České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Petr Blšťák**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **Distribuovaná multiplayer aréna pro edukativní hry**

Pokyny pro vypracování:

1. Analyzujte možnosti rozšíření herní arény projektu hrave.cz pro distribuovanou hru velkého množství hráčů.
2. Navrhněte řešení a ověřte technologické možnosti využití Amazon Web Services pro implementaci arény.
3. Navrhnete a implementujte modul pro párování hráčů a jejich distribuci mezi instance arén.
4. Implementujte a otestujte prototyp distribuované arény pro jednoduchou hru.

Seznam odborné literatury:

dokumentace projektu hrave.cz
dokumentace Amazon Web Service
Singh Huns: Service-Oriented Computing

Vedoucí: doc. Jiří Vokřínek Ing., Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

prof. Ing. Jiří Žára, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 4. 11. 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů

Diplomová práce

Distribuovaná multiplayer aréna pro edukativní hry

Bc. Petr Blšťák

Vedoucí práce: doc. Ing. Jiří Vokřínek, Ph.D.

Studijní program: Otevřená informatika, Navazující magisterský

Obor: Softwarové inženýrství

Leden 2015

Poděkování

Zde bych rád poděkoval vedoucímu práce doc. Ing. Jiřímu Vokřínkovi, Ph.D. za dohled a připomínky k její korektní formě. Dále společnosti Educasoft za možnost zpracování tohoto tématu i za prostoru k tomu, jej dále rozvíjet. Poslední část patří rodině, za mnohaletou podporu ve studiu, která nepolevila ani při psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 5. ledna 2015 .....

Abstrakt

Tato práce se zabývá analýzou a implementací vědomostní arény pro více hráčů, kterou bude možné provozovat na cloudových technologiích. Vychází z projektu Hravě.cz, jehož modul arény jí posloužil jako odrazový můstek. Práce má dva hlavní cíle. Za prvé optimalizaci pro cloudové prostředí, jehož distribuovaná povaha vyžaduje specifické přístupy v návrhu architektury i v implementaci. Za druhé zajištění real-time komunikace mezi hráči v tomto prostředí.

Řešení se opírá o moderní technologii websocketů a jejich implementaci pomocí frameworku Atmosphere. Pro cloudové prostředí využívá služeb Amazon Web Services včetně Elastic Load Balancingu, automaticky škálovaných EC2 instancí s vlastním Jetty serverem a rychlého datového úložiště ElastiCache. Výsledný projekt by měl být v roce 2015 implementován do Hravě.cz jako plnohodnotná multiplayer aréna.

Klíčová slova

Cloud; cloudové technologie; Amazon Web Services; AWS; websocket; Atmosphere; multiplayer, real-time

Abstract

This thesis deals with analysis and implementation of a multiplayer knowledge arena, that will be able to run on cloud technolgies. The idea for this work originated from the Hrave.cz project and it's single player arena, that will serve as a stepping stone. This work has two main goals. First, to optimize the application for the cloud environment, which requires some specific approaches and architecture styles, due to its distributed nature. Second, to ensure a real-time communication between players in this environment.

The solution is built on top of modern websocket technology implemented using the Atmosphere framework. The cloud environment is represented by the Amazon Web Services including the Elastic Load Balancing, auto scaled EC2 instances with customised Jetty server and the ElastiCache, a high speed data storage. In 2015, the resulting project should be implemented in the Hrave.cz as a full-featured multiplayer arena.

Keywords

Cloud; cloud technologies; Amazon Web Services; AWS; websocket; Atmosphere; multiplayer, real-time

Obsah

Seznam obrázků

1 Úvod

Hry, ať už v jakékoli formě, byly vždy nejen zábavou, ale i prostředkem k dosažení určitých cílů, které se samotným aktem hraní neměly příliš společného. Skrze věky, hraní her slouží k socializaci a navázání kontaktů, stejně jako k porovnání schopností a dovedností ať už jedince nebo kolektivu. V moderní době se hry také stále více používají jako nositel informace, která je účastníkovi během procesu hraní si předána a navíc je její zapamatovatelnost umocněna prožitkem ze hry. Vzniká stále více takzvaných edukativních her, vyrobených především za účelem snadnějšího předání informací těm, kteří by o ně jinak neměli velký zájem.

Webová aplikace Hravě, v jejímž rámci byla praktická část této práce vyvíjena, se řadí do této kategorie. Jejím úkolem je usnadnit středoškolským studentům přípravu ke státní maturitě a k přijímacím zkouškám na vysoké školy. Od jiných projektů svého druhu se však liší tím, že se nesnaží pouze zprostředkovat učební obsah. Snaží se jej navíc podat zábavnou herní formou, která typicky udrží pozornost studenta déle, než čisté konstatování informace. Student tak má v rámci aplikace svou herní postavu, která se zlepšuje adekvátně jeho pokrokům ve studiu. Srovnání dosažených atributů herních postav s ostatními uživateli Hravě je pak jednou formou motivace studenta. Daleko silnější motivací však pro mnohé mohou být souboje jejich postav ve vědomostní aréně.

Aréna projektu Hravě má velice jednoduchý koncept, dosud ovšem existovala pouze ve formě single-playeru, tedy hry jednoho hráče proti počítači. Hráč dostává otázky ze zvoleného oboru a musí na ně co nejrychleji správně odpovědět. Při správné odpovědi je zasažen soupeř a snižuje se jeho zdraví, při špatné odpovědi je naopak zasažen hráč. Zdraví hráče také postupně klesá nehledě na odpovědi, má tedy na poražení soupeře pouze omezený čas.

Ačkoli je tento způsob procvičování otázek zajímavější než pouhé vyplňování odpovědí do formuláře, není pro mnoho lidí takovou výzvou, jakou je srovnání se s živým soupeřem. Nesčetné on-line hry i fitness aplikace těží právě z tohoto modelu, kdy je hnacím motorem pro jejich uživatele právě srovnávání se v rámci uživatelské komunity. Pro projekt Hravě je tedy rozšíření současné arény o hru dvou a více živých hráčů proti sobě (dále jen multiplayer), dalším logickým krokem v jeho rozvoji.

Do budoucna se také počítá se značným nárůstem počtu uživatelů aplikace a bylo rozhodnuto o jejím přesunu na cloudové technologie, které nabídnou škálovatelný výpočetní výkon, úložný prostor, zvýšenou bezpečnost i dostatečnou konektivitu. Připravovaná multiplayer aréna musí tedy být na cloudové prostředí optimalizována a maximálně jej podporovat.

1.1 Cíle práce

Cílem práce je navrhnout a implementovat koncept multiplayer arény, který bude připraven na provoz v cloudu. Cloudová podpora znamená optimalizaci pro distribuovaný systém, kde aplikace existuje v několika rovnocenných instancích (instancí je myšlena jedna kopie aplikace), z nichž se typicky každá nachází na jiném serveru (virtuálně nebo i fyzicky). Z hlediska uživatele je naprosto jedno, která instance zpracuje jeho požadavek, systém se mu navenek jeví jako jediná aplikace. Uvnitř spolu však instance musí být schopny komunikovat, protože každá uchovává o uživateli určitá data a musí být schopna je s ostatními sdílet.

Bude třeba implementovat nový modul pro párování hráčů do soubojů v jednotlivých arénách. Do tohoto modulu se dostane pouze přihlášený hráč, což bude simulovat příchod z původní aplikace. Propojení hráčů musí být možné napříč všemi aktivními instancemi a ty musí být schopny uživatele přesunout z jedné instance na jinou. Přesun hráčů zajistí, že dva soupeři v aréně se nacházejí na stejné instanci a jejich komunikace probíhá uvnitř instance, nikoli mezi dvěma instancemi, což zmenší komunikační vytížení celého systému.

V samotné aréně je pak potřeba zajistit real-time zprostředkování akcí hráčů jejich soupeřům. Hráč nemůže zjistit, že jej soupeř zasáhl až 10 vteřin poté, co se tak opravdu stalo. V klasickém request/response komunikačním modelu z původní arény se data ze serveru dostávají k uživateli pouze v případě, že si o ně uživatel zažádá. Server tedy nemůže ze své vlastní iniciativy informovat uživatele o tom, jakou akci provedl jeho soupeř. Stávající model zde musí být nahrazen pokročilejší technologií, jako je long-polling či websocket, která zajistí serveru možnost zasílat klientovi data, aniž by si o ně klient sám řekl.

Implementovaný koncept musí obsahovat všechny technologie nezbytné pro běh popsaného systému tak, aby pozdější rozšíření do plnohodnotné arény bylo možné pouze v rámci stávajících a funkčních technologií. Práce musí také obsahovat testy prokazující funkčnost systému v zátěži v reálném cloudovém prostředí.

1.2 Struktura práce

Práce začne popisem stávajícího projektu Hravě a jeho technologií, které budou případně k dispozici pro řešení této práce. Jelikož se jedná z velké části o standardní systém, popisy nebudou zabíhat příliš do detailů.

Poté práce přejde na popis nových technologií, které budou potřeba pro splnění vytyčených cílů. Těmi bude technologie websocketů a služby Amazon Web Services.

Čtvrtá kapitola se bude věnovat tomu hlavnímu - architektuře a implementačním detailům celého systému navrženého v této práci. Architektura bude probrána z několika pohledů, jelikož nebude popisována pouze samotná aplikace, ale i celý cloudový systém včetně přidružených služeb. Implementace bude pak prezentována zejména na hlavních komponentách aplikace.

Poslední dvě kapitoly se věnují testům celého systému a přidružené služby ElastiCache pro inter-instanční komunikace. Nakonec je zařazeno jeho celkové zhodnocení.

2 Stávající projekt

Současná aréna sdílí s výukovým modulem projektu značnou část implementace a nebylo by v tomto případě efektivní oddělovat ji jako samostatnou aplikaci postavenou jinak a na jiných technologiích. Vzhledem k architektuře celého systému by to sice bylo možné, ale celou řadu funkcí by bylo třeba implementovat znovu. Proto i multiplayer aréna využije části původního projektu. Nejprve bude představena jeho technologická stránka. Sekce 2.3 pak představí některé klíčové prvky jeho architektury.

2.1 Backend

Backendová část je postavena na platformě Java, konkrétně její Enterprise Edici. Spolu s frameworkem Spring a aplikačním serverem Jetty tvoří poměrně výkonné a zároveň kompaktní řešení. Tyto tři hlavní backendové technologie budou představeny pouze v krátkosti, vzhledem k jejich obecné rozšířenosti. Několika slovy bude zmíněna i použitá databáze a build nástroj Maven.

2.1.1 Java EE

Platforma Java v rozšíření Enterprise Edition (jinak také J2EE) je standardem při vývoji webových aplikací. Je podporována většinou web serverů a frameworků a její nativní knihovny dávají k dispozici velké množství užitečných technologií. Mnoho jich však využijeme zprostředkovaně, pomocí frameworku Spring. Z čisté Javy budeme v projektu používat například Servlety pro vystavení koncových bodů některých serverových služeb. Verze Javy EE použitá v současném projektu Hravě je 1.6, tato práce však pracuje s verzí 1.7 zejména kvůli lepší podpoře v cloudových technologiích.

2.1.2 Spring

Open-source framework pro platformu Java. Mimo jiné znatelně usnadňuje vývoj webových projektů zjednodušením jejich konfigurace a automatickou správou životních cyklů některých backendových komponent. Zavádí systém Controllerů, které nahrazují Enterprise JavaBeans a umožňují vytvořit plnohodnotné REST (Representational State Transfer) rozhraní. Většinu konfigurace systému je možné provádět pomocí anotací, které nahrazují konfigurační XML soubory. Inversion of Control Container nám pak umožní vytvořit interní Servisy - jakési knihovny vlastních funkcí, které je možné použít kdekoli v aplikaci pomocí Dependency Injection. Systém se sám stará o jejich korektní vytváření, zavádění i destrukci. V neposlední řadě pak využijeme podporu velkého množství databázových frameworků jako je například Hibernate. Ačkoli je již nějakou dobu k dispozici Spring verze 4, pro kompatibilitu s původním projektem bude použit Spring verze 3.2.2.

2.1.3 Jetty

Volně dostupný web server vyvíjený skupinou Eclipse Foundation. Jedná se o velmi lehký server, což znamená, že jeho distribuce obsahuje jen to nejnutnější pro jeho provoz. Chybí mu jakékoli uživatelské rozhraní pro konfiguraci a monitoring a vše se děje přes několik konfiguračních a logovacích souborů. To může být překážkou pro velké a výpočetně náročné aplikace, proto také Jetty nemá velké zastoupení v korporátní sféře.

Absence jakékoli nadstavby se ovšem velice příznivě projevuje na jeho velikosti, která je oproti některým web serverům i desetinová (např. JBoss), a také na velmi malých požadavcích na operační paměť, viz obr. 1. Obojí bylo v začátcích projektu jedním z nejdůležitějších parametrů a pro provoz web serveru na cloudu je to velké plus. Svědčit o tom může i využití Jetty na Google App Enginu, v současnosti jednom z největších poskytovatelů cloudových služeb.

Naopak výkonem nemá Jetty problém srovnat se s jinými velkými web servery [1] [2], jakými je JBoss, GlassFish, WebSphere nebo Tomcat, zvláště v oblasti zpracovávání konkurenčních požadavků z mnoha tisíc současných spojení. Navíc (krom jiných) podporuje pro nás potřebné technologie: Java 1.7, Websockety a Servlet 3.1. Verze Jetty použitá pro tuto práci je 9.2.5.

	Freebsd + jetty – 6082	Freebsd + tomcat 6083	Centos + jetty 6092	Centos + tomcat 6093
chyby	1 connection timeout	65 c.t. a dve vazne chyby	8 connection timeout	5 connection timeout
1 vlakno x 300 cyklu	37,33	45,67	46,67	39,67
20 vlaken x 100 cyklu	70,43	93,33	87,77	81,53
50 vlaken x 100 cyklu	213,45	211,35	216,88	215,23
200 vlaken x 20 cyklu	675,56	922,44	786,11	779,05

FreeBSD + Jetty

Průměrný load: prvni půle kolem 0,8, druhá půle kolem 0.5
Procesor: více méně pořád kolem 2%
Paměť: během testu vyrostla spotřeba o 52MB

FreeBSD + Tomcat
Průměrný load: kolem 1.33
Procesor: více méně pořád kolem 3% (obcasne 10%)
Paměť: během testu vyrostla spotřeba o 123MB

CentOS + Jetty

Průměrný load: prvni půle kolem 1.5, druhá půle kolem 0.3
Procesor: v prvni půli spotřebováno 100%, v druhé kolisal kolem 40%
Paměť: během testu vyrostla spotřeba o 32MB

CentOS + Tomcat
Průměrný load: prvni půle kolem vzrostl az na 1.7, druhá půle kolem 0.3
Procesor: v prvni půli spotřebováno 100%, v druhé kolísal kolem 30%
Paměť: během testu vyrostla spotřeba o 141MB

Obrázek 1 Data dodaná Hravě.cz ukazující testování kombinací dvou OS se servery Jetty a Tomcat. Zdroj k obrázku uvádí následující:
Při výběru serveru pro projekt Hravě byla hlavním kritériem jeho velikost a paměťová zátěž. Připadal v úvahu Tomcat a Jetty a tento mini test posloužil k porovnání jejich výkonu.
Connection timeout události vznikly vždy v posledním testu 200 vláken x 20 cyklů. Čísla v buňkách jsou průměrné ms na jeden HTTP požadavek (každý cyklus se skládal ze tří požadavků: read, write a create)
zdroj: interní dokumentace Hravě.cz

2.1.4 MySQL databáze

Pro tuto práci bylo využití databáze pouze okrajové a je implementováno především z důvodu ověření funkcí relační databáze na cloudu. Amazon, poskytovatel cloudových služeb využívaných v této práci, nabízí několik typů databází zahrnující většinu v současnosti používaných řešení, včetně vlastní databáze Aurora. V projektu Hravě je

použita databáze MySQL verze 5.6.15, v brzké době se ovšem počítá s její výměnou za PostgreSQL.

2.1.5 Maven

Velice rozšířený nástroj pro management buildů aplikací, zejména na platformě Java. Popisuje jak vytvořit build (sestavení) aplikace a její závislosti na externích knihovnách. Tyto knihovny umí okamžitě stáhnout v příslušné verzi a importovat do projektu. V jeho databázi se jich nachází obrovské množství a pokud vámi potřebná knihovna obsažena není, můžete se odkázat na jinou, než hlavní Maven databázi, případně si vytvořit vlastní. Výhodou také je, že při sdílení projetu nemusíte vámi použité knihovny kopírovat, stačí konfigurační Maven soubor a každému vývojáři se knihovny stáhnou samy v definované verzi.

2.2 Frontend

Frontend využívá standardní technologie webové prezentace: **HTML** verze 5.0, **CSS** 3.0 a **JavaScript** (JS). Technologie jako JavaServer Pages(JSP) nebo JavaServer Faces(JSF) nejsou použity z důvodů, které budou uvedeny v sekci 2.3. JavaScript sám o sobě je velice dobrým nástrojem pro dynamickou webovou prezentaci, zvláště pokud je rozšířen o knihovnu JQuery a jsou použita serverová volání typu AJAX. Hlavní technologií frontendu je ovšem framework AngularJS, kterému se tato část věnuje především.

2.2.1 Standardní technologie

JQuery výrazně usnadňuje práci s JS přidáním funkcí umožňujících jednodušší navigaci mezi prvky zdrojového HTML dokumentu (DOM - Document Object Model) a manipulaci s nimi, vytváření animací, AJAX volání, automatickou podporu formátu JSON a další rozšířitelnost o mnoho pluginů. Použitá je verze 2.0.3.

AJAX - Asynchronous JavaScript and XML je způsob volání vzdáleného serveru s požadavkem na data. Používá se nejčastěji ke změně části HTML stránky, například dat tabulky, aniž by musela být celá stránka znovu nahrána. Případně k operacím uživateli úplně skrytým, například ověření, že je uživatel přihlášen do aplikace. Jeho důležitým aspektem je, že volání je asynchronní, prohlížeč tedy nečeká až server odpoví, pouze pošle požadavek a pokračuje dál v práci. Teprve když je prohlížeči doručena odpověď, provede specifikovanou akci v závislosti na odpovědi.

2.2.2 AngularJS

Je JavaScript framework na jehož tvorbě se mimo mnoha nezávislých přispěvatelů podílela skupina Brat Tech LLC a Google. První verze byla vydána v roce 2010 a dle jeho tvůrců je Angular "tím co by bylo HTML, kdyby bylo vytvořeno pro budování webových aplikací" [3]. V této práci je použitá verze 1.2.25.

Na frontendu vytváří Model-View-Controller (MVC) strukturu a transformuje jej na samostatnou webovou aplikaci na klientské straně. Ta dostává od backendu pouze surová data a zpracovává si je dle vlastní potřeby. Vzhledem k poměrně krátké existenci tohoto frameworku a širokému využití v této práci, se na jeho strukturu a funkcionalitu podíváme podrobněji.

Základem je **modul**, který obsahuje definici a konfiguraci konkrétní instance Angularu. Ta je navázána na určitou HTML stránku (typicky index) v rámci níž funguje celá webová aplikace. Definice modelu určuje jeho jméno a případně použité pluginy, jichž existuje velká škála pro podporu routování, překladů, práci se soubory, spolupráci s Google analytics atd. V konfiguraci se pak určí nastavení těchto pluginů. V rámci modulu je možné definovat controllery.

Controller je jednotka typicky ovládají jeden view. Uchovává veškerá data v něm použitá a obsahuje logiku pro manipulaci s nimi. **View** je název pro HTML šablonu - ta definuje kde mají být jaká data zobrazena a v jakých typech prvků (div, odstavec, formulář, tabulka...). Veškeré proměnné controlleru v nichž jsou uložená data zobrazovaná ve view jsou pak v rámci MVC architektury chápány jako model.

Uveďme si příklad, kdy máme tabulku dat. Její view obsahuje pouze její název a poté tabulku o jednom řádku a jednom sloupci s odkazy na příslušná data v controlleru a direktivou pro jejich vyplnění. V závislosti na konkrétních datech vygeneruje controller tabulky příslušné buňky a vyplní je daty. Pokud se data změní, controller automaticky updatuje tabulku. Pokud budeme mít na jedné stránce tabulek více, pro každou z nich bude vytvořena jedna instance controlleru, které se navzájem neovlivňují.

Předpokládejme, že by tabulka obsahovala pouze číselná data a chtěli bychom je vynásobit nějakým číslem. Pod tabulku bychom vložili input pole s odkazem na nějakou proměnnou controlleru a tlačítko k provedení příkazu. Proměnná je dynamicky svázána s hodnotou zobrazenou v input poli a jakmile se její hodnota změní, controller updatuje její zobrazení v poli a naopak, při přepsání hodnoty pole je automaticky updatována proměnná. V controlleru bude dále funkce na vynásobení dat tabulky číslem v proměnné. Ta je napojena na tlačítko po tabulkou. V okamžiku jeho stisku se provede přepočet všech dat dle aktuální hodnoty proměnné a výsledná data jsou okamžitě aktualizována v tabulce.

Tyto funkce bychom pochopitelně dokázali naprogramovat i bez použití Angularu. S jeho použitím je ovšem práce výrazně jednodušší, navíc se zmenšuje objem napsaného kódu i jeho přehlednost. Zde je několik funkcionalit, které Angular poskytuje pro view:

- Funkce *switch* a *if* pro podmíněné přidání/odebrání elementů z DOM
- Show/hide elementy v závislosti na proměnné pomocí CSS třídy
- Disable/enable elementů formuláře v závislosti na proměnné
- Automatické vyplňování polí hodnot do libovolných elementů pomocí funkce repeat (vyplňování tabulek, selectů, listů)
- Dvoucestné provázání proměnné skriptu a zobrazené hodnoty ve view
- Dynamické přidání/odebrání tříd elementu v závislosti na proměnné
- Zobrazení příslušného view v závislosti na URL

Poslední uvedená funkcionalita je již zmiňované routování. To samo o sobě znamená navigaci dle URL adresy a není ničím převratným. V Angularu má ten rozdíl, že při něm není načítána celá nová stránka, ale pouze nové view. To může mít libovolný rozsah od jednoho elementu až po kompletní tělo stránky. Rámec aplikace tvoří iniciální HTML soubor obsahující typicky definici *head* části a přiřazení modulu Angularu k *body* části této stránky pomocí deklarace *ng-app*. V URL adrese zůstává název iniciálního HTML souboru a routování probíhá přidáním značky # (hash) za níž následují identifikátory jednotlivých view. Stránka kontaktů pak může mít adresu například

www.myApp.cz/index.html#/contacts případně *www.myApp.cz/#/contacts* v závislosti na nastavení serveru.

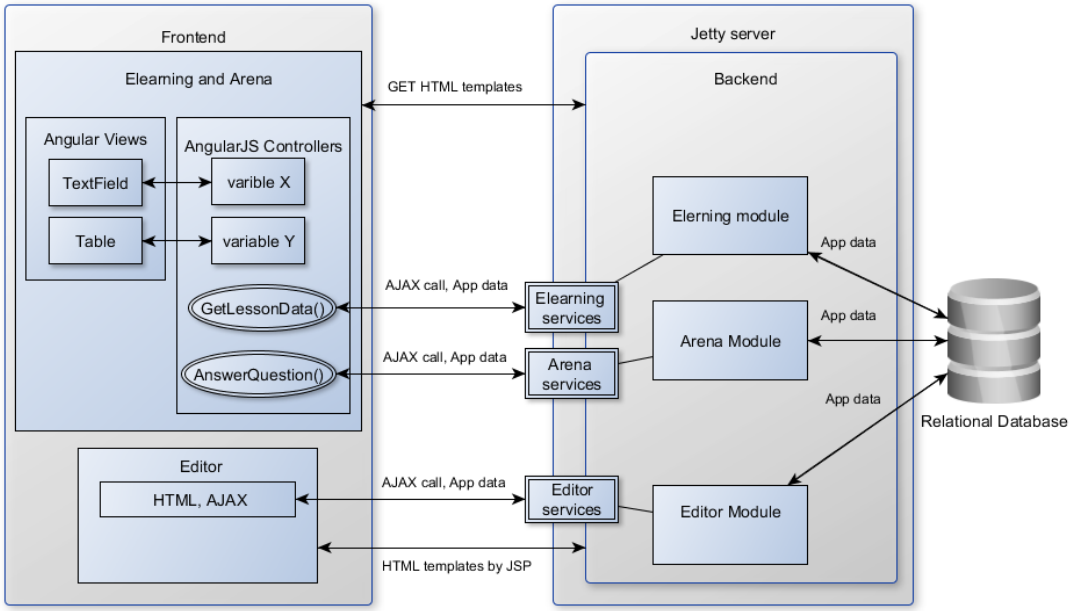
Routování Angularu je užitečné, jelikož při jeho konfiguraci můžete rovnou přiřazovat controllery jednotlivým view, přidávat controllerům dodatečné závislosti, definovat parametry rout nebo řešit přesměrování uživatele.

2.2.3 Bootstrap

Známý webový framework pro HTML, CSS i JS. Podobně jako Angular velice usnadňuje vývoj webu, zejména jeho vizuální stránky. V tomto projektu stejně jako v současné verzi Hravě je využit minimálně, nicméně je uveden pro úplnost. Bootstrap je použit ve verzi 3.0.0.

2.3 Architektura

Aplikace Hravě má celkem tři části. E-learningový modul, jehož obsahem jsou veškeré lekce a je stěžejním pro uživatele aplikace, Arénu a Editor. Editor slouží pro vytváření obsahů e-learningu a vznikal jako první, je proto částečně postaven na starší technologii JSP. Zbylé dvě části jsou postaveny výhradně na systému servis poskytovaných backendem jako přípojné body a jejich volání pomocí AJAXu ze strany frontendu. Je to systém podobný REST rozhraní, ovšem nedodržuje všechny jeho zásady.



Obrázek 2 Schématické zobrazení architektury stávajícího systému s důrazem na popsané technologie a komunikaci mezi komponentami.

Obr. 2 schématicky zobrazuje rozložení komponent frontendu a backendu a jejich komunikaci. HTML šablony, tedy převážně view jsou zasílány jako odpověď na standardní GET requesty. Controllery Angularu ovšem komunikují pouze se servisami vystavenými třemi moduly backendu a to pomocí AJAX volání na což server odpovídá požadovanými daty. Controllery se také automaticky starají o update dat ve view. Modul editoru

je specifický kvůli současnému použití JSP a AJAXu, což souvisí s jeho postupnou modernizací. V současné době se pracuje na jeho kompletním převedení do frameworku Angular a zbavení se technologie JSP.

Angular je totiž hlavním důvodem pro postupné opuštění JSP (potažmo JSF). Princip funkce obou technologií je podobný. Systém obsahuje šablony, definující jaká data a kde se mají zobrazit. Při odpovědi na uživatelskou akci se šablona naplní odpovídajícími daty, systém z ní vygeneruje HTML stránku a tu zobrazí uživateli. Rozdíl je ve straně, která stránky generuje. Zatímco u JSP je to server, u Angularu je to klient.

Druhou výhodou využití Angularu je snížení objemu komunikace s backendem. Odpovědí na AJAXová volání klienta jsou pouze surová data ve formátu JSON. U JSP to je celá HTML stránka, ačkoli změna, která se na ní udála může být pouze malá. Nutno poznamenat, že i v JSP je možno použít AJAX a pokročilejší JSF podporuje parciální aktualizace stránky skrze AJAX nativně. V obou případech je ovšem složitost implementace značně vyšší, což dobře demonstruje tento článek [4]. Pro nás je navíc výhodné přenechat vykreslování stránek klientské straně, jelikož tak dochází k určitému snížení zátěže backendu.

Třetí věcí ve prospěch Angularu je jeho zaměření na budování webových aplikací, které se projevuje ve velké úspoře kódu a jednoduchosti implementace některých funkcionalit. Z toho důvodu je možné přesunout na klienta nejen vykreslování stránek, ale také část aplikační logiky. Tento princip takzvaného tlustého klienta, je velkým přínosem při přesunu projektu do cloudového prostředí. V cloudu se typicky platí za využitý výpočetní výkon a objem datové komunikace, z čehož oboje je tímto přístupem znatelně redukováno.

Poslední věcí k vyzdvihnutí architektury postavené čistě na AJAXové komunikaci je nezávislost implementace backendu a frontendu. Obě strany jsou svázány pouze servisami, které přijímají požadavky frontendu a posílají mu zpět požadovaná data. Jakým způsobem a technologií je požadavek na backendu zpracován je z pohledu frontendu naprosto nepodstatné a naopak. Dokud se nezmění název servisy, přijímané parametry, nebo struktura odeslaných dat, budou obě strany bez problémů spolupracovat nehledě na změny interní funkcionality, programovacího jazyka či platformy.

Výčet výše uvedených vlastností poměrně jasně ukazuje velkou výhodu architektury postavené na Angularu, oproti starším technologiím. Zůstane proto zachována i v této práci, což přinese dobrou kompatibilitu s projektem Hravě. Velkým pozitivem je i vhodnost této architektury pro cloudové prostředí. Další část práce se již věnuje analýze nových požadavků a jejich technologickému provedení.

3 Analýza nových technologií

V úvodu práce bylo uvedeno několik cílů. Technologie z minulé kapitoly umožňují splnění pouze části z nich, je tedy potřeba tuto technologickou základnu rozšířit. V první řadě o real-time komunikaci, která zajistí doručení zpráv mezi dvěma hráči s minimálním zpožděním, ideálně zanedbatelným v rámci plynulého běhu hry. O této technologii budou pojednávat první dvě sekce této kapitoly.

Druhým nutným rozšířením je zařízení běhu na cloudových technologiích. Ty se v základu příliš neliší od technologií běžných single-server řešení. Pokud má ovšem aplikace plně využívat výhod cloudu, je nutné posunout se za tyto základy a využít paralelního běhu aplikace na více serverových instancích, jejich automatické škálování, load balancing uživatelských požadavků a pokročilá datová úložiště.

Výsledný systém musí tyto technologie kombinovat do efektivního celku a optimalizovat jejich použití pro co nejlepší poměr cena/výkon. V cloudu je totiž zpoplatněna velká část datových přenosů i nevyužitý výpočetní výkon objednané služby. Jejich neefektivní používání se tedy může velice prodražit.

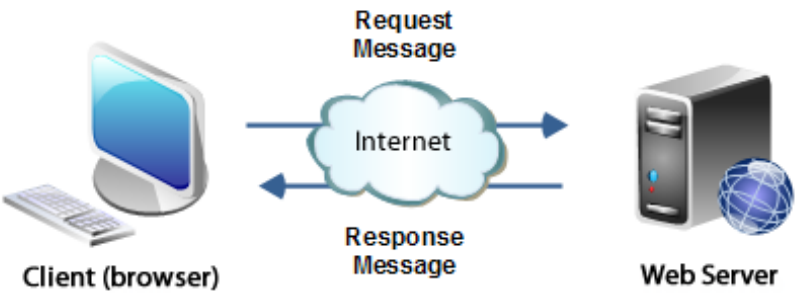
O jednotlivých cloudových službách a vhodnosti jejich využití bude pojednávat sekce 3.3 a celkové shrnutí 3.4.

3.1 Real-time komunikace

Co si tedy přesně představit pod pojmem real-time komunikace. Běžný telefonní hovor je jejím příkladem. To, co jedna strana říká, se okamžitě přenese k druhé straně, pouze s minimálním zpožděním, které člověk nepostřehne. Na úrovni fyzického spojení to znamená vytvoření takzvaného plně duplexního spojení, což je komunikační kanál, jehož obě strany mohou v jeden okamžik přijímat i odesílat data.

Spojení serveru a klienta webové aplikace je plně duplexní, alespoň co se týče fyzického přenosu dat. Technologie používané pro tuto komunikaci ovšem neumí vždy tento fakt využít.

Při vývoji komunikačního protokolu pro webové stránky se s potřebou real-time komunikace neuvažovalo. Takový typ komunikace byl používán pouze v desktopových aplikacích například pomocí protokolu TCP. Pro webové stránky byl vyvinut protokol HTTP, který ovšem v principu funguje jinak. Přímo určuje stranu, která musí vždy komunikaci otevřít a nedává tak prostor pro plné využití potenciálu této komunikace.



Obrázek 3 Standardní komunikace klienta a serveru pomocí request/response technologie
zdroj: http://tutorials.jenkov.com/web-services/message-formats.html

Na obr. 3 je zobrazena standardní komunikace klienta a serveru. Iniciátorem této komunikace musí vždy být klient. Ten vytvoří požadavek - *request*, odešle jej serveru na zpracování a ten pošle zpět odpověď - *response*. Důležité je, že response serveru je vždy vázána na request, který mimo jiné obsahuje i identifikaci klienta. Bez něj by tedy server v první řadě ani nevěděl, kam má data poslat.

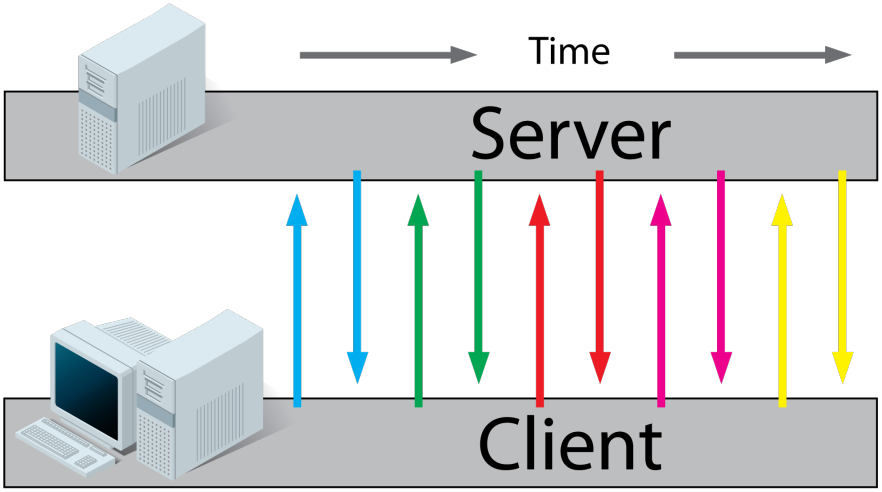
Real-time komunikace je při použití tohoto modelu nemožná z logického důvodu. V okamžiku kdy uživatel A pošle serveru request s daty pro uživatele B, nemůže server data doručit, dokud neobdrží také request od uživatele B, který se bude ptát zda pro něj nejsou nějaká data připravena. To může trvat libovolně dlouhou dobu v závislosti na konkrétním chováním klienta. Nemůžeme tedy zaručit, že data budou mezi uživateli předávána okamžitě. Postupně však bylo vyvinuto několik technologií, které takovou takový typ komunikace zaručit mohou.

3.1.1 Short-polling

Také nazývané AJAX polling, je jednoduše opakované dotazování se serveru zda se něco stalo. Při ustanovení komunikačního kanálu je spuštěna JavaScriptová funkce obsahující časovač, který v daném intervalu volá server pomocí AJAXu s požadavkem na nová data. AJAXové volání je navíc asynchronní, takže neblokuje klienta v provádění dalších operací a vše běží na pozadí normálního fungování webové aplikace.

Problémem je že interval časovače musí být velmi krátký, dejme tomu 0.5 až 1 vteřinu, aby klient obdržel zprávu opravdu v okamžiku kdy na serveru nastane. Tím vzniká obrovská zátěž komunikačního kanálu, která navíc neobsahuje žádná důležitá data, pokud na serveru nedošlo k žádné události. V případě, že bychom chtěli tuto zátěž snížit a prodloužili bychom interval, zvyšujeme riziko, že se o události na serveru dozvíme se zpožděním.

Tento model má tedy smysl pouze v případě, že by se na serveru opravdu měnila pro nás relevantní data každou vteřinu, což by mohl splnit chat se stovkou uživatelů v jedné místnosti, případně některé webové on-line hry. Běžné aplikace ovšem takového komunikačního vytížení nedosahují.

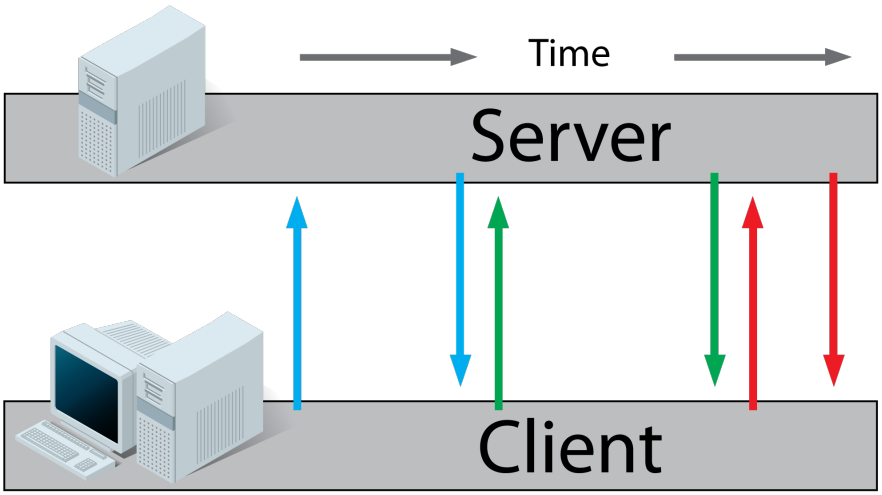


Obrázek 4 Short-polling, každá barva představuje jednu dvojici request/response
zdroj: http://stackoverflow.com/questions/11077857

3.1.2 Long-polling

Je v principu velice podobný short-pollingu, řeší ovšem jeho nedostatek mnoha "zbytečných" dotazů na server. Klient pošle serveru request s žádostí o nová data, server ovšem neodpoví hned. Pozdrží odpověď do doby, kdy jsou nová data k dispozici a teprve poté pošle klientovi odpověď. Ten ji zpracuje a okamžitě pošle serveru další request.

Výhod tohoto modelu je několik. Citelně snižuje objem komunikace mezi klientem a serverem. Klient se o události na serveru dozví opravdu v okamžiku, kdy se stala. V neposlední řadě je long-polling podporován i ve starších internetových prohlížečích, což mu dává výhodu oproti novějším technologiím. Jeho hlavní nevýhodou je velká zátěž serveru na udržování otevřených spojení - v případě velkých webových aplikaci jich v paměti musí držet i desítky tisíc. Přesto je tato technologie velice rozšířená a využívaná.

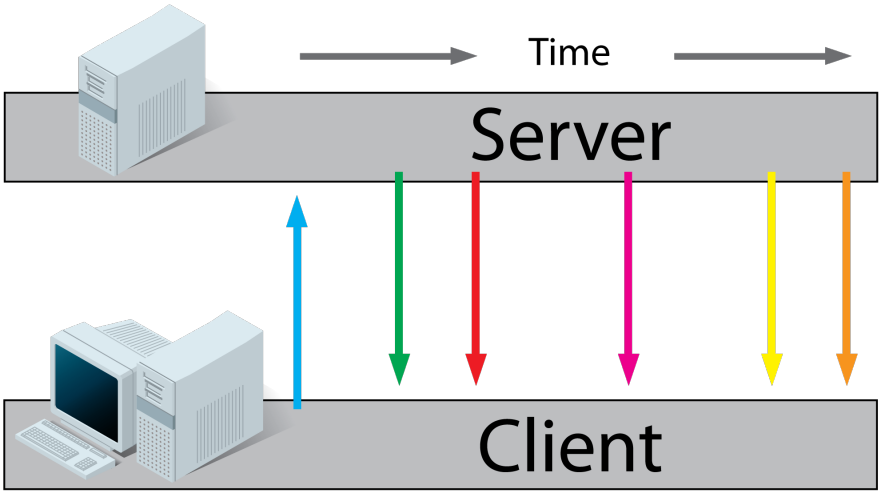


Obrázek 5 Long-polling, každá barva představuje jednu dvojici request/response
zdroj: http://stackoverflow.com/questions/11077857

3.1.3 Server sent events

V podstatě nevytváří duplexní spojení. Klient pošle request o připojení se k určitému kanálu na serveru a od té doby mu server automaticky posílá přes tento kanál nová data. Princip je podobný jako když si člověk nechává posílat email o nových příspěvcích v tématu nějakého fóra.

Kromě chybějící podpory tohoto systému v prohlížeči Internet Explorer, je nevýhodou, že klient přes tento kanál data posílat nemůže. Se serverem nadále komunikuje běžnými HTTP requesty. To nemusí v některých případech vadit. U klienta jde především o to, aby se dozvěděl o událostech na serveru v okamžiku, kdy se stanou, což SSE splňuje. Přesto je implementačně jednodušší mít jeden komunikační kanál pro oba směry. V rámci této práce má však SSE ještě jednu nevýhodu a tou je chybějící podpora CORS (Cross-origin resource sharing). Vysvětlením tohoto mechanismu a jeho použitím se bude detailněji zabývat kapitola 4 a její sekce 4.2.3.

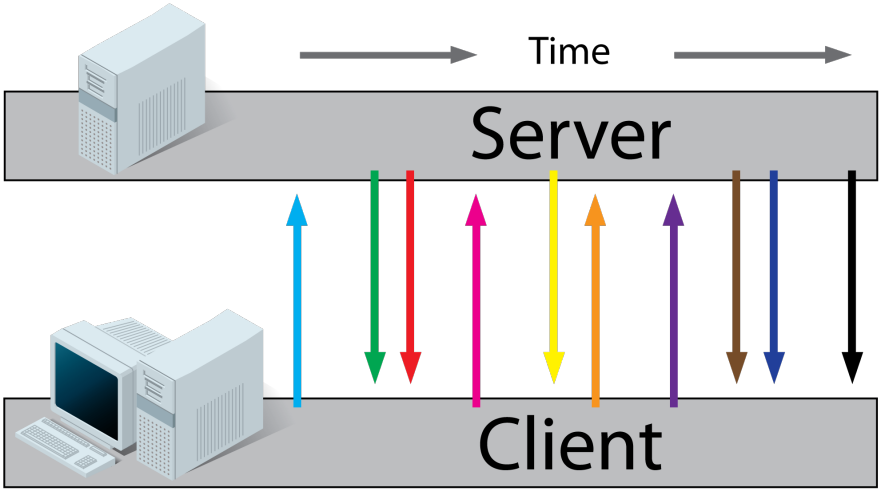


Obrázek 6 Server sent events, modrá šipka je žádost o spojení, další jsou pak odpovědi serveru v okamžiku nějaké události
zdroj: http://stackoverflow.com/questions/11077857

3.1.4 Websockety

Poměrně nová technologie, jejíž specifikace byla standardizována v roce 2011 a ve webových prohlížečích je součástí HTML verze 5. Vytváří plně duplexní komunikační kanál pomocí přímého TCP (Transmission Control Protocol) spojení mezi serverem a klientem. Právě využití TCP umožňuje plně duplexní spojení oproti HTTP užívanému předchozími třemi technologiemi. TCP je součástí transportní vrstvy zatímco HTTP se nachází v hierarchicky nižší aplikační vrstvě [5], datový HTTP packet je tedy součástí TCP packetu, který jej zapouzdřuje. Pro technologii websocketů to znamená, že není omezena polo duplexitou HTTP.

Jediné co websockety mají s HTTP má společného, je iniciální požadavek na vytvoření spojení. Ten musí přijít standardním HTTP requestem obsahující hlavičku s žádostí o upgrade spojení na úroveň TCP. Pokud je tento upgrade ze strany serveru podporován, vytvoří se žádané spojení a je umožněna plně duplexní komunikace serveru a klienta, dokud není spojení jednou ze stran zrušeno.



Obrázek 7 Websocket spojení, po prvním requestu od klienta mohou obě strany posílat data bez omezení.
zdroj: http://stackoverflow.com/questions/11077857

Největší výhodou této technologie je, že se jedná opravdové obousměrné real-time spojení, které u předchozích technologií mělo vždy nějaký nedostatek. Dalším plusem je velice malý objem datové komunikace. Zprávy HTTP protokolu mají takzvaný overhead v řádu stovek až tisíců bajtů. Jsou to data v hlavičce obsahující všemožné dodatečné informace pro přijímající stranu. Zde jich ovšem není potřeba, jelikož spojení je již ustanoveno a tyto informace byly předány. Overhead websocket zprávy je v závislosti na její velikosti 2 až 12 bajtů.

Zátěž serveru je také nižší. Long-polling i SSE vytvářejí pro každého připojeného klienta vlastní proces, který musí držet v paměti po dobu trvání spojení. Websocket server má typicky jeden socket a tedy jeden proces na zpracování všech requestů od uživatelů, v případě potřeby může vytvořit socketů a procesů více. Celkově jich však bude řádově méně, než u předchozích řešení. Například server Jetty ve verzi 9 tvrdí, že je schopen udržovat až stovky tisíc websocket spojení. [2] Nutno dodat, že toto číslo je také značně závislé na hardwarových prostředcích serveru.

Přes svou poměrně krátkou existenci jsou websockety podporovány všemi velkými webovými prohlížeči [6], web servery a jsou také součástí standardu Java 7 EE. U prohlížečů ovšem musíme stále pamatovat na zpětnou kompatibilitu, a nenechat websocket spojení jako jedinou možnost komunikace. Některé prohlížeče totiž přidaly websocket podporu teprve nedávno (IE 10, Opera 26) jejich starší verze by se se serverem nedomluvily.

Poslední zmínka kladných vlastností websocketů patří podpoře CORS. Jedinou nevýhodou této technologie tak může být složitější implementace, zvláště na straně serveru. Existuje ovšem několik frameworků, které jsou přímo orientovány na podporu websocket komunikace. Pro JavaScript je to například Socket.io [7], pro Javu pak Atmosphere [8], který obsahuje knihovnu i pro JavaScript.

Websockety tedy mají téměř všechna pro, které u komunikační technologie hledáme. Starší technologie ovšem stále není radno zavrhovat z důvodu kompatibility se staršími systémy. Ideální je, zkombinovat více těchto technologií dohromady a nabídnout každému uživateli pro něj nejlepší možnou volbu. V následující části si ukážeme framework, který nám takovou implementaci značně zjednoduší.

3.2 Atmosphere

Framework podporující websocket komunikaci pro Javu EE/JavaScript vyvinutý Jean-Francoisem Arcandem (představitel skupiny Async-IO.org). Cílem frameworku je usnadnit vývojářům práci s implementací real-time komunikace ve webových aplikacích. Pro klientskou stranu framework poskytuje JavaScript knihovnu obsahující funkce nejen pro websocket komunikaci, ale i pro SSE a long-polling. Umožňuje pro ně také automatický fallback v případě, že primární zvolená technologie pro komunikaci není podporovaná.

Pro serverovou stranu nabízí výběr mezi nativní implementací technologií, nebo pokud ji server neobsahuje, případně programátor z určitých důvodů nechce, použití knihoven Atmosphere [9], dostupných přes Maven. Implementace pomocí knihoven Atmosphere teoreticky zaručuje přenositelnost kódu mezi různými servery, zatímco nativní knihovny mohou poskytnout lepší výkon.

Tato práce používá první řešení. Přenositelnost implementace mezi různými web servery však testována nebude. Představu, jak Atmosphere a vůbec technologie websocketů pracuje na straně serveru i klienta, poskytuje následující souhrn z implementačního tutoriálu [10].

3.2.1 Server

Nejdůležitější komponentou je rozhraní **AtmosphereHandler**, které představuje přípojný bod pro klienty. Implementuje se pomocí anotace

```
@AtmosphereHandlerService(path = "/{path}")
```

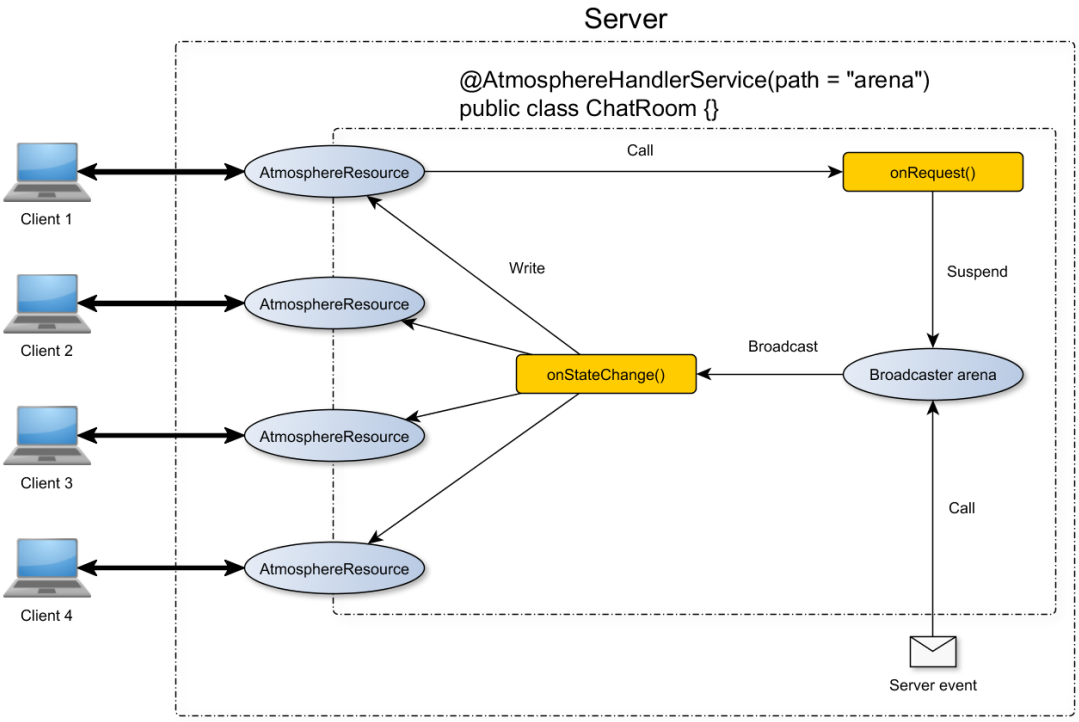
přičemž je nutné specifikovat proměnnou *path* určující název konkrétního přípojného bodu. Rozhraní dále definuje tři funkce, přesněji řečeno callbacky:

- **onRequest** - zavolána při obdržení zprávy (requestu) od klienta
- **onStateChange** - volána pokud je uzavřeno spojení od připojeného klienta, nebo pokud broadcaster tohoto AtmosphereHandleru provede operaci *broadcast*
- **destroy** - zavolána při ukončení činnosti Atmospheru

Druhou komponentou je **AtmosphereResource**. Ten představuje komunikační kanál mezi serverem a jedním klientem. Pomocí něj je možné od klienta přijmout zprávy, poslat mu zprávy a především spojení uspat, pokud není aktivně používáno. Každý AtmosphereResource má svůj identifikátor *uuid* pomocí kterého je možné resource kdykoli najít a zavolat.

Třetí komponenta se nazývá **Broadcaster** a můžeme jej přirovnat ke komunikačnímu uzlu. Každý broadcaster má svoje jméno, které jej identifikuje a umožňuje jej vyhledat a připojit se k němu. Každý AtmosphereResource, který se připojí k serveru, musí být napojen na jeden nebo více broadcasterů. Když je pak na nějakém broadcasteru zavolána metoda *broadcast("zpráva")*, odešle se zpráva všem připojeným resourcům.

Broadcaster je vždy vytvořen při definici AtmosphereHandleru a je pojmenován podle jeho proměnné *path*. AtmosphereResource přistupující k serveru přes tento přípojný bod, je pak automaticky napojen na tento broadcaster. Také je možné libovolný broadcaster vytvořit pomocí konstruktoru.



Obrázek 8 Schématické zobrazení komunikačního procesu na Atmosphere serveru při obdržení requestu od klienta a při události na serveru.

Obr. 8 reprezentuje komunikaci klienta se serverem přes přípojný bod *arena*. Request klienta doputuje na server a je transformován na objekt AtmosphereResource. Ten je jako parametr předán funkci *onRequest* a v ní zpracován dle potřeby. Když je zpracování zprávy dokončeno, resource je uspán. Jakmile proběhne na serveru událost, o které mají být informováni klienti připojeni k bodu *arena*, je zavolán stejnojmenný objekt broadcaster a jeho funkce *broadcast*. Ta probudí všechna spojení příslušných klientů a zavolá funkci *onStateChange*, která pošle klientům zprávu.

Toto je velice základní princip fungování Atmospheru a postupem času byly přidány další funkce a logika, které zdokonalují jeho schopnosti. Plná anotace handleru by nyní vypadala následovně:

```
@AtmosphereHandlerService(path = "/chat",
   broadcasterCache = UUIDBroadcasterCache.class,
   broadcaster = RedisBroadcaster.class,
   interceptors = { AtmosphereResourceLifecycleInterceptor.class,
      BroadcastOnPostAtmosphereInterceptor.class,
      TrackMessageSizeInterceptor.class,
      HeartbeatInterceptor.class
   })
public class ChatRoom extends OnMessage<String> {}
```

- **BroadcasterCache** slouží k překlenutí času, kdy by došlo k výpadku připojení klienta a zajišťuje, že všechny zprávy odeslané v době výpadku budou klientovi doručeny.

- **RedisBroadcaster** přidává cloudovou podporu. K jeho použití je potřeba ještě dodatečná konfigurace ale ve výsledku zajišťuje připojení broadcasterů na PubSub kanál Redis serveru a sdílení zpráv mezi instancemi webové aplikace na různých serverech. O Redis serveru detailněji pojednává část práce věnující se Amazon technologii ElastiCache.

Interceptory obecně jsou filtry, které jsou vždy volány před a po provedení funkce *onRequest*. Atmosphere jich obsahuje více, ale tři obecně důležité jsou:

- **AtmosphereResourceLifecycleInterceptor** se stará o automatické uspání a opětovnou aktivaci AtmosphereResource.
- **TrackMessageSizeInterceptor** umožňuje kontrolovat velikost doručovaných zpráv. V případě větší délky zpráva nemusí přijít v jednom celku a systém díky této kontrole ví, kdy má ještě čekat na další části zprávy.
- **HeartbeatInterceptor** slouží k udržování uspaných spojení. Některé firewally nebo proxy povolují spojení být neaktivní pouze určitou dobu. Tento interceptor proto posílá v daném časovém intervalu bílé znaky, aby nedošlo k uzavření spojení.

Poslední změnou oproti původnímu kódu je rozšíření třídy *OnMessage*. Ta sama implementuje původní tři funkce AtmosphereHandleru a nahrazuje je funkcemi *onOpen* pro příchozí zprávy a *onMessage* pro zprávy odchozí. Přidává navíc některé další, jako *onResume*, *onTimeout* a *onDisconnect*.

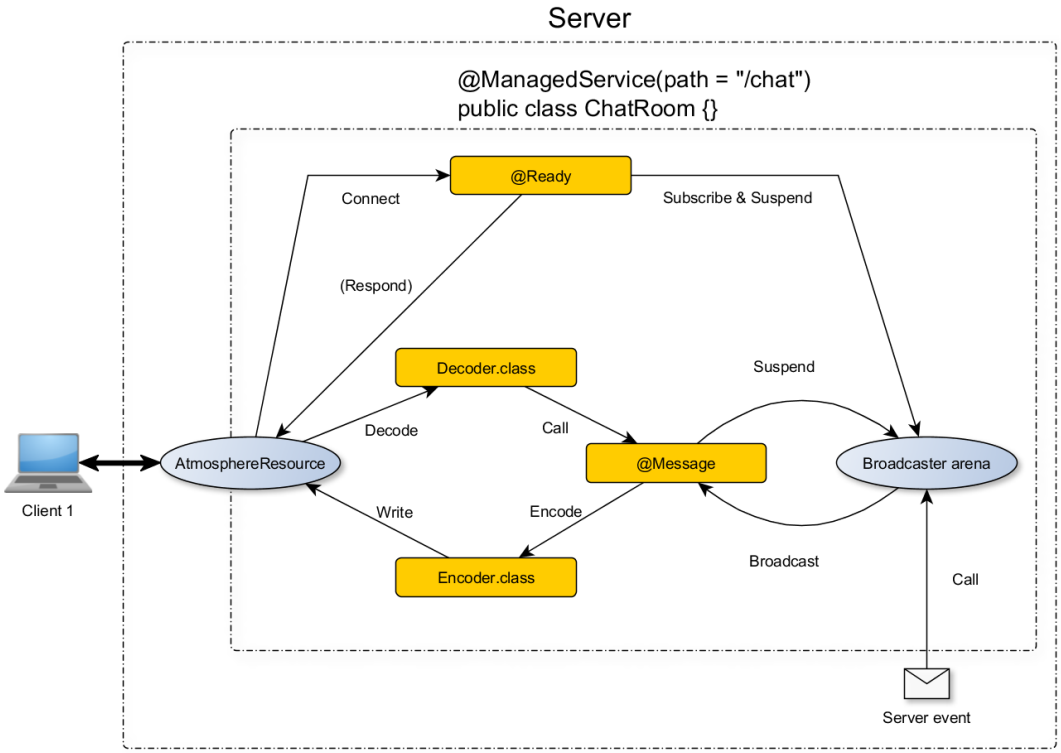
Jelikož se uvedené parametry anotace *@AtmosphereHandlerService* používají pro velkou část případů, vytvořili tvůrci novou anotaci *@ManagedService*, do které všechny dosud popsané funkcionality implementovali. Kód nutný pro serverovou stranu se tím velice zjednodušil a pro funkční aplikaci Chatu nyní stačí následující:

```
@ManagedService(path = "/chat")
public class ChatRoom {
   private final Logger logger = LoggerFactory.getLogger(Chat.class);

   @Ready
   public void onReady(final AtmosphereResource r) {
      logger.info("Browser {} connected.", r.uuid());
   }

   @Disconnect
   public void onDisconnect(AtmosphereResourceEvent event) {
      if (event.isCancelled()) {
         logger.info("Browser unexpectedly disconnected");
      } else if (event.isClosedByClient()) {
         logger.info("Browser closed the connection");
      }
   }

   @Message(encoders = {JacksonEncoder.class}, decoders =
       {JacksonDecoder.class})
   public Message onMessage(Message message) throws IOException {
      logger.info("{} just send {}", message.getAuthor(),
          message.getMessage());
      return message;
   }
}
```

Kód by dokonce mohl být ještě redukován odstraněním metod *onDisconnect* a *onReady*, které zde mají pouze informativní funkci.

Všechny funkce této třídy se nyní definují pomocí anotací. *@Ready* a *@Disconnect* označují funkce volané kdykoli se připojí, respektive odpojí klient. Anotace *@Message* definuje funkci pro zpracování zpráv a to jak příchozích tak odchozích. Funkcí anotovaných pomocí *@Message* může být vícero a liší se pouze ve svých parametrech. Ty musí korespondovat s definovanými encodery a decodery, což jsou filtry pro příchozí a odchozí zprávy. Pokud je u *@Message* funkce definován encoder, je jeho třída použita ke zpracování odchozí zprávy. Decodery naopak zpracovávají příchozí zprávy. Tyto dodatečné filtry se dají velice užitečně použít například pro mapování JSON zpráv na Java objekty.

Nová anotace *@ManagedService* také přidává možnost definovat základní CRUD funkce: *@Post*, *@Get*, *@Put* a *@Delete*.



Obrázek 9 Schématické zobrazení komunikačního procesu na Atmosphere serveru při použití @ManagedService anotace

Při použití této anotace se poněkud mění serverové schéma Atmospheru a ukazuje jej obr. 9. Při prvním připojení je klienta je zavolána funkce @Ready, která mu může odpovědět vlastní zprávou, resource je přiřazen broadcasteru *arena* a je uspán. Při každé další zprávě je nejprve volán dekodér (pokud je specifikován) a poté funkce @Message, která vstupním parametrem odpovídá výstupu dekodéru. Po zpracování zprávy je resource opět uspán. Postup při události na serveru je obdobný, funkce broadcast zavolá metodu @Message, opět podle vstupního parametru, ta předá zprávu encoderu (pokud je definován) a výstup je zapsán do AtmosphereResource pro doručení klientovi.

Starší AtmosphereHandler ovšem zůstává stále zachován a podporován. Proto zde byly uvedeny obě architektury. Na starší jsou lépe vidět základní principy a její implementace se hodí pokud je potřeba dostat se blíže k jádru Atmospheru, případně si některé nižší funkce upravit pro svou potřebu.

ManagedService je velkým zjednodušením frameworku, což je praktické pro základní vývoj. Pokud je ovšem potřeba složitějších konstrukcí, můžeme narazit na omezení v podobě pevně definovaných parametrů či funkcí. Je proto důležité oběma architekturám nejprve rozumět a poté zvolit dle požadavků aplikace tu vhodnější.

3.2.2 Klient

Strana klienta je již podstatně jednodušší. Nejprve je vytvořen objekt *request* obsahující základní informace pro připojení k serveru:

```
var request = { url: document.location.host + 'chat',
contentType : 'application/json',
transport : 'websocket' ,
fallbackTransport: 'long-polling',
timeout: 300000};
```

Povinný je pouze atribut url pro určení přípojného bodu serveru. Ostatní atributy mají své default hodnoty, které jsou použity pokud je klient nespecifikuje sám. Jejich kompletní seznam je k dispozici zde [11]. Význam několika nejdůležitějších:

- **contentType** - typ dat, která budeme odesílat
- **transport** - typ spojení, který bude primárně zvolen pro komunikaci
- **fallbackTransport** - typ spojení, pokud server nepodporuje primární typ
- **timeout** - maximální doba životnosti spojení pokud nejsou přijímány ani posílány žádné zprávy
- **enableXDR** - povolení CORS

Jakmile je objekt vytvořen, klient se připojí zavoláním funkce:

```
var socket = atmosphere.subscribe(request);
```

Od této chvíle je možné používat objekt *socket* k odesílání dat pomocí:

```
socket.push(data);
```

Na objektu request jsou pak definované veškeré další callbacky obsluhující vytvořené spojení. Nejdůležitější je *onMessage*, která přijímá data od serveru:

```
request.onMessage = function(response){
// process response
};
```

Objekt *response* obsahuje kromě samotných dat od serveru *responseBody* také informace jako *state*, *transport*, *headers* a podobné. Některé další callbacky objektu request jsou:

- **onOpen** - zavolána když je spojení otevřeno
- **onClose** - zavolána při ukončování spojení
- **onError** - volána pokud spojení nahlásí chybu
- **onTransportFailure** - zavolána pokud server odmítne nastavený typ transportu

V okamžiku, kdy klient chce ukončit spojení, zavolá funkci:

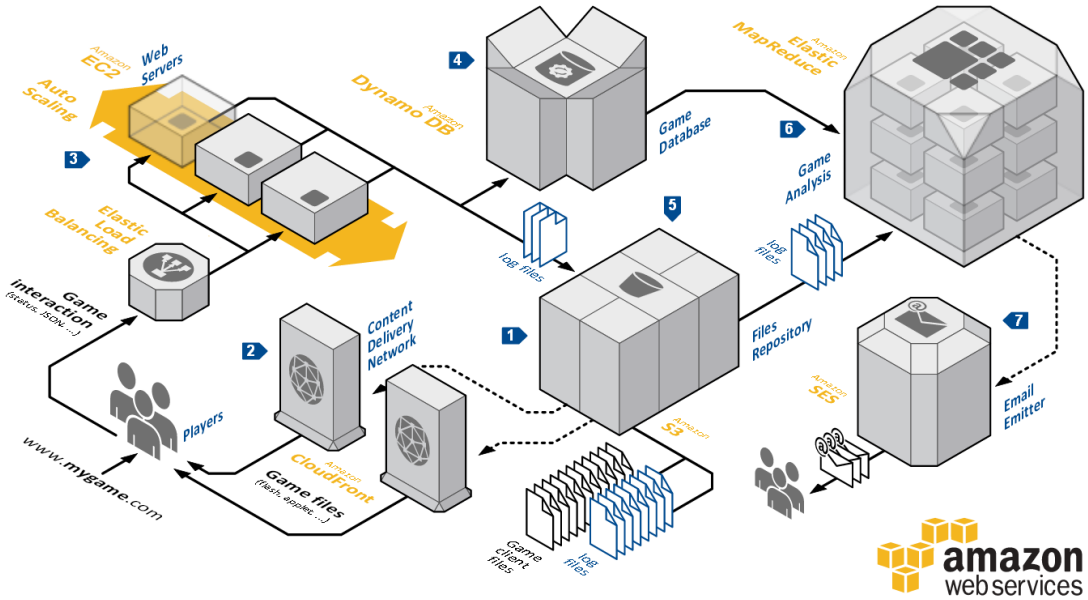
```
atmosphere.unsubscribe();
```

Ta před samotným ukončením spojení ještě volá funkci *onClose*. Pokud není *unsubscribe* zavolána před opuštěním stránky, zůstává spojení automaticky otevřeno až do uplynutí doby stanovené proměnnou *timeout*, případně do zavření celého browseru. Proto je poměrně dobrou praxí detekovat opuštění stránky uživatelem a uzavřít spojení i při této události.

Framework Atmosphere je velice silným nástrojem, který umožňuje obohacení aplikace nejen o websocket technologii, ale také o množství dalších podpůrných funkcí pro komunikaci serveru a jeho klientů. Dokumentace je v některých částech nedostatečná, zejména pokud se programátor pustí mimo několik málo standardních použití. Ovšem objem kódu a čas který framework celkově ušetří je dostatečnou kompenzací.

3.3 Amazon Web Services

Zkráceně AWS, je soubor cloudových technologií a služeb poskytovaných společností Amazon. Definicí toho, co je cloud a cloudová služba, je mnoho a tato práce se rozebíráním těchto pojmů zabývat nebude. Jejich problematika je dobře popsána například v práci Veroniky Noskové [12]. Tato část práce se bude soustředit na popis AWS služeb které v ní budou využity, případně na zdůvodnění proč.



Obrázek 10 Ukázka komplexní AWS architektury použité pro webovou on-line hru
zdroj: http://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_games_10.pdf

Obr. 10 ilustruje, jak může vypadat architektura AWS služeb pro aplikaci představující webovou on-line hru. Architektura této práce zdaleka tak složitá není, což bude prezentováno v další kapitole. Tento obrázek však poslouží k představě, kde se jaká služba nachází, jaký má význam a se kterými dalšími službami může interagovat.

3.3.1 Elastic Compute Cloud

V AWS označován jako EC2 je základní komponentou většiny architektur. Na obr. 10 je reprezentován číslem 3 a představuje instanci virtuálního serveru. Její konkrétní využití je na uživateli - web server, aplikační server, vlastní load balancer, instance dedikovaná čistě na matematické výpočty - možností je stejně neomezeně jako v případě vlastního fyzického serveru. Amazon nabízí velkou škálu typů instancí, které se liší jednak výkonem a jednak zaměřením:

- **T** a **M** - obecný typ s vyváženým výpočetním, paměťovým a síťovým výkonem
- **C** - výkonné výpočetní instance s posílenou síťovou podporou
- **R** - instance optimalizované pro velké paměťové nároky vhodné pro výkonné databázové a cachovací systémy
- **G** - instance osazené výkonnými grafickými kartami pro GPU výpočty nebo práci s video streamy
- **I** - instance pro rychlé operace s velkým množstvím dat při náhodném přístupu, obsahují množství rychlých SSD disků, ideální pro NoSQL databáze
- **HS** - podobně jako typ I zaměřeny na operace s daty s rozdílem orientace na sekvenční přístup a poskytujících kapacity až 48 TB

U každého typu je ještě možno vybrat model určující počet jader procesoru, velikost paměti a diskového prostoru.

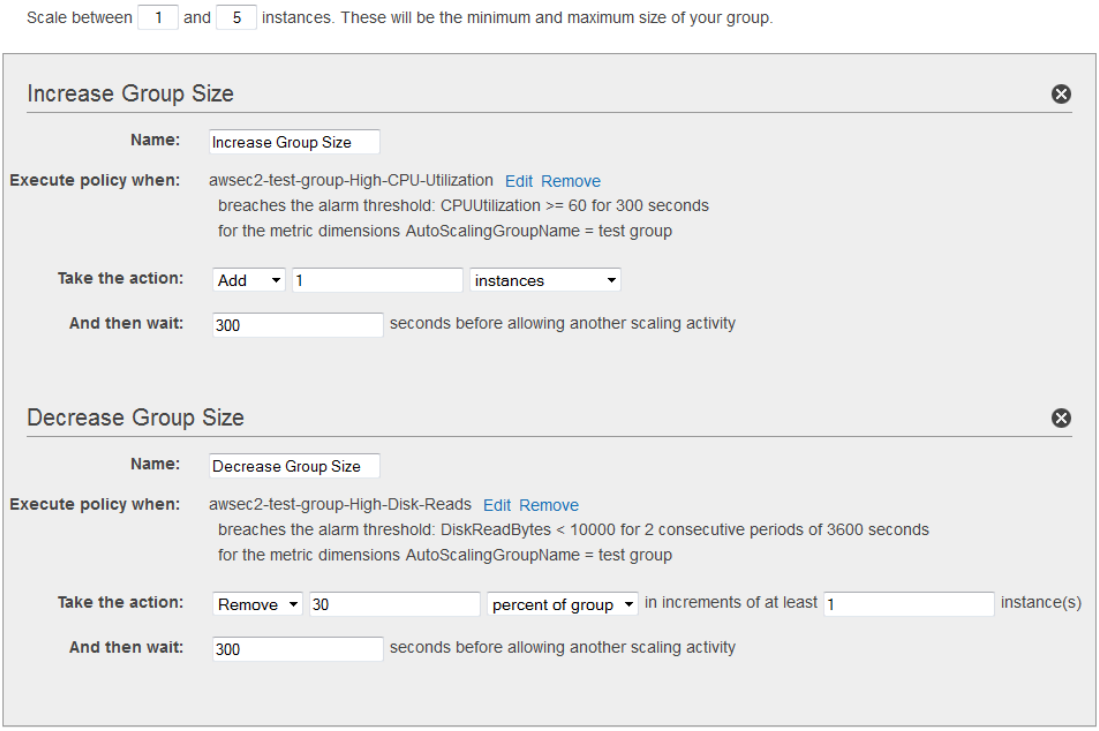
Model	vCPU	Mem (GiB)	SSD Storage (GB)	Model	vCPU	Mem (GiB)	SSD Storage (GB)
m3.medium	1	3.75	1 x 4	i2.xlarge	4	30.5	1 x 800
m3.large	2	7.5	1 x 32	i2.2xlarge	8	61	2 x 800
m3.xlarge	4	15	2 x 40	i2.4xlarge	16	122	4 x 800
m3.2xlarge	8	30	2 x 80	i2.8xlarge	32	244	8 x 800
c3.large	2	3.75	2 x 16	r3.large	2	15.25	1 x 32
c3.xlarge	4	7.5	2 x 40	r3.xlarge	4	30.5	1 x 80
c3.2xlarge	8	15	2 x 80	r3.2xlarge	8	61	1 x 160
c3.4xlarge	16	30	2 x 160	r3.4xlarge	16	122	1 x 320
c3.8xlarge	32	60	2 x 320	r3.8xlarge	32	244	2 x 320

Obrázek 11 Dostupné modely Amazon EC2 instancí několika typů a jejich konfigurace
zdroj: http://aws.amazon.com/ec2/instance-types/

Pokud není potřeba specifických nastavení instance, je k dispozici několik předdefinovaných konfigurací pro konkrétní programovací jazyky (JavaScript, Ruby, PHP, Java...) nabízejících model PaaS (Platform as a Service) s automatickým přednastavením load balancingu a škálování. V tomto případě je možné si zvolit verzi jazyka a operační systém Windows nebo Linux v několika různých distribucích včetně vlastní Amazon distribuce Linuxu, například aplikační server je ovšem pevně daný.

V případě vyšších nároků na konfigurovatelnost instance disponuje Amazon i modelem IaaS, kdy je pro start instance zvolen pouze operační systém a veškerá další nastavení a instalace softwaru je v rukou uživatele. Obraz takto vytvořené instance je navíc možné uložit jako vlastní AMI (Amazon Machine Image) a použít jej k vytvoření dalších instancí aniž by bylo nutné provádět celou konfigurační proceduru znovu. Model IaaS byl zvolen i pro tuto práci kvůli instalaci vlastního Jetty serveru, který Amazon standardně nenabízí.

Nejdůležitější vlastností EC2 instancí je jejich škálovatelnost. AWS nabízí možnost škálovat instance manuálně i automaticky. U manuálního škálování je na uživateli, aby sledoval statistiky a zatížení jeho instancí a dle nich přidával nebo odebíral instance. Automatické škálování obsahuje několik metrik, u nichž je možné nastavit hranice pro škálování nahoru nebo dolů, stejně jako počet přidaných i odebraných instancí v závislosti na momentálním stavu. Metriky zahrnují vytížení CPU, objem přijímaných nebo odeslaných dat a počty operací zápisu nebo čtení z disku. U každého typu je možné určit za jaký časový úsek jsou hodnoty měřeny, zda se jedná o hodnoty minimální, maximální, průměrné nebo o jejich součet. Při každé změně je také možné zaslat uživateli upozornění.

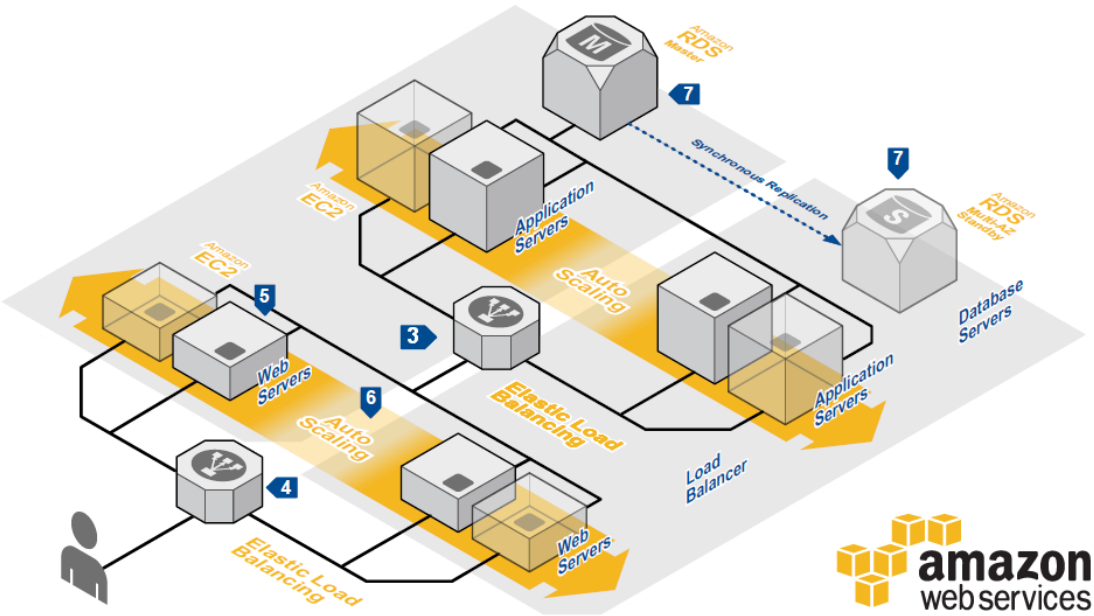


Obrázek 12 Ukázka možností nastavení automatického škálování instancí EC2.
zdroj: AWS EC2 Management Console

Automatické nastavení vyžaduje nějaký čas pro jeho testování. Zvláště škálování směrem nahoru je nutné nastavit pečlivě a určit maximální možný počet instancí. Pokud systém nedopatřením naškáluje na 100 středních instancí typu C, bude jeho provoz stát 50$ za hodinu. Při správném nastavení ovšem může být výsledkem systém, který je vždy ideálně vytížen na 80% a je jedno zda je k němu připojeno 20 uživatelů nebo 10 000.

3.3.2 Elastic Load Balancer

Zkratkou ELB, se nachází těsně před EC2 instancemi. Zajišťuje rovnoměrné rozvrstvení požadavků na každou z nich a zároveň hlídá, že připojené instance jsou zdravé - tedy že jsou v provozu a komunikují. Amazon se stará o jeho automatické škálování, nemělo by se tedy stát, že by byl úzkým hrdlem systému. Může být použit jak pro požadavky přicházející zvenku, tedy od připojených klientů (obr. 13, číslo 4), tak na vnitřní požadavky, například od web serverů na aplikační servery (obr. 13, číslo 3). Všechny požadavky jsou také monitorovány a případně logovány.



Obrázek 13 Použití ELB pro balancování requestů uživatele a interních requestů web serverů. *zdroj: http://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_web_01.pdf*

ELB může operovat ve dvou režimech. Prvním je režim aplikační vrstvy pracující přímo s HTTP/HTTPS požadavky. To znamená, že ELB má přístup k obsahu HTTP requestu a může jej měnit, což zpřístupňuje to dvě poměrně důležité technologie:

- **Sticky sessions** - obsahem requestu je cookie AWSELB, která určuje na jakou konkrétní instanci má být tento request přeposlán. Používají se pro aplikace, které si o uživateli uchovávají určité aktuální informace v paměti v takzvané session. Může se jednat například o informaci zda je uživatel v aplikaci přihlášen, detaily z jeho profilu, případně jaké stránky navštívil. Když pak systém tyto informace potřebuje, má je k dispozici rovnou a nemusí je hledat v databázi.Tyto informace se ale uchovávají v paměti konkrétní instance a jsou tedy k dispozici pouze jí. Pokud by požadavek uživatele došel na jinou instanci, ta zmíněné informace k dispozici nemá a mohla by po uživateli požadovat aby se například opět přihlásil. Řešením je buď ukládání těchto sessions na sdíleném úložišti, nebo právě přesměrování uživatele vždy na stejnou instanci k níž přistoupil poprvé.

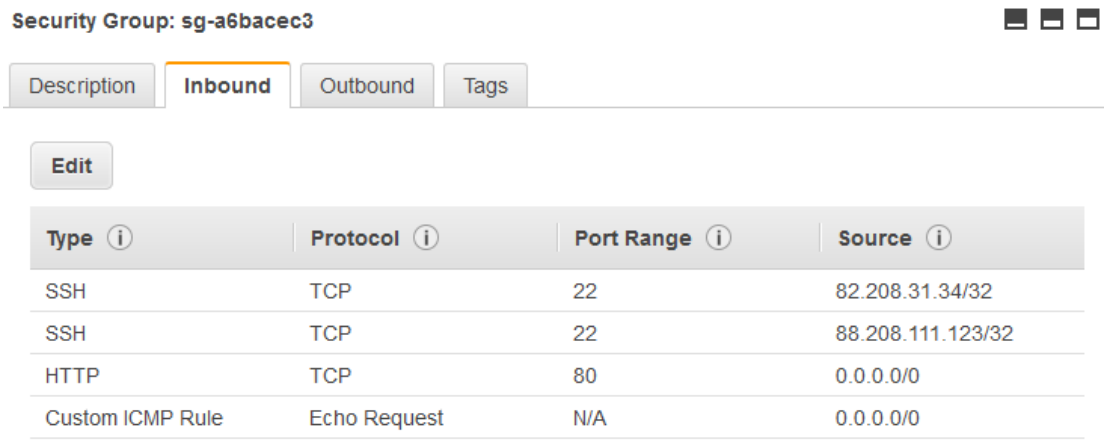
- **X-Forwarded Headers** - do requestu je přidána hlavička **X-Forwarded-For** v níž jsou zaznamenány IP adresy jak odesílatele requestu, tak všech případných proxy, kterými tento požadavek prošel. Každý request totiž typicky obsahuje IP adresu zdroje odkud přišel. Pamatuje si však pouze tu poslední. Pokud projde proxy nebo load balancerem, adresa zdroje bude ukazovat na tyto prvky, ne na originálního klienta. Stejně fungují další dvě hlavičky **X-Forwarded-Proto** pro originální protokol a **X-Forwarded-Port** pro originální port.

Druhým režimem ELB je práce v transportní vrstvě na úrovni protokolu TCP/SSL. Jak bylo již zmíněno v sekci 3.1.4 protokol TCP je nadřazen HTTP takže obaluje jeho datové packety svou vlastní hlavičkou a řídí se pouze podle ní. Data potřebná pro obě výše zmíněné technologie se však nachází až v HTTP requestu, který v současné době ELB neumí extrahovat a zpracovat. TCP režim tak neumožňuje technologii *Sticky sessions* ani *X-Forwarded Headers*. Je ovšem nutný, pokud bychom s EC2 instancemi chtěli například komunikovat pomocí websocketů, nebo použít TCP pro streamování dat.

3.3.3 Security Groups

Veškerá bezpečnostní politika služeb AWS je řízena pomocí takzvaných bezpečnostních skupin. Každé službě, musí být přiřazena nějaká bezpečnostní skupina, která určuje jak a s kým má služba povoleno komunikovat. Skupina má své jméno a obsahuje pravidla pro příchozí a odchozí komunikaci. Jedno pravidlo vždy určuje typ spojení, protokol, port a zdroj, kterým může být IP adresa nebo jiná bezpečnostní skupina.

Příklad nastavení je na obr. 14. Typ a protokol jsou povinné parametry, port je nastavován v závislosti na prvních dvou položkách a zdroje může označovat konkrétní prvky, skupiny, nebo kohokoli, což je reprezentováno řetězcem samých nul.



Type	Protocol	Port Range	Source
SSH	TCP	22	82.208.31.34/32
SSH	TCP	22	88.208.111.123/32
HTTP	TCP	80	0.0.0.0/0
Custom ICMP Rule	Echo Request	N/A	0.0.0.0/0

Obrázek 14 Nastavení vstupních pravidel Security Group pro EC2 instanci. SSH je omezeno na konkrétní adresy kvůli Admin přístupu k instanci, HTTP požadavky je naopak možné provádět odkudkoli,stejně jako příkaz ping.
zdroj: AWS EC2 Management Console

3.3.4 Databáze

AWS nabízí dva druhy databází. Klasickou relační SQL databázi, označovanou jako Amazon RDS (Relational Database Service) na obr. 13 označenou číslem 7 a NoSQL databázi Amazon DynamoDB obr. 11 pod číslem 4.

Amazon RDS nabízí výběr z pěti databázových systémů: MySQL, Oracle, Microsoft SQL Server, PostgreSQL nebo Amazon Aurora. Umožňuje tak přechod z používání klasické databáze na EC2 instanci na RDS téměř beze změny kódu. RDS je dle testů [13] o něco rychlejší než provozování vlastního DB serveru, navíc poskytuje automatické logování, údržbu a zálohování. Databázi je možné replikovat dvěma způsoby orientovanými buď na zvýšení bezpečnosti dat nebo na jejich škálovatelnost, přičemž oba přístupy lze i kombinovat. Výkon databáze také určuje typ instance na kterém je provozována, přičemž pro RDS Amazon nabízí instance typu T, M a C.

DynamoDB je vlastní NoSQL databází Amazonu, která zajišťuje především vysoký výkon společně s téměř neomezenou kapacitou. Platí se za objem vašich uložených dat, za čtecí a zápisové jednotky určující výkon databáze a nakonec za objem přenosu dat ven z DB.

Pro tuto práci bude využita RDS databáze k uchování dat uživatelů. DynamoDB prozatím využita nebude, pro ukládání uživatelských session ji nahradí ElastiCache.

3.3.5 ElastiCache

Principem představuje ElastiCache (EC) také NoSQL key-value databázi, která se od těch klasických liší v umístění ukládaných dat. Ta se zde nacházejí v paměti, místo na pevném disku, což sice omezuje jejich celkovou možnou velikost, výměnou ovšem nabízí daleko rychlejší odezvu především při čtení dat. Spíše než jako NoSQL databáze se tedy označuje jako cachovací systém.

EC nabízí výběr mezi provozem pomocí systému Redis a MemCache. MemCache je o několik let starší a Redis jej postupně dohnal a překonal co se týče dostupných funkcí. Od vydáni Redisu verze 3.0 ztratil MemCache poslední výhodu, kterou byla podpora clusteringu. Ten umožňuje současný běh několika instancí, jež spolu spolupracují a tváří se navenek jako jedna. Clustering tedy značí vysokou možnost škálování. V současných verzích je výkon a paměťová náročnost obou systémů porovnatelná, ale Redis nabízí navíc několik důležitých funkcí:

- různé typy ukládaných dat - hash, list, sorted set
- automatické zálohy dat na disk
- transakce
- pub/syb systém
- nativní clustering (MC nabízí pouze pomocí třetí strany)

Redis je také používán pro cloudovou podporu Atmosphere frameworku a jeho sdílení broadcastovaných zpráv mezi instancemi v cloudu, byl tedy zvolen pro použití v této práci. Jak bylo zmíněno v předešlé části, Redis bude využit pro uložení uživatelských session. Jeho primárním použitím ale bude sdílení založených her mezi instancemi a to pomocí dvou možných architektur využívajících listy a pub/sub kanál. Jejich popis a testování bude ukázáno v následujících kapitolách.

3.4 Shrnutí technologií

Soupis dostupných technologií pro real-time komunikaci ukazuje, že ideální volbou pro její realizaci budou websockety s případným fallbackem na long-polling pro podporu starších prohlížečů. V rámci cloudových technologií je naznačeno jedno omezení, které bude třeba obejít a tím je nemožnost provozování load balanceru současně s technologií sticky sessions a s podporou websocketů. Jeho řešení se věnuje část 4.2.3. Pro snadnější implementaci websocketů jak na klientovi tak na serveru bude použit framework Atmosphere. Serverové knihovny pro websockety budou navíc podrobeny testování, pro zjištění výkonnostních rozdílů mezi jejich nativní verzí z Jetty a univerzální verzí z Atmospheru.

Co se týče výběru cloudových služeb a technologií, nejprve k výběru jejich poskytovatele. Ten byl pro tuto práci byl předem dán ze zadání, její autor se však podílel na výběru poskytovatele i pro projekt Hravě. Proto několik slov k této volbě. Primárním důvodem byl velký kredit, který Hravě dostalo od společnosti Amazon v rámci jeho podpory startup projektů. Samotná rešerše dostupných cloudových řešení ukázala, že na současném trhu je několik velkých poskytovatelů, mezi nimiž kromě AWS vyniká Google Cloud Computing, Microsoft Azure a Rackspace. Uvažováni byli pouze poskytovatelé IaaS (Infrastructure as a Service) řešení, jelikož byla požadována co nejširší možná kontrola nad systémem.

Porovnání poskytovatelů je poměrně komplexní úloha a dosavadní studie (například [14] a [15]) neurčují žádného jasného leadera. Portfolio nabízených služeb je vždy poměrně široké a zahrnuje množství různých typů výpočetních serverů, SQL i NoSQL databáze, CDN (Content Delivery Network) pro poskytnutí obsahu z ideálního umístění, různá datová úložiště, load balancery a v neposlední řadě monitoring všech těchto služeb. Měření výkonu závisí na mnoha faktorech, jejichž velká část je uživateli nedostupná a výsledky se i při opakování testu na stejné konfiguraci mohou výrazně lišit například v závislosti na denní době. Kromě výkonu je také nutné volit podle přesných specifikací požadovaných služeb a jejich ceny. Ve výsledku je ideální vyzkoušet si svou aplikaci na několika řešeních a potom se rozhodnout. Jelikož u Amazonu nebyla shledána absence služeb, které by byly pro projekt potřeba, bylo rozhodnuto zvolit toto řešení vzhledem k finanční stránce a tuto práci použít pro jeho otestování.

Nyní k samotným službám. Práce byla vyvíjena v rámci AWS Free Tier [16], které dovolují uživateli roční využívání většiny AWS služeb zdarma. Na každou službu existuje datové či časové omezení, které je možné vyčerpat v jednom měsíci. U EC2 nebo RDS je navíc omezení na typ a model instance, které je možné použít, typicky jsou ty nejnižší. Pro seznámení se s cloudovými technologiemi je ovšem Free Tier ideální, je pouze potřeba dávat si pozor na úroveň využití služeb, aby nedošlo k překročení limitů kdy se služby automaticky zpoplatní. Pro tuto práci bude tedy využita následující sestava technologií:

- 2x EC2 t2.micro instance (1 vCPU, 1GB RAM, 8GB SSD), instalován Jetty server verze 9.2.5, Java 1.7, Amazon Linux 64-bit
- Elastic Load Balancer, HTTP režim se Sticky Session
- ElastiCache s Redis serverem verze 2.8.6, t2.micro instance (0.5GB RAM)
- Amazon RDS s enginem MySQL 5.6.21, t2.micro instance, 5GB SSD

Výkon této konfigurace se ukáže v testech reálné zátěže. V cloudovém prostředí obecně je těžké odhadnout výkon poskytnutého virtuálního serveru. Pro pozdější implementaci produkčního řešení poslouží tento systém jako dobrá reference.

4 Architektura a implementace

Tato kapitola se věnuje návrhu a realizaci celého systému z různých úhlů pohledu a v různém měřítku. První část bude orientovaná na pohled uživatele. Ten je ideálním vstupním bodem, jelikož popisuje jak aplikace vypadá a jak se chová navenek. Typicky jsou to požadavky, specifikující jaké akce je možné provést na které stránce, za jakých podmínek, jak se lze mezi různými částmi aplikace pohybovat a podobně. Tento pohled poskytne povědomí o tom, co aplikace umí a co si představit pod různými pojmy jako je založení hry nebo souboj v aréně. Také zde budou přestaveny tři komponenty aplikace: Welcome page, Matchmaker a Aréna.

Druhá a třetí část se zaměří na celkový pohled na systém, tedy nejen na samotnou aplikaci, ale i na přilehlé cloudové služby. Zapojování těchto služeb probíhalo postupně tak, jak byly plněny požadavky práce. Stejný posun bude ukázán v několika krocích návrhu celkové architektury, od single-server řešení až po optimalizovaný cloudový systém. Tyto části ukáží především komunikaci jednotlivých služeb a optimalizaci jejich využití.

Čtvrtá část již odhlédne od architektury a bude se krátce věnovat standardní HTTP komunikaci frontendu a backendu. Současně s ní budou detailněji představeny dvě hlavní komponenty Spring frameworku, Controller a Service.

Poslední část se pak bude věnovat jednotlivým komponentám aplikace a to jak z klientské tak ze serverové strany. Bude ukázána jejich grafická stránka a především implementační detaily, které jsou pro komponentu důležité nebo vyžadovaly nestandardní řešení. Tato část především by měla pomoci budoucí implementaci podobného systému.

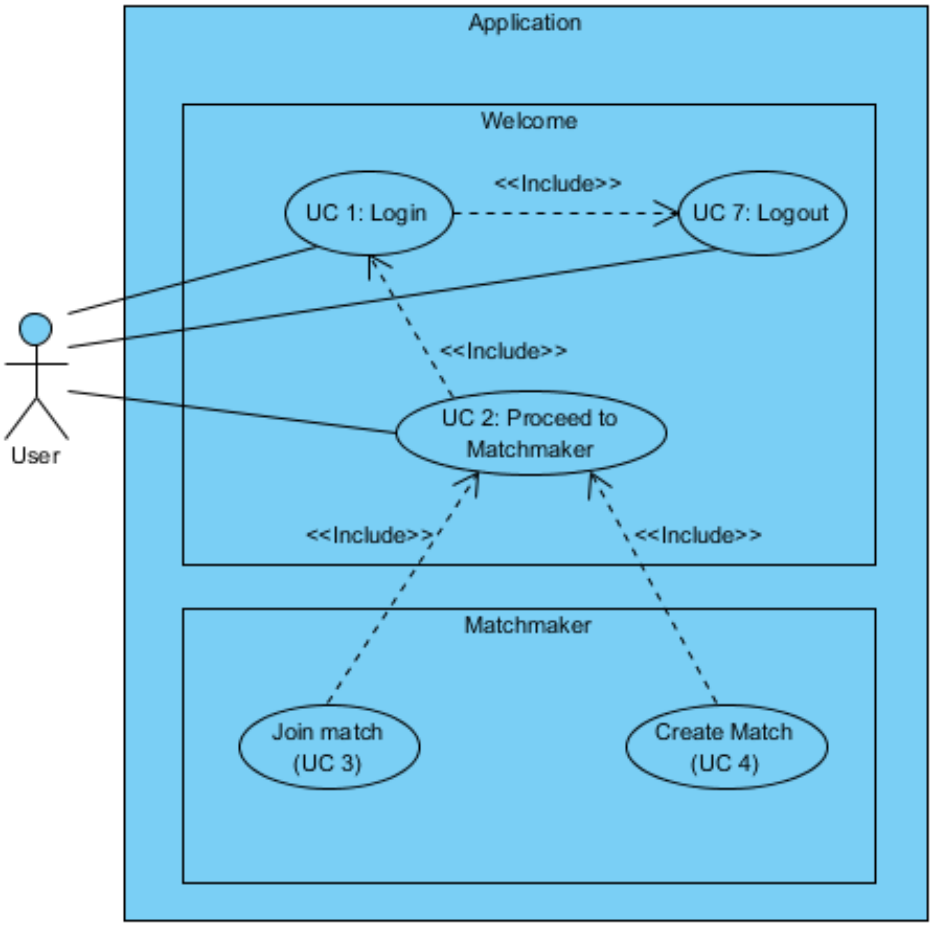
4.1 Aplikace z pohledu uživatele

Jak bylo řečeno, tento pohled popisuje jak se aplikace jeví navenek. Procesy probíhající uvnitř, jako například komunikace mezi službami, nebo způsob zpracování uživatelské akce, z tohoto pohledu nejsou vidět a nezatěžují nás. Bude se jim věnovat samostatná část kapitoly.

Použijeme zde UseCase diagramy, každý věnovaný jedné komponentě, které zobrazí i vztahy mezi jednotlivými uživatelskými akcemi, aby bylo jasné v jakém pořadí mohou být provedeny. V popisech diagramů jsou pak zmíněny i některé akce aplikace pro zobrazení logiky jejího chování.

4.1.1 Welcome page

Aplikace musí disponovat úvodní stránkou, kde uživatel vstoupí a identifikuje se. Identifikace koresponduje s požadavkem práce, aby aplikaci mohli používat pouze přihlášení uživatelé. Tato úvodní stránka simuluje reálnou aplikaci Hravě, kterou bude později v produkci nahrazena, a přechod do multiplayer arény.



Obrázek 15 UseCase diagram pro úvodní stránku, ukazuje přihlášení do aplikace a přechod do modulu matchmakeru.

UC 1: Login

Popis

Uživatel se přihlásí do aplikace.

Aktéři

Uživatel (User), Aplikace - komponenta Welcome

Podmínky pro spuštění

Uživatel se nachází na stránce welcome a není přihlášen.

Základní tok

1. Uživatel zadá své uživatelské jméno a odešle jej Aplikaci
2. Aplikace ověří uživatelské jméno a přihlásí uživatele

Chybný tok

2.1 Uživatel zadal neplatné uživatelské jméno a systém jej nepřihlásí

Podmínky pro dokončení

Uživatel je přihlášen v aplikaci

UC 2: Proceed to Matchmaker

Popis

Uživatel přejde na stránku matchmakeru.

Aktéři

Uživatel (User), Aplikace - komponenta Welcome

Podmínky pro spuštění

Uživatel se nachází na stránce welcome a je přihlášen.

Základní tok

1. Uživatel klikne na tlačítko pro přesun do Matchmakeru
2. Aplikace přesune uživatele na stránku Matchmakeru

Podmínky pro dokončení

Uživatel se nachází na stránce Matchmakeru

UC 7: Logout

Popis

Uživatel se odhlásí z aplikace.

Aktéři

Uživatel (User), Aplikace - komponenta Welcome

Podmínky pro spuštění

Uživatel se nachází na stránce welcome a je přihlášen.

Základní tok

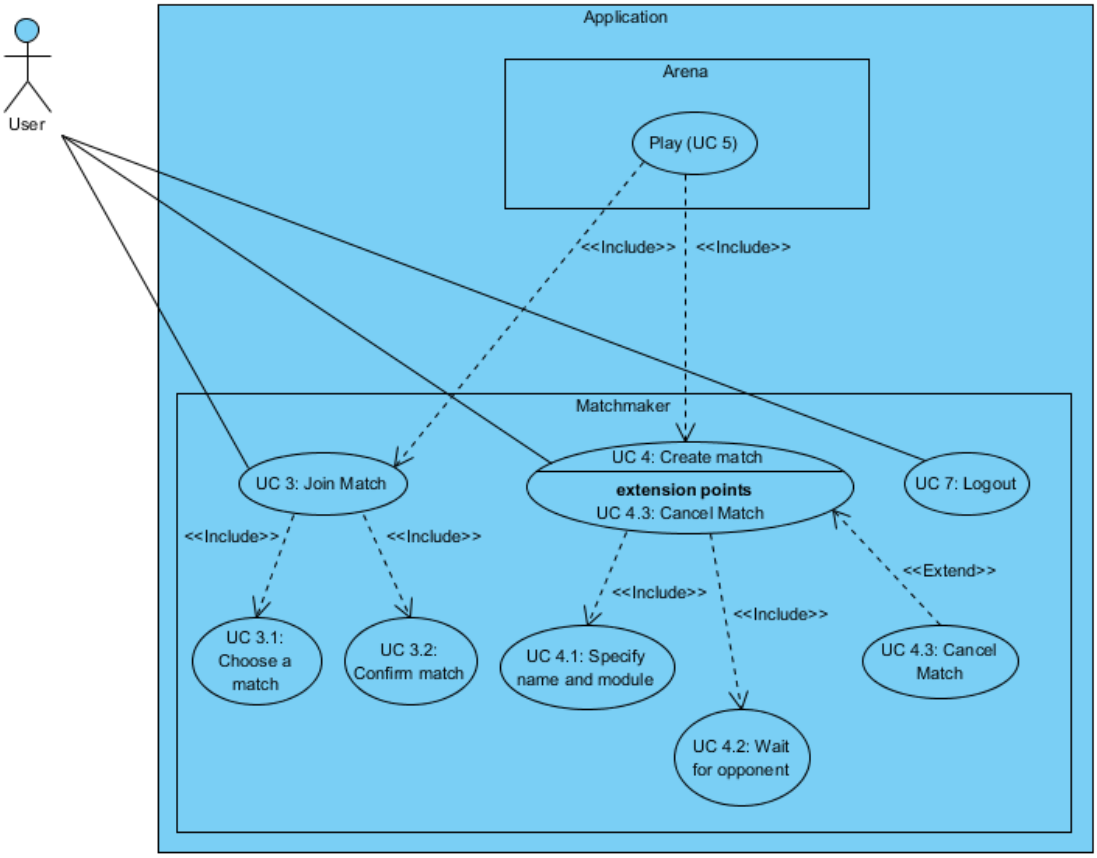
1. Uživatel klikne na tlačítko *Logout*
2. Aplikace odhlásí uživatele

Podmínky pro dokončení

Uživatel se nachází na stránce Welcome a je odhlášen.

4.1.2 Matchmaker

Je párovací modul, kde hráči zakládají nové hry a hledají své oponenty do arény. Po příchodu sem má hráč volbu. Buď založí novou hru, která se zobrazí na seznamu všech dostupných her, a čeká na připojení soupeře. Nebo si vybere ze seznamu již založených her a do nějaké se připojí jako soupeř. Jakmile se k založené hře připojí druhý hráč, je pro tuto dvojici vytvořena aréna a oba jsou do ní přesměrováni.



Obrázek 16 UseCase diagram pro Matchmaker, ukazuje zakládání hry a připojení se k ní.

UC 3: Join Match

Popis

Uživatel si vybere hru a připojí se k ní.

Aktéři

Uživatel (User), Aplikace - komponenta Matchmaker

Podmínky pro spuštění

Uživatel je přihlášen, je se na stránce Matchmaker, není v procesu zakládání hry.

Základní tok

1. Uživatel si zvolí hru z globálního seznamu her (UC 3.1) a klikne na tlačítko *Join*
2. Aplikace zobrazí detaily vybrané hry
3. Uživatel potvrdí připojení do zvolené hry (UC 3.2)
4. Aplikace oba hráče přesměruje do arény

Alternativní tok
3.1 Uživatel zruší výběr zvolené hry
4.1 Aplikace zavře detail hry

Podmínky pro dokončení
Uživatel se nachází v aréně vybrané hry

UC 4: Create Match

Popis
Uživatel založí hru, do které se připojí oponent, nebo je hra zrušena.

Aktéři
Uživatel (User), Aplikace - komponenta Matchmaker

Podmínky pro spuštění
Uživatel je přihlášen na stránce Matchmaker, není v procesu připojování se do hry.

Základní tok
1. Uživatel klikne na tlačítko *Create Game*
2. Aplikace zobrazí okno pro vytvoření hry
3. Uživatel zadá jméno hry, modul a klikne na tlačítko *Create* (UC 4.1)
4. Aplikace zkontroluje zadané údaje, založí hru a otevře okno pro čekání na soupeře
5. Uživatel čeká na připojení soupeře (UC 4.2)
6. Soupeř se připojí a Aplikace oba hráče přesměruje do arény

Alternativní tok 1
3.1 Uživatel zruší vytváření hry kliknutím na tlačítko *Cancel* (UC 4.3)
4.1 Aplikace zavře okno pro vytvoření hry

Alternativní tok 2
5.1 Uživatel zruší vytvořenou hru kliknutím na tlačítko *Cancel* (UC 4.3)
4.1 Aplikace smaže vytvořenou hru a zavře okno pro čekání na soupeře

Chybový tok
4.1 Uživatel zadal nevalidní údaje, systém zobrazí chybu a nevytvoří hru

Podmínky pro dokončení
Uživatel se nachází v aréně vytvořené hry

UC 7: Logout

Popis
Uživatel se odhlásí z aplikace.

Aktéři
Uživatel (User), Aplikace - komponenta Matchmaker

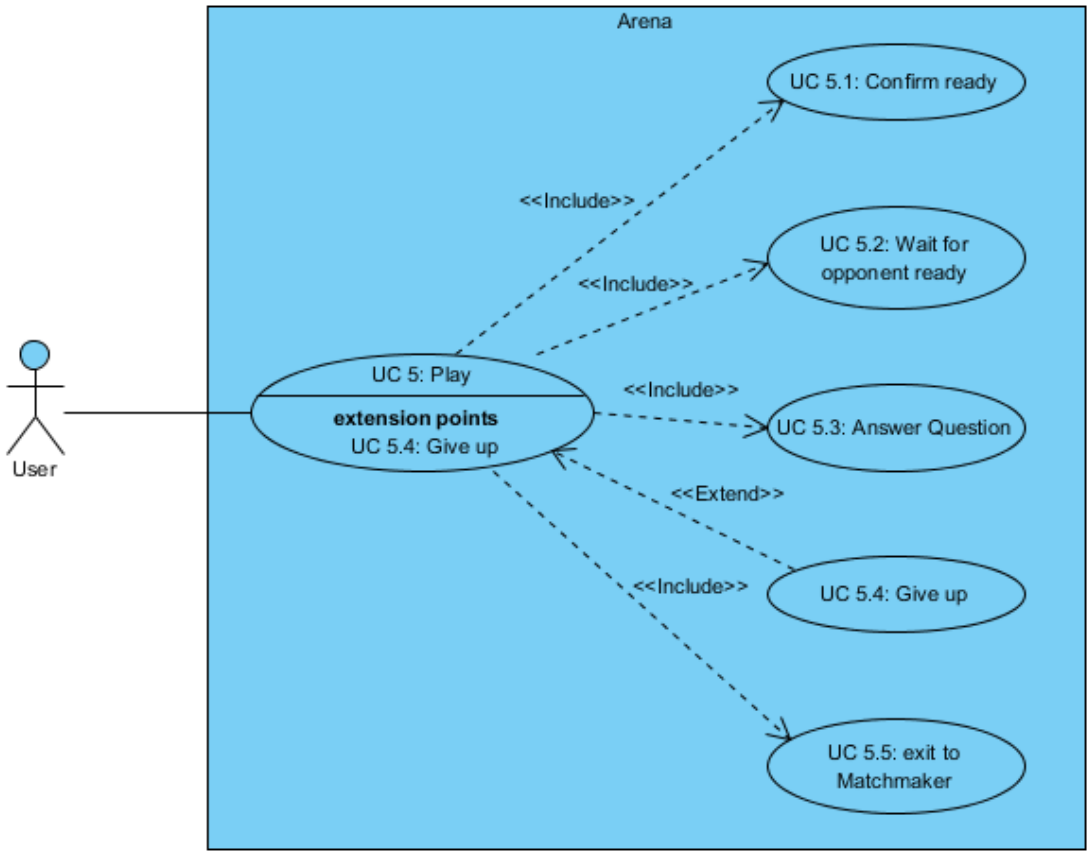
Podmínky pro spuštění
Uživatel je přihlášen, nachází se na stránce Matchmaker a není v procesu zakládání hry ani připojování se do hry.

Základní tok
1. Uživatel klikne na tlačítko *Logout*
2. Aplikace jej odhlásí a přesměruje na stránku Welcome

Podmínky pro dokončení
Uživatel je odhlášen a nachází se na stránce Welcome

4.1.3 Aréna

Hlavní modul, který zprostředkovává vědomostní souboj mezi oběma soupeři. Oba příchozí hráči musí nejprve potvrdit, že jsou připraveni a počkat až totéž udělá jejich oponent. Ihned po potvrzení se oběma ukáže první otázka a začíná souboj. Ten končí jakmile zdraví jednoho hráče klesne na nulu. Poté musí hráči arénu opustit a vrací se do Matchmakeru. Kdykoli během hry může jeden hráč hru vzdát, což určí jeho soupeře jako vítěze hry a oba se opět vrací do Matchmakeru.



Obrázek 17 UseCase diagram pro Arénu, ukazuje průběh hry a odchod z arény.

UC 5: Play

Popis

Uživatel hraje proti soupeři v aréně.

Aktéři

Uživatel (User), Aplikace - komponenta Aréna

Podmínky pro spuštění

Uživatel je přihlášen, nachází se na stránce Arény a dosud nezačal hru.

Základní tok

1. Uživatel je připraven ke hře a klikne na tlačítko *Start Fight* (UC 5.1)
2. Aplikace zaregistruje hráče jako připraveného
3. Uživatel čeká na oponenta až bude připraven (UC 5.2)
4. Oponent potvrdí že je připraven

5. Aplikace registruje oba soupeře jako připravené a zobrazí první otázku
6. Uživatel odpoví na otázku (UC 5.3)
7. Aplikace vyhodnotí otázku, pokud je odpověď správná, ubere soupeři zdraví, pokud je odpověď špatná, ubere zdraví uživateli
8. Aplikace zkontroluje, zda zdraví některého ze soupeřů nekleslo na nulu
9. Pokud je zdraví obou soupeřů nenulové, zobrazí uživateli novou otázku a opakuje se bod 6
10. Pokud zdraví jednoho uživatele kleslo na nulu, hra jej určí jako poraženého, jeho protivníka jako vítěze a zobrazí tlačítko k opuštění arény
11. Uživatel klikne na tlačítko *Leave Arena* (UC 5.5)
12. Aplikace jej přesměruje na stránku Matchmakeru

Alternativní tok

6.1 Uživatel vzdá hru kliknutím na tlačítko *Give up* (UC 5.4)
7. Aplikace jej určí jako poraženého, jeho soupeře jako vítěze a zobrazí tlačítko k opuštění arény.
8. Pokračuje se bodem 11 základního toku.

Podmínky pro dokončení

Uživatel se nachází na stránce Matchmakeru

Toto je kompletní funkcionalita z pohledu uživatele, kterou nabídne implementace této práce. V produkční verzi pro projekt Hravě je plánováno rozšiřování funkcionality například o možnost filtrování her v Matchmakeru, opakování hry se stejným soupeřem nebo zapisování bodů za vyhrané zápasy do databázových záznamu uživatele. To budou ovšem pouze alternace funkcionalit již implementovaných. Proto jsou v této verzi, představující koncept finální arény, vynechány.

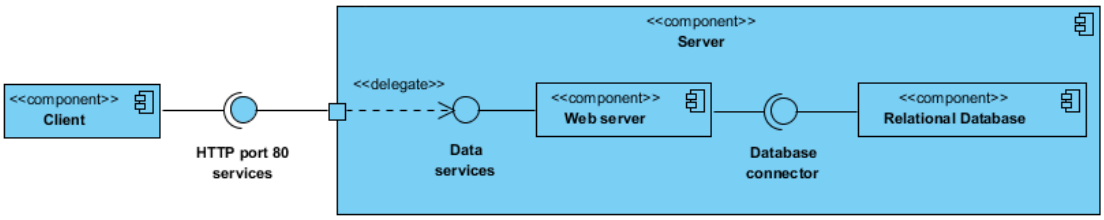
Tři představené komponenty - Welcome, Matchmaker a Aréna - představují kompletní aplikaci, která se nachází na jedné EC2 instanci. Pro její běh jsou ovšem potřeba další služby, které tvoří celý cloudový systém. V další sekci bude tento systém představen postupně tak, jak vznikal při přechodu z běžného single server řešení na Amazon Web Services.

4.2 Návrh architektury

Architektura vznikala postupně z klasické architektury webové aplikace o jednom serveru s relační databázi, ke kterému přistupuje klient pomocí webového prohlížeče. Vývoj byl rozdělen do čtyř částí ohraničených implementací real-time komunikace, přesunem na cloud a nakonec jeho optimalizací.

4.2.1 Single server

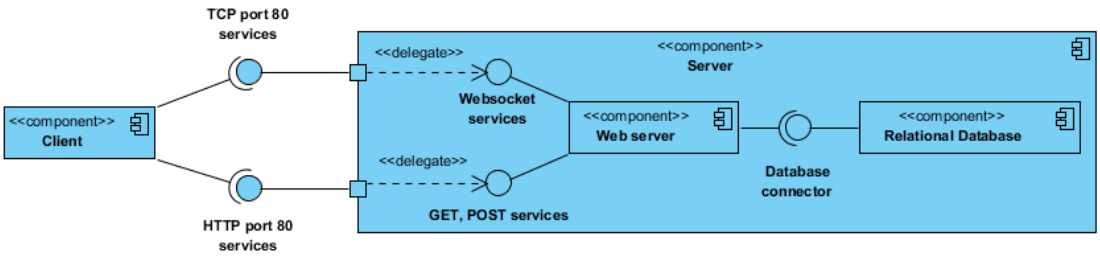
Diagram komponent na obr. 18 ukazuje, že web server i databáze se nachází na jednom stroji a komunikují spolu pomocí interního rozhraní. Klient k serveru přistupuje přes port 80 a využívá jeho služby pomocí HTTP request/response. Sekce 3.1 sice ukazuje, že i pomocí klasických HTTP služeb je možné dosáhnout real-time komunikace, ukazuje ale také mnoho nevýhod takového spojení a navrhuje využití pokročilejší technologie v podobě websocketů.



Obrázek 18 Diagram komponent pro jednoduchou architekturu webové aplikace

4.2.2 Real-time single server

Obr. 19 ukazuje diagram, kde klient nově komunikuje se serverem dvěma způsoby. HTTP protokol je používán pro požadavky typu GET (statické soubory, žádost o upgrade na WS) a POST (datové requesty na serverové servisy). TCP protokol je pak používán výhradně pro komunikaci pomocí websocketů. Web server také poskytuje jiné API pro každý typ servis.



Obrázek 19 Diagram komponent pro architekturu webové aplikace využívající websockety pro real-time komunikaci

Tato architektura zajišťuje real-time komunikaci, pro cloud je ovšem nevhodná. Zejména neumožňuje škálování tak, aby aplikace byla přístupná přes jednotnou adresu. Na každý server by se muselo přistupovat přes jeho vlastní adresu, což je velice nepraktické. Také by nebylo možné sdílet data o uživatelích mezi jednotlivými servery.

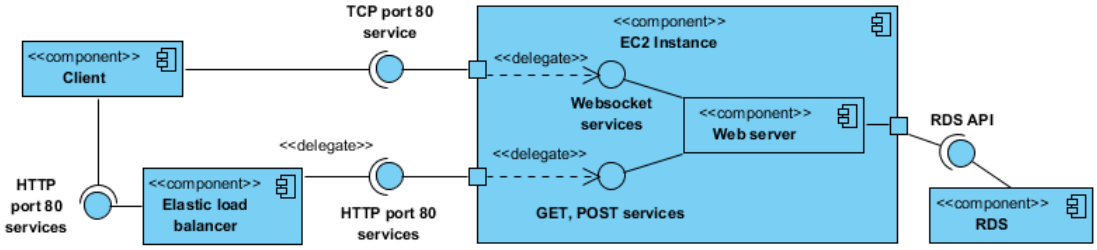
4.2.3 Real-time cloud

Architektura na obr. 20 je již pro cloud vhodnější a obsahuje dvě podstatné změny:

- Databázi na samostatném serveru
- Zavedení load balanceru

Jelikož se nyní pohybujeme v oblasti cloudových služeb Amazonu, databáze je představována službou RDS, ke které se webový server připojuje pomocí jejího API. V závislosti na zátěži databáze, může RDS obsahovat jednu nebo více instancí databázových serverů. Z hlediska naší aplikace je však podstatné pouze to, že nám RDS zprostředkovává přístup k DB pomocí jednoho přípojného bodu a ten je společný pro všechny EC2 instance provozující naši aplikaci. Tím je zaručeno sdílení dat mezi všemi instancemi, nehledě na jejich proměnlivý počet.

Zavedení Elastic Load Balanceru jednak zaručí jednotnou adresu pro všechny instance aplikace a jednak se postará o rovnoměrnou distribuci požadavků od klientů. Jeho zavedení ovšem přináší do systému komplikaci zmíněnou v sekci 3.3.2. ELB může operovat v jednom ze dvou režimů: HTTP nebo TCP.



Obrázek 20 Diagram komponent pro cloudovou architekturu webové aplikace využívající websockety pro real-time komunikaci

Pro používání aplikace se klient musí přihlásit a systém si jeho přihlášení zapamatuje. Minimálně tato informace je tedy o klientovi uchovávána a je vhodné, aby byla dostupná po celou dobu jeho přihlášení. Pro tuto funkcionalitu je nutné, aby ELB operoval v režimu HTTP se zapnutou technologií Sticky Sessions, nebo aby se klientská session ukládala v databázi.

Pokud je session uložena v DB a klient pošle požadavek na instanci, kde ještě není přihlášen (jeho session není v paměti instance), aplikace nejprve zkusí vyhledat klientskou session v DB a teprve v případě neúspěchu bude po klientovi vyžadovat přihlášení. Pokud si ovšem aplikace ukládá o uživateli více dat než jen jeho přihlášení musí být session v DB updatována pokaždé, když se klientská data změní. Vzniká tak nutnost poměrně časté komunikace s databází k udržení aktuální session každého uživatele.

Po několika požadavcích jednoho klienta je jeho session nahraná a udržována v paměti každé instance, což také n-násobně zvyšuje nároky na paměť serveru, kde n je počet běžících instancí. Jedinou výhodu tohoto způsobu implementace je velice přesný load balancing, který probíhá při každém requestu. V případě použití Sticky Sessions probíhá load balancing pouze při prvním requestu uživatele a pak znovu až v okamžiku kdy je otevřen a zavřen prohlížeč, nebo vyprší klientská session.

Pro tuto aplikaci je mnohem důležitější optimalizace paměťových nároků a objemu komunikace,budou proto použity Sticky sessions a tedy ELB poběží v režimu HTTP. Díky tomu je ale znemožněno, aby přes load balancer šla i TCP komunikace použitá pro websockety. ELB v tomto režimu s celými TCP packety neumí správně pracovat a pokus o navázání komunikace pomocí WS tedy selže.

Řešením je duální komunikace. Jak je vidět z obr. 20 nebo 21, klient ke komunikaci se systémem využívá dva kanály. Standardní HTTP požadavky jdou přes ELB, WS komunikace probíhá přímo s instancí aplikace. Load balancing není WS komunikací nijak ovlivněn. Klient nejprve navazuje komunikaci standardním HTTP požadavkem a teprve poté proběhne spojení pomocí WS s konkrétní instancí přiřazenou ELB. Z pohledu klienta se tato dualita vizuálně neprojeví, jelikož WS spojení běží na pozadí aplikace a URL adresa zůstává stále stejná, mířící na ELB. Poslední důležitou věcí je nutnost zapnutí podpory CORS pro WS spojení.

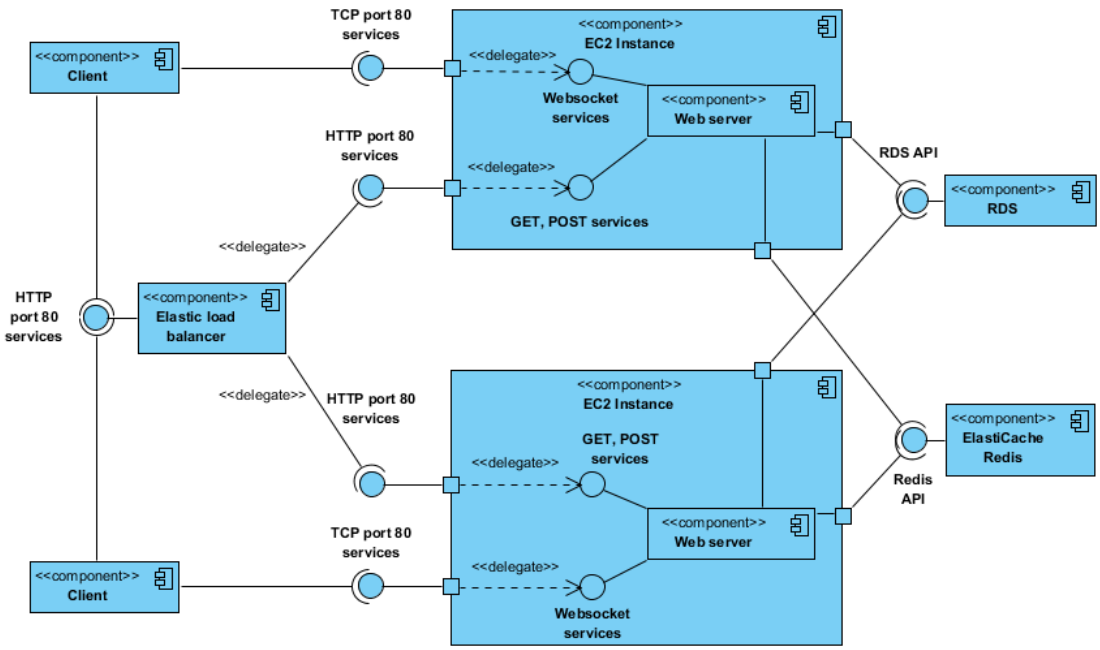
Tento, již dvakrát zmiňovaný mechanismus, je nutný v případě, že se klient nachází na nějaké webové adrese a přitom posílá požadavek na jinou. V tomto případě je klient připojen na URL load balanceru, ale úvodní HTTP požadavek na upgrade spojení pro websockety posílá přímo instanci EC2, která má vlastní URL. Hlavička HTTP požadavku vždy obsahuje obě tyto adresy a tento křížový požadavek je servery iniciálně zakázán z důvodu bezpečnosti. Je ovšem možné jej povolit jak v konfiguraci serveru, tak v běhu aplikace při zpracování požadavku. Atmosphere řeší povolení CORS pomocí parametru v iniciálním requestu klienta. Tento mechanismus bude ovšem potřeba ještě v Matchmakeru při přesunu klienta mezi instancemi. V implementačních detailech této komponenty tedy bude zobrazen i čistě programové zpracování CORS požadavku.

4.2.4 Real-time cloud optimalizace

Takto navržená architektura je funkční, ovšem pro sdílení dat mezi instancemi obsahuje pouze relační databázi. Ta je vhodná pro data, jejichž charakter je dlouhodobý. Například informace o uživateli, případně by zde mohly být uloženy klientské sessions (záleží na frekvenci jejich změn). Mezi instancemi je ovšem potřeba sdílet i data jiného charakteru.

V okamžiku, kdy nějaký uživatel založí hru, je vytvořen záznam o této hře. Iniciálně je ovšem viditelný pouze pro uživatele přihlášené na stejné instanci jako zakládající uživatel. Je nutné jej propagovat na ostatní aktivní instance a bylo by tedy možné jej uložit do relační databáze, odkud by si jej ostatní instance přečetly. Tento záznam ovšem existuje pouze od okamžiku založení hry do připojení oponenta. Jakmile oponent potvrdí vstup do hry, je vytvořena instance arény pro tyto dva hráče, informace ze záznamu hry jsou jí předány a samotný záznam je smazán. Jeho existence tedy trvá typicky několik desítek vteřin. Pokud bude někde uložen seznam všech založených her, v závislosti na počtu uživatelů na každé instanci se takový seznam může měnit několikrát za vteřinu. Na uchovávání takových dat jsou relační databáze nevhodné.

Naopak velice vhodným systémem je buď messaging nebo NoSQL databáze. Ve fázi návrhů architektury byly zpracovány obě možnosti, obě realizované pomocí zmiňované služby ElastiCache a Redis serveru. Detaily jejich architektury jsou podrobněji popsány v následující sekci a finální rozhodnutí o použití jedné z nich bude učiněno až na základě testů.



Obrázek 21 Diagram komponent pro kompletní cloudovou architekturu webové aplikace s real-time komunikací pomocí websocketů a cachovacím systémem Redis serveru. Představuje dva klienty připojené ke dvěma různým instancím aplikace.

Obr. 21 ukazuje zapojení ElastiCache do celkové architektury. Stejně jako RDS, EC představuje službu běžící na samotné instanci, se kterou aplikační instance komunikují pomocí jejího API. Vnitřní uspořádání není z tohoto pohledu podstatné, důležitý je jeden přístupový bod pro všechny instance. Tento obrázek také ukazuje připojení více klientů, z nichž každý je pomocí ELB směrován na svou instanci a k té je také připojen WS spojením.

Celková architektura systému není ve výsledku příliš složitá. Hlavním rozdílem oproti běžnému serveru je přesun určitých částí aplikace na samostatné stroje a jejich použití v podobě externích služeb. Právě tyto transformace však bývají zdrojem nutných změn v chování aplikace.

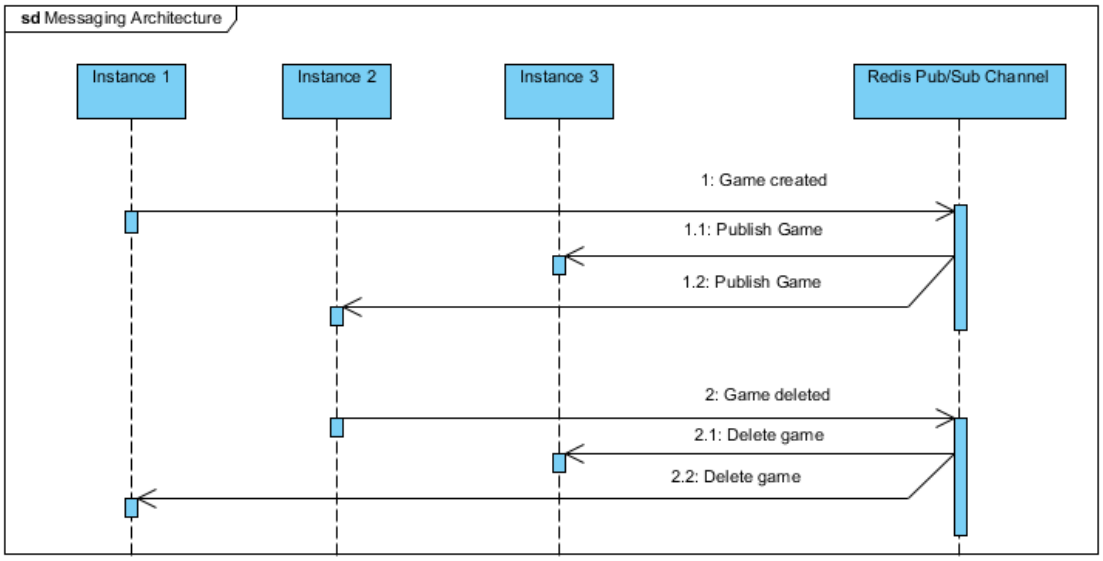
Přesun databáze na vlastní server je poměrně běžnou praxí i mimo cloudové prostředí. Load balancing klientských požadavků pouze kopíruje systém rozdělování těchto požadavků jednotlivým vláknům samotné aplikace. Jeho transformace do samostatné služby se však také neobešla bez komplikací v podobě nutnosti duální komunikace klienta. Nahrazení sdílené interní paměti serveru externím messaging systémem případně externí sdílenou pamětí, je již podstatná změna. Celkově zabrala nejvíce času a jejím detailům se věnuje následující část.

4.3 Inter-instanční komunikace

Byly zmíněny dvě architektury pro sdílení her mezi instancemi pomocí Redis serveru. Messaging a NoSQL databáze. Existují i další způsoby sdílení dat, například Java Messaging Services, Amazon Simple Notification Service nebo Amazon Simple Queue Service. Možnosti poskytnuté Redisem byly ovšem dostačující a v teoretickém porovnání rovnocenné zmíněným službám. Nebyl tak důvod rozšiřovat stávající systém o další komponenty.

4.3.1 Messaging

Tímto termínem je zde myšleno vytvoření globálního kanálu, ke kterému jsou připojeny všechny instance. V okamžiku události (založení nebo smazání hry) na jedné instanci je zpráva o ní publikována do globálního kanálu a všechny připojené instance ji okamžitě obdrží. Tento systém se nazývá publish/subscribe a Redis jej nabízí pod zkratkou pub/sub.



Obrázek 22 Sekvenční diagram komunikace instancí a Redis serveru při použití pub/sub kanálu.

Sekvenční diagram na obr. 22 ukazuje, že Redis vždy okamžitě sdílí zprávu o události. V této architektuře si každá instance udržuje kompletní seznam všech her a updatuje jej jednak podle událostí nastalých na ní a jednak podle zpráv od ostatních instancí. Sdílení dat mezi instancemi je tedy řešeno pomocí zpráv a Redis je v této architektuře prostředníkem, starajícím se o jejich transport.

Kritické pro tuto architekturu je přidání nové instance. Jakmile se instance připojí, začne přijímat zprávy od ostatních instancí a sama vytvářet nové hry. Nemá však ponětí o hrách, které byly vytvořeny před jejím zapojením do systému.

Tuto mezeru je možné vyplnit tak, že nová instance pošle zvláštní příkaz, kterým požádá ostatní (případně jednu konkrétní) instance, aby jí poslaly kompletní seznam dosavadních her. To vyžaduje ovšem dobrou synchronizaci, aby obdržený seznam byl aktuální vzhledem k ostatním informacím o vytvořených a smazaných hrách. Tedy aby nová instance neobdržela seznam her a následně zprávu informující o smazání hry, která

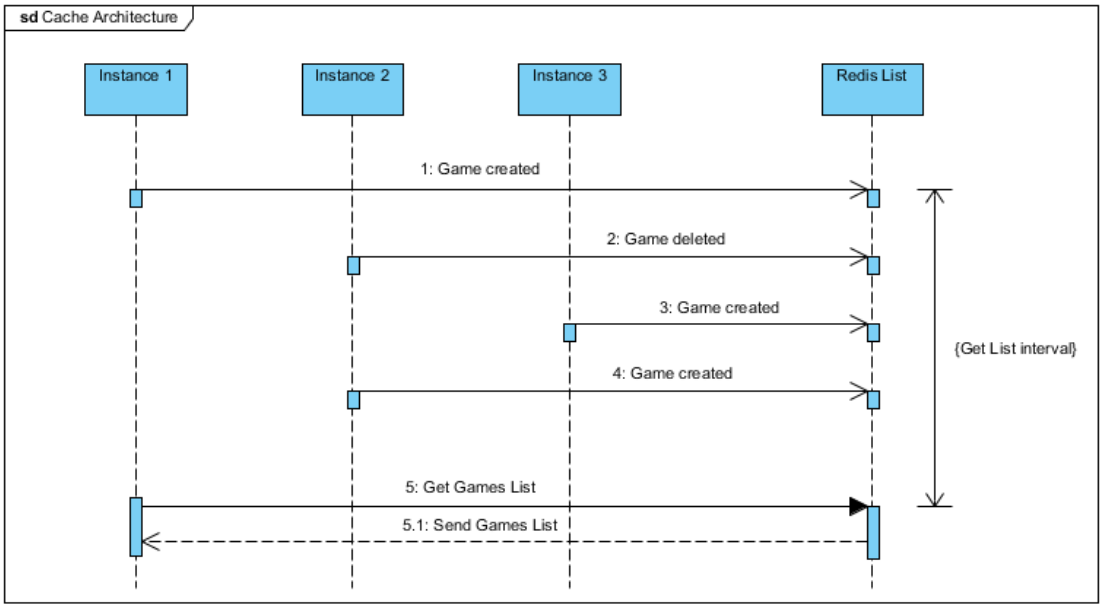
ovšem na seznamu není, jelikož byla založena v době, kdy nová instance zpracovávala doručený seznam. Zasílaný seznam her může být také poměrně datově obsáhlý.

Druhou možností po připojení nové instance je ignorovat hry založené před jejím zapojením. Jak bylo řečeno, záznam o založené hře typicky existuje pouze několik desítek vteřin. Proto přibližně první minutu nebude mít instance kompletní seznam všech dostupných her. Po uplynutí této doby by již měly být všechny hry založené před jejím zapojením pryč ze seznamu.

K přidání instance bude navíc docházet až při zatížení ostatních instancí určitým vyšším počtem hráčů. Na původních instancích tedy bude existovat dostatek hráčů na to, aby pokryli jako oponenti hry založené před přidáním nové instance. Tento přístup je jednodušší a především komunikačně méně náročný, bude proto preferován.

4.3.2 NoSQL databáze

Redis je primárně NoSQL databáze a pub/sub systém je pouze jeho doplňkem. V této architektuře plní tedy svou primární roli, jako rychlá sdílená paměť. V Redisu je vytvořen seznam založených her jako struktura set (neseřazený key-value seznam), který je průběžně aktualizován všemi instancemi. Instance ovšem průběžně pouze posílají data a server jim nijak neodpovídá ani neinformuje ostatní instance, pouze updatuje svůj set. V určitém pevně daném časovém intervalu si každá instance vyžádá od Redisu tento set, který k danému okamžiku představuje kompletní aktuální seznam her. Tím poté přepíše svůj dosavadní seznam her, který si udržuje ve své vlastní paměti. Komunikaci v tomto typu architektury představuje sekvenční diagram na obr. 23.



Obrázek 23 Sekvenční diagram komunikace instancí a Redis serveru při použití sdíleného seznamu her

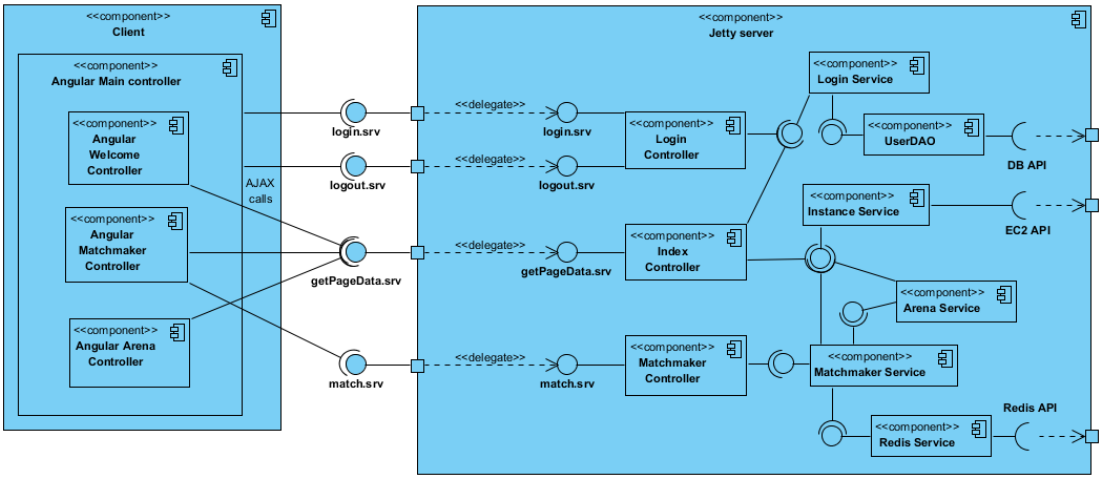
Výhodou tohoto řešení je relativní zmenšení objemu komunikace mezi instancemi a Redisem. Také nemusíme řešit připojení nové instance a prodlevu při její postupné aktualizaci seznamu her. Po připojení si nová instance zažádá Redis o kompletní seznam her a v okamžiku jeho zpracování si je s ostatními instancemi rovnocenná.

Nevýhodou je naopak redundance posílaných dat. Čím častěji si bude instance žádat o kompletní seznam her, tím menší bude procento dat, které se v něm liší oproti původnímu seznamu. Proto je zmenšení objemu komunikace oproti messagingu pouze relativní - záleží jak často budou instance požadovat data od Redisu. Poslední nevýhodou je možnost překročení vyčerpání paměti Redisu při těchto příkazech. Ten totiž při dotazu na kompletní set vystaví jeho kopii do bufferu příslušného spojení, odkud je set následně stažen volající instancí. Proces stažení však nějakou dobu trvá a pokud bude set velký a zažádá o něj několik instancí najednou, může se stát, že vytvořené kopie se nevejdou do paměti a server nebude schopen pracovat.

Kde jsou limity obou architektur a která je efektivnější vzhledem k objemu komunikace a zátěži Redis serveru se ukáže při jejich testování. Další sekce se již bude věnovat uspořádání samotné aplikace na jedné instanci a její komunikaci s klientem.

4.4 HTTP komunikace

Způsob HTTP komunikace frontendu a backendu byl nastíněn již v části 2.3 popisující architekturu původní aplikace Hravě. Tento způsob je společný pro všechny tři hlavní komponenty aplikace a bylo rozhodnuto zachovat pro jeho výhody zahrnující malý datový objem komunikace a nezávislost technologií použitých k řešení obou stran. Backend vystaví konkrétní přípojné body pro jednotlivé komponenty klienta a ten od nich pak pomocí AJAXových volání získává data v univerzálním formátu JSON. Kromě GET requestů pro vyžádání html, css a js souborů a websocket spojení, je toto jediný způsob komunikace backendu a frontendu.



Obrázek 24 Diagram komponent znázorňující komunikaci frontendu a backendu pomocí Servis vystavených backendem a AJAX volání z frontendu

Diagram komponent na obr. 24 ukazuje, že backend obsahuje tři hlavní controllery, což jsou objekty frameworku Spring. Tyto controllery vystavují jednotlivé přípojné body, které označí určitým názvem. Každý název má koncovku *srv* značící, že se jedná o běžnou servisu. Používá se například i koncovka *do* pro odlišení servis, které může frontend volat pouze pokud je jeho uživatel přihlášen. Takové servisy se používají typicky pro Admin funkce. Koncovky jsou dány konvencí projektu Hravě, nejsou tedy funkční součástí Springu. Definice Springového controlleru a jeho přípojného bodu vypadá následovně:

```
@Controller
public class MatchmakerController extends BaseController {

   @Autowired
   protected MatchmakerService matchmakerServicet;

   @RequestMapping(value = "/match.srv", method = RequestMethod.POST)
   public void match(
      final @RequestParam(value = "action", required = true) String action,
      final @RequestParam(value = "name", required = false) String name,
      final @RequestParam(value = "moduleId", required = false) Long moduleId,
      final @RequestParam(value = "gameId", required = false) Long gameId,
      final @RequestParam(value = "uuid", required = false) String uuid,
      HttpServletRequest request, HttpServletResponse response,
      final HttpSession session) { ... }
}
```

Anotace *@Controller* u definice třídy říká Springu o jaký typ jeho objektu se jedná. Anotace *@RequestMapping* u definice funkce *match* pak určuje jméno servisy a typ požadavku, kterým je možno servisu volat. Tento controller tedy vytváří servisu *match.srv*, přístupnou na adrese *http://address_of_this_app/match.srv*, kterou je možno zavolat metodou *POST*. Tato servisa má několik parametrů se kterými je možno ji volat, pouze parametr *action* je povinný, ostatní parametry nemusí být při jejím volání uvedeny. Uvnitř metody *match* je pak dle parametru *action* provedena příslušná akce její výsledek je odeslán zpět volajícímu klientovi jako data.

Anotace *@Autowired* vytváří takzvanou Dependency Injection a zpřístupní tomuto controlleru objekt *MatchmakerService*. Zpřístupněný objekt je dalším Springovým typem, jak název napovídá, jedná se o servis. Ten je definován následovně:

```
@Service
public class MatchmakerService {

   @Autowired
   protected InstanceService instanceService;
   @Autowired
   protected RedisService redisService;
   @Autowired
   protected AtmosphereService atmosphereService;
    ...
}
```

Servisy jsou typicky třídy starající se o vnitřní aplikační logiku. Z obr. 24 je vidět, že každý controller využívá několik různých objektů typu servis podle toho, jaká akce je po něm vyžadována. Objekt servis může také využívat ostatní servisy, jak je vidět i z ukázky kódu výše. Ideálně má každý na starost jednu logicky ohraničenou část aplikace. *RedisService* se tak například stará o napojení na Redis server a komunikaci s ním, *InstanceService* o získání dat o EC2 instanci na níž se aplikace nachází, jako je její ID nebo veřejná IP adresa. Aplikace obsahuje více Springových objektů typu servis, než ukazuje obr. 24. Zde jsou ovšem zobrazeny ty hlavní a také jsou zobrazeny jejich vazby s controllery a mezi sebou navzájem.

HTTP komunikace, která zahrnuje dynamická data aplikace (tedy ne statické soubory, ale například data tabulky) a příkazy pro ni, tedy začíná AJAXovým požadavkem z Angular Controlleru klienta. Ten je nasměrován na některou servisu aplikace vystavenou na přípojném bodě daném Controllerem frameworku Spring. Controller požadavek přijme, dle jeho parametrů určí akci požadovanou klientem a zavolá určitý servis pro provedení dané akce. Jakmile je akce dokončena, servis předá zpět controlleru její výsledek buď v podobě požadovaných dat, nebo informace, jak akce dopadla s případnou dodatečnou zprávou pro klienta. Controller vše převede do JSON formátu a odešle klientovi jako HTTP odpověď.

Tato komunikace je zajímavá především z hlediska zmiňované separace frontendu a backendu co se týče technologického provedení. Použitý JSON formát dat je univerzální a pokud by bylo rozhodnuto přepsat backendovou část aplikace například do .NET, na frontendu by nebylo potřeba změnit jedinou řádku kódu. Spring a jeho objekty jsou zde také velice nápomocné, jelikož zjednodušují kód a dělí aplikaci na logické celky s jednoduše rozlišitelným účelem a zodpovědností za určité ucelené spektrum služeb.

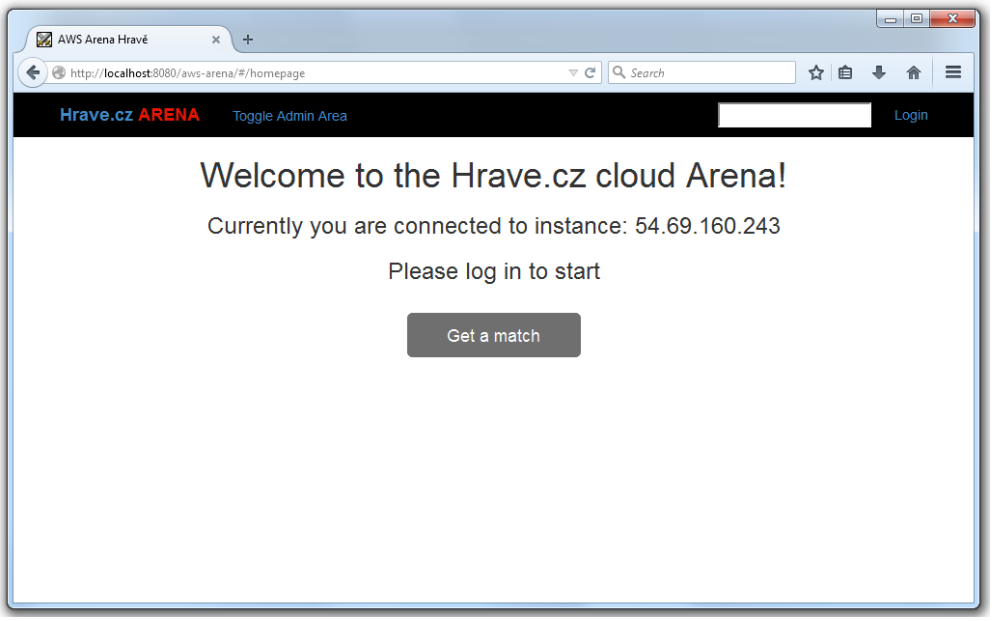
Nyní se již přestaneme věnovat aplikaci jako celku a zaměříme se na tři její hlavní komponenty.

4.5 Detaily komponent a implementace

Jednotlivé komponenty aplikace byly již představeny v první části této kapitoly, ovšem pouze z vnějšího pohledu. Tato část ještě doplní předchozí vnější pohled o grafické zpracování komponent a poté ukáže jak jsou řešeny uvnitř, jak na serverové tak na klientské straně.

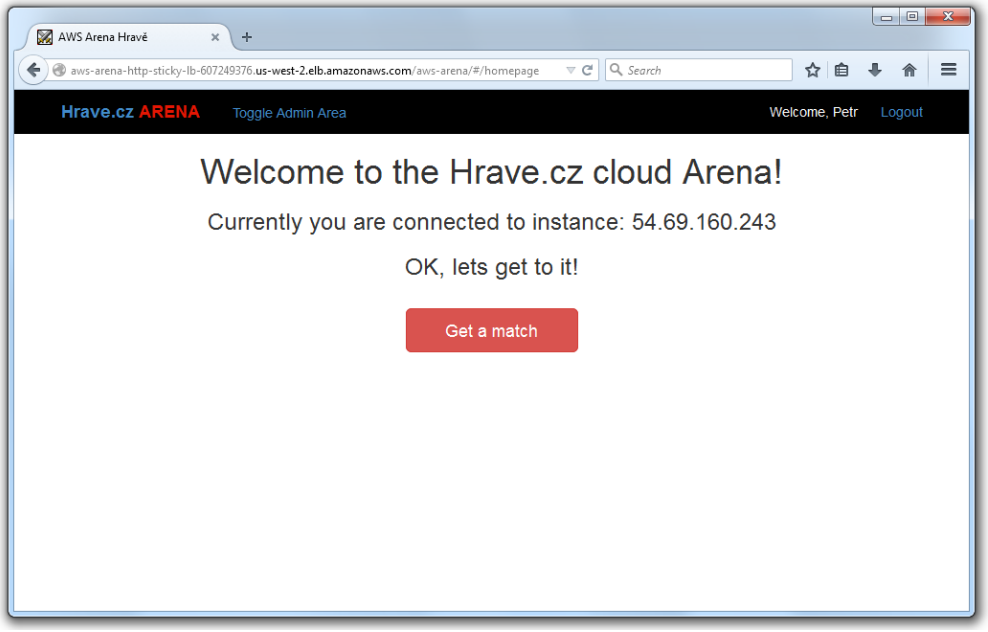
4.5.1 Welcome

Úvodní stránka aplikace, pro přihlášení uživatele a přechod do modulu Matchmakeru.



Obrázek 25 Ukázka grafického řešení welcome při prvním vstupu do aplikace

Jak již bylo řečeno, tento úvodní modul by měl být později v produkční verzi zastoupen samotnou aplikací Hravě. Je počítáno s tím, že v okamžiku přechodu do multiplayer arény, konkrétně do modulu Matchmakeru pro spárováním s oponentem, bude uživatel v aplikaci již přihlášen, aby o něm mohla být udržována určitá data.



Obrázek 26 Ukázka grafického řešení welcome page po přihlášení uživatele

Obr. 25 ukazuje tuto stránku při prvním příchodu uživatele, obr. 26 pak po jeho přihlášení. V pravém horním rohu je jednoduchý přihlašovací formulář požadující pouze uživatelské jméno. Přihlašovací proces je co nejvíce zjednodušen, takže zde neexistuje registrace. Pokud systém uživatele nezná, automaticky jej registruje. Zjednodušení má důvod právě v pozdějším nahrazení tohoto procesu vlastním přihlašovacím procesem Hravě, nebylo proto nutné plnohodnotné přihlašování registraci implementovat. Zde je tedy jednak uživatel identifikován a dále je otevřeno spojení s databází, kde je uživatel vyhledán. Pokud není nalezen, je o něm přidán nový záznam. Modul tedy slouží i k ověření funkčního spojení s databází na Amazon RDS.

Spojení s databází je realizováno naprosto stejně jako s běžnou MySQL databází například na lokálním serveru. Adresa RDS je veřejně přístupná i z prostředí mimo AWS, z důvodu jednoduššího vývoje. V produkční verzi aplikace by byla přístupná pouze pro EC2 instance. Spojení je definováno v aplikačním XML konfiguračním souboru:

```
<bean id="dataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver" />
   <property name="url" value="${db.url.local}" />
   <property name="username" value="${db.username}" />
   <property name="password" value="${db.password}" />
</bean>
```

V aplikaci je pak spojení s databází navázáno získáním beany *dataSource* z aplikačního kontextu:

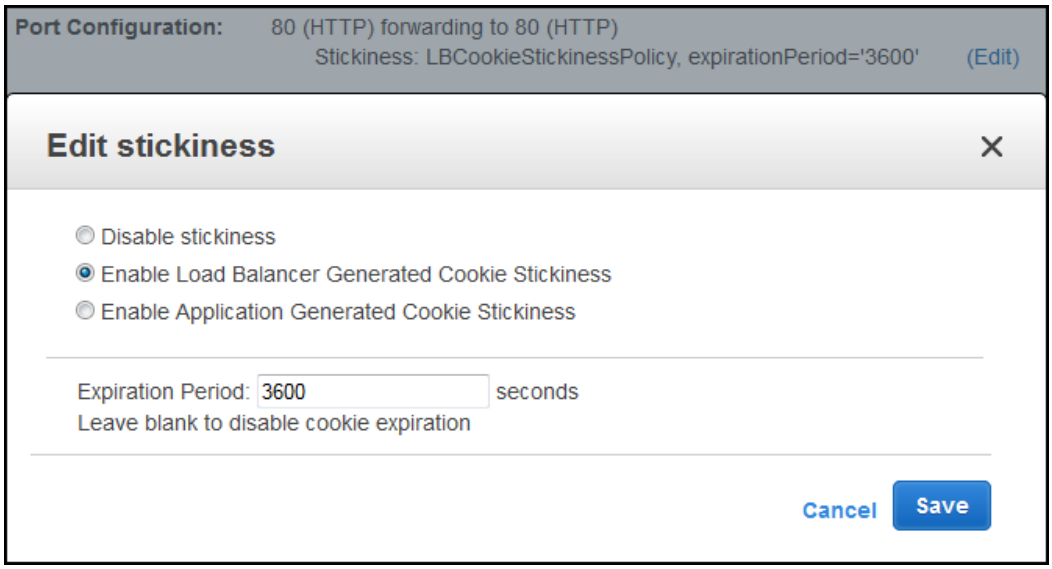
```
public class UserDAO implements ApplicationContextAware {

   private ApplicationContext applicationContext = null;
   private DataSource dataSource = null;

   private Connection getConnection(){
      if(dataSource == null){
         dataSource = (DataSource) applicationContext.getBean("dataSource");
      }
      try {
         return dataSource.getConnection();
      } catch (SQLException e) {
         throw new RuntimeException(e);
      }
   }

   public User getUserByName(String name) {

      String sql = "SELECT * FROM user WHERE NAME = ?";
      Connection conn = null;

      try {
         conn = getConnection();
         PreparedStatement ps = conn.prepareStatement(sql);
         ps.setString(1, name);
         User user = null;
         ResultSet rs = ps.executeQuery();
         if (rs.next()) {
            user = new User(rs.getLong("ID"), rs.getString("NAME"),
                rs.getInt("XP"));
         }
         rs.close();
         ps.close();
         return user;
      } catch (SQLException e) {
         throw new RuntimeException(e);
      } finally {
         if (conn != null) {
            try {
               conn.close();
            } catch (SQLException e) {
               throw new RuntimeException(e);
            }
         }
      }
   }
}
```

Welcome modul má pak ještě dvě další úlohy. První není přímo součástí tohoto modulu, ale děje se při prvním příchodu uživatele a to je nastavení klientské session, která udržuje uživatele přihlášeného. Tuto session vytváří automaticky server Jetty. Každý HTTP požadavek, který server zpracovává, prochází nejprve Interceptorem. Ten zachytí

požadavek a pomocí ThreadLocal mechanismu naváže na obsluhující vlákno CallContext objekt. CallContext obsahuje všechny důležité informace o uživateli, které získají právě z klientské session. Tímto způsobem uživatelská data dostupná kdekoliv v kódu prostým zavoláním CallContextu při zpracovávání uživatelského požadavku. Interceptor vypadá následovně:

```
public class ContextInterceptor extends HandlerInterceptorAdapter {

   protected final Logger log = Logger.getLogger(this.getClass());

   @Override
   public boolean preHandle(HttpServletRequest request,
   HttpServletResponse response, Object handler) throws Exception {
      CallContext.free();
      HttpSession session = request.getSession();
      CallContext.set(SessionInfoService.getSessionInfo(session), session);
      log.info("[ContextInterceptor] Setting context, loggedIn: " +
          CallContext.get().getLoggedIn());
      return true;
   }
}
```

Aby byla dostupná tato uživatelská session, musí pochopitelně uživatelský požadavek směřovat na instanci, kam nějaký požadavek od tohoto uživatele již jednou šel. To je zajištěno pomocí několikrát zmíněných Sticky Sessions od ELB. Při prvním připojení uživatele na load balancer, mu je v odpovědi zaslána cookie pojmenovaná AWSELB, kterou si uživatel uloží v prohlížeči a ELB si přitom zapamatuje, na kterou instanci uživatele s touto cookie přesměroval. Každý další požadavek pak obsahuje hodnotu této cookie, kterou si ELB přečte přesměruje uživatele na "jeho" instanci. Nastavení ELB pro tuto funkci je na obr 27.



Obrázek 27 Nastavení Elastic Load Balanceru pro HTTP režim a Sticky Sessions

ELB nastavuje svou vlastní cookie na 1 hodinu. Je také možné nastavit navázání na vlastní cookie, jejíž délka života se pak řídí podle nastavení aplikace.

Posledním úkolem Welcome modulu je zobrazení informace o veřejné adrese instance, na které se uživatel momentálně nachází. Tuto informaci je také možné vidět na obou obrázcích 25 a 26. Zde sloužila primárně pro testovací účely, kdy bylo potřeba například spárovat pro zápas hráče z různých EC2 instancí. Informace zde tedy byla uvedena, aby bylo ihned jasné na jaké instanci se uživatel nachází. Běžně je tato informace nedostupná, ani URL adresa se v závislosti na různých instancích nemění. Systém proto z vnějšku opravdu vypadá jako by se jednalo o jednu aplikaci na jednom serveru. Zjištění informací o instanci je ovšem důležité při zakládání hry, aby bylo jasné kde pak poběží samotný zápas v aréně. O identifikaci her bude detailněji pojednávat následující sekce. Zde však ještě ukážeme, jak tyto informace dostaneme.

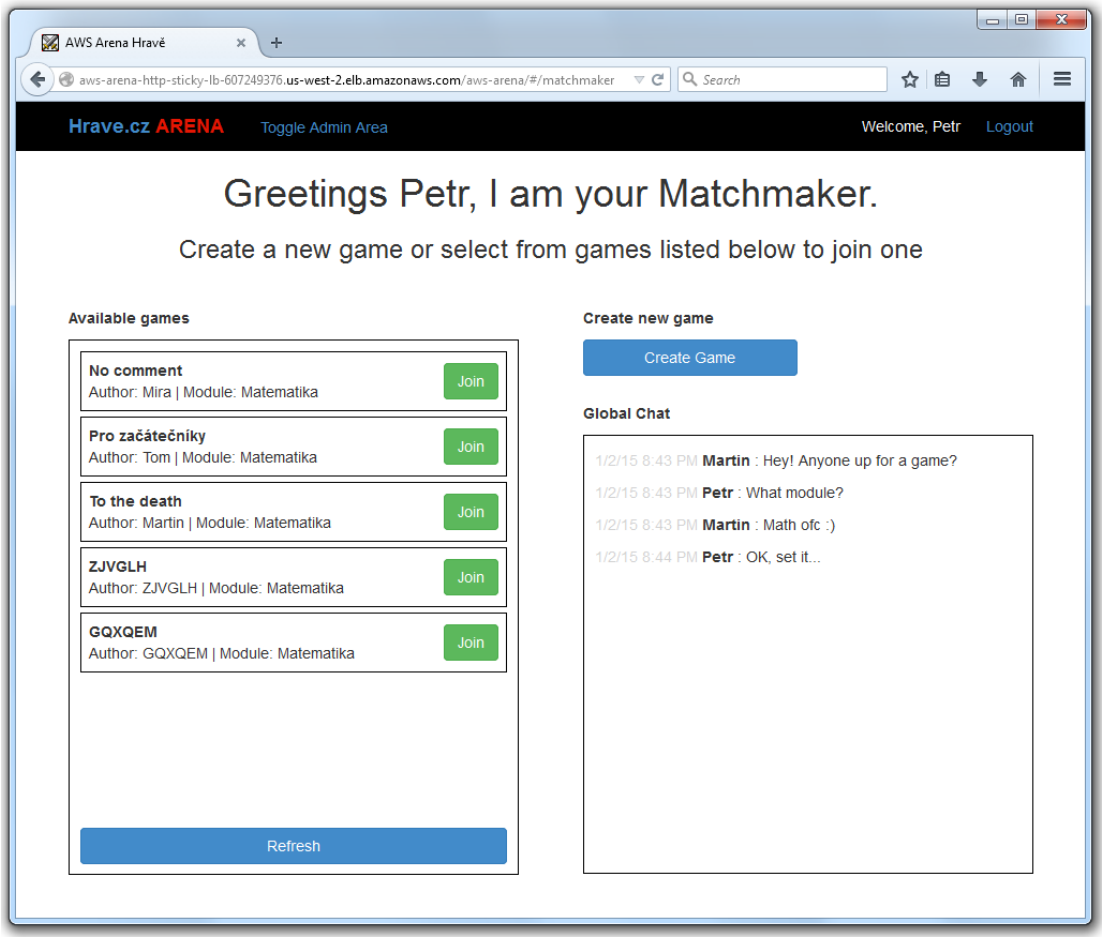
Každý AWS systém obsahuje stejnou interní adresu, kde je možné doptat se na sadu informací o konkrétní instanci. Zde jsou tyto informace zjišťovány pomocí třídy *InstanceService*:

```
@Service
public class InstanceService {

   private String instanceId = null;

   public String getInstanceId(){
      if(instanceId == null){
         instanceId = getInstanceKey("instance-id");
      }
      return instanceId;
   }

   private String getInstanceKey(String key){
      String EC2Id = null;
      String inputLine;
      try {
         URL EC2MetaData = new URL("http://169.254.169.254/latest/meta-data/"
             + key);
         URLConnection EC2MD = EC2MetaData.openConnection();
         BufferedReader in = new BufferedReader(new
             InputStreamReader(EC2MD.getInputStream()));
         while ((inputLine = in.readLine()) != null) {
            EC2Id = inputLine;
         }
         in.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
      return EC2Id;
   }
}
```

Na zavolání adresy *http://169.254.169.254/latest/meta-data/* z EC2 instance systém odpoví kompletní sadou informací o instanci. Pokud je za adresu ještě přidán název konkrétního informačního klíče, například *instance-id*, systém odpoví pouze hodnotou tohoto klíče. Bohužel je nutné získávat informace tímto vnějším voláním, jelikož Java knihovny dodávané Amazonem vývojářům takovou funkci zatím neobsahují.

4.5.2 Matchmaker

Než může hráč vůbec vstoupit do arény, musí si pochopitelně najít soupeře pro hru. Hru je také třeba vytvořit a oba hráči se k ní musí připojit. K tomu všemu slouží modul Matchmaker.



Obrázek 28 Ukázka základní obrazovky modulu Matchmaker

Tento modul je celkově nejkomplexnější částí aplikace, využívá totiž veškeré technologie v ní použité. Obr. 28 ukazuje jeho hlavní obrazovku. Horní lišta zůstává stejná a v okamžiku, kdy by se uživatel odhlásil, byl by přesměrován zpět na Welcome page.

V levé části se nachází seznam dostupných her, který se průběžně aktualizuje. Každá je identifikována názvem, přičemž je zobrazeno i jméno uživatele, který jí založil a modul otázek, který pro ni bude použit. V této práci je modul pouze jeden - matematika. Jeho otázky se generují na backendu, v produkční verzi pak ovšem budou nahrávány z databáze.

V pravé části obrazovky je tlačítko na založení nové hry a pod ním globální chat pro případnou domluvu s protihráči. V tomto momentě by bylo ještě dobré zmínit používané spojení "založená hra", kterým je myšlena instance hry vytvořená zde v Matchmakeru za pomoci tlačítka *Create Game*. Ta představuje pouze informační objekt sloužící k propojení dvou hráčů. V okamžiku spárování dvou hráčů, je tento jejich sdílený

objekt smazán a data z něj jsou použita k opravdovému vytvoření hry, kterým se myslí vytvoření instance arény pro tuto dvojici. Nyní se podíváme na technické provedení jednotlivých akcí, možných v tomto modulu.

Založení hry

Založení hry začíná uživatelským kliknutím na tlačítko *Create Game*, které otevře modální Bootstrap okno zobrazené na obr. 29. Zde uživatel zadá název hry a zvolí požadovaný modul. V tento okamžik ještě může zakládání hry zrušit bez interakce backendu. V okamžiku kdy klikne na modré tlačítko *Create Game*, provede se validace polí formuláře a pokud obsahují platné informace, spustí se AJAXové volání:

```
cookieVal = $.cookie("AWSELB");
var params = {
   action: "createGame",
   name: $scope.createGameName,
   moduleId: $scope.createGameModule,
   uuid: $scope.connection.uuid,
   lbCookieVal: cookieVal};

jcsAjaxCall(BASE_URL + "/match.srv", params, false, function(response) {
   if(response.result == true){
      $scope.createdGame = parseGameId(response.game_id);
   } else {
      $scope.createGameMessage = response.message;
      $scope.showCreateGameMessage = true;
   }
   $timeout(function(){});
});
```

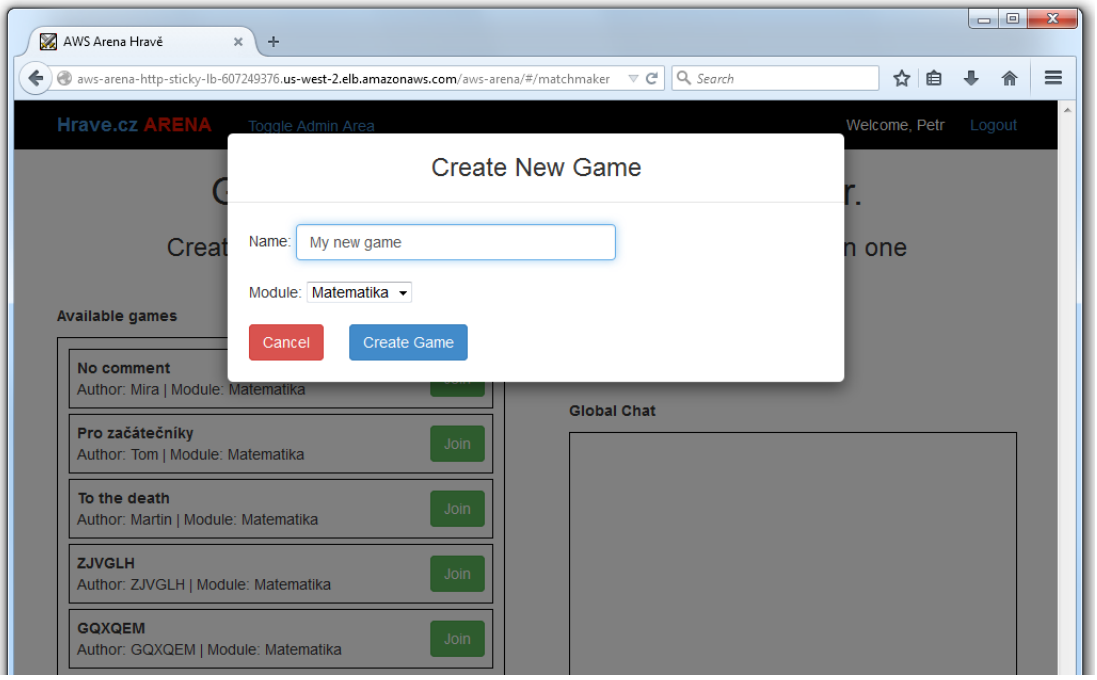
Zde stojí za povšimnutí sada parametrů, které jsou odesílány backendu. Neobsahují pouze data zadaná do formuláře, ale také *uuid*, což je Atmosphere identifikátor uživatele a také hodnotu AWSELB cookie. Oba jsou velice důležité při připojování soupeře k založené hře a budeme se jim detailně věnovat u popisu této akce.

Na backendu je požadavek zpracován a je vytvořen objekt založené hry, který je jednak uložen do paměti instance a jednak odeslán na Redis server:

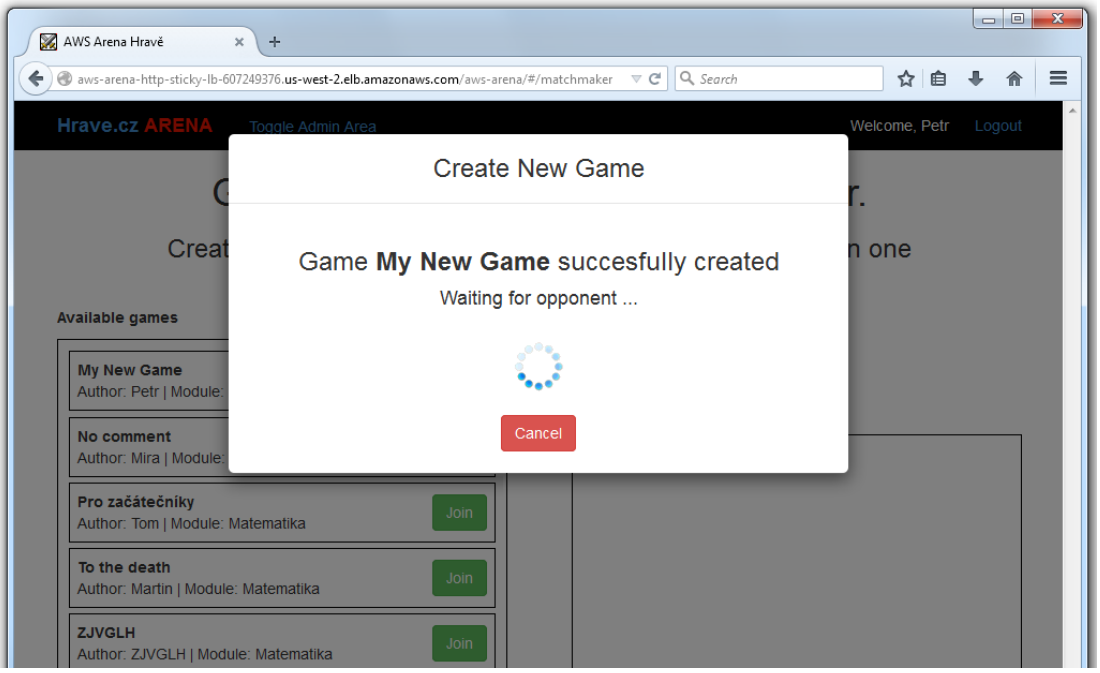
```
Game newGame = new Game(fullGameId, simpleGameId, gameName, playerId,
    playerName, EGameModul.getFromId(moduleId), instanceIp, lBCookieValue,
    uuid);
waitingGames.put(simpleGameId, newGame);
return redisService.createGame(fullGameId);
```

Zde je parametrů ještě o něco více, přibyly konkrétně dvě vygenerovaná ID založené hry a také je uložena veřejná IP adresa instance, na níž byla hra založena. Parametr *simpleGameId* je pseudonáhodné číslo typu Long. *fullGameId* je pak řetězec složený ze všech ostatních parametrů hry. Je také vidět, že do lokální paměti, tedy do mapy *waitingGames*, je uložen objekt hry pod klíčem *simpleGameId*, zatímco na Redis server je odesláno *fullGameId*. Informace z Redis serveru totiž slouží frontendu, který rozparsováním řetězce *fullGameId* zpětně získá kompletní informace o hře.

Pokud vše proběhne v pořádku, server odpoví uživateli, že hra je založena a zobrazí mu čekací okno, ukázané na obr. 30. V porovnání s obrázkem 29 je vidět, že v seznamu dostupných her již založená hra přibyla.



Obrázek 29 Modální okno Matchmakeru pro vytvoření hry



Obrázek 30 Modální okno Matchmakeru při založení hry a čekání na příchod soupeře

Pokud nyní chce zakládající uživatel hru zrušit a klikne na tlačítko *Cancel*, je odeslán požadavek na backend, který musí vymazat hru z Redisu i z lokálního seznamu her.

Aktualizace seznamu her

Při aktualizaci seznamu her aplikace poprvé využívá technologie websocketů. Teoreticky by zde ještě nebyla naprosto nutná, ovšem tento modul ji využívá i pro připojení soupeře do hry a pro okno chatu, bylo proto pohodlnější použít ji i zde. Klient tedy nemusí neustále posílat serveru požadavky na update seznamu, naopak server sám v určitém intervalu informuje všechny připojené klienty o změnách.

Jednou z prvních akcí, kterou klient udělá po příchodu do modulu Matchmakeru, je vytvoření WS spojení se serverem:

```
var socket = atmosphere;
var subSocket = null;

var request = {
   url: 'http://' + $rootScope.instanceIp + BASE_URL + '/matchmaker',
   contentType: 'application/json',
   logLevel: 'debug',
   transport: 'websocket',
   trackMessageLength: true,
   reconnectInterval: 5000,
   enableXDR: true,
   timeout: 300000,
};
...
function init(){
   getInfo();
   $scope.refreshGames();
   subSocket = socket.subscribe(request);
}
```

Zde musí být povolená podpora CORS pomocí parametru *enableXDR*, jelikož klient je sice připojen na URL ELB, ale posílá požadavek přímo na EC2 instanci pomocí parametru *$rootScope.instanceIp*. To je také první funkční využití informace o konkrétní instanci, tedy její veřejná IP adresa.

Na straně serveru spojení realizováno @ManagedService třídou Matchmaker:

```
@ManagedService(path = "/matchmaker")
public final class Matchmaker{

   @Message(encoders = {MatchmakerMessageEncDec.class},
      decoders = {MatchmakerMessageEncDec.class})
   public MatchmakerMessage onMessage(MatchmakerMessage mm) {
      return mm;
   }
}
```

Uvedena je pouze jedna metoda, která zajišťuje přijímání i odesílání zpráv na broadcasteru */matchmaker*. Pro přijmutí zpráv je využita pouze pokud nějaký klient odešle zprávu do globálního chatu. V takovém případě je zpráva okamžitě přeposlána všem klientům připojeným k tomuto broadcasteru. Zpráva není dál nijak zpracovávána nebo ukládána.

Pokud naopak odesílá zprávu server, používá třídu AtmosphereService:

```
@Service
public class AtmosphereService {
   ...
   private Broadcaster matchmakerBroadcaster =
       broadcasterFactory.lookup("/matchmaker");

   public void matchmakerBroadcast(MatchmakerMessage mm){
      matchmakerBroadcaster.broadcast(mm);
   }

   public void matchmakerBroadcastSingleResource(String resourceId,
   MatchmakerMessage mm){
      AtmosphereResource ar = atmosphereResourceFactory.find(resourceId);
      matchmakerBroadcaster.broadcast(mm, ar);
   }
}
```

Může tak poslat zprávu všem klientům připojeným k broadcasteru */matchmaker* pomocí funkce *matchmakerBroadcast(...)*, nebo pouze vybranému klientovi pomocí funkce *matchmakerBroadcastSingleResource(...)* kde je navíc vyhledán AtmosphereResource daného klienta podle jeho ID.

Automatické rozesílání seznamu her provádí třída RedisService:

```
@Service
public class RedisService implements
    ApplicationListener<ContextRefreshedEvent>{

   @Override
   public void onApplicationEvent(ContextRefreshedEvent arg0) {
      ...
   }

   @Scheduled(fixedDelay = 5000)
   private void getGamesUpdate(){
      allGames = jedis.smembers(AppGlobalsService.getRedisSetGameIds());
      atmosphereService.matchmakerBroadcast(
         new MatchmakerMessage(EMatchmakerMessageType.GAMESLIST, allGames));
   }
}
```

Ta implementuje rozhraní *ApplicationListener* a přepisuje jeho metodu *onApplicationEvent(...)*, která zajistí připojení se k Redis serveru v okamžiku plné inicializace aplikace. Metoda *getGamesUpdate()* je anotována pomocí *@Scheduled*, což je anotace Spring frameworku. Ta zajistí, že každých X milisekund se tato metoda spustí, kde X je definováno parametrem *fixedDelay*.

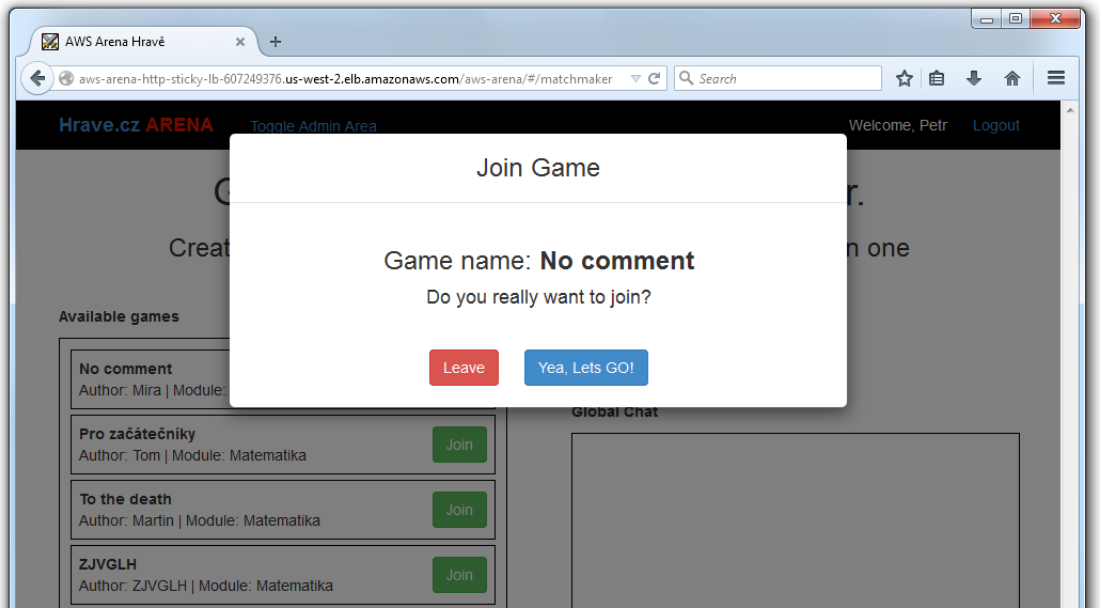
Metoda nejprve vznese dotaz na všechny prvky herního seznamu na Redisu pomocí *jedis.smembers(...)* a poté celý seznam pošle všem svým klientům pomocí již zmíněného AtmosphereServisu. Obsahem seznamu jsou všechny řetězce *fullGameId* aktuálně dostupných her.

Za zmínku ještě stojí upřesnění, že výše uvedený kód popisuje sdílení her pomocí systému popsaného v sekci 4.3.2, tedy pomocí Redisu jako NoSQL databáze. Při sdílení pomocí messagingu nedochází v této funkci k volání Redis serveru, pouze k rozeslání seznamu. Jeho aktualizace pomocí zpráv probíhá v této třídě odděleně a průběžně při každé události na nějaké EC2 instanci, nikoli v pevně daném časovém intervalu.

Připojení do hry

Ještě než se dostaneme k samotnému připojení se ke hře, zopakujeme si některé informace, které založená hra nese ve svém identifikačním řetězci *fullGameId*. Pro připojení do hry jsou primárně důležité tři: *simpleGameId*, *instanceIp* a *lBCookieValue*. První z nich slouží k jednoduché identifikaci hry na instanci, kde byla založena a kde je uložen její objekt v paměti instance. Založené hry jsou však sdílené mezi všemi instancemi. Je tedy nutné hru jasně identifikovat i označením instance, kde se fyzicky nachází. K tomu slouží druhý parametr *instanceIp*.

Třetí parametr je hodnota cookie AWSELB uživatele, který založil tuto hru. Je využit k přesunu připojujícího se soupeře na stejnou instanci, kde je hra vytvořena. Bylo by sice možné, aby každý ze soupeřů byl při souboji v aréně připojen k jiné EC2 instanci. Znamenalo by to ovšem velké množství inter-instanční komunikace, která by zbytečně zatěžovala systém. Je daleko jednodušší přesměrovat připojujícího se uživatele na stejnou instanci, kde se nachází zakládající hráč a objekt založené hry. Proto v okamžiku, kdy je schváleno připojení druhého hráče k založené hře, je mu přenastavena hodnota AWSELB cookie v prohlížeči a jeho příští HTTP požadavek je na ELB přesměrován na stejnou instanci, kde se nachází založená hra a její autor.



Obrázek 31 Modální okno Matchmakeru připojování se do zvolené hry

Proces připojení tedy vypadá následovně. Uživatel si vybere jednu ze seznamu dostupných her a klikne na její tlačítko *Join*. V tom okamžiku mu systém otevře modální okno, vyžadující potvrzení o připojení k vybrané hře, ilustrováno obrázkem 31. Tento mezikrok není nezbytný, slouží ovšem nechtěnému připojení do hry, pokud se uživatel uklikne.

Při otevírání tohoto okna neprobíhá žádná komunikace s backendem, frontend má veškeré informace o hře z rozparsovaného řetězce *fullGameId*. Backend je volán až v okamžiku, kdy uživatel potvrdí výběr hry kliknutím na tlačítko *Yea, lets GO!*. Tehdy je konkrétní instanci backendu poslán AJAX požadavek na připojení uživatele do vybrané hry:

```
$scope.startGame = function(gameId){
   var params = {action: "joinGame", gameId: gameId, name: $scope.player.name};
   var instanceUrl = "http://" + $scope.joinedGame.instanceIP + BASE_URL;

   jcsAjaxCall(instanceUrl + "/match.srv", params, true, function(response) {
      if(response.result == true){
         $.cookie("AWSELB", response.message, { path : "/" });
         $("#joinGameModal").modal('hide').on('hidden.bs.modal',
         function (event) {
            $rootScope.instanceIp = $scope.joinedGame.instanceIP;
            $scope.$apply(function(){
               $location.path("/arena/" + $scope.joinedGame.simpleId);
            });
         });
      } else {
         $scope.joinMessage = response.message;
         $timeout(function(){});
      }
   });
};
```

Nastavovaný parametr *gameId* je skutečné *simpleGameId* vybrané hry. Na třetím řádku je vidět, že AJAX volání bude směřovat přímo na konkrétní instanci místo na ELB, jak je tomu u všech jiných AJAX volání (kromě navazování WS spojení). Zde tedy bude také potřeba podpora CORS, jelikož požadavek jde opět z URL ELB na URL konkrétní instance.

Pokud server odpoví kladně a hra je stále dostupná pro připojení, je nejprve přenastavena hodnota AWSELB cookie a IP adresa současné instance a poté je uživatel přesměrován do modulu Aréna do konkrétní připravené instance hry.

V případě, že nebylo možné připojit se ke hře, tedy že se například stihl připojit někdo jiný dříve a server nestačil ještě poslat aktualizovaný seznam her, je zobrazena chybová hláška zaslaná serverem.

Při přijmutí připojovacího požadavku na straně serveru je nejprve nastavena podpora CORS pro odpověď na tento konkrétní požadavek přidáním hlavičky *Access-Control-Allow-Origin*:

```
else if (action.equals("joinGame")) {
   response.addHeader("Access-Control-Allow-Origin", "*");
   ...
   Tuple<Boolean, String> result = matchmakerServicet.joinGame(gameId, name);
   ...
}
```

Teprve poté je požadavek zpracováván dále. Pokud by totiž na backendu nastala výjimka, server by nemusel tuto hlavičku stihnout nastavit a klient by nedostal na svůj požadavek žádnou odpověď, jelikož bez upravené hlavičky by server zakázal odeslání odpovědi.

Ve třídě MatchmakerService je poté požadavek zpracován následovně:

```
public Tuple<Boolean, String> joinGame (Long simpleGameId, String name){
   if(!waitingGames.containsKey(simpleGameId)){
      return new Tuple<Boolean, String>(false, "Game " + simpleGameId
         + " is no longer availible.");
   }
   Game toStart = waitingGames.remove(simpleGameId);
   redisService.deleteGame(toStart.getFullId());

   Tuple<Boolean, String> creationResult = arenaService.createArena(toStart);
   if(!creationResult.getFirst()){
      return creationResult;
   }
   atmosphereService.matchmakerBroadcastSingleResource(
      toStart.getCreatorResourceUuid(), new MatchmakerMessage(
      EMatchmakerMessageType.GAMEJOINED, true));
   return new Tuple<Boolean, String>(true, toStart.getLBCookieValue());
}
```

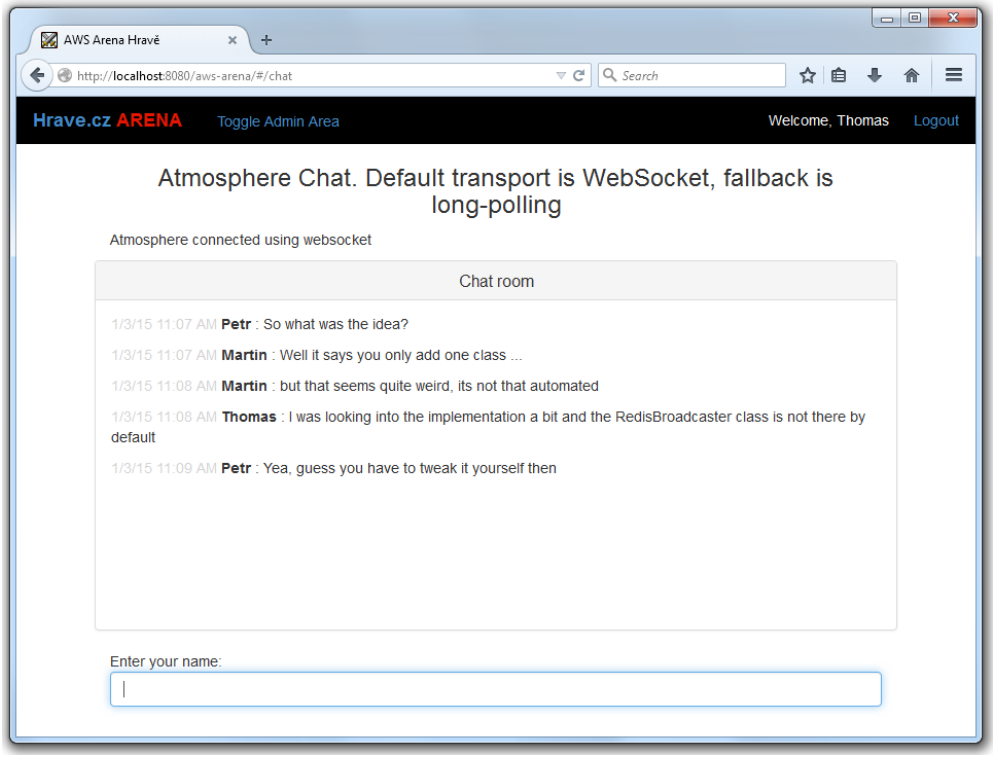
Po ověření, že je požadovaná hra stále dostupná, je smazána z interního seznamu her, smazána z Redis serveru a předána třídě ArenaService na vytvoření herní instance. V případě úspěšného provedení všech akcí je pomocí WS spojení informován zakládající uživatel, že k jeho hře se připojil soupeř a může přejít do arény. Připojujícímu se uživateli je odpověď zaslána klasickou HTTP odpovědí.

Přesně toto využití dvou různých technologií odpovědi ukazuje nutnost použití WS spojení, jelikož bez něj by se zakládající uživatel musel neustále doptávat serveru, zda se již někdo nepřipojil k jeho hře. A v naprosté většině případů by dostal zápornou odpověď, která by pouze plýtvala komunikačními prostředky obou stran.

Zakládající uživatel je do instance arény přesměrován stejným způsobem jako jeho soupeř s tím rozdílem, že není přenastavována jeho AWSELB cookie a IP adresa jeho instance.

Chat

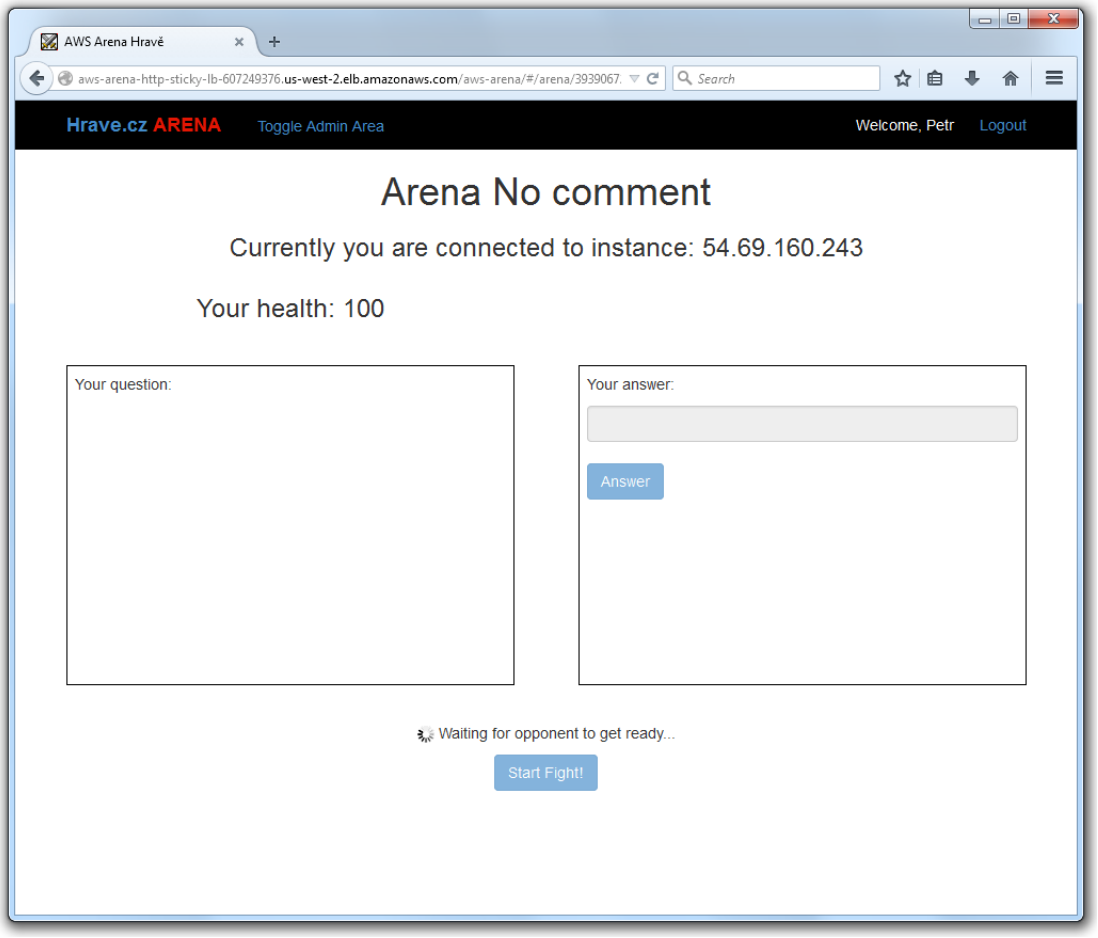
Krátká zmínka zde patří ještě oknu globálního Chatu. Text na obr. 28 do něj byl dodán ručně. Chat je totiž implementován, ale jelikož není ještě jisté jestli bude v produkční verzi zobrazen přímo v modulu Matchmakeru, nachází se prozatím na samostatné stránce *.../aws-arena/#/chat*. Jeho stránku ukazuje obr. 32.



Obrázek 32 Stránka pro globální chat v aplikaci, momentálně přesunutá mimo modul Matchmaker

4.5.3 Aréna

Přesto, že je hlavním modulem aplikace, není již tak složitá jako Matchmaker, alespoň z implementačního hlediska. Představuje soubojový prostor pro dvojici hráčů, přičemž každá hra se odehrává v unikátní instanci arény vytvořené pro tuto dvojici.



Obrázek 33 Modul arény ve fázi přípravy souboje. Hráč potvrdil, že je připraven k souboji a nyní čeká na stejné potvrzení od oponenta.

Instance arény je vytvořena v okamžiku připojení soupeře k založené hře, jak bylo ukázáno v minulé sekci. Tato akce je provedena ve třídě AtmosphereService:

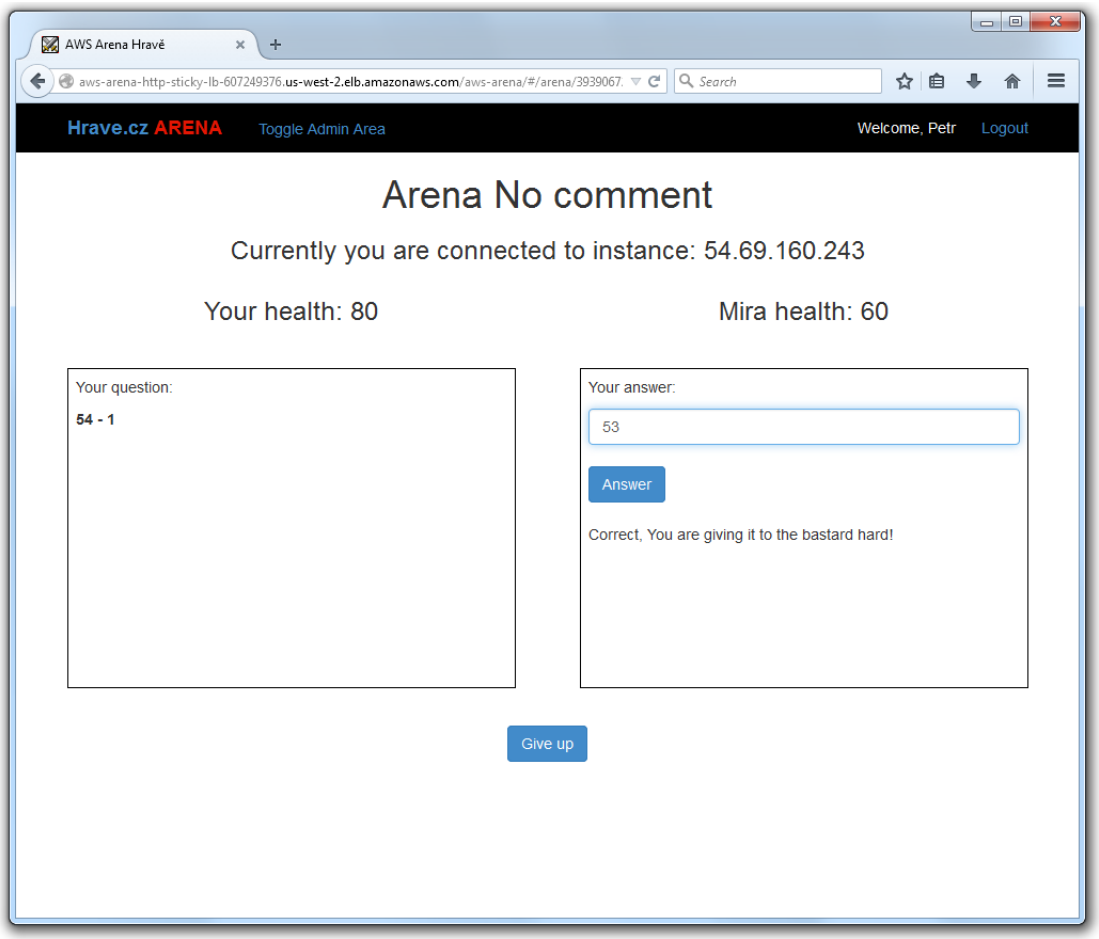
```
public Tuple<Boolean, String> createArena(Game game){
   Broadcaster broadcaster = broadcasterFactory.lookup(
         ARENA + game.getSimpleId(), true);
   broadcaster.setBroadcasterLifeCyclePolicy(
         BroadcasterLifeCyclePolicy.EMPTY_DESTROY);
   Arena newArena = new Arena(game, broadcaster);
   activeArenas.put(game.getSimpleId(), newArena);
   return new Tuple<Boolean, String>(true, null);
}
```

Instanci arény je přiřazen její vlastní Atmosphere broadcaster a samotná instance je uložena do mapy aktivních arén *activeArenas* pod ID shodným se *simpleGameId* para-

metrem založené hry, ze které vznikla. Vlastní broadcaster má také identifikátor shodný s tímto parametrem a je definován jako */arena/{id: [0-9]*}*, tedy řetězec "arena/" po němž následuje číselné ID arény. Nyní aréna čeká na připojení obou soupeřů.

Po přesměrování uživatele do arény z modulu Matchmakeru je u něj zkontrolováno jeho přihlášení do systému. Při příchodu z jiné instance je uživatel automaticky přihlášen na nové instanci právě v tomto okamžiku. Pokud vše proběhne v pořádku, uživateli se zobrazí stránka arény na obr. 33.

Pro uživatele je aréna identifikována jejím jménem, zde tedy *No comment*. V adrese je ovšem identifikována svým číselným ID stejně jako její broadcaster. Pod jménem je uvedena i IP aktuální instance. Tato informace opět sloužila převážně pro testovací účely a rychlé ověření přesměrování uživatelů na korektní instanci. V levé části je prostor, kde se při souboji zobrazují otázky, v pravé pak místo na odpověď uživatele. V okamžiku, který zde je zobrazen, hráč přišel do arény a potvrdil připravenost k zápasu kliknutím na tlačítko *Start Fight*, které je nyní již neaktivní. Nad ním je nyní informační zpráva, že systém čeká, než to samé provede i soupeř.



Obrázek 34 Modul arény ve fázi souboje.

Obr. 34 již ukazuje arénu v průběhu souboje. Tlačítko pro začátek hry zmizelo a bylo nahrazeno možností vzdát se soupeři. V polích pro otázku je zobrazen matematický příklad a pole pro odpovědi je aktivní. Nad těmito poli je zobrazeno zdraví obou soupeřů.

Modul arény jako jediný nemá vlastní Springový controller. Základní informace o stránce arény jsou získány HTTP požadavkem na servisu *getPageData.srv* stejně jako u předchozích dvou modulů. Zbytek komunikace probíhá čistě pomocí WS spojení. Vlastní controller tedy nebyl potřeba.

Navázání spojení ze strany klienta je stejné jako u modulu Matchmakeru, liší se pouze identifikátor broadcasteru. Nejzajímavější funkcí frontendu je *onMessage(...)*, která přijímá a zpracovává zprávy ze serveru:

```
request.onMessage = function(response){
   var responseBody = atmosphere.util.parseJSON(response.responseBody);
   if(responseBody.messageType == "ANSWER"){
      if(responseBody.message == "OK"){
         $scope.systemMessage = "Correct!";
         $scope.oppHit();
      } else if(responseBody.message == "HIT"){
         $scope.systemMessage = "Wrong, Looser!";
         $scope.takeHit();
      } else {
         $scope.systemMessage = "received unknown command " +
             responseBody.message;
      }
   }
   ...
   else if(responseBody.messageType == "QUESTION")
   {
      $scope.arenaQuestion = responseBody.message;
      $scope.arenaAnswer = null;
   }
   else if(responseBody.messageType == "SYSTEM")
   {
      ...
      } else if(responseBody.message == "CONNECT"){
         $scope.amIleftPlayer = responseBody.playersSide;
         $scope.gameName = responseBody.name;
      }
      ...
   }
   else if(responseBody.messageType == "READY")
   {
      ...
   }
   ...
};
```

Každá zpráva serveru je převedena do objektu *responseBody*. Zpráva obsahuje typ a samotné tělo zprávy, které může být dále strukturováno. Zde je ukázáno pouze několik příkladů typů a zpracování zpráv. Například zpráva *CONNECT* typu *SYSTEM* při úspěšném připojení k instanci arény, obdržení nové otázky pomocí typu *QUESTION*, nebo zpracování vyhodnocení odpovědi v typu *ANSWER*. Ve srovnání s Matchmakerem je frontend Arény poměrně přímočarý a není ani tak obsáhlý. Jeho hlavní rozšíření přijde v produkční verzi, kde bude v první řadě propracovanější grafika.

Backend obsahuje podobnou distribuci zpracování zpráv mezi jednotlivé funkce třídy

AtmosphereService. V tomto ohledu zde nedocházelo ke komplikacím s jeho řešením. Jedinou složitější věcí byl Atmosphere @ManagedService tohoto modulu při přijímání a odesílání zpráv. Dokumentace Atmospheru totiž příliš nepopisuje jak rozdělit zpracování příchozích a odchozích zpráv ve funkcích anotovaných jako @Message. Zde bylo použito následující řešení:

```
@ManagedService(path = "/arena/{id: [0-9]*}")
public class Arena {

   @PathParam("id")
   private String arenaIdString;

   private final static String ARENA = "/arena/";
   private long arenaId;

   @Ready(encoders = {ArenaMessageEncDec.class})
   public ArenaMessageOut onReady(final AtmosphereResource r) {
      arenaId = Long.parseLong(arenaIdString);
      return new ArenaMessageOut(EArenaMessageType.INIT,
         "Welcome user "+r.uuid()+" to broadcaster " + ARENA + arenaId);
   }

   @Message(decoders = {ArenaMessageEncDec.class})
   public void onIncomingMessage(ArenaMessageIn message){
      ArenaService.getArenaService().processMessage(message, arenaId);
   }

   @Message(encoders = {ArenaMessageEncDec.class})
   public ArenaMessageOut onMessage(ArenaMessageOut message){
      return message;
   }
}
```

V první řadě zde stojí za zmínku regulární výraz použitý pro definování názvu broadcasteru a pozdější využití parametru *id* v kódu @Ready metody.

Hlavní v této třídě jsou ale dvě funkce @Message. Bylo totiž nutné použít jednu pro příchozí a jednu pro odchozí zprávy a rozlišit je typem vstupního parametru, jelikož jejich odlišný název není brán v potaz. Funkce *onIncomingMessage(...)* pro příchozí zprávy tak má parametr typu *ArenaMessageIn* a návratový typ *void*. Pokud by totiž návratový typ obsahovala, zpráva přijatá touto funkcí by byla obratem odeslána všem klientům daného broadcasteru, což není momentálně žádoucí.

Funkce *onMessage(...)* pro odchozí zprávy s parametrem typu *ArenaMessageOut* na druhou stranu nijak nezpracovává danou zprávu a pouze jí pošle dál. Tato metoda slouží k odesílání zpráv ze systému klientům a jelikož obsahuje dekodér, který by zprávu přicházející od klienta dovedl převést na typ korespondující s jejím vstupním parametrem, není při příchozí zprávě zavolána. Ze obdobného důvodu absence enkodéru pak není volána funkce *onIncomingMessage(...)* při odesílání zprávy ze systému.

Kvůli tomuto specifickému chování Atmospheru obsahuje systém dvě třídy reprezentující zprávy modulu Aréna, *ArenaMessageOut* a *ArenaMessageIn*, ačkoli kromě názvu jsou naprosto shodné.

Uzavírání WS spojení

Vůbec poslední zmínkou k implementaci všech tří komponent je životní cyklus WS spojení. Jak je otevřeno bylo ukázáno vícekrát. K jeho uzavření může dojít buď zavřením internetového prohlížeče, vypršením daného limitu jeho nečinnosti nebo ukončením spojení třetí stranou, například proxy nebo firewallem. Pokud ovšem uživatel přechází mezi stránkami aplikace, nebo aplikaci opustí a přejde na jinou webovou stránku, spojení zůstává otevřeno, což je problém při jeho opětovném návratu do této aplikace a otevření nového WS spojení. Klientská část aplikace by pak mohla obsahovat několik WS spojení, jejichž několikanásobné systémové zprávy by vytvářely chyby v aplikaci.

Proto je nutné spojení korektně uzavírat, kdykoli uživatel opouští danou stránku. V modulech Arény i Matchmakeru byly do jejich klientských částí přidány následující funkce:

```
$scope.$on('$locationChangeStart', function () {
   console.log("User is changing page...");
   socket.unsubscribe();
});

$(window).bind('beforeunload', function(){
   console.log("User is leaving app...");
   socket.unsubscribe();
});
```

První funkce je zavolána v okamžiku, kdy je prováděno vnitřní přesměrování v rámci aplikace. *$locationChangeStart* označuje událost nastávající při volání Angular funkce *$location.path(...)*. Druhá funkce zajišťuje uzavření spojení, pokud uživatel odchází z aplikace na jinou webovou stránku.

4.6 Shrnutí architektury a implementace

Tato kapitola ukázala v několika úrovních a pohledech celý systém. Jejím úkolem bylo především sesadit dohromady veškeré technologie představené v předchozích částech a ukázat jejich spolupráci a komunikaci. Architekturu celkového systému bylo možno popsat vzhledem k poměrně malému počtu jeho komponent. Vnitřní architektura aplikace je naopak v některých částech poměrně komplikovaná ačkoli nijak vysoce nestandardní, příkladem může být vzájemné používání Springových servisů. Proto byla ukázána pouze v části HTTP komunikace, která popsala její důležitou vlastnost - oddělení technologií frontendu a backendu.

Jednotlivé komponenty aplikace byly ukázány jak z uživatelského hlediska, pro ilustraci současných schopností aplikace, tak z hlediska programátora, který by chtěl využít technologie popisované v této práci. Implementační popis se soustředil na zajímavé nebo problematické části implementace, popisující spolupráci s Amazon službami, nebo využití nových technologií ze třetí kapitoly. Zbytek implementace je pokryt spíše standardními technologiemi, případně je implementace analogická k poskytnutým ukázkám.

Nyní následuje poslední část práce, která ukáže, jak a zda aplikace opravdu funguje v reálném provozu na Amazon cloudu.

5 Testování

Otestovat celý systém, je úkol velice komplexní. Zpravidla se začíná unit testy malých funkčních částí systému, mnohdy až k úrovni jednotlivých funkcí. Toto testování probíhá pouze programově je velice podrobné a vhodné spíše pro systémy, kde výpočetní operace a aplikační logika dalece převyšují grafickou stánku aplikace. Pokud aplikace bude pracovat minutu na nějakém úkolu než ukáže jeho výsledek, je dobré vložit testy dovnitř tohoto procesu, aby bylo možné ověřit správnost řešení úkolu v různých fázích zpracování.

Pokud aplikace na každou uživatelskou akci odpovídá grafickým výstupem za kterým není složitá business logika, je na zvážení, zda je potřeba vkládat programové testy dovnitř aplikace. Druhou možností jsou proto pro vizuální aplikaci vhodnější klikací testy. Automatické programy, které umějí buď nahrát uživatelskou aktivitu na GUI aplikace a poté jí opakovat, případně je možné jim popsat tuto aktivitu pomocí programovacího jazyka.

V této kapitole bude nejprve otestována celá aplikace pomocí klikacích testů, aby bylo ověřeno, že snese zátěž současného připojení mnoha uživatelů. Poté proběhne test soustředěný pouze na jednu službu AWS, kterou je ElastiCache a Redis server. Účelem bude zjistit efektivitu dvou různých architektur pro inter-instanční komunikaci. Druhý test bude probíhat převážně programově, hlavním výstupem ovšem budou stejně jako u prvního testu výsledné grafy zátěže služeb.

5.1 Celková zátěž aplikace

První test má simulovat reálnou zátěž. Musí tedy probíhat přes grafické rozhraní aplikace aby bylo ověřeno, že i ovládací prvky se v zátěži chovají jak mají. Popišme si nejprve jak vypadá průchod uživatele aplikací:

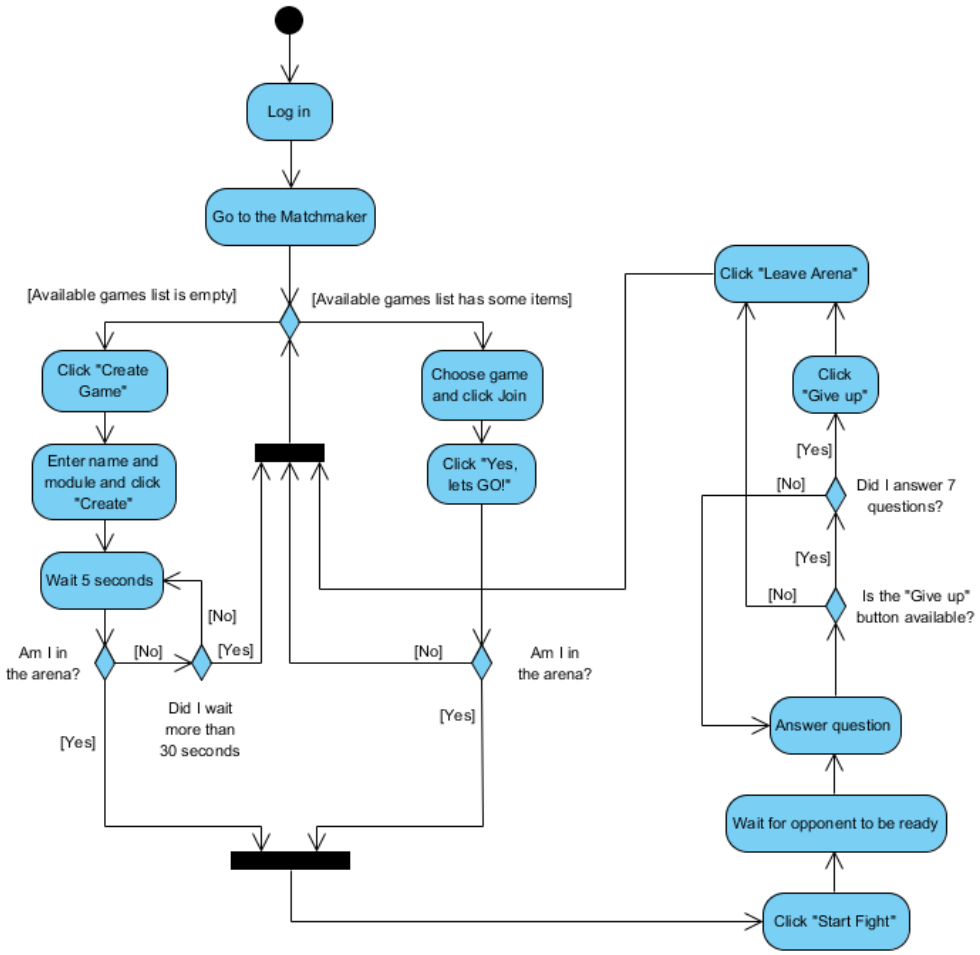
- Přihlas se a vstup do Matchmakeru
- Rozhodni se zda založíš hru nebo se k nějaké připojíš
- Pokud se připojuješ
  - Zvol hru a klikni na *Join*
  - Ověř že si zvolil správnou hru a klikni na *Lets GO!*
  - Pokud hra nebyla dostupná, zavři modální okno a opakuj postup
- Pokud vytváříš hru
  - Klikni na *Create Game*
  - Vyplň potřebné údaje a klikni na *Create*
  - Počkej na připojení oponenta
- Klikni na *Start Fight* a počkej až soupeř udělá to samé
- Odpovídej na otázky
- Po skončení hry opusť arénu kliknutím na tlačítko *Leave Arena*

Takto by mohl vypadat jeden průchod uživatele skrz aplikaci. Kromě toho kdy má uživatel vzdát hru, pokrývá všechny scénáře. I toto rozhodnutí se však dá do testu přidat. Teď je potřeba nástroj pro takový test.

5.1.1 Selenium a nastavení testu

Konkrétně jeho WebDriver projekt [17], je výborným nástrojem na vytvoření automatického klikače pro testování. Selenium začínalo jako plugin do prohlížeče Firefox, který uměl již zmíněné nahrání uživatelské akce a její následné přehrání. WebDriver později přidal možnost toto klikání do prohlížeče nastavit programově a nyní dokonce podporuje většinu velkých prohlížečů včetně několika jejich verzí.

Knihovny Selenia jsou dostupné v několika programovacích jazycích, pro Javu i v repositáři Mavenu, vytvoření projektu je tedy velmi rychlé. Dokumentace je dostupná na zmíněném webu projektu, zde bude stačit informace, že aktivní elementy, které má Selenium používat musí být identifikovány pomocí atributu *id* nebo *name*, v případě linku stačí i jeho text. Zdrojový kód klikacího testu je na CD přiloženém k této práci. Diagram aktivit pro průchod klikače aplikací vypadá následovně.



Obrázek 35 Activity diagram pro první test

Téměř kopíruje průchod uživatele aplikací, který byl popsán na minulé stránce. Liší se pouze v tom, že po zodpovězení 7 otázek v aréně vzdá hru a po opuštění arény

opět vytváří hru nebo se k nějaké připojuje. Vytváří tak nekonečnou smyčku, která je ukončena až přímým vypnutím samotného testu.

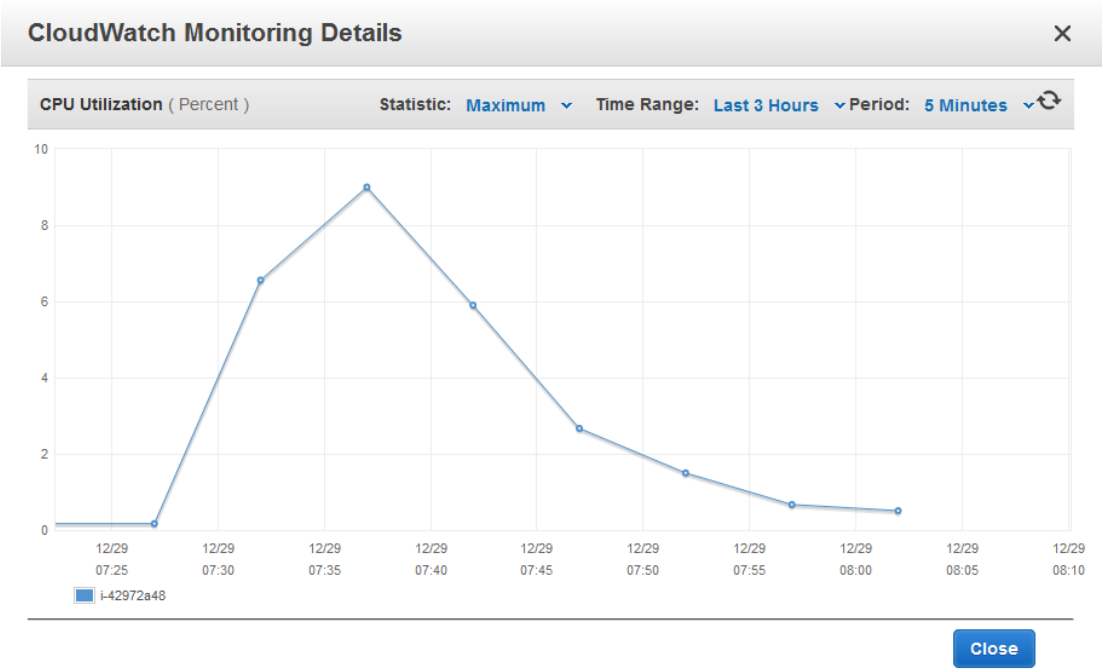
Pro běh testu vyl vytvořen spustitelný jar soubor, který akceptuje dva parametry. Prvním je počet paralelně běžících instancí klikače, druhým je počet opakování testu. Ten může být nastaven i na nekonečno opakování jak je ukázáno v diagramu na obr. 35. Každá instance klikače spustí svůj prohlížeč a běží nezávisle na ostatních. Počet instancí je tak omezen pouze pamětí stroje, kde test běží, jelikož každý prohlížeč potřebuje cca 100 MB RAM.

5.1.2 Výsledky

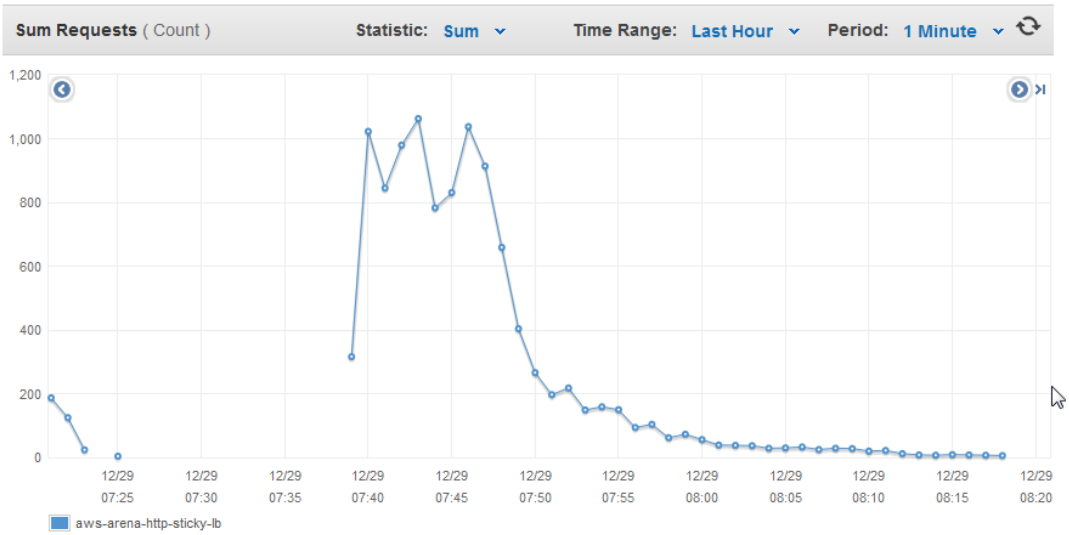
Test byl prováděn ve školních laboratořích Fakulty elektrotechniky. K dispozici bylo 20 strojů a na každém bylo spuštěno 10 instancí klikačů. Test byl opakován dvakrát, poprvé s jednou běžící EC2 instancí aplikace, podruhé se dvěma. Celá konfigurace systému je zde:

- 2x EC2 t2.micro instance (1 vCPU, 1GB RAM, 8GB SSD), instalován Jetty server verze 9.2.5, Java 1.7, Amazon Linux 64-bit
- Elastic Load Balancer, HTTP režim se Sticky Session
- ElastiCache s Redis serverem verze 2.8.6, t2.micro instance (0.5GB RAM)
- Amazon RDS s enginem MySQL 5.6.21, t2.micro instance, 5GB SSD

V prvním testu bylo zvyšování zátěže postupnější, jelikož zároveň probíhalo logování se na testovací stroje. Výsledky ukazují obrázky 36 a 37. Maximální vytížení CPU jedné instance se dostalo na 9%, ELB ukazuje zpracování kolem 1000 HTTP požadavků za minutu.

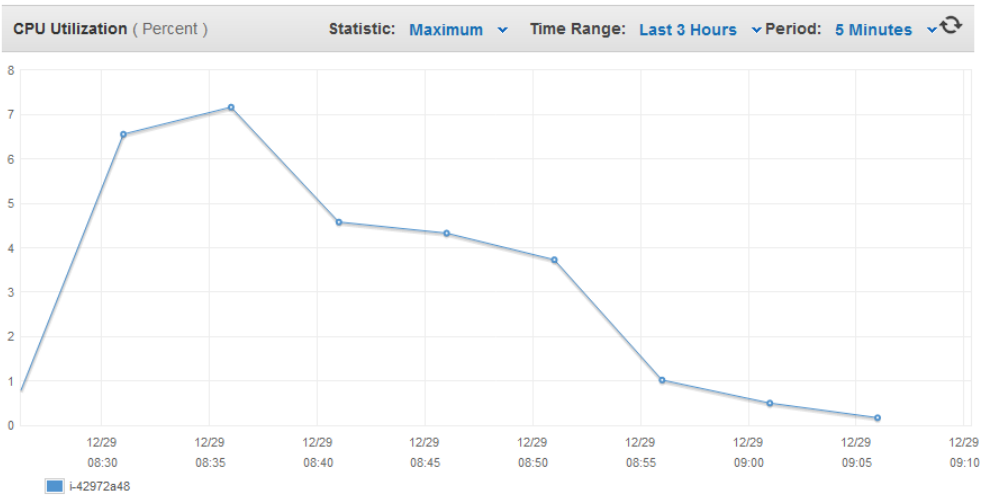


Obrázek 36 Test 1.1, CPU zátěž

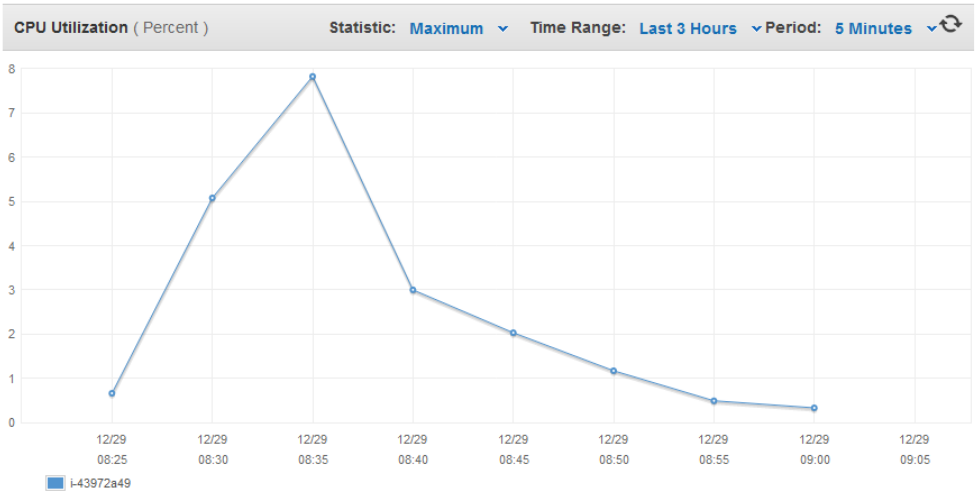


Obrázek 37 Test 1.1, ELB suma příchozích požadavků

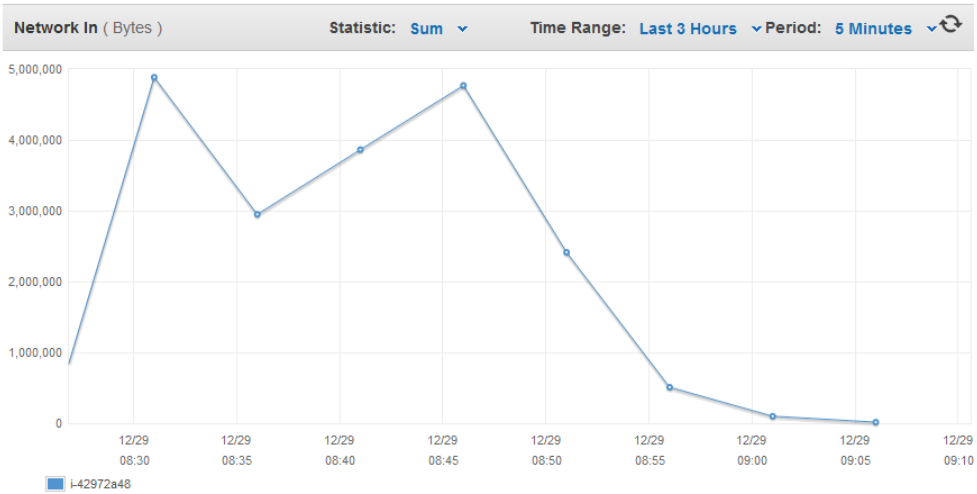
Již první test ukázal poměrně velkou slabinu klikačů pro zátěžový test. Program je nastaven tak, aby rozpětí mezi jednotlivými kliknutími bylo 1 až 2 vteřiny, v závislosti na konkrétní akci. Pokud ovšem aplikace na moment zpomalí a nezobrazí očekávaná data včas, klikací program nahlásí chybějící element, případně neproveditelnou akci s ním a ukončí se. Odhadem polovina klientů tak skončila chybou po jednom či dvou průchodech. Objevila se také zajímavá chyba, kdy se klikač korektně přihlásil do aplikace, ovšem systém jej přesto odmítal pustit dále. Při prvním přihlášení do aplikace se tato chyba dala obejít zvětšením časových rozestupů mezi jednotlivými kliknutími. Při přechodu z Arény zpět do Matchmakeru ovšem nebylo již snadné chybu replikovat a nepodařilo se ji odstranit. Problém může být v příchodu klienta do Matchmakeru instance, kde předtím nebyl přihlášen, bylo by ovšem potřeba delšího zkoumání pro řešení tohoto problému. Pro druhý test byly tedy klikače mírně upraveny na větší časové rozpětí u problematických částí testu. Byly také spouštěny téměř těsně za sebou, takže všech 200 klientů se připojilo během cca 4 minut.



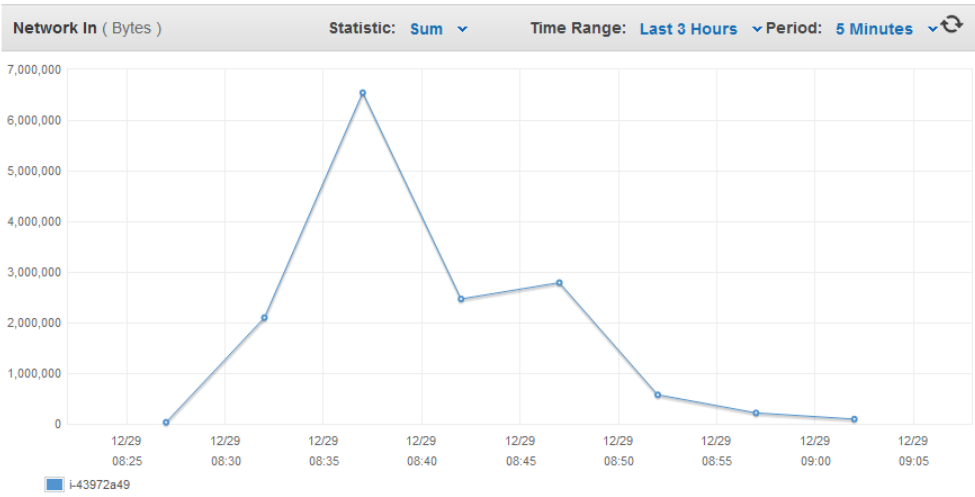
Obrázek 38 Test 1.2, CPU zátěž instance 1



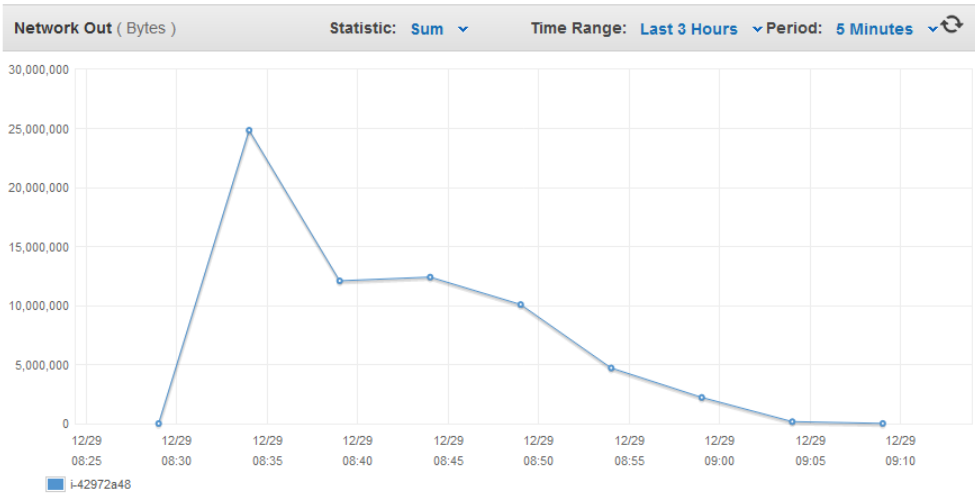
Obrázek 39 Test 1.2, CPU zátěž instance 2



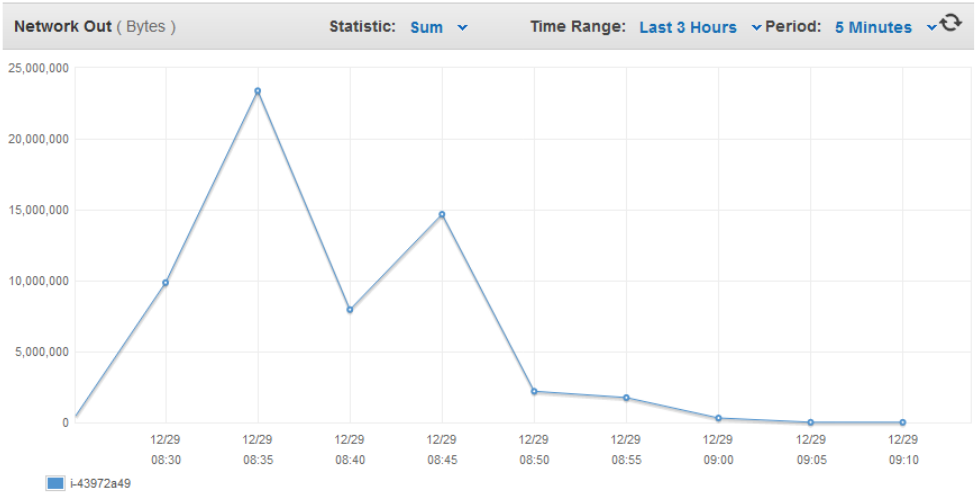
Obrázek 40 Test 1.2, Data In instance 1



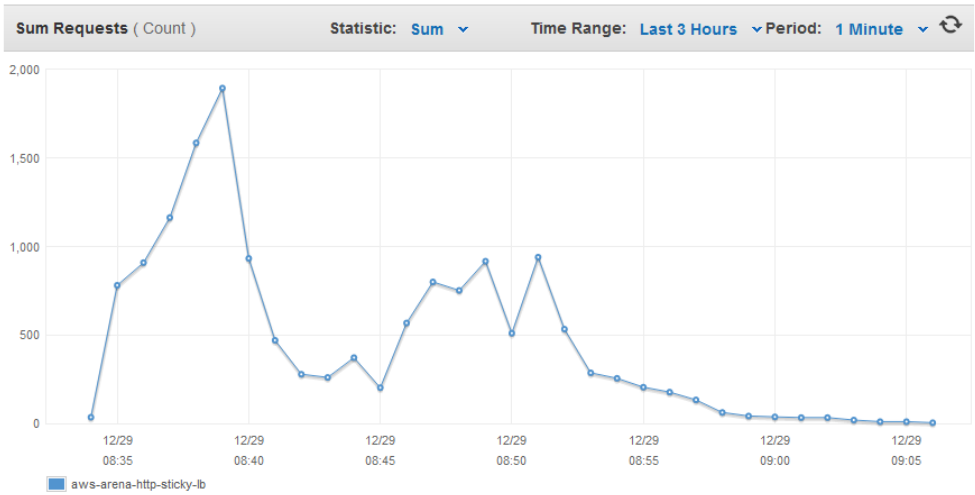
Obrázek 41 Test 1.2, Data In instance 2



Obrázek 42 Test 1.2, Data Out instance 2



Obrázek 43 Test 1.2, Data Out instance 2

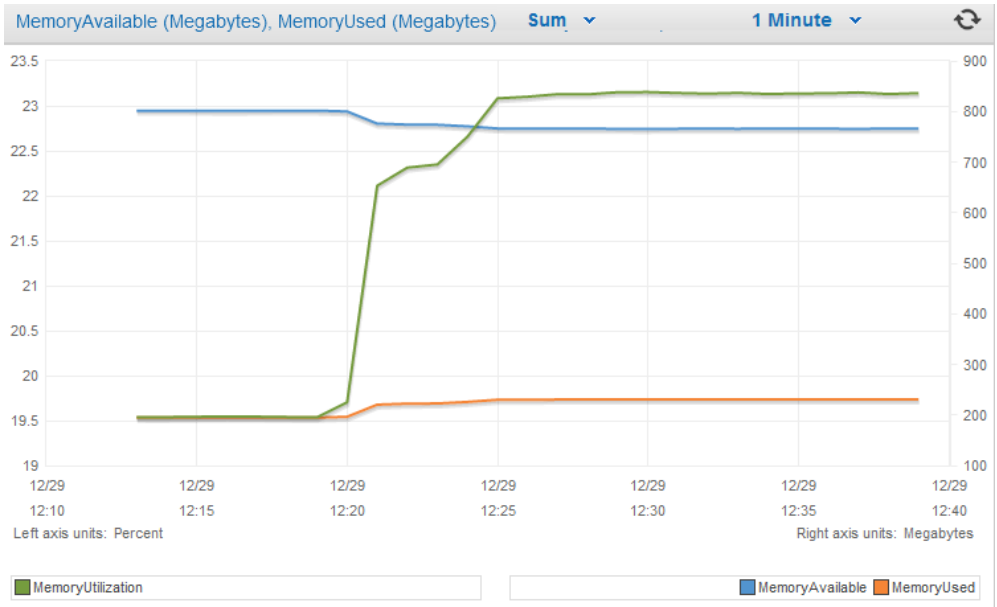


Obrázek 44 Test 1.2, ELB suma příchozích požadavků

Výsledky druhého testu ukazují znatelně vyšší zatížení systému. Ani tentokrát nevydržely klikače příliš dlouho. Průchod aplikací byl většinou v pořádku, ale při přechodu z Arény zpět do Matchmakeru se začala výrazněji projevovat chyba, kdy systém odmítl uznat, že je uživatel přihlášen. Část klikačů tak i přes úpravu testu odpadla. Podle vizuální kontroly jednotlivých testovacích stanic ale stačily naběhnout všechny klikače, než začaly první z nich chybovat. Druhý test je tedy možné považovat za plnohodnotné zatížení systému 200 klienty, kteří navíc používají aplikaci znatelně rychleji než lidský uživatel.

Obě instance byly zatíženy na 7 až 8 procent, což ukazuje i fungující load balancing požadavků, ačkoli klienti na první instanci vydrželi testování déle. Oproti prvnímu testu byly přidány grafu přijatých a odeslaných dat obou instancí, které vykazují stejné charakteristiky jako zatížení instancí - rovnoměrné zatížení a o něco pozvolnější pokles na první instanci. Objem dat odeslaných systémem ukazuje maximálně 25 MB za 5 minut, což je při 100 klientech na instanci 0,5 MB/min, tedy velice zanedbatelný přenos dat. Objem přijatých dat byl ještě přibližně čtyřikrát menší. Poslední graf ukazuje opět sumu příchozích požadavků na load balanceru. Připojení přibližně dvojnásobného počtu klientů než při prvním testu odpovídá také přibližně dvojnásobnému objemu těchto požadavků.

Monitorovací služba Amazonu CloudWatch bohužel neumí automaticky monitorovat využitou paměť na jednotlivých EC2 instancích. Měření je nutné nastavit manuální instalací měřících skriptů přímo do operačního systému instance. Tento nedostatek byl bohužel zjištěn až po skončení testů ve školních laboratořích. Byl proto spuštěn ještě jeden dodatečný test na jedné instanci s celkovým počtem 40 klientů s dostupnými počítači mimo laboratoře. Obr. 45 ukazuje, že po připojení všech klientů mezi časy 12:18 a 12:25 vzrostlo využití paměti z 19,5% na 23% a v absolutních číslech bylo využito cca 50 MB paměti z dostupného 1 GB.



Obrázek 45 Test 1.3, dodatečný test na využití paměti instance

Celý zátěžový test tedy prokázal, že aplikace funguje korektně i při zátěži většího počtu klientů a její teoretické limity jsou kolem 1000 klientů na jednu instanci.

5.2 Redis, porovnání architektur

Druhým provedeným testem, bylo porovnání navržených architektur pro inter-instanční komunikaci v praxi. Obě architektury byly představeny v části 4.3 a bylo konstatováno, že jejich teoretické porovnání z hlediska výkonu je složité. Proto bylo rozhodnuto udělat zátěžový test, který by porovnal obě architektury proti sobě.

Z minulého testu celkové zátěže bylo jasné, že pomocí klikačů bychom nikdy nebyli schopni založit dostatečné množství her pro zaznamenatelné vytížení Redis serveru. Jeden klikač stíhal založit jednu hry každých odhadem 30 vteřin. Zde bylo potřeba zakládat a mazat stovky her během vteřin. Test proto musel proběhnou čistě mezi EC2 instancemi a Redisem. Klient, který test řídil byl pouze informován o jeho průběhu.

Pro tento test byla tedy vytvořena vlastní třída TestService obsahující metodu pro spuštění parametrizovatelného počtu konkurenčních vláken, jež budou zakládat a mazat hry. Každému vláknu je pak určen maximální počet her, jež může udržoval založený. Jediné další využívané třídy byl RedisService a AtmosphereService. První z nich pochopitelně pro využití jejích metod pro manipulaci se záznamy her, druhá pro zpřístupnění Matchmaker broadcasteru, který odesílal informace o průběhu testu. Pro test bylo také zakázáno odesílání informací o založených hrách připojeným klientům. Bylo potřeba změřit data, která každá instance odesílá pouze Redis serveru, a data založených her pro klienta by tato čísla zkreslovala.

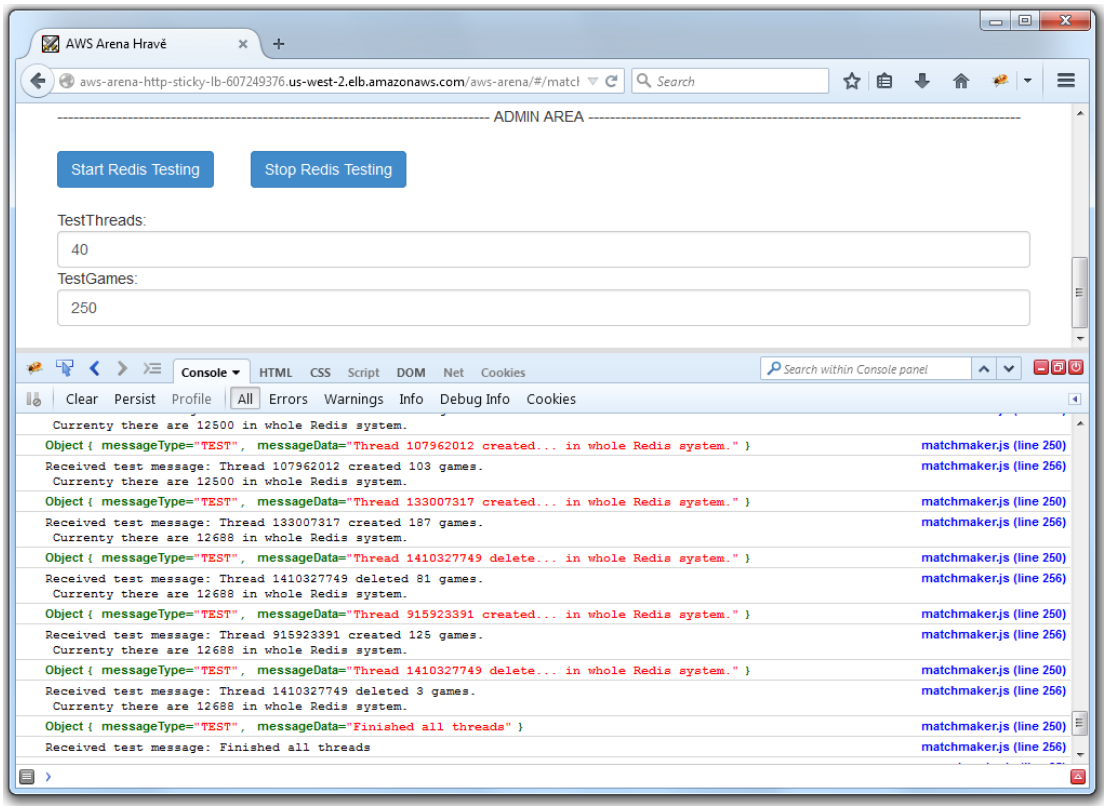
Ovládací prvky testu byly umístěny do Admin části stránky Matchmakeru, jelikož tento modul již disponuje globálním WS spojením, přes které bylo možno posílat informace. Průběh testu na jedné instanci byl tedy následovný:

- Klient se přihlásí, vstoupí do Matchmakeru a zobrazí Admin rozhraní
- Zvolí počet vláken testu a maximální počet založených her na vlákno
- Odstartuje test
- Instance nastartuje příslušný počet vláken
- Každé vlákno opakuje nekonečný cyklus
- Na začátku cyklu se rozhodne zda bude vytvářet nebo mazat hry
- Vygeneruje náhodné číslo určující počet her
- Vytvoří nebo smaže daný počet her
- Informuje klienta o počtu provedených operací a celkovém počtu her v systému
- Opakuje cyklus dokud není test přerušen

Na obou architekturách byly provedeny tři testy s postupně vzrůstajícím počtem vláken i her. První test obsahoval 5 vláken a maximálně 10 her na vlákno. Sloužil pouze k ověření korektního běhu testu. Druhý test spouštěl 20 vláken s maximem 50 her a třetí test 40 vláken s maximem 250 her. Testy byly vždy spouštěny na dvou EC2 instancích zároveň. Ukázány budou výsledky převážně třetích testů, kde docházelo již ke znatelnému rozdílu.

Jelikož každé vlákno rovnoměrně mazalo a vytvářelo hry, průměrně udržovalo polovinu maximálního možného počtu her. Při nejvyšší zátěži to ovšem tvoří průměrně 10 000 her současně sdílených v systému ((250 her x 40 vláken / 2) x 2 instance). To je již zátěž, která se na Redis serveru projeví.

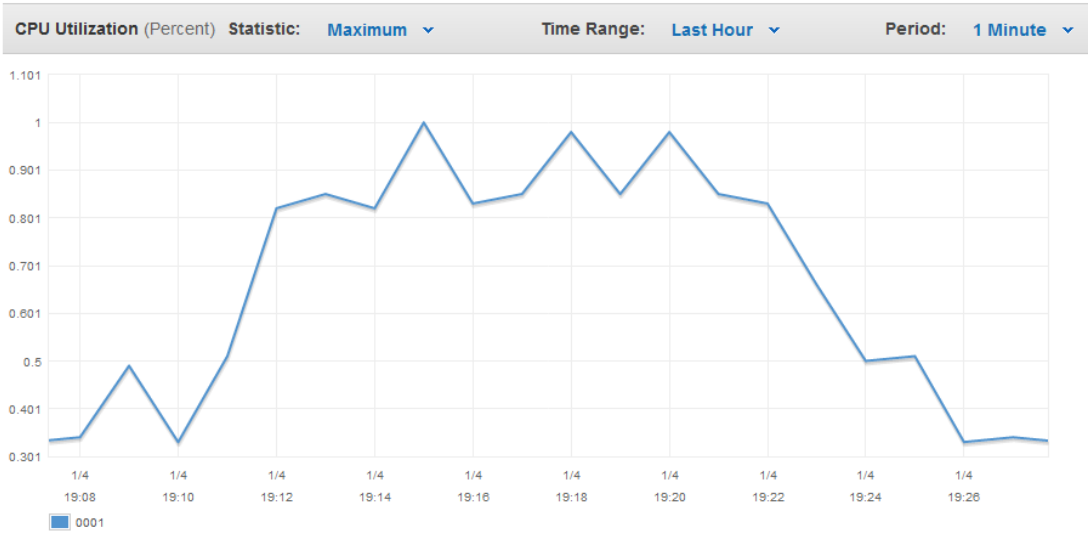
Obr. 46 ukazuje klienta na konci testu. V konzoli jsou vidět průběžně zasílané informace o stavu testu.



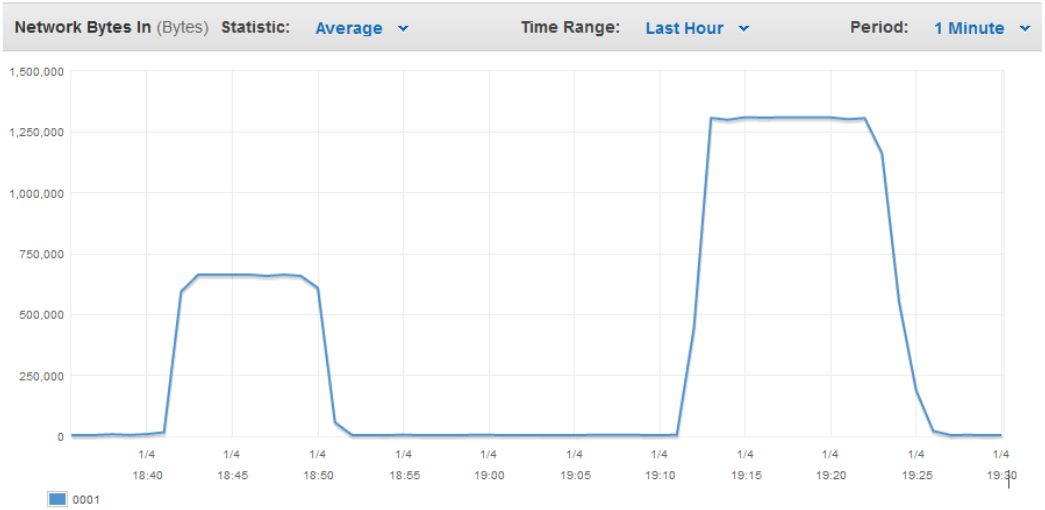
Obrázek 46 Test 2, okno klienta ukazující Admin panel a konzoli s informacemi o průběhu testu

5.2.1 Výsledky testu - Messaging

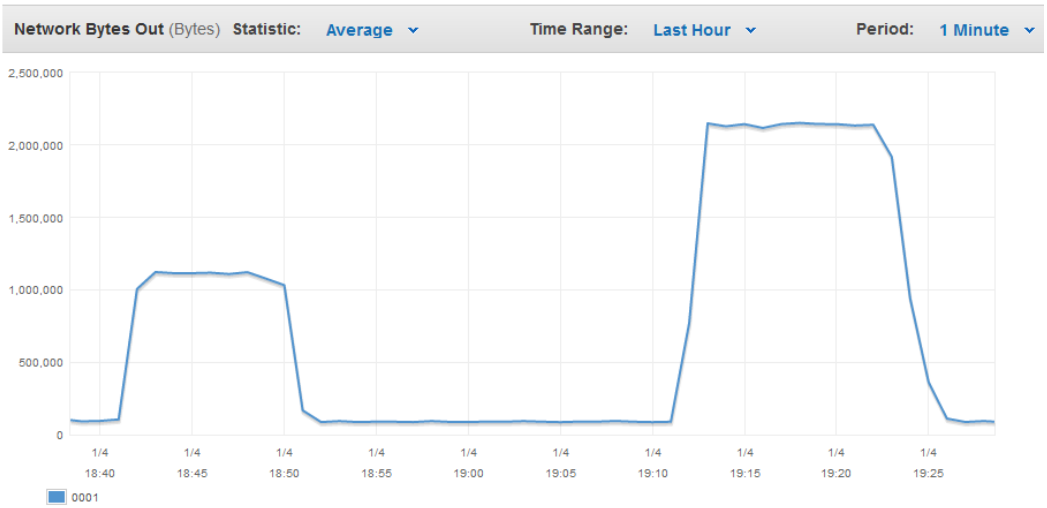
Jako první byla testována architektura messagingu. Zde každá instance udržuje kompletní seznam založených her u sebe v paměti a Redis server, respektive jeho Pub/Sub kanál, je využíván pouze pro zasílání zpráv mezi instancemi informujícími o nově založených nebo naopak smazaných hrách.



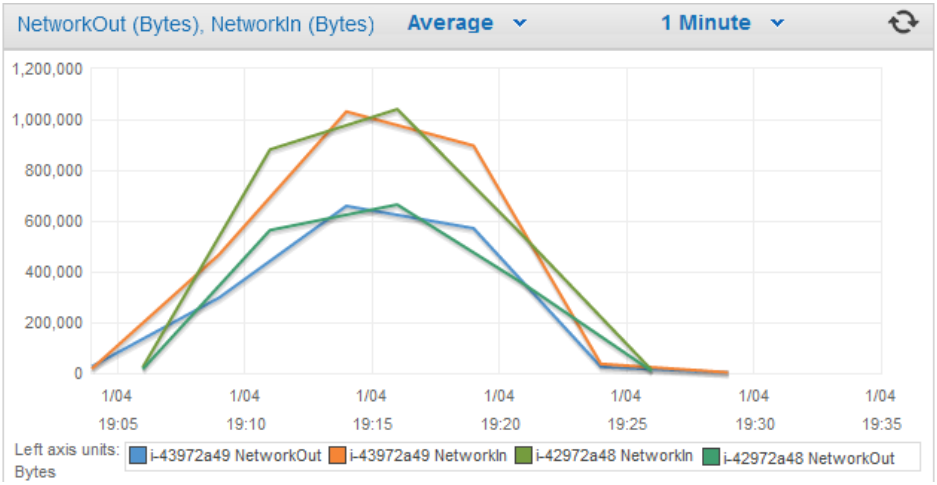
Obrázek 47 Test 2.1, CPU zátěž Redisu, 40x250 her



Obrázek 48 Test 2.1, Data In Redis server, test 20x50 a 40x250 her



Obrázek 49 Test 2.1, Data Our Redis server, test 20x50 a 40x250 her



Obrázek 50 Test 2.1, Data In/out na EC2 instancích, 40x250 her

Maximální vytíženi CPU na instanci Redis serveru dosáhlo 1%, což je opravdu zanedbatelné. Na druhou stranu server slouží pouze pro přeposílání zpráv, neprovádí žádné výpočetní operace.

Datový tok směrem na Redis server byl kolem 650 KB/sec v případě testu 20x50 her, 1300 KB/sec v případě testu 40x250 her. Tento tok zahrnuje všechny zprávy posílané instancemi do PubSub kanálu.

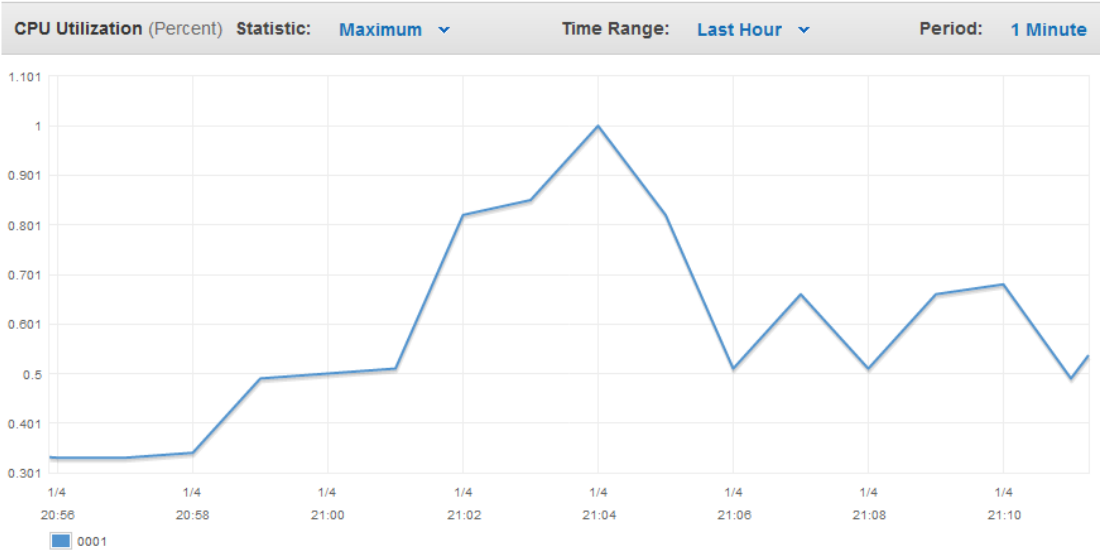
Datový tok z Redis serveru k EC2 instancím má zhruba dvojnásobné hodnoty. To odpovídá, jelikož instance byly v testu dvě, každá zpráva, která na Redis přišla, byla poté odeslána na obě EC2 instance.

Poslední obrázek ukazuje datový tok dovnitř a ven EC2 instancí v testu 40x250 her. Tento graf také koresponduje s ostatními dvěma. Každá instance přijímala maximálně kolem 1 MB/sec, což odpovídá odchozímu datovému toku mírně nad 2 MB/sec u Redis serveru.

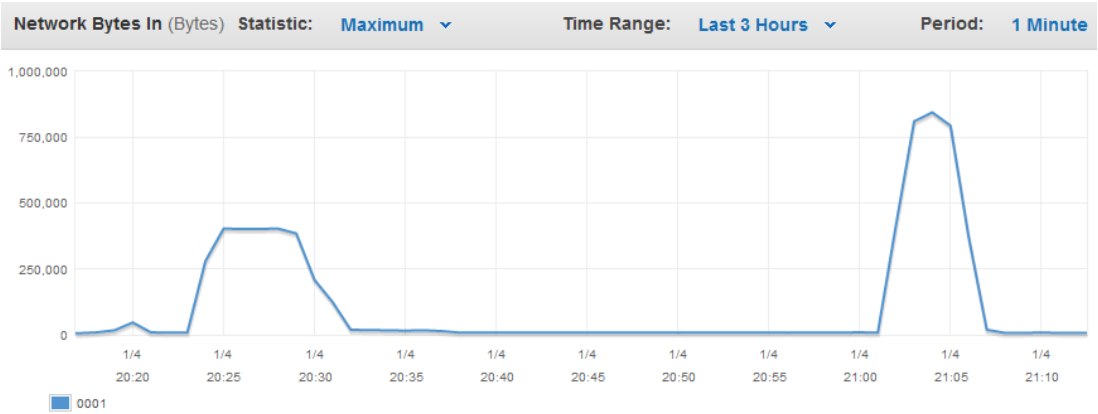
Nebyl zde zahrnut graf vytížení vnitřní paměti Redisu, jelikož ten byl téměř nulový a v průběhu testů se neměnil. Při testu této architektury nenastaly žádné chyby.

5.2.2 Výsledky testu - NoSQL

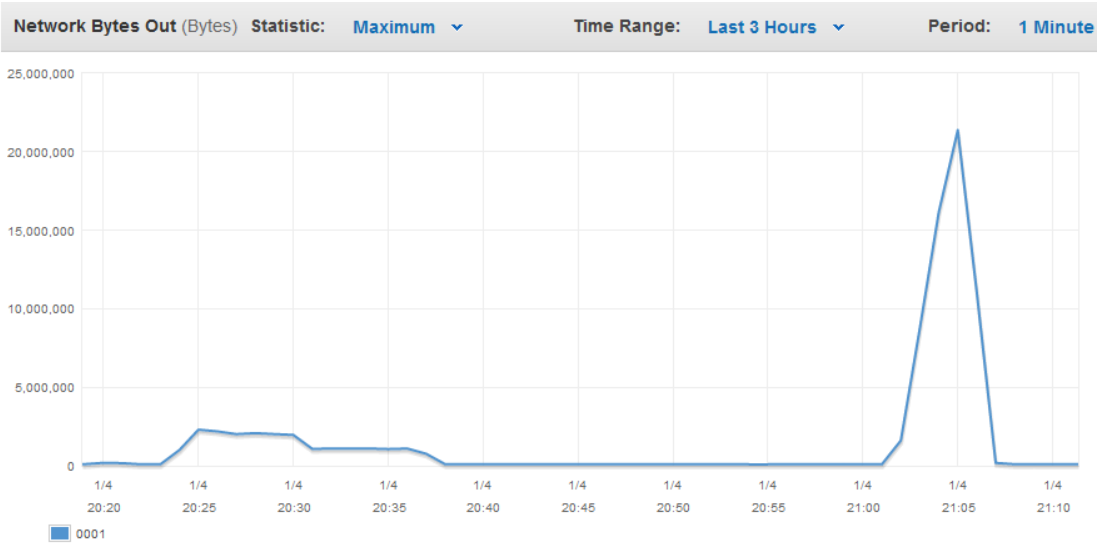
Druhou testovanou architekturou byl Redis v módu NoSQL databáze, kdy informace o založených a smazaných hrách jsou odesílány na server a ten zde udržuje kompletní seznam všech dostupných her. V daném časovém intervalu si instance tento seznam periodicky stahují a odesílají jej připojeným klientům.



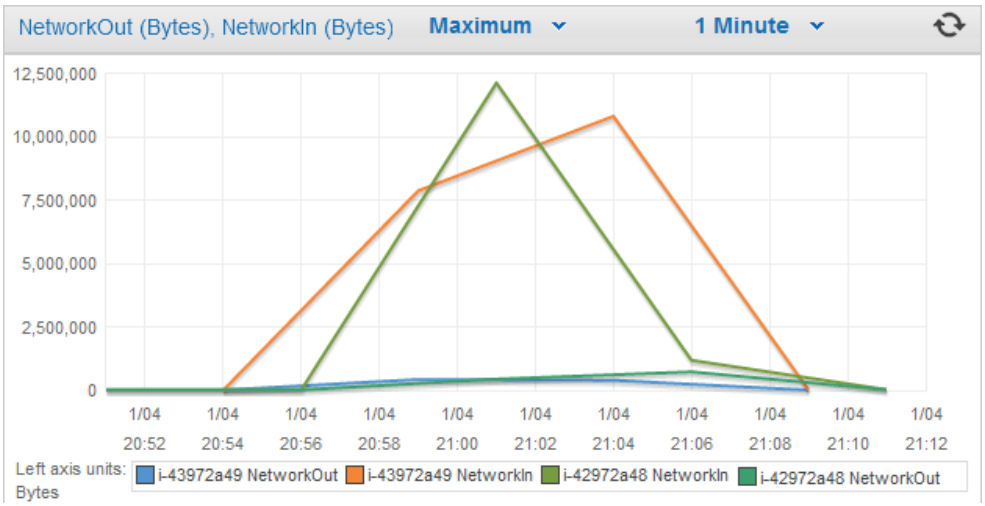
Obrázek 51 Test 2.2, CPU zátěž Redisu, 40x250 her



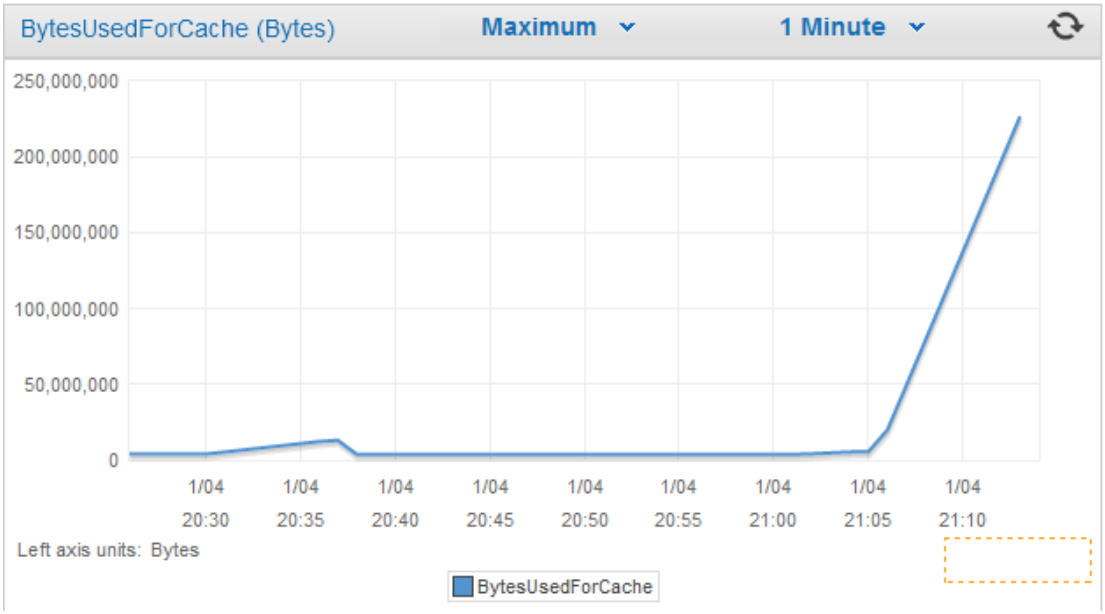
Obrázek 52 Test 2.2, Data In Redis server, test 20x50 a 40x250 her



Obrázek 53 Test 2.2, Data Out Redis server, test 20x50 a 40x250 her



Obrázek 54 Test 2.2, Data In/out na EC2 instancích, 40x250 her



Obrázek 55 Test 2.2, Velikost paměti využité pro uložení seznamu her, 40x250 her

Maximální vytížení CPU Redis serveru dosáhlo také maximálně 1%, což je poměrně překvapivé. Server udržoval poměrně velký seznam, který byl konstantně měněn. Předpoklad byl na vyšší vytížení CPU.

Objem přijímaných dat je menší než u předchozí architektury. Přibližně 400 KB/sec v testu 20x50 her, 850 KB/sec v testu 40x250 her. Objem příchozích dat byl předpokládán podobný jako u předchozí architektury, což se řádově potvrdilo. Jeho nižší objem může být způsoben kratší dobou trvání testů, jelikož na této architektuře nebyl jejich průběh naprosto bezproblémový.

V objemu odchozích dat je ovšem obrovský skok. Oproti přijímaným datům i oproti odchozím datům předchozí architektury vzrostl o jeden řád na maximálních 20 MB/sec v případě testu 40x250. To je nárůst opravdu značný a projevuje se zde velká redundance zasílaných dat.

Graf dat přijatých a odeslaných dat na EC2 instancích opět koresponduje s předchozími grafy datových toků na Redis serveru. Oproti minulé architektuře je zde ovšem zahrnut i graf využití interní paměti Redis serveru na obr. 55. Mezi časy 20:30 a 20:40 je vidět využití paměti pro test 20x50 her, které činilo přibližně 10 MB. Měření u testu 40x250 je zkresleno faktem, že po ukončení testu Redis dále z neznámého důvodu replikovat uložená data a docházelo k nárůstu jejich objemu. V době ukončení testu ve 21:08 ovšem jejich objem činil přibližně 80 MB. Tento nárůst relativně odpovídá předpokládanému nárůstu na desetinásobek vzhledem k desetinásobnému zvýšení počtu uložených her.

Při testování této architektury se bohužel objevila chyba. Běh vlákna skončil výjimkou v okamžiku kdy se pokoušelo stáhnout aktuální seznam her z Redis serveru. Je velmi pravděpodobné, že při tomto objemu stahovaných dat již přestávala stačit paměť Redisu vyhrazená pro vystavení dat ke stažení. O tomto možném problému se zmiňovala část 4.3.2. Test nicméně mohl běžet alespoň tak dlouho, aby poskytl relevantní data pro tuto architekturu.

5.2.3 Messaging vs. NoSQL

Z provedených testů obou architektur vychází poměrně jednoznačný vítěz, kterým je architektura Messagingu využívající PubSub kanál Redis serveru. Její předností je především malý objem přenášených dat v obou směrech a také téměř nulové nároky na vnitřní paměť Redisu. Management založených her na EC2 instancích je mírně složitější, ovšem na jejich zátěži se to neprojeví. V průběhu všech testů byly jejich CPU zatíženy na 100%. Ostatní statistiky obou architektur byly v průběhu testů srovnatelné.

Pokud by vadilo, že v této architektuře nově připojená instance nemá první minutu až dvě informaci o všech založených hrách, bylo by možné implementovat žádost o seznamy vlastních, kterou by nově spuštěná instance zaslala ostatním již běžícím. Jak bylo ovšem diskutováno v sekci 4.3.1, nemělo by k této situaci docházet příliš často a v běhu systému ani nijak nepřekáží.

5.3 Shrnutí testů

Aplikace a celkově celý systém splnit očekávání. V celkové zátěži bez problémů vydržel současnou práci 200 uživatelů a omezením se ukázaly možnosti testovacího prostředí. Podle výsledků, by neměl být pro jednu EC2 instanci problém, zvládnout zatížení až 1000 uživatelů. Programové klikače, navíc procházely aplikaci znatelně rychleji, než je schopen lidský uživatel. Také použitá konfigurace AWS systému je nejnižší možná. Na druhou stranu, aplikace v modulu arény například vůbec nekomunikuje s databází. Při nasazení v produkci budou arénové otázky a jejich řešení uloženo v DB a častá komunikace s databází bude nutností. Zvýší se také zátěž na instanci při jejich zpracování, jelikož datový objem některých z nich může být znatelně větší. Vliv těchto úprav ovšem bude nutné ověřit až při vývoji produkční verze.

Pro verzi aplikace v této práci by bylo vhodné ověření odhadnutého počtu 1000 uživatelů na instanci. To by ovšem vyžadovalo lepší naprogramování klikačů, aby byly schopné reagovat na zpoždění aplikace a případně se samy restartovat při zmíněné chybě, kdy je aplikace odmítne uznat jako přihlášené uživatele. Také by bylo nutné zapojit více strojů, aby bylo možné takový objem klikacích programů spustit.

Porovnání architektur pro sdílení založených her bylo také úspěchem, jelikož zjištěné rozdíly jsou opravdu markantní. V konečném důsledku mohou přinést i nezanedbatelné finanční úspory, jelikož bylo zmíněno, že i přenosy dat v cloudových službách jsou zpoplatněny. Použití messagingové architektury je zde evidentně lepší volbou a bude zvoleno pro produkční verzi aplikace. Objem možných sdílených dat na Redis serveru je také velice uspokojivý. Na limit 10 000 založených her se aplikace jen tak nedostane ani v produkční verzi.

Ve výsledku tedy je možné říci, že testy prokázaly požadovanou funkčnost aplikace a její dostačující výkon i pro další vývoj a konečné nasazení do produkce.

6 Závěr

Cílem práce byl návrh a implementace funkční multiplayer arény na cloudových technologiích. Část návrhu, například systém komunikace backendu a frontendu aplikace, vycházela z projektu Hravě. Také mnoho použitých technologií, včetně frameworků Spring a AngularJS bylo zvoleno kvůli kompatibilitě s tímto projektem. Pouze to by však ke splnění zadaných cílů nestačilo. Proto práce přinesla novou technologii websocketů pro skutečnou real-time komunikaci ve webových aplikacích. Ta umožnila dosud nedostupnou plynulou hru dvou lidských hráčů proti sobě. Realizace pomocí frameworku Atmosphere se sice neobešla bez mírných komplikací, ale ve výsledku je tento framework projektu velkým přínosem zjednodušujícím management komunikačních kanálů na stran serveru i klienta.

Dalším krokem bylo prozkoumání cloudových služeb společnosti Amazon, která byla zvolena poskytovatelem cloudu pro tuto práci. Přenos aplikace do tohoto distribuovaného prostředí vyžadoval postupné úpravy architektury aplikace a její spolupráce s narůstajícím počtem vnějších služeb. Nejnáročnějším, ale zároveň také velice zajímavým, se stal problém možného pohybu uživatelů mezi různými instancemi aplikace a výměna dat mezi instancemi obecně. Pro nalezení ideálního způsobu bylo navrženo více možností a až konečným testováním aplikace byl určen ten nejvhodnější.

Cloudové prostředí obecně přineslo daleko větší množství služeb a technologií, ze kterých je možné vybírat, než je běžné u single-server řešení webových aplikací. Na jednu stranu k nim dává jednoduchý přístup a za programátora udělá velké množství práce při jejich iniciálním nastavení a propojení s dalšími komponentami. Na druhou, trvá nějakou dobu, než se člověk naučí komplexní systém ovládat a hlubší změny konfigurací, případně doinstalování chybějících služeb, je mnohdy daleko náročnějším procesem, než na běžném aplikačním serveru. Autor této práce se například naučil vzdáleně instalovat a konfigurovat Jetty server na unixové konzoli, jelikož mezi předinstalovanými aplikačními servery nabízenými Amazonem nebyl k dispozici.

Celý systém byl nakonec úspěšně navržen i implementován, o čemž svědčí provedené zátěžové testy. Podle nich by její současná verze v neměla mít problém se zvládnutím stovek až tisíců uživatelů na jednu instanci aplikačního serveru a to i při provozu na nejnižší možné konfiguraci Amazonu. Vývoj aplikace zde však zdaleka nekončí. Jak bylo řečeno v úvodu, je plánována její implementace do projektu Hravě, jako plnohodnotné multiplayer arény. To bude obnášet jednak množství grafických změn, ale také dodání funkcionalit pro lepší uživatelský zážitek jako je globální chat, vyzývání přátel na souboje, zužitkování odměn získaných v e-learningu Hravě a mnoho dalších. Tato práce tedy slouží jako koncept a průkopník nových technologií, na kterých bude dále stavět tým projektu Hravě v čele s autorem práce.

Webovou aplikaci, popsanou v této práci je k 5.1. 2015 možno najít na adrese:
http://aws-arena-http-sticky-lb-607249376.us-west-2.elb.amazonaws.com/aws-arena/

Literatura

[1] Simon Maple. *The Great Java Application Server Debate with Tomcat, JBoss, GlassFish, Jetty and Liberty Profile*. URL: http://zeroturnaround.com/rebellabs/the-great-java-application-server-debate-with-tomcat-jboss-glassfish-jetty-and-liberty-profile/ (cit. 12. 12. 2014).

[2] Webtide. *Why Choose Jetty*. URL: https://webtide.com/why-choose-jetty/ (cit. 12. 12. 2014).

[3] Google Brat Tech LLC. *AngulraJS main page*. URL: http://angularjs.org/ (cit. 12. 12. 2014).

[4] Frans van Buul. *Web forms with Java: AngularJS and other approaches*. URL: http://blog.trifork.com/2014/03/20/web-forms-with-java-angularjs-and-other-approaches/ (cit. 12. 12. 2014).

[5] IEC ISO. *ISO/IEC 7498-1. Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*. Second edition. 1994. Kap. 6. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip (cit. 13. 12. 2014).

[6] Alexis Deveria. *Can I use - web service*. URL: http://caniuse.com/#search=websockets (cit. 13. 12. 2014).

[7] *Socket.io*. URL: http://socket.io/ (cit. 14. 12. 2014).

[8] JeanFrancois Arcand. *Atmosphere*. URL: https://github.com/Atmosphere/atmosphere/wiki (cit. 14. 12. 2014).

[9] JeanFrancois Arcand. *Installing AtmosphereServlet with or without native support*. URL: https://github.com/Atmosphere/atmosphere/wiki/Installing-AtmosphereServlet-with-or-without-native-support (cit. 14. 12. 2014).

[10] JeanFrancois Arcand. *Understanding the Atmosphere Framework*. URL: http://async-io.org/tutorial.html (cit. 14. 12. 2014).

[11] JeanFrancois Arcand. *Atmosphere JQuery API*. URL: https://github.com/Atmosphere/atmosphere/wiki/jQuery.atmosphere.js-atmosphere.js-API (cit. 15. 12. 2014).

[12] Veronika Nosková. "Specifika přechodu do cloudového prostředí". Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2013. Kap. 1.

[13] Michiel Van Vlaardingen. *Amazon RDS vs DIY MySQL on EC2 Benchmark*. URL: http://blog.observu.com/2011/05/rds-vs-mysql-on-ec2-benchmark/ (cit. 19. 12. 2014).

[14] Peter Wayner. *Ultimate cloud speed tests: Amazon vs. Google vs. Windows Azure*. URL: http://www.infoworld.com/article/2610403/cloud-computing/ultimate-cloud-speed-tests--amazon-vs--google-vs--windows-azure.html (cit. 17. 12. 2014).

[15] Peter Wayner. *Cloud review: 8 public cloud services put to the test*. URL: http://www.infoworld.com/resources/16303/cloud-computing/cloud-review-8-public-cloud-services-put-to-the-te (cit. 17. 12. 2014).

[16] *AWS Free Tier*. URL: http://aws.amazon.com/free/ (cit. 27. 12. 2014).

[17] *Selenium WebDriver*. URL: http://www.seleniumhq.org/projects/webdriver/ (cit. 29. 12. 2014).

Příloha A

Obsah přiloženého CD

- **PDF** - Verze této práce ve formátu PDF
- **Source** - Zdrojové kódy aplikace
- **Test** - Zdrojové kódy a spustitelný soubor testovacího programu
- **Tex** - Verze této práce ve formátu LaTeX
- **Tex/Pictures** - Veškeré obrázky použité v této práci

Příloha B

Seznam použitých zkratek

Tento seznam obsahuje pouze zkratky, které se vyskytly v textu práce a jejich význam byl vysvětlen. Některé zkratky běžně používaných technologií zde uvedeny nejsou, jelikož i text práce předpokládá znalost jejich významu (HTML, CSS, URL ...)

AJAX	Asynchronous JavaScript and XML
AMI	Amazon Machine Image
AWS	Amazon Web Services
CORS	Cross-origin resource sharing
DOM	Document Object Model
EC2	Elastic Compute Cloud
ELB	Elastic Load Balancer
JMS	Java Messaging Service
JS	JavaScript
JSF	JavaServer Faces
JSP	JavaServer Pages
MVC	Model-View-Controller
RDS	Relational Database Service
REST	Representational State Transfer
SSE	Server Sent Events
TCP	Transmission Control Protocol
UC	UseCase
WS	Websocket