s	45 MB
90	0 m 58 s	42 MB

Tabulka 5.1: Vliv nastaveného `GOP` na výslednou velikost videa a čas zpracování.

případů také lépe. Jako poslední důvod lze uvést vyšší náročnost na `HW` uživatele při zpětném dekódování.

Z důvodu subjektivně výrazného rozdílu mezi datovým tokem 0,5 Mbps a 1 Mbps jsem nakonec pro použití v aplikaci zvolil kombinaci kodeku `VP8` s datovým tokem 1 Mbps. Čas zpracování mezi jednotlivými datovými toky se již tolik neliší a 1 Mbps se tak zdá být **optimální kompromis** mezi kvalitou a velikostí videa.

Pro úplnost je v grafu uveden i kodek `H.264` s horšími výsledky než má kodek `VP8` a to v obou sledovaných parametrech. Překódování všech videí rovnou do `H.264`, který je lépe podporovaný, by tak znamenalo přibližně **dvojnásobný čas zpracování**.

Vstupní video o velikosti 58,4 MB tak bylo zmenšeno na 45 MB a překódování trvalo přibližně 15 % času potřebného k přehrání videa.

Se zvolenou kombinací byl dále vykonán test s volbou hodnoty parametru `-g` udávající velikost `GOP` (Group of Pictures) struktury. Výsledky jsou shrnuty v tabulce 5.1.

Lze vidět, že nastavená hodnota **nemá vliv na čas zpracování** a ovlivňuje výslednou velikost videa jen minimálně. Z důvodu lepší práce s menšími segmenty videa (základní jednotka) a podrobnější možností posuvu ve videu byla pro aplikaci zvolena hodnota parametru `-g` rovna 60.

5.2 Srovnání hashovacích funkcí

Ačkoliv již není nutné v prohlížečích implementovat hashovací funkce ručně, ale stačí použít volání nad objektem `window.crypto`, bylo nutné vybrat funkci, kterou bude možno počítat

F(x) / video		scene1-source (121 MB)	scene2-source (140 MB)	scene3-source (24 MB)
SHA-384	**čas**	64,4 s	75,1 s	13,2 s
	CPU	16,3 %	16,7 %	16,1 %
	RAM	278,6 MB	313,0 MB	77,2 MB
SHA-256	**čas**	53,1 s	62,1 s	10,8 s
	CPU	17,5 %	16,4 %	16,6 %
	RAM	301,9 MB	304,1 MB	77,4 MB
SHA-1	**čas**	24,3 s	28,5 s	5,2 s
	CPU	16,4 %	16,5 %	16,1 %
	RAM	231,9 MB	280,9 MB	110,8 MB

Tabulka 5.2: Naměřené hodnoty vlastností hashovacích funkcí.

rychle a s minimálními nároky na hostitelský systém. Otisk je počítán pro každý 32 kB chunk.

- **Hardware**: procesor `Intel Core i7-6700HQ` (4 jádra), 8192 MB RAM
- **Software**: `Windows 10 64b, Chrome 50`

Pro měření časových úseků je zde i v dalších testech používán časovač s vysokou přesností - `window.perfomance.now()`. Každá hodnota je výsledkem průměru tří měření. Hodnoty `CPU` a `RAM` jsou vždy průměrné za dobu měření.

Z naměřených dat uvedených v tab. 5.2 vychází funkce `SHA-1` jako **nejrychlejší** ve všech testech. Využití procesoru vycházelo zhruba stejně u všech testovaných funkcí. `SHA-1` v poslední testované scéně vykazuje vyšší paměťovou náročnost. Vzhledem k tomu, že bude funkce aplikována na menší shluky dat to však není tak podstatné.

V implementaci je tedy použita funkce `SHA-1`.

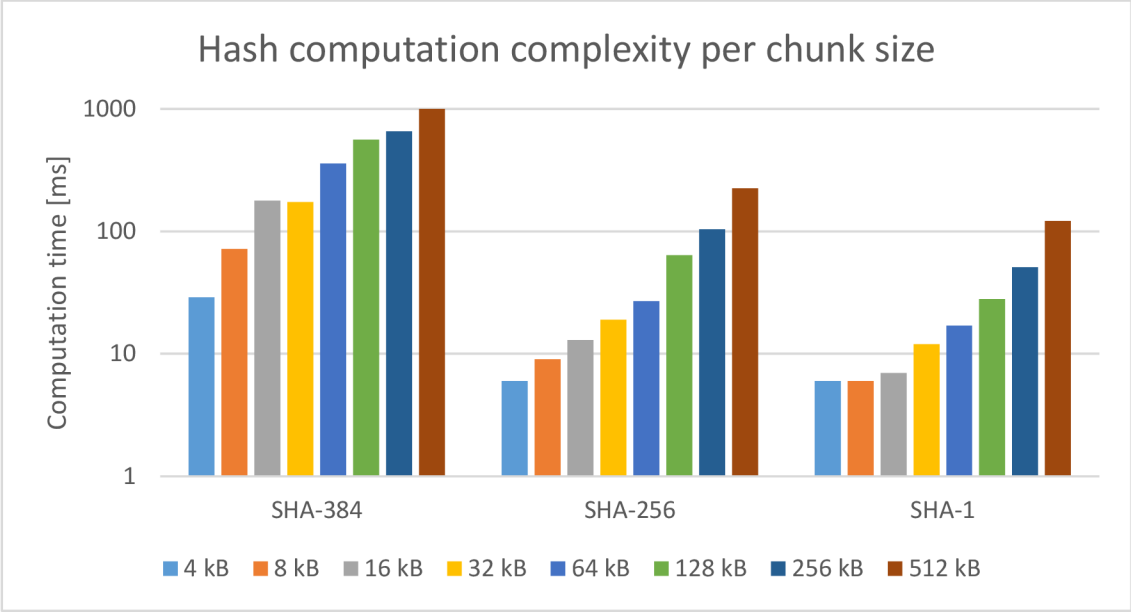
Dále je v souvislosti s volbou hashovací funkce experimentováno s velikostí chunku. Graf 5.2 znázorňuje nejvhodnější velikost chunku pro každou testovanou hashovací funkci. Každé měření bylo vykonáno 100krát ve vlastním okně prohlížeče.

Lze vidět, že výpočetní čas téměř logaritmicky **roste s velikostí chunku** od jisté hraniční hodnoty, která eliminuje vliv statistické chyby. V případě `SHA-1` je tak hraniční hodnotou velikost chunku 16 kB. Nejvíce dat za jednotku času funkce spočítá stejně pro velikosti chunku 32 a 64 kB. To s ohledem na další důsledky (viz část 4.4) zakládá na výslednou volbu hodnoty 32 kB .

5.3 Vektor stahovaných částí

Byl zkoumán vliv počtu stahovaných dílů práce ze signalizačního serveru na celkový čas dokončení stažení videa. Jedná se o parametr `bufferChunks`. Čím menší hodnoty nabývá, tím lépe uzel zohledňuje aktuální stav (má častěji aktuální data), ale je nutná častější komunikace se serverem, který práci rozděluje.

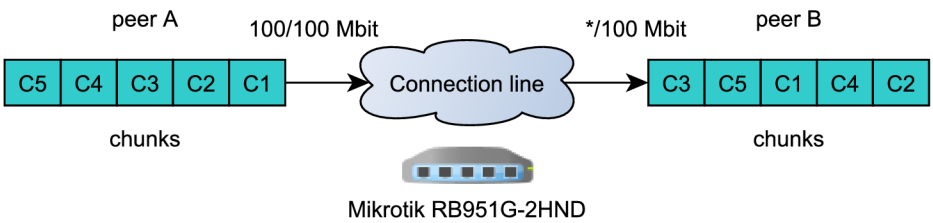
- **Hardware A**: procesor `Intel Core i7-6700HQ` (4 jádra), 8192 MB RAM



Obrázek 5.2: Volba velikosti chunku na základě doby výpočtu per funkce.

- **Software A**: `Windows 10 64b, Chrome 51`
- **Hardware B**: procesor `Intel Core i5-430M` (2 jádra), 8192 MB RAM
- **Software B**: `Fedora 21 64b, Chrome 51`
- **Video**: soubor `record2.webm`, záznam různorodé práce uživatele na PC, rozlišení 1920x1200, 30 FPS, délka 0:58 min, velikost 8,4 MB, zdrojový kodek `VP8` 1 Mbps variabilní bitrate, enkodér `ffmpeg N-81025-g25ca74d`.

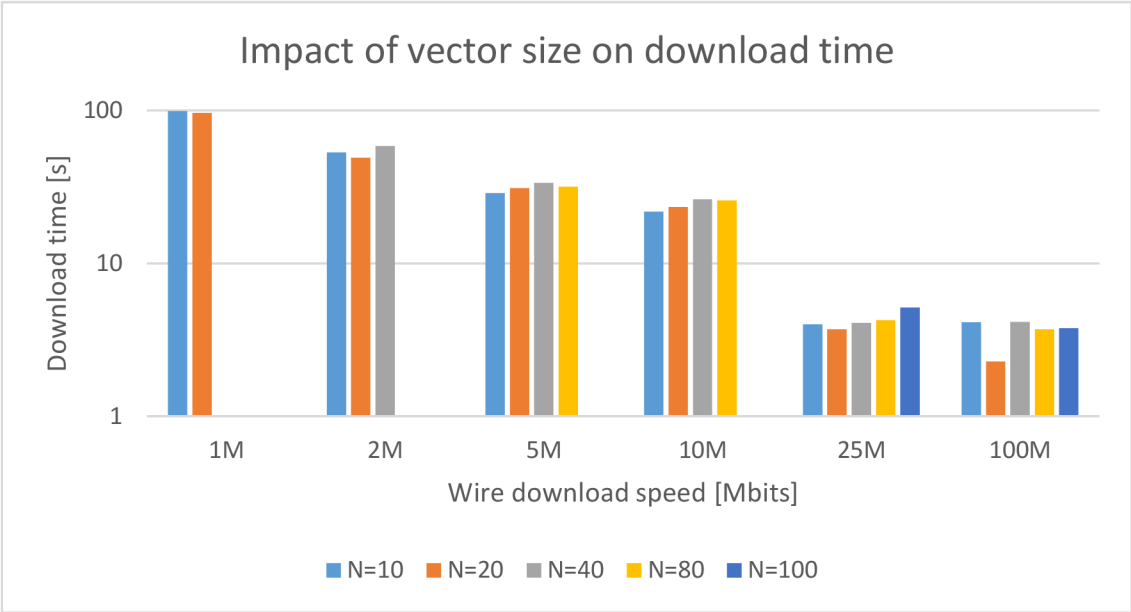
Do experimentu byly zahrnuty dva uzly (A a B), kdy uzel A disponuje všemi částmi distribuovaného videa. Uzel B se k uzlu A připojuje přes postupně různě **limitované přenosové pásmo** a stahuje vždy celé video výhradně od uzlu A. Rychlost linky byla limitována pouze pro uzel B a to na mezilehlém prvku `Mikrotik RB951G-2HND` metodou jednoduché fronty (viz obr. 5.3).



Obrázek 5.3: Síťová topologie pro omezení rychlosti přenosového kanálu.

Během experimentu je pro eliminaci stahování ze serveru `TURN` vypnuta funkce prioritního stahování (viz část 4.7). Vždy je tak využit pouze 1 stahovací slot (uzel B) a 1 slot odesílací (uzel A).

Oba prohlížeče jsou před každým experimentem restartovány a vynesené hodnoty odpovídají průměru tří po sobě jdoucích měření.



Obrázek 5.4: Závislost rychlosti stažení videa na velikosti chunku.

Graf 5.4 s logaritmickým měřítkem potvrzuje **výrazný vliv rychlosti linky** na výsledný čas stažení souboru. Současně však také ukazuje, že volba počtu prvků vektoru má na rychlost jen **minimální vliv**.

Při nízkých rychlostech linky a vysokých počtech prvků ve vektoru docházelo k výpadkům stahování na hlídacích časovačích prostojů ve frontách a zajištění kvality přenosu. Aplikace se snažila z takových stavů zotavovat (typicky pokusem o stažení z `TURN` serveru či přidáním uzlu na `blacklist`). Tyto hodnoty nebyly dále uvažovány.[3]

Z grafu dále vyplývá, že od jisté hraniční rychlosti (zde 25 Mbps) již téměř **nezáleží na rychlosti linky**, protože aplikace neumí takový potenciál řádně využít. Pro stahování videa to nevadí, pro co nejrychlejší přenos souborů v obecné `P2P` síti by se mohlo jednat o úzké hrdlo, které je třeba zoptimalizovat. Stále se však jedná o stahování pouze z jednoho uzlu. V případě více uzlů se problém minimalizuje. Současně je vzhledem k typicky asymetrickým přípojkám výrazně méně uzlů schopných odesílat rychlosti vyšší než 25 Mbps.

Pro vyloučení statistické odchylky v případě velmi vysokých rychlostí (25 a 100 Mbps), kdy je stahování dokončeno již po několik vteřinách, byl test zopakován s delším videem. Naměřené hodnoty zde neuvádím, protože jen potvrdily zjištění z předchozího experimentu – vliv počtu částí vektoru na výslednou rychlost je minimální.

Velikost vektoru tak byla zvolena dle poměru mezi komunikační režií[4] a pružností reflektování změn v `P2P` síti (více viz část 5.3).

Pro $N = 20$ a video o délce $L = 200$ chunků (velikost $S = L{*}chunkSize = 200{*}32 = 6,4$ MB) je nutné komunikovat se serverem $\frac{L}{N} = \frac{200}{20} = 10$krát. Pružnost je v případě předpokládané rychlosti stahování 2 Mbps rovna $\frac{N{*}chunkSize}{speed} = \frac{20{*}32}{(\frac{2{*}1024}{8})} = 2,5$ s. Každé 2,5 s tak teoreticky budou stahována aktuální data. Ve skutečnosti jsou však vektory stahovány

[3] a to je důvod, proč je v grafu u některých rychlostí méně naměřených sloupců

[4] dvojnásobná velikost vektoru znamená poloviční potřebu komunikace se serverem

extendTreshold	čas stažení
0,2	2 m 13 s
0,5	2 m 18 s
0,7	2 m 24 s
0,9	2 m 26 s

Tabulka 5.3: Vliv nastaveného parametru `extendThreshold` na čas stažení videa.

v předstihu[5] a aktuální data jsou získávána každých $\frac{2,5}{2} = 1,25$ s.

Zvolená výchozí hodnota pro velikost vektoru je tedy $N = 20$.

Zmíněný práh pro včasné stahování nových vektorů (`extendTreshold`, více viz část 4.4) byl předmětem dalšího testu. Na stejném videu a stejné topologii sítě s upravenou rychlostí na `5M/5M` pro oba uzly.

Čas stažení v závislosti na hodnotě prahu je uveden v tab. 5.3. Vyplývá z něj **mírná výhoda** v použití nižšího prahu. **Čím nižší práh, tím rychleji** jsou potřebné chunky připraveny ke stažení, ale také **tím více zbytečně přenášených dat**. **Proto byl jako kompromis zvolen práh s hodnotou 0,5.** Při dokončeném stažení poloviny chunků z fronty `Q` se tak stáhne další vektor o velikosti 20 prvků.

5.4 Počet stahovacích slotů

Experimentem je sledováno, zdali více současně otevřených spojení k více uzlům znamená nárůst rychlosti stažení videa. Vždy byly do sítě postupně přidávány uzly s kompletně staženým videem a vždy stahoval současně v síti pouze jediný uzel. Pro vyloučení odchylek bylo během testů odpojeno vykreslovací jádro a byla tak testována čistě přenosová `P2P` vrstva aplikace.[6]

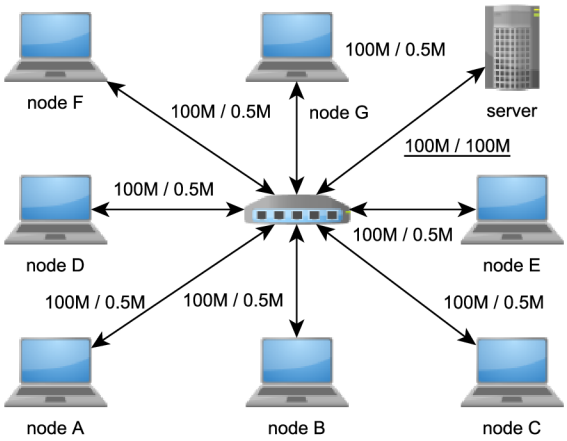
S omezením rychlosti

První experiment zahrnuje až 7 uzlů spuštěných ve virtuálních instancích na společném hostitelském systému. Tyto uzly jsou opět připojeny do společného síťového prvku `Mikrotik`, který je nastaven pro omezení rychlosti odesílání dat každého uzlu zvlášť na **0,5 Mbps** (viz obr. 5.5). Toto se nevztahuje na signalizační server a není ani dotčena rychlost stahování žádného uzlu.

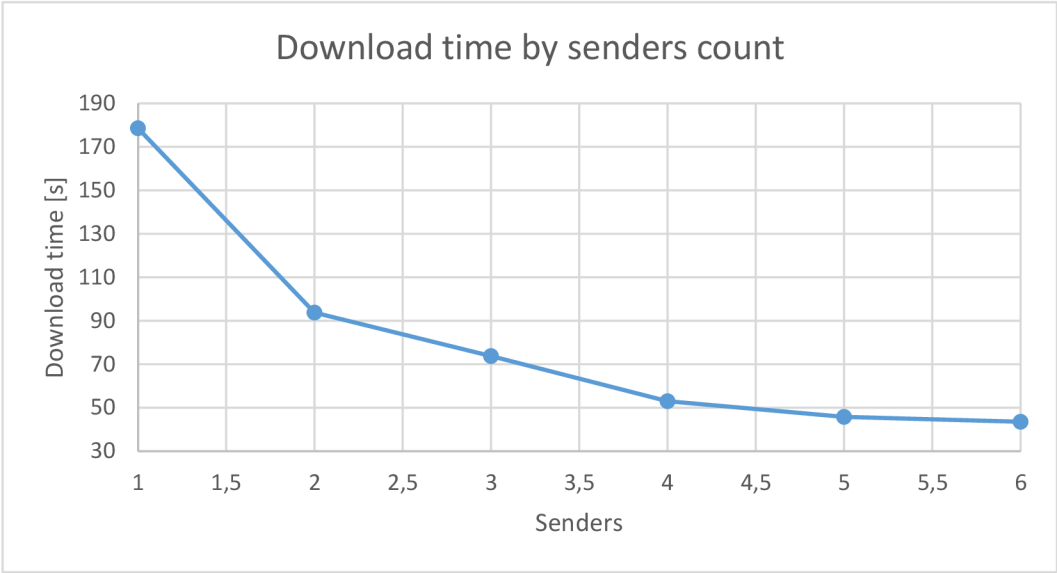
- **Hardware hostitel:** procesor `Intel Core i7-6700HQ` (4 jádra), 8192 MB RAM
- **Software hostitel:** `Windows 10 64b, Chrome 51`
- **Hardware instance:** až 4 jádra, 2048 MB RAM
- **Software instance:** `Linux Mint 17 64b, Chrome 51`
- **Video:** soubor `record2.webm`, viz část 5.3

[5] určeno hodnotu `extendThreshold`, viz test dále

[6] Ukázalo se totiž, že v případě více uzlů má na slabších strojích vykreslení videa a grafické sítě až dvojnásobný čas stažení videa.



Obrázek 5.5: Síťová topologie pro experiment s počtem DL slotů.



Obrázek 5.6: Vliv počtu odesílatelů na rychlost stažení videa.

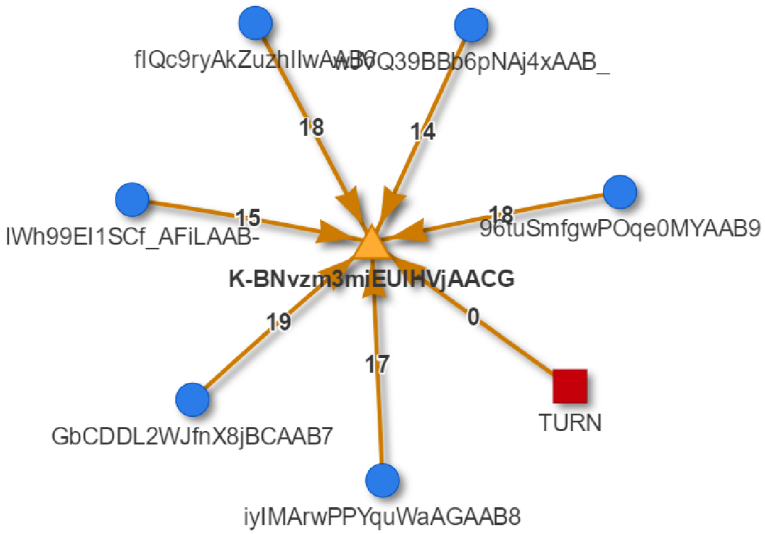
Graf 5.6 znázorňuje, jak s rostoucím počtem uzlů čas stahování **klesá**. Nicméně zhruba od čtyř uzlů již klesání **výrazně zpomaluje** a lze se domnívat, že by ještě větší počet současně připojených uzlů rychlost nezvýšil ba naopak pravděpodobně snížil kvůli komunikační režii. Je však nutno podotknout, že jednotlivé uzly běžely na stejném hostitelském systému jehož prostředky byly v případě 6 virtualizovaných strojů značně vyčerpány.

Pro zajímavost byl ještě vykonán test s maximálním počtem odesílatelů, ale limitovaným počtem stahovacích slotů u příjemce. Hodnoty jsou uvedeny v tabulce 5.4.

Dle očekávání lze vidět, že omezení počtu slotů vede k rychlostem obdobným jako v případě reálné dostupnosti právě tolika odesílatelů. Lehce vyšší hodnoty než v grafu 5.6 jsou nejspíše způsobeny odchylkami heuristik pro výběr odesílatele a tedy případné možnosti přepojení k jinému dostupnému odesílateli.

počet DL slotů	čas stažení
10	43,54 s
4	56,58 s
2	98,42 s

Tabulka 5.4: Omezení počtu souběžných stahování při šesti odesílatelích.



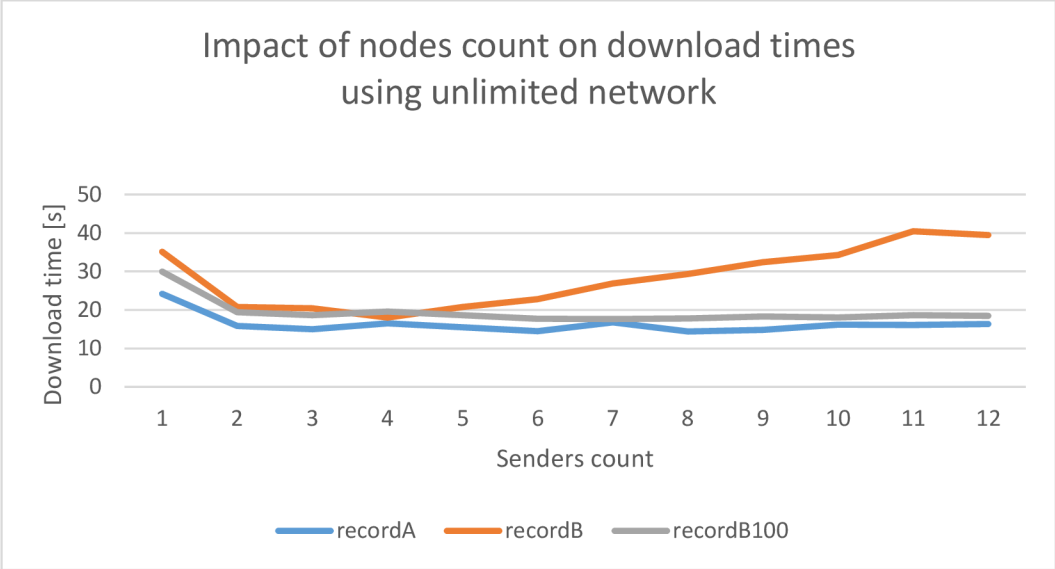
Obrázek 5.7: Ukázka, kdy uzel stahuje od 6 ostatních bez nutnosti použití `TURN` serveru.

Bez omezení rychlosti

Druhý experiment byl vykonán v reálné situaci v prostorách laboratoří `CVT FIT VUT` v Brně. Jako přenosový kanál sloužila místní lokální gigabitová síť. Vzhledem k tomu, že nebylo možné rychlost uměle snížit a experimentovat tak i s rychlostí komunikace, experimenty probíhaly **za plné rychlosti** jinak nevyužívané sítě. K dispozici bylo více stanic, než bylo možné v rámci virtualizace spustit.

- **Hardware uzly:** `Intel Haswell i3-4360` (2 jádra), 8192 MB RAM
- **Software uzly:** `Cent OS 6, Firefox 45`
- **Video A:** soubor `record3.webm`, záznam různorodé práce uživatele na PC, rozlišení 1920x1200, 30 FPS, délka 7:11 min, velikost 45,4 MB, zdrojový kodek `VP8` 1 Mbps variabilní bitrate, enkodér `ffmpeg N-81025-g25ca74d.`
- **Video B:** soubor `record4.webm`, záznam přehrávání filmu, rozlišení 1920x1200, 30 FPS, délka 31:39 min, velikost 152,8 MB, zdrojový kodek `VP8` 1 Mbps variabilní bitrate, enkodér `ffmpeg N-81025-g25ca74d.`

Vzhledem k tomu, že nebyla přenosová rychlost mezi uzly jakkoliv omezena, od hraničního počtu uzlů (zde cca 3) již dle grafu 5.8 **nemá smysl** stahovat video od více uzlů. To však platí za velmi specifických podmínek (dostatek velmi rychlých odesílatelů na stejné `LAN`). S rostoucím počtem takových odesílatelů začne rychlost stahování dokonce klesat z důvodu komunikační režie. Pro zajímavost je v grafu vynesen i test se záznamem videa



Obrázek 5.8: Vliv počtu odesílatelů na rychlost stažení videa (`LAN`).

`recordB`, ale s nastaveným počtem prvků vektoru na $N = 100$. Lze vidět, že je tímto **výrazně omezen** neduh zpomalujícího stahování kvůli režie. Je to způsobeno méně častou komunikací se signalizačním serverem, kdy se musí čekat na přidělení další práce jednotlivým uzlům. Ty jsou tak rychlé, že je tato komunikace úzkým hrdlem.

Bohužel **nebylo možné** otestovat stejnou situaci na reálnějších podmínkách **s nastavením omezení rychlosti** jako v případě předchozího testu. Výsledky s tolika nezávislými stanicemi by mohly být velmi zajímavé a jsou tedy předmětem navazujících prací.

5.5 Video on Demand služby

V rámci zajištění `VoD` služeb bylo testováno, zdali prioritní režim skutečně pomáhá v plynulosti přehrávání a jak dlouho trvá stažení potřebných částí videa při náhodném **posuvu ve videu**. Současně jsou tyto testy parametrizovány nastavením velikostí klouzavého okna (viz část 4.7) a je zkoumán její vliv na **plynulost přehrávání**.

Síťová topologie je ustavena stejně jako v případě testu 5.3. Stejně tak použitý `HW`, `SW` i testované video. Dva uzly tedy komunikují postupně různě omezovanou rychlostí. Změna je v nastavené strategii výběru chunku, která je místo běžné `RarestFirst` **změněna na** `LatestFirst`. Tím je zajištěno, že stahované chunky od začátku, popř. při posuvu ve videu, vynucuje prioritní režim a ne strategie serveru.[7] Oba přístupy tak stahují data proti sobě.

V tab. 5.5 je ve sloupci „čekání na data“ uveden čas, kdy video stálo z důvodu chybějících segmentů pro pokračování v přehrávání. Tento čas nezahrnuje čas do prvního spuštění přehrávání. Ten je uveden ve sloupci „start přehrávání“.

Lze vidět, že velikost okna má **výrazný vliv** na zmíněné čekání na data i na rychlost stažení celého videa. S vyšší velikostí není ani více využíván `TURN` server.

Za povšimnutí stojí, že větší okno **nezrychlí čas do startu** přehrávání videa. To je způsobeno režií se získáním prvního chunku, který spustí sekvenční stahování. Tento přístup by tedy měl být vylepšen.

[7] takové chunky lze při znázornění vidět jak přibývají odzadu

velikost okna	rychlost uzlu	start přehrávání	celé stáhnutí	turn/p2p	čekání na data
W=3s	**1 M**	4,11 s	54,85 s	155/103	16 s
	2 M	3,01 s	37,57 s	131/127	8 s
	5 M	3,44 s	24,25 s	86/172	0 s
W=5s	**1 M**	5,02 s	48,62 s	158/100	8 s
	2 M	3,63 s	33,32 s	119/139	0 s
	5 M	4,02 s	22,91 s	81/177	0 s

Tabulka 5.5: Vliv velikosti klouzavého okna a rychlosti uzlu na kvalitu `UX`.

velikost okna	rychlost uzlu	čas spuštění
W=3s	**1 M**	9,12 s
	2 M	7,17 s
	5 M	3,85 s
W=5s	**1 M**	2,58 s
	2 M	2,79 s
	5 M	2,05 s

Tabulka 5.6: Vliv velikosti klouzavého okna a rychlosti uzlu na zotavení z přesunu ve videu.

Následoval test náhodného posuvu ve videu se sledováním, po jaké době je přehrávání opět obnoveno. Z hodnot v tab. 5.6 je patrné, že větší velikost okna **výrazně zlepšuje** `UX` přehrávače, protože jsou data vybaveny **mnohem rychleji**.

Jako výchozí hodnota velikosti okna tak byla zvolena hodnota 5 vteřin. Tu je však nutno brát s rezervou, protože přesahy segmentů mohou způsobit označení za prioritní i delší úseky než 5 s (viz obr. 4.21).

Připojování částí videa po segmentech je také důvodem, proč nelze kalkulovat pouze s datovým tokem, který by při dosažení uzlu měl být dostatečný k plynulému přehrání videa. Vždy se čeká na stažení celého segmentu, protože se video nepřipojuje po jednotlivých bajtech. I proto trvá prvotní start videa několik vteřin navíc.

5.6 Kvalita přenosového média

Vzhledem k tomu, že protokol `SCTP` umožňuje volbu **spolehlivosti přenosu** stejně tak jako volbu **garance pořadí doručení** dat,[8] bylo vhodné tyto kombinace otestovat v reálném prostředí a vybrat pro tuto aplikaci nejen teoreticky ale i prakticky nejvíce vyhovující variantu.

Pro simulaci různých neduhů reálných sítí byl po celou dobu vývoje i během testování využíván nástroj `Clumsy`.[9] Ten umožňuje detailní nastavení potřebných parametrů přenosového kanálu pro další experimenty.

Stejně jako v předchozím experimentu, i nyní je použita opět stejná síťová topologie a shodné uzly použité při testování 5.3. Prioritní stahování je však opět vypnuto pro eliminaci stahování z `TURN` serveru. Uzel A disponuje celým videm a poskytuje ho ke stažení

[8] Více o teoretické volbě parametrů viz 4.8.

[9] Podpora pro `Windows`, licence `MIT`.

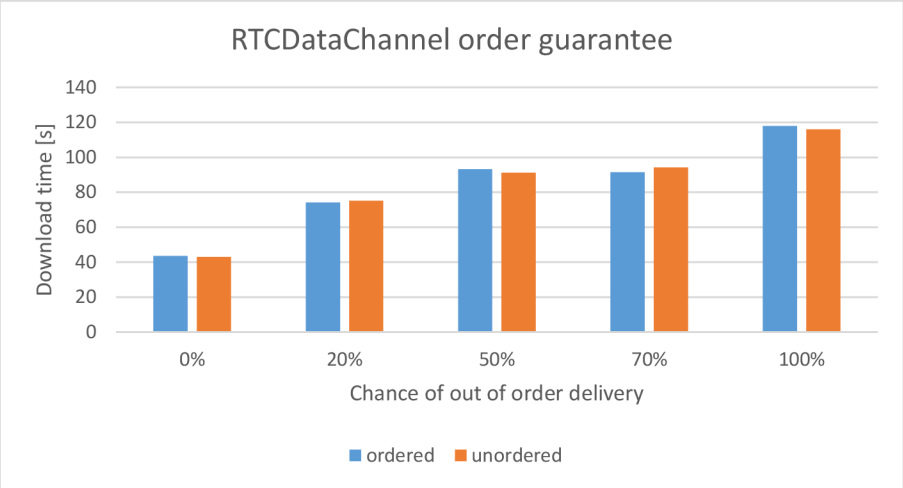
uzlu B. Žádné jiné uzly v síti nefigurují.

- **Rychlost uzlů**: 100/100 Mbps
- **Video A**: soubor `record3.webm` (pro test pořadí zpráv, více viz část 5.4)
- **Video B**: soubor `record2.webm` (pro test spolehlivosti doručování, více část 5.4)

Pořadí zpráv

Aplikace byla navržena tak, ať je každá přenášená zpráva **samostatnou jednotkou** (viz část 4.8). Není tak zavedeno žádné čekání na zprávy v pořadí a tudíž není překvapením, že graf 5.9 potvrzuje podobné časy stažení videa v obou případech. Menší režie negarantovaného kanálu se zde tolik neprojevila, protože byl přítomen jediný uzel. V případě více uzlů by se tento rozdíl adekvátně zvýšil.

Důvod pomalejších přenosů v případě častějšího doručování mimo pořadí může být způsoben **fragmentací paketů**. Každá zpráva, byť v kanálu nezaručujícím pořadí, je fragmentována dle principu fragmentace IP datagramů. Transportní vrstva `SCTP` tak každopádně musí čekat na pořadí doručení těchto fragmentů před zpřístupněním zprávy aplikační vrstvě.



Obrázek 5.9: Garantovaný (`ordered`) a negarantovaný (`unordered`) `RTCDataChannel`.

Není důvod používat kanál garantující pořadí doručení a proto je tato vlastnost kanálu v implementaci vypnuta.

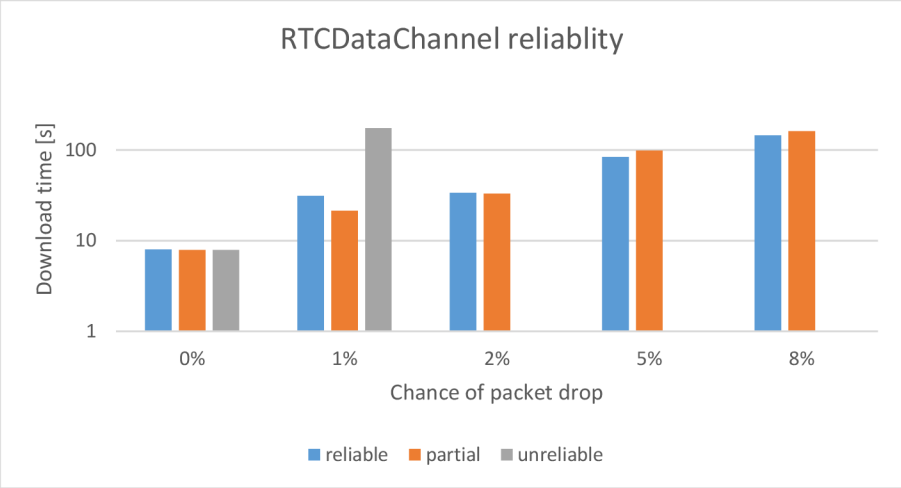
Spolehlivost doručování

Z hlediska rychlosti aplikace důležitějším parametrem přenosového kanálu je to, zdali vůbec garantuje doručení zprávy či nikoliv. Výkonový rozdíl mezi `TCP` a `UDP` je výrazný[10] a tudíž bylo i toto nastavení `SCTP` kanálu podrobeno experimentu.

Mimo spolehlivé či nespolehlivé doručování dovoluje `SCTP` i **částečně spolehlivé** doručování (viz část 4.8). Toto bylo testováno s hodnotou $maxPacketLifeTime = 5$ s.[11]

[10] Pokud není nutné přenášet ztracené datagramy znovu.

[11] Což je hodnota shodná s maximální dobou čekání na doručení chunku, po které již aplikační vrstva chunk stejně zahodí.



Obrázek 5.10: Spolehlivý (`reliable`), nespolehlivý (`unreliable`) a částečně spolehlivý (`partial`) `RTCDataChannel`.

Graf 5.10 s logaritmickým měřítkem shrnující naměřené údaje v první řadě ukazuje, že použití nespolehlivého kanálu, kdy nedoručené chunky řeší aplikační vrstva jejich znovu odesláním, je v praxi nepoužitelné. Již od 2% šance ztráty paketu stažení videa trvá nepoužitelně dlouho a dále tedy není nespolehlivý kanál testován.

Částečně spolehlivý kanál však funguje překvapivě dobře a v praxi nejvíce reálném komunikačním prostředí s hodnotou `packet loss` do 1 % vykazuje **znatelně rychlejší přenos** (31,24 s versus 20,31 s). V případě častější ztráty paketů (nad 2 %) je pak na tom jen o málo hůře než přenos spolehlivý.

Nemá tedy cenu uvažovat nespolehlivý přenos ani přenos spolehlivý. Pro aplikaci je nejvhodnější implementace částečné spolehlivosti přenosového kanálu.

5.7 Flash crowd

Posledním experimentem byla simulace fenoménu známého jako „Flash crowds“, což značí **nárazový zájem o jeden konkrétní obsah.** Testována byla tedy situace, kdy video, které nemá žádné odesílatele začne ve stejný okamžik stahovat několik uživatelů.

Je sledováno, kolik dat se díky `P2P` síti ušetří na přenosovém pásmu `TURN` serveru a také jak je ovlivněna rychlost stahování videa z takové sítě, kde dochází k výměnám právě stahovaných videí.

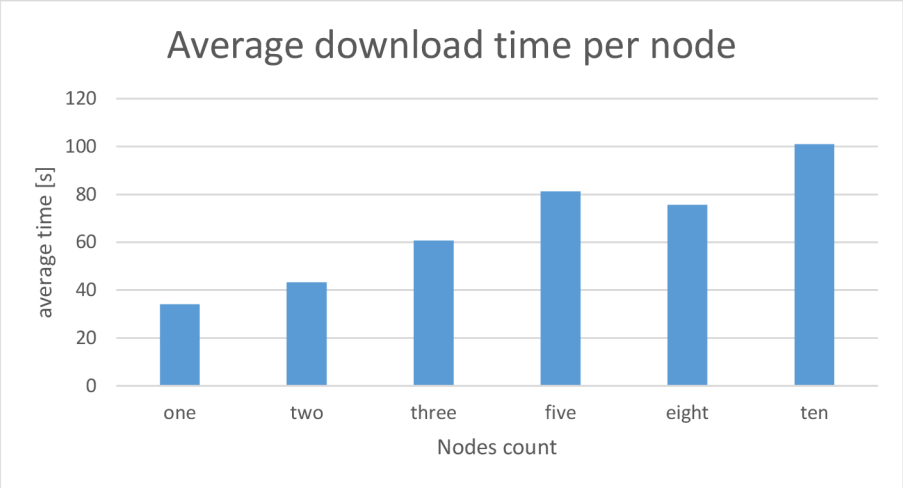
Podmínky experimentu byly stejné jako v případě experimentu 5.4. Jednalo se tedy o rychlostně **neomezenou** školní síť s mnoha stanicemi. Použito bylo video `record4.webm`.[12]

Rychlost stahování

Vliv nárazového zájmu o video na průměrnou rychlost stažení takového videa každým uzlem je uveden v grafu 5.11. V případě jediného uzlu se jedná o nejrychlejší variantu, protože se stahuje přímo z `TURN` serveru. Až na jednu výjimku rychlost stahování téměř lineárně klesá s počtem aktuálně pracujících uzlů. To je opět způsobeno neomezenou rychlostí sítě na což aplikace není primárně nastavena. Průběh je tak podobný jako v případě experimentu

[12] parametry videa viz 5.4

s počtem stahovacích slotů na neomezené síti (viz graf 5.8). Je však nutné podotknout, že se ve všech případě stále jedná o velmi rychlé stažení videa.[13]



Obrázek 5.11: Průměrná rychlost stažení videa uzlem při hromadném stahování.

Poměr zdrojů přenosů

Z hlediska sítě zajímavější pozorování znázorňuje graf 5.12. Zdroj dat v případě jediného uzlu je jasný - vše musí být staženo z `TURN` serveru. V případě 2, 3 a 5 současně stahujících uzlech se poměr dat přenesených `TURN` serverem **lineárně zmenšuje.** Pak však nastává obrat a mezi 8, 10 a pravděpodobně i více uzly se poměr přenosů **opět zvyšuje.**

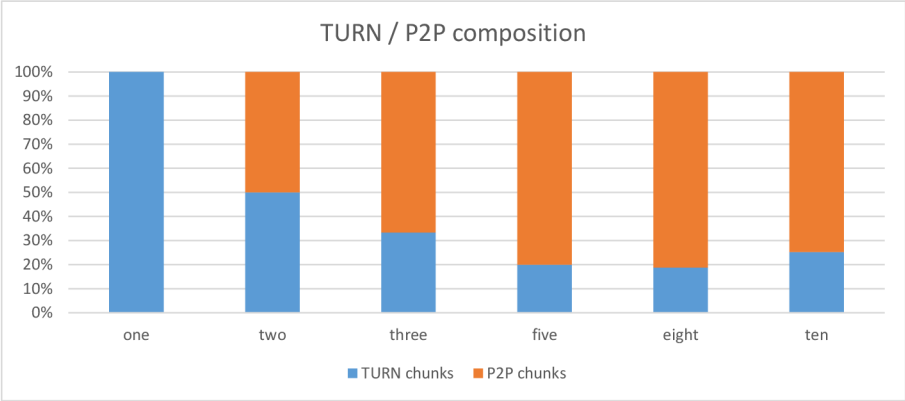
Toto pozorování lze vysvětlit omezeným počtem odesílacích slotů. Ten je ve výchozím nastavení roven 4. V případě 5 uzlů tak teoreticky 1 celou dobu stahuje z `TURN` serveru a 4 současně od něj. V případě více než 5 uzlů však může lehce docházet ke dvěma komplikacím:

- Server generující úkolový vektor, stejně tak jako jednotlivé uzly v síti, **neví**, zdali je uzel ve vektoru vytížený[14] nebo ne. Je tak nutné se postupně všech uzlů doptat, což je pomalé[15] a dojde tak k **vypršení časovače** prostoje ve frontách, což může zapříčinit až stažení z `TURN` serveru (pokud požadovaná data nikdo další nemá).
- Strategie `RarestFirst` rozděluje uzlům práci tak, ať stahují pokud možno **disjunktní množiny** chunků. To v kombinaci s obsazenými sloty ještě snáz způsobí situaci popsanou v předchozím bodě.

Situace by se však neměla výrazně zhoršovat s rostoucím počtem uzlů, protože bude mít každý uzel více kandidátů, od kterých může data zkusit stáhnout. Tím se zvyšuje šance, že bude mít některý volný odesílací slot nebo že jej alespoň jeden kandidát uvolní během tohoto postupného zkoušení. Stahování z `TURN` serveru je totiž vždy až poslední varianta po tom co se vyzkouší veškeré uzly.[16]

Stále jsou však výsledky poměru `P2P` přenosů ku `TURN` přenosům **velmi dobré**. Pro další použití je však nutné tuto situaci dále zanalyzovat za použití více uzlů s omezenou šířkou pásma.

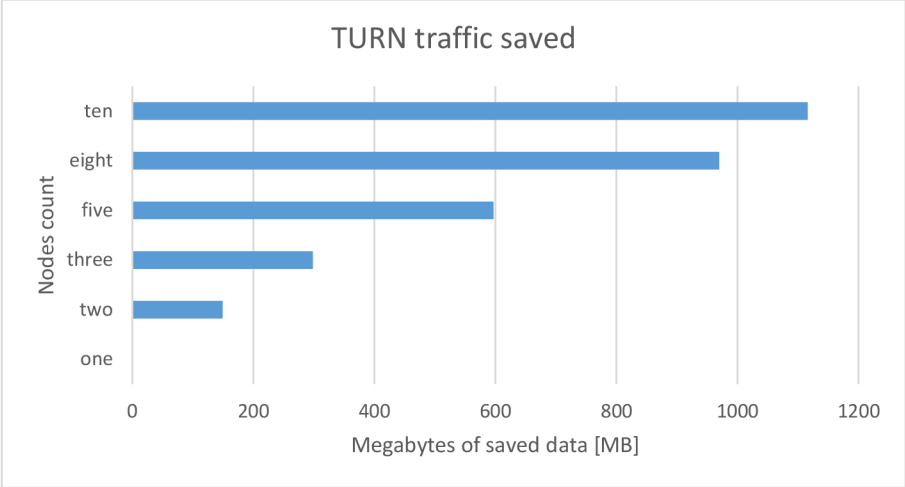
[13] 31 minut dlouhé video staženo průměrně do 1 minuty
[14] co do počtu odesílacích slotů
[15] především během stahování a odesílání plnou rychlost, kdy je i signalizační kanál vytížený
[16] kromě situace s prioritním stahováním, které je v rámci tohoto testu vypnuto



Obrázek 5.12: Poměr dat stažených z P2P a TURN serveru v závislosti na počtu uzlů během flash crowd.

Ušetřené přenosové pásmo

Pro přenášené video jsou konkrétní počty MB ušetřených z přenosového pásma `TURN` serveru uvedeny v grafu 5.13. **V případě 10 uzlů bylo ušetřeno 1116 MB dat během zhruba minuty co přenos trval.** Ještě lepší relativní hodnoty ukazují přenosy v případě 2 až 5 uzlů z důvodu uvedených v předchozí podsekci.



Obrázek 5.13: Absolutní čísla ušetřených dat při přenosu P2P.

5.8 Shrnutí experimentů

V experimentu 5.1 se ukázalo, že je vhodné použít starší kodek `VP8` s datovým tokem 1 Mbps. Dále, že druhý kodek `H.264` je při překódování až 2krát pomalejší. Vliv počtu klíčových snímků nemá vliv na čas zpracování a má jen minimální vliv na velikost videa.

Z hashovacích funkcí v sekci 5.2 vychází nejlépe kombinace `SHA-1` s 32/64 kB chunky.

Experiment 5.3 ověřil výrazný vliv rychlosti linky na čas stažení souboru. Dále vyvrátil vliv počtu prvků stahovaného vektoru na rychlost stažení videa. Nad cca 25 Mbps již rychlost linky nehraje roli, protože aplikace neumí využít takový potenciál. Dřívější stahování úkolového vektoru zrychluje stahování, ale neúměrně zatěžuje signalizační server.

Dále experiment 5.4 ukazuje znatelné zrychlení stahování v případě 4–5 odesílatelů. Poté již rychlost stagnuje či zpomaluje. Na neomezené LAN síti je tato hranice ještě nižší.

Prioritní režim zkoumaný v sekci 5.5 významně zlepšuje `UX` přehrávače a ukázal výrazný vliv velikosti klouzavého okna na rychlost stažení a pružnost při posuvu ve videu.

Potvrzena byla vhodnost volby přenosového kanálu bez garance pořadí doručování zpráv a především výhody částečně spolehlivého doručování v experimentu 5.6.

Závěrečný experiment 5.7 poukázal opět na neoptimální chování aplikace na neomezené síti `LAN`. Současně však ukázal výrazné ušetření přenosového pásma serveru při nárazovém stahování obsahu.

Jedná se o velice zajímavá čísla pro reálný provoz a i přes několik zjištěných komplikací, které musí být ještě vyřešeny, je výsledek testování pro autora práce **více než uspokojivý.**

Kapitola 6

Závěr

Práce se zabývala nejnovějšími technologiemi pro sdílení pracovní plochy v prostředí prohlížečů, pořizováním záznamů, jejich zpracováním a následnou distribucí mezi sledující uživatele pomocí hybridní `Video on Demand P2P` sítě. Ta zajišťuje vysokou horizontální škálovatelnost a snížení provozních nákladů ve srovnání s `client-server` architekturou.

Vzhledem k charakteru nedokončených specifikací `API` pro získání záznamu pracovní plochy uživatele a nedostatku zdrojů informací pro implementaci může analýza těchto specifikací sloužit jako přehledný zdroj pro další vývojáře.

S vydáváním nových verzí prohlížečů bylo nutné práci průběžně aktualizovat. Za nejdůležitější změnu lze považovat implementaci `MediaRecorder API` v prohlížeči `Chrome 49`, kdy smysl aplikace nabral nový rozměr.[1]

Video získané z prohlížeče není použitelné pro následnou `P2P` distribuci se zachováním `VoD` služeb a proto bylo nutné jeho následné zpracování. Úpravy videa v rámci prohlížeče uživatele[2] nejsou z hlediska rychlosti použitelné.

Přenosy dat zrychluje použitý transportní protokol `SCTP` umožňující volbu negarantovaného pořadí doručovaných zpráv a jen částečně spolehlivé doručování.

Prohlížeče umožňují záznam videa pouze do formátu `WebM` a testování ukázalo, že potřebné konverze jsou v tomto formátu téměř dvakrát rychlejší než v případě `MPEG-4`.[3]

Vývoj `VoD P2P` sítě představoval větší výzvu než vývoj klasické `P2P` sítě kvůli zajištění plynulého přehrávání videa již během distribuce s možností libovolného posuvu ve videu. Hybridní architektura se stále dostupným serverem poskytující videa[4] zajištění `VoD` služeb umožňuje. Stále je však implementace `P2P` sítí mezi prohlížeči nová neprobádaná oblast.

Bylo otestováno, že i video dlouhé přes 30 minut zvládá bez problémů hrát plynule i na pomalejší lince (2 Mbps) a to s možností rychlého posuvu ve videu.

Výsledná implementace splnila a předčila zadání diplomové práce stejně jako má očekávání. Aplikace dokáže spolehlivě pořizovat videa a distribuovat je mezi uživateli. Plně podporované jsou prohlížeče `Chrome`, `Firefox` a `Opera` na běžných platformách.

Již nyní implementovaná síť **výrazně šetří přenosové pásmo** přenosem dat, které by jinak musely být poskytovány centrálním serverem (viz výsledek 5.7). To je její hlavní účel, který tak splňuje. Dále také síť dobře zvládá **rychlejší stažení** dat v případě současného

[1] Dále např. přidání podpory kodeku `VP9` v prohlížeči `Firefox 47` a `Chrome 49`, což znamenalo potřebu otestovat připravenost jejich použití a případně tak pokračovat v implementaci s jejich použitím.

[2] pomocí nástroje `ffmpeg` zkompilovaného do `asm.js`

[3] Navzdory kapitole 3, kde bylo doporučeno používat kodek `VP9` kdykoliv je to možné, konečná konverze probíhá do formátu `VP8`, kvůli výrazně vyšší rychlosti zpracování (viz experiment 5.1).

[4] avšak jen pro případ nutnosti, jinak se vždy preferuje P2P

spojení s několika uzly (viz část 5.4).

Na práci lze navázat implementací obecné `P2P VoD CDN` sítě umožňující jednoduchým nasazením do existující aplikace ušetřit provozovateli přenosové pásmo. Vzhledem ke komplexnosti aplikace jsou další návrhy na rozšíření uvedeny v příloze A.

Klientská i serverová část byla vyvíjena v jazyce `JavaScript`. Jeho neblokující asynchronnost v kombinaci s návrhovým vzorem `Promise` perfektně zapadá do potřeb aplikace.[5] Výsledné velmi rychlé zpracování zajišťuje potřebnou vertikální škálovatelnost řešení.

Výsledky realizace lze považovat za důkaz, že má smysl na aplikaci dále pracovat a obecně, že má smysl se zabývat `P2P` přenosy i v prostředí webových prohlížečů.

Na tuto diplomovou práci autor naváže v rámci spuštění online projektu, který na výsledcích práce staví.[6] Spuštění je plánováno během 4. Q 2016 na adrese `https://ctrlv.tv`.

[5] časté čekání na I/O či síťové operace

[6] S nasazením v různorodém prostředí sítě Internet je očekáváno mnoho dalších komplikací a situací, které nebyly testovány a bude třeba je vylepšit.

Literatura

[1] Brown, T. B.; Butters, K.; Panda, S.: *Jump Start HTML5*. SitePoint, první vydání, 2 2014, ISBN 9780980285826.
URL http://amazon.com/o/ASIN/0980285828/

[2] Bugzilla@Mozilla: Bug 624883- iframe with src='view-source...' should be treated as an unknown scheme, 2014.
URL https://bugzilla.mozilla.org/showbug.cgi?id=624883

[3] Cozzi, P.: *WebGL Insights*. CRC Press, 2015, ISBN 9781498716086.
URL https://books.google.cz/books?id=6crECQAAQBAJ

[4] Developers, G.: WebRTC Plugin-free realtime communication.
URL http://io13webrtc.appspot.com/

[5] Grigorik, I.: *High Performance Browser Networking: What every web developer should know about networking and web performance*. O'Reilly Media, první vydání, 9 2013, ISBN 9781449344764.
URL http://amazon.com/o/ASIN/1449344763/

[6] Inc, G.: Stable channel update, 2013.
URL http://googlechromereleases.blogspot.com/2013/07/stable-channel-update.html

[7] International, E.: ECMAScript? 2015 Language Specification. 2015.
URL http://www.ecma-international.org/ecma-262/6.0/

[8] Johnston, A.: Session Description Protocol (SDP) Offer/Answer Examples. RFC 4317, December 2005.
URL https://tools.ietf.org/html/rfc4317

[9] Josefsson, S.: The Base16, Base32, and Base64 Data Encodings. RFC 4648, October 2006.
URL https://tools.ietf.org/html/rfc4648

[10] Libby, A.: *HTML5 Video How-To*. Packt Publishing, 10 2012, ISBN 9781849693646.
URL http://amazon.com/o/ASIN/1849693641/

[11] Lombardi, A.: *WebSocket: Lightweight Client-Server Communications*. O'Reilly Media, první vydání, 9 2015, ISBN 9781449369279.
URL http://amazon.com/o/ASIN/1449369278/

[12] Loreto, S.; Romano, S. P.: *Real-Time Communication with WebRTC: Peer-to-Peer in the Browser*. O'Reilly Media, první vydání, 5 2014, ISBN 9781449371876.
URL http://amazon.com/o/ASIN/1449371876/

[13] Manson, R.: *Getting Started with WebRTC*. Packt Publishing, 9 2013, ISBN 9781782166306.
URL http://amazon.com/o/ASIN/1782166300/

[14] Oksanen, I.; Hazaël-Massieux, D.; Kostiainen, A.: HTML Media Capture. Last call WD, W3C, Červen 2014,
http://www.w3.org/TR/2014/WD-html-media-capture-20140619/.

[15] Parker, D.: *JavaScript with Promises*. O'Reilly Media, první vydání, 6 2015, ISBN 9781449373214.
URL http://amazon.com/o/ASIN/1449373216/

[16] Pfeiffer, S.: *The Definitive Guide to HTML5 Video (Expert's Voice in Web Development)*. Apress, 2011 vydání, 12 2010, ISBN 9781430230908.
URL http://amazon.com/o/ASIN/1430230908/

[17] Pfeiffer, S.; Green, T.: *Beginning HTML5 Media: Make the most of the new video and audio standards for the Web*. Apress, druhé vydání, 6 2015, ISBN 9781484204610.
URL http://amazon.com/o/ASIN/1484204611/

[18] Saint-Andre, P.; Smith, K.; Tronçon, R.: *XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies*. O'Reilly Media, 2009, ISBN 9780596555597.
URL https://books.google.cz/books?id=SG3jayrd41cC

[19] Shen, X.; Yu, H.; Buford, J.; aj.: *Handbook of Peer-to-Peer Networking*. Lecture Notes on Coastal and Estuarine Studies, Springer US, 2010, ISBN 9780387097510.
URL https://books.google.cz/books?id=nXk_AAAAQBAJ

[20] Simpson, K.: *You Don't Know JS: ES6 & Beyond*. O'Reilly Media, první vydání, 12 2015, ISBN 9781491904244.
URL http://amazon.com/o/ASIN/1491904240/

[21] Smith, P.: *Professional Website Performance: Optimizing the Front-End and Back-End*. Wrox, první vydání, 11 2012, ISBN 9781118487525.
URL http://amazon.com/o/ASIN/1118487524/

[22] Sosinsky, B.: *Networking Bible*. Wiley, první vydání, 9 2009, ISBN 9780470431313.
URL http://amazon.com/o/ASIN/0470431318/

[23] Svačina, L.: *Distribuovaný rendering na platformě WebGL*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2013.

[24] Tian, Y.; Liu, Y. C.; Bhosale, A.; aj.: All Your Screens Are Belong to Us: Attacks Exploiting the HTML5 Screen Sharing API. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, Washington, DC, USA: IEEE Computer Society, 2014, ISBN 978-1-4799-4686-0, s. 34–48, doi:10.1109/SP.2014.10.
URL http://dx.doi.org/10.1109/SP.2014.10

Příloha A

Navazující rozšíření

Během realizace této komplexní aplikace byly postupně sepisovány návrhy na navazující práci a případná rozšíření. Jejich výčet zařazený do patřičných kategorií je uveden v této příloze.

A.1 Záznam videa

- Zpřístupnit záznam z dalších zdrojů videa a audia jako jsou webové kamery.
- Implementovat oddělený záznam zvukové stopy z webové kamery v prohlížeči `Chrome` (`Firefox` podporuje nativně) a na serveru video a audio sesynchronizovat při překódování. Pokud již není v době čtení nativně implementováno i v `Chrome` (verze 52 nepodporuje).
- Volba datového toku pořizovaného záznamu v závislosti na rozlišení videa.
- Dostat adresu služby na výchozí povolený seznam prohlížeče `Firefox`. Pak není nutné rozšíření.

A.2 Zpracování videa

- Prozkoumat možnosti omezení nutnosti překódování videí. Třeba v případě dostatku segmentů přímo od prohlížeče (novější verze).
- Přidat možnost ořezu videa, přidání vodoznaku apod.

A.3 Přenos videa

- Odlehčit a zoptimalizovat signalizační kanál. Kumulativní potvrzování vektorů.
- Rozšířit převzetí signalizačního kanálu `P2P` kanálem po úspěšném spojení.
- Prioritní časovače hlídající maximální dobu čekání přenosu relativně prodlužovat dle pořadí chunku (a tím jeho reálné prioritě).
- Omezení neprioritního stahování v případě prioritního požadavku (ať se uvolní pásmo, pozastavením?).

- Preferovat ty uzly, které mají více dat. Ať se zbytečně neotevírá spojení (pomalé) s uzly, které nemají moc co nabídnout.
- Obdržení dat po vypršení aplikačního časovače nezahazovat, pokud nejsou v procesu stahování od jiného uzlu.
- Uzly s pomalým odesíláním (které stejně způsobí `timeout` a stažení z `TURN` serveru) zabanovat přímo na serveru a nevracet je v seznamu uzlů.
- Udělovat vyšší prioritu pro odesílání uzlům, které odesílají hodně (`Tit-for-Tac` princip z `BitTorrent` sítě).
- Optimalizovat chování na neomezených sítích `LAN`.
- Optimalizovat chování sítě při `flash crowd` situacích.
- Vylepšení strategií a heuristik pro volbu dat ke stažení, uzlů, kandidátů k odpojení či vyřazení ze sítě apod.

A.4 Přehrávání videa

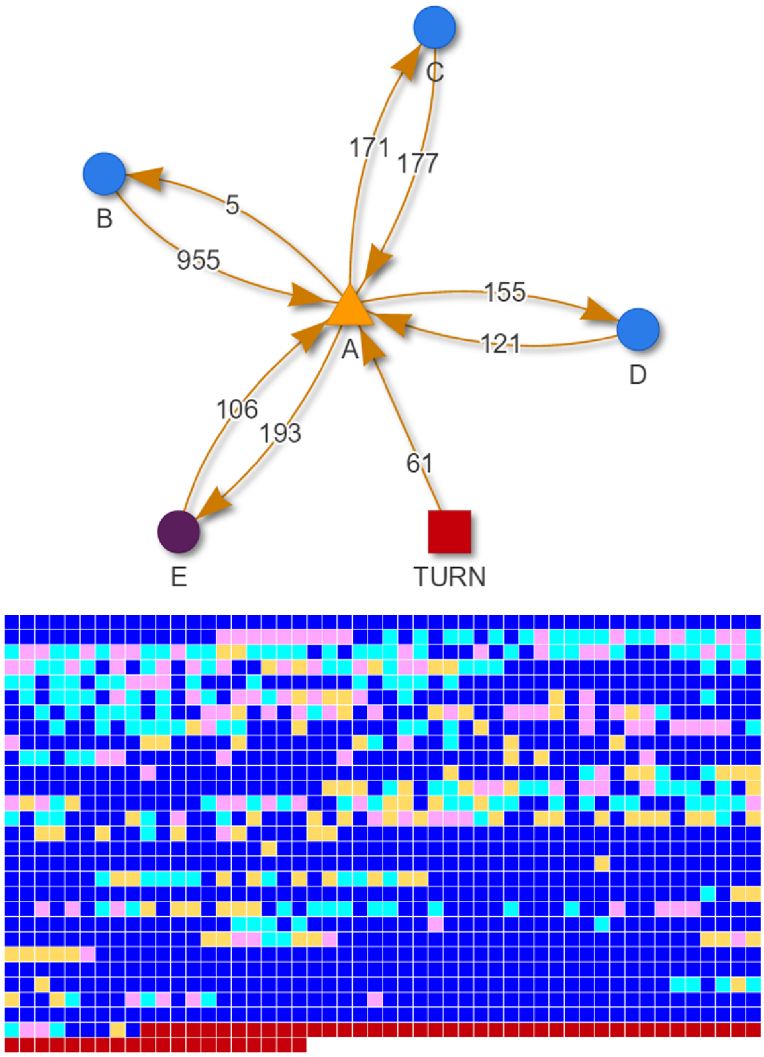
- Pro prohlížeče nepodporující formát `WebM` přidat `fallback` na `MPEG-4` variantu, která se dodatečně na serveru překóduje.
- Pro prohlížeče nepodporující `P2P` přenosy zpřístupnit video klasicky přes `<video>` element.
- Ukládat v paměti prohlížeče jen omezený počet „nejlepších“ chunků pro následné sdílení nebo data persistovat.
- Znovu prozkoumat aktuální situaci na poli `VP9` / `H.265` kodeků.

A.5 Testování

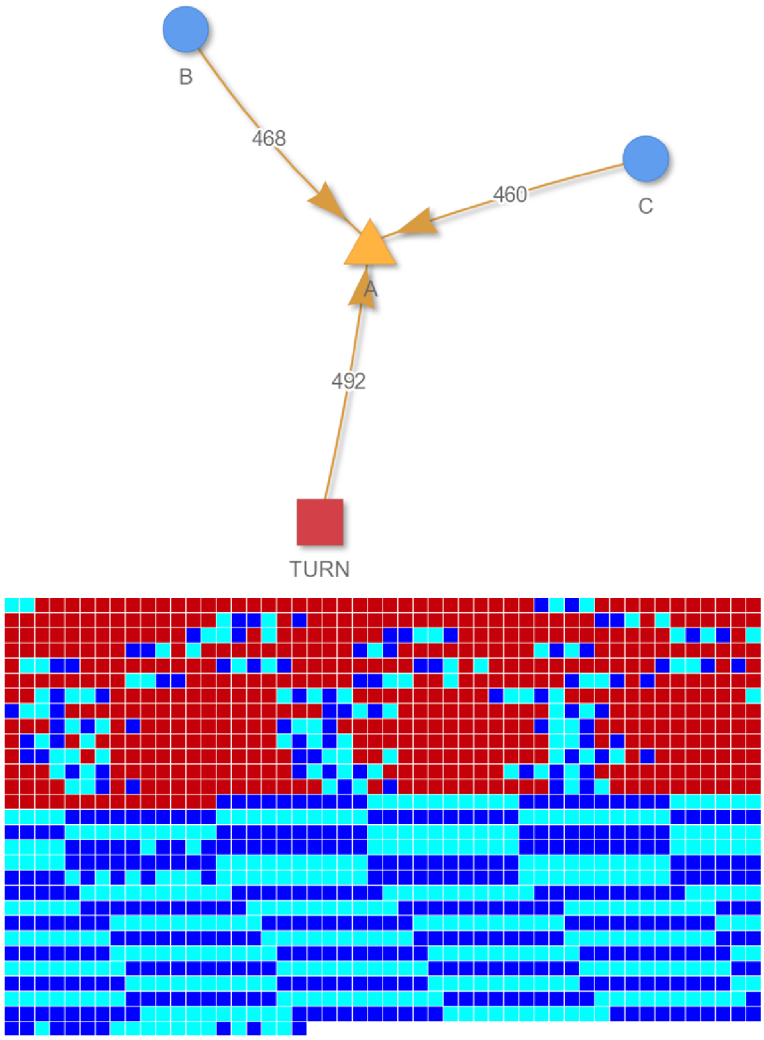
- Vliv počtu stahovacích slotů na čas stažení videa v případě mnoha nezávislých uzlů na rychlostně omezené síti.
- Spustit prioritní stahování dříve a to při sekvenčním načtení prvních několika segmentů videa namísto stahování prvního libovolného segmentu.

Příloha B

Znázornění přenosů



Obrázek B.1: Vzájemná výměna dat mezi 4 uzly již během stahování (smyčky). Dopomoc `TURN` serveru v případě nedostupnosti některých částí.



Obrázek B.2: Prioritní stahování proti strategii `LatestFirst` na pomalé lince. Jen občas se stihne stáhnout prioritní část od připojených uzlů (modrá v červené oblasti).

Příloha C

Instalace

C.1 Prerekvizity

Je nutné mít nainstalovaný webový server `nginx` (1.11.1), interpret `NodeJS` (6.2.1), `NPM` (součástí `NodeJS`, 3.8.5). V případě platformy `Windows 10 64b` není nutné kompilovat externí nástroje ručně, protože jsou zkompilované binární soubory již přítomny.

Na ostatních platformách je dále nutné zkompilovat a umístit do `app/server/tools/` následující nástroje: ffmpeg, sample_muxer, mse_json_manifest (pomocí kompilátoru jazyka GO) a mkclean.

Předpokládá se připravená doména, na které aplikace poběží (možno i `localhost`) včetně důvěryhodného certifikátu (jinak prohlížeč v určitých částech nedovolí spojení se serverem).

Pro fungující síťové připojení je nutné otevření portů 80, 443, 7776, 7777 a 7780.

C.2 Virtual hosts

Pro zvolenou doménu `domain.tld` směřující na adresu 127.0.0.1 je potřeba nakonfigurovat `nginx` virtuální hosty dle vzoru C.1. `SSL` je pro aplikaci nutné, protože nelze využívat některá `API` prohlížečů přes nezabezpečený kanál.

Ukázka kódu C.1: Šablona konfigurace virtuálních hostů `nginx.conf`.

```
server {
    listen 80;
    server_name  domain.tld;
    return 301 https://domain.tld$request_uri;
}

server {
    listen       443 ssl;
    server_name  domain.tld;
    fastcgi_read_timeout 600;
    proxy_read_timeout 600;

    ssl_certificate      "path-to-certificate";
    ssl_certificate_key  "path-to-key";

    location / {
        root   "path-to-dvd-root/app/client";
        index production.htm; # or playground.htm
```

```
19          rewrite ^/([a-zA-Z0-9]+)$ /?fileid=$1 break;
20      }
21
22      location /public/ {
23          root   "path-to-dvd-root/app/";
24      }
25
26      location /api/ {
27        proxy_pass https://domain.tld:7777/;
28      }
29  }
```

C.3 Příprava aplikace

Před samotným spuštěním aplikace je dále nutné:

- umístit do složky `certs` validní certifikát `domain.pem`, intermediate certifikáty do souboru `domain.chain` a privátní klíč `domain.key`,
- nastavit zvolenou doménu `domain.tld` do konfiguračního souboru `app/shared/config.js` (položka `config.domain`),
- spustit instalaci závislostí pomocí příkazu `npm install`,
- spustit kompilaci JS souborů pomocí příkazu `gulp`.

C.4 Spuštění aplikace

Dále lze:

- spustit v terminálu `API` server příkazem `node app/server/bin/api`,
- spustit v dalším terminálu signalizační server příkazem `node app/server/bin/signalling`,
- navštívit v prohlížeči adresu `https://domain.tld`.

Vždy při změně konfiguračního souboru `app/shared/config.js` je nutné překompilovat `JS` soubory opětovným voláním příkazu `gulp`. Případně lze spustit automatickou kompilaci příkazem `gulp watch`.

Přenos a zobrazení videa v prohlížeči pro záznam pracovní plochy

Ústav počítačové grafiky a multimédií, FIT VUT BRNO 2016

Autor: Bc. LUKÁŠ SVAČINA
Vedoucí: Ing. MICHAL ŠPANĚL, Ph.D.

ÚVOD

Chcete **jednoduše** a hlavně rychle nahrát video toho co právě s počítačem děláte přímo z prohlížeče bez dalšího SW?

A co takhle rovnou získaný krátký odkaz videa komukoliv rozeslat nebo kdekoliv vložit?

Hromadné otevření takového odkazu však klade vysoké požadavky na zdroje poskytujícího server.

Tak to video začněme rovnou během načítání distribuovat v P2P síti! Přímo v prohlížeči a bez potřeby rozšíření.

Budoucnost spojení prohlížečů je tady.

TECHNOLOGIE

Málokdo ví, že moderní prohlížeče dnes již umožňují přistupovat nejen ke kameře uživatele, ale také k jeho pracovní ploše.

Současně lze takové streamy dat za běhu nahrávat a odesílat v binární podobě na server.

Tam již stačí video zpětně složit, provést úpravy umožňující zpětné přehrání a je to.

Ještě zajímavější je však to, že prohlížeče nabízejí prostředky pro P2P spojení mezi uživateli. To umožňuje implementovat výměnnou síť dat obdobně jako třeba známý BitTorrent. A to bez nutnosti rozšíření.

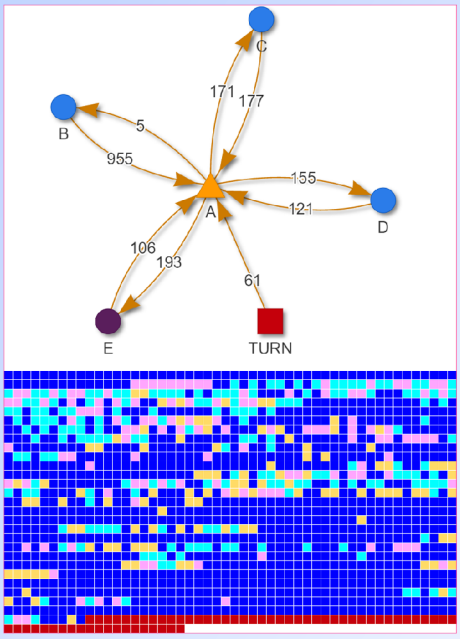
OBSAH PRÁCE

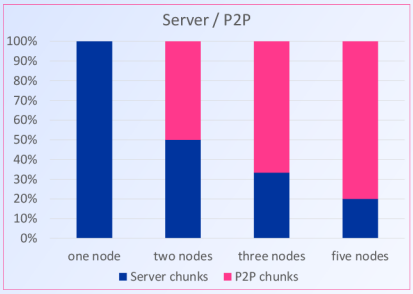
Práce si dala za úkol zkombinovat tyto moderní až experimentální API a vytvořit unikátní webovou službu.

Kromě zpracování videí je implementována hybridní Video on Demand P2P síť.

To znamená, že si kdokoliv může sledovat cokoliv a to v libovolném čase díky posouvání ve videu! A přitom rovnou odlehčuje serveru nabízením stažených částí ostatním sledujícím.

Že je JavaScript dobrý leda pro skrývání tlačítek na webu? Pohání klientskou i serverovou část a jeho asynchronnost je pro tuto aplikaci ideální!





POVEDLO SE?

Ano! Objem dat přenášených serverem klesá a rychlost stahování s počtem uzlů stoupá.

Důkazem budiž veřejné spuštění služby během 4. Q 2016 pod adresou https://ctrlv.tv