Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta

Adaptivní prioritní fronty

Bakalářská práce

Ondřej Melichar

Vedoucí práce: Ing. Václav Novák, CSc.

České Budějovice 2014

Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta

ZADÁVACÍ PROTOKOL BAKALÁŘSKÉ PRÁCE

Student: **Ondřej Melichar**
(jméno, příjmení, tituly)

Obor – zaměření studia: 1801R001 / Aplikovaná informatika

Katedra/ústav, kde bude práce vypracovávána: Ústav aplikované informatiky

Školitel: Ing. Václav Novák, CSc., vacnovak@prf.jcu.cz, M: 606 666 694
(jméno, příjmení, tituly, u externího š. název a adresa pracoviště, telefon, fax, e-mail)

Garant z PřF:
....
(jméno, příjmení, tituly, katedra – jen v případě externího školitele)

Školitel – specialista, konzultant: ....
(jméno, příjmení, tituly, u externího š. název a adresa pracoviště, telefon, fax, e-mail)

Téma bakalářské práce: Adaptivní prioritní fronty

Cíle práce:

Řešený zahlcení daty v malých systémech je v poslední době hojně diskutována. Řešeních je celá řada, od posílení výkonu až k postupnému dlouhodobému zpracování příchozích zpráv. Nastává problém s prioritou zpracování událostí zejména tehdy, když se mění scénáře priorit v čase.

V oblasti sportu, například pro maratony a městské běhy je charakteristický vysoký počet účastníků a hostů. To vytváří tlak na toky dat mezi datovými systémy vedoucí až k zahlcení. Tím mohou informace pro zdravotníky uvíznou ve frontě dat. Je tedy komplikované řídit komunikaci s nadměrnými toky dat ze servu k zájemcům o výsledky závodů a zároveň akceptovat některé požadavky na informace v reálném čase ve směru klient server, o prioritách pro hasiče či zdravotníky nemluvě. To je vlastně praktický úkol pro studenta.

1. Navrhněte a implementujte adaptivní prioritní systém rozhodování podle důležitosti sdělení a dalších uživatelských nastavení, adaptující se v čase podle scénáře priorit a momentálního stavu zahlcení v síti.
2. Realizujte on-line řešení informačního systému umožňující distribuovat data z centra pro jednotlivé žadatele s rozlišením, zda se jedná o diváky, organizátory, závodníky a ostatní uživatelsky definované skupiny.

3. Přijímat on-line do centra informace z rozlišením jednotlivých skupin. Zejména vyhledávat zprávy s maximální prioritou.

Aplikaci otestujte na kombinaci Windows server a Android nebo Windows Mobile klient.

Základní doporučená literatura:

- Javaalgoritmy.wz.cz: ADT. Javaalgoritmy.wz.cz: Implementace ADT prioritní fronta [online]. 1. vyd. [cit. 2014-01-19]. Dostupné z: http://javaalgoritmy.wz.cz/pfronta.htm
- Dynamické datové struktury.: Seznam. Fronta. Zásobník. Strom. Dynamické datové struktury. [online]. 2014 [cit. 2014-01-20]. Dostupné z: http://web.natur.cuni.cz/~bayertom/Prog2/prog2_2.pdf

Financování práce: ....

Vedoucí práce: Ing. Václav Novák, CSc. .... podpis: ....

U externích vedoucích fakultní garant práce: .... podpis: ....

Garant oboru bak. studia, pokud je obor zajišťován jinou katedrou/ústavem, než ze které je školitel (nepožaduje se u oboru biologie): .... podpis: ....

Vedoucí katedry: RNDr. Libor Dostálek podpis: ....

Případný souhlas vedoucího ústavu AV: .... podpis: ....

V Českých Budějovicích dne 31.1.2014 Podpis studenta: ....

Bibliografické údaje

Melichar O., 2014: Adaptivní prioritní fronty.
[Adaptive priority queues. Bc. Thesis, in Czech.] – 26 p., Faculty of Science, The University of South Bohemia, České Budějovice, Czech Republic.

Anotace

Tato bakalářská práce se zabývá řešením zahlcení daty v malých systémech. V oblasti sportu, například pro maratony a městské běhy je charakteristický vysoký počet účastníků a hostů. To vytváří tlak na toky dat mezi datovými systémy vedoucí až k zahlcení. Tím mohou informace pro zdravotníky uvíznout ve frontě dat. Cílem práce je řídit komunikaci s nadměrnými toky dat ze serveru k zájemcům o výsledky závodů a zároveň akceptovat některé požadavky na informace v reálném čase ve směru klient server, o prioritách pro hasiče či zdravotníky nemluvě.

In English

This bachelor thesis concerns a solution of a data congestion in small systems. It is typical for sports, for example marathons and urban runs are characterised with a large number of participants and guests. This creates a pressure on a data flow between data systems and it leads to the congestion. The information for paramedics can thereby stuck in a data queue. The aim of the work is to control a communication with the excessive data flow from a server to takers of races results and at the same time to accept some requirements for information in realtime in the direction client server, not speaking of priorities for firemen or paramedics.

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.
Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích, dne 12.12. 2014

Podpis

Poděkování

Rád bych poděkoval panu Ing. Václavu Novákovi, CSc. za ochotu a pomoc při odborných konzultacích, které přispěly k uskutečnění této bakalářské práce.

Obsah

1 Úvod....1
1.1 Cíle práce....1
1.2 Postup realizace cílů....2
2 Výběr vhodné dynamické datové struktury....2
2.1 Dynamické datové struktury....3
2.2 Binární halda....4
3 Návrh aplikace....7
3.1 Požadavky na systém....7
3.2 Návrh scénářů priorit....8
3.3 Use-case diagram....8
3.4 Architektura systému....9
4 Implementace....11
4.1 Databáze....11
4.2 Webový server....12
4.3 Adaptivní prioritní fronta....14
4.4 Robot....19
4.5 Klientská aplikace....21
5 Testování systému....24
5.1 Testování klientské aplikace....24
5.2 Testování adaptivní prioritní fronty....25
6 Závěr....26
7 Seznam použité literatury....27
8 Seznam obrázků....28
9 Seznam tabulek....28
10 Seznam příkladů....28

1 Úvod

Problém zahlcení daty malých systémů je v poslední době hojně diskutován. Řešeních je celá řada. Od posílení výkonu až k postupnému dlouhodobému zpracování příchozích zpráv. Nastává problém s prioritou zpracování událostí zejména tehdy, když se mění scénáře priorit v čase.

V oblasti sportu, například pro maratony a městské běhy, je charakteristický vysoký počet účastníků a hostů. To vytváří tlak na toky dat mezi datovými systémy vedoucí až k zahlcení. Tím mohou informace pro zdravotníky uvíznout ve frontě dat. Je tedy komplikované řídit komunikaci s nadměrnými toky dat ze serveru k zájemcům o výsledky závodů a zároveň akceptovat některé požadavky na informace v reálném čase ve směru klient server, o prioritách pro hasiče či zdravotníky nemluvě.

Tato bakalářská práce se zabývá dalším možným řešením zahlcení malých systémů daty, a to návrhem a implementací adaptivního prioritního systému rozhodování podle důležitosti sdělení, adaptující se v čase podle scénáře priorit a momentálního stavu zahlcení v síti. Server s tímto systémem rozhodování odpovídá přednostně uživatelům s momentálně nejvyšší prioritou. Řešení by tedy mělo odstranit problém uvíznutí důležitých požadavků ve frontě dat při zahlceném serveru.

Pro otestování rozhodovacího systému je další motivací této práce vytvořit informační systém obsahující data z oblasti sportu pro jednotlivé žadatele. Uživatelé systému zobrazují výsledky sportovních událostí pomocí klientské aplikace, která běží na mobilním zařízení s operačním systémem Android.

1.1 Cíle práce

1. Navrhnout a implementovat adaptivní prioritní systém rozhodování podle důležitosti sdělení, adaptující se v čase podle scénáře priorit a momentálního stavu zahlcení v síti.
2. Realizovat on-line řešení informačního systému umožňující distribuovat data z centra pro jednotlivé žadatele s rozlišením, zda se jedná o diváky, organizátory, závodníky a ostatní uživatelsky definované skupiny.
3. Přijímat on-line do centra informace s rozlišením jednotlivých skupin. Zejména

vyhledávat zprávy s maximální prioritou.

4. Aplikaci otestovat na kombinaci Windows server a Android

1.2 Postup realizace cílů

Realizace hlavního cíle, navrhnutí adaptivního prioritního systému rozhodování, se skládá z několika částí:

- Vytvořit frontu, do které se ukládají požadavky od uživatelů informačního systému a zároveň skupina (role), do které uživatel patří
- Třídit požadavky ve frontě pomocí algoritmů vhodné dynamické datové struktury s ohledem na aktuální scénář priorit pro jednotlivé požadavky
- Vytvořit softwarový webový server, který bude ukládat požadavky do fronty a zároveň odpovídat na požadavky, které budou na vrcholu této fronty
- V průběhu času měnit scénáře priorit pro jednotlivé skupiny uživatelů, na základě zahlcení webového serveru požadavky a aktuálního stavu systému

V rámci této práce je realizována klientská aplikace, která slouží uživatelům informačního systému k získávání výsledků závodů, souřadnic nehod a dalších dat uložených v datovém centru. Postup realizace této aplikace je následující:

- Aplikaci vytvořit, aby běžela na mobilních zařízeních s operačním systémem Android
- Uživatelé mohou mít jednu z pěti definovaných rolí
- Aplikace má sportovní tématiku, především sportovní události s vysokým počtem účastníků jako jsou městské běhy.
- Uživatel bude moci vykonávat řadu akcí, které jsou popsány v požadavcích na systém (v praktické části práce)

2 Výběr vhodné dynamické datové struktury

Po stanovení postupu realizace cílů bakalářské práce byl proveden sběr informací. Cílem bylo vybrat vhodnou dynamickou datovou strukturu. Podle algoritmů této struktury se ukládají a zároveň třídí požadavky od uživatelů ve frontě. Bylo nutné vybrat strukturu, která funguje jako prioritní fronta. Klasická fronta je seznam prvků seřazených podle času příchodu. Příklad takové fronty si lze představit jako frontu nakupujících v obchodě. Do fronty přicházejí nový nakupující, a tak se fronta plní. Zároveň z fronty nakupující odcházejí a fronta ubývá. Kdo přijde do fronty dříve, také dříve odejde [1]. Oproti tomu prioritní fronta je seznam prvků seřazený podle priorit. Z reálného světa si lze takový typ fronty představit jako frontu lidí, kde někteří předbíhají ostatní.

V této práci je místo klasického spojového seznamu nebo pole pro prvky prioritní fronty vytvořena samostatná databázová tabulka, kde každý záznam tabulky reprezentuje prvek fronty. První záznam v tabulce je kořenový prvek, tedy prvek s nejvyšší prioritou a poslední záznam v tabulce reprezentuje prvek s nejnižší prioritou ve frontě. Tato práce používá pro ukládání požadavků databázi, protože nad frontou požadavků se bude v průběhu času provádět velký počet operací. Například vkládání a následné třídění, společně s odebíráním a následným tříděním. Zároveň bude v programu přistupovat do fronty více komponent. Kvůli těmto faktorům by při použití klasického seznamu nebo pole nastal problém se správným pořadím prvků a mohlo by dojít k narušení integrity dat. Databázový systém pro zajištění integrity dat musí zaručovat – a*tomičnost*, *konzistenci*, *izolovanost*, *trvalost* [2]. Za pomoci těchto vlastností bude mít fronta vždy správnou strukturu.

2.1 Dynamické datové struktury

Jak už bylo napsáno výše, k realizaci adaptivního prioritního systému bylo nutné vybrat dynamickou datovou strukturu, která funguje jako prioritní fronta. Text této [3] publikace, představuje realizaci prioritní fronty pomocí uspořádaného pole, neuspořádaného pole, uspořádaného seznamu a neuspořádaného seznamu. Pro splnění hlavních cílů této bakalářské práce nebyly tyto datové struktury vhodné z důvodu vysoké časové složitosti při vkládání do uspořádaného seznamu nebo při odebírání z neuspořádaného seznamu. Případ, kdy algoritmus musí procházet celou dynamickou datovou strukturou je příliš zdlouhavý. Tabulku znázorňující časové složitosti jednotlivých operací všech čtyř zmíněných

dynamických datových struktur lze vidět v tabulce 1.

	vlož	vyber největší (nejmenší) prvek	najdi největší (nejmenší) prvek
uspořádané pole	N	1	1
neuspořádané pole	1	N	N
uspořádaný seznam	N	1	1
neuspořádaný seznam	1	N	N

Tabulka 1: Časové složitosti operací – vkládání a odebírání z fronty, realizace polem a seznamem [3]

2.2 Binární halda

Další dynamická datová struktura, která byla zkoumána se nazývá binární halda. Binární halda je binární strom, kde platí, že každý potomek vrcholku má nižší (vyšší) nebo stejnou hodnotu nežli vrcholek sám [4]. Binární halda se chová jako prioritní fronta, protože vlastnost býti haldou je rekurzivní – všechny podstromy haldy jsou také haldy [4]. Na vrcholek celé haldy musí vždy vystoupit prvek s nejvyšší prioritou. Této vlastnosti se využívá ve velmi efektivním řadícím algoritmu nazývaném *heapsort*, založeném na porovnávání dvou prvků.

V textu této publikace [5] stojí, že implementace prioritní fronty pomocí seznamu není nejlepším řešením, a to z důvodu vysoké časové složitosti při vkládání do fronty - *O(n)* a při třídění - *O(n log n)*. Text dodává, že klasický způsob implementace je pomocí binární haldy, protože při vkládání i odebírání prvku z fronty je časová složitost – *O(log n)*. V další části publikace je zmíněn zajímavý rozdíl mezi vizualizací haldy a reálnou implementací. Při vizualizaci halda vypadá jako strom, zatímco při implementaci je halda realizována klasickým seznamem prvků (indexovaným). Pro realizaci adaptivního prioritního systému se na základě zjištěných poznatků rozhodl autor této bakalářské práce použít dynamickou datovou strukturu halda.

2.2.1 Operace s binární haldou

Binární halda má tyto základní operace:

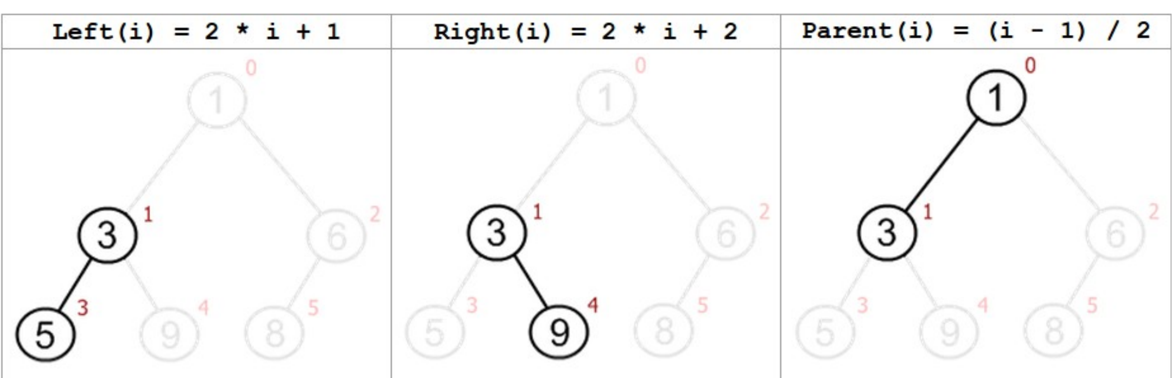
- *Repair top* - pokud kořen haldy nerespektuje správné uspořádání, tato operace přesune kořen na správnou pozici. Algoritmus operace porovnává uzel s jeho potomky, a jestliže má jeden z potomků vyšší prioritu, tak se oba uzly prohodí.

Pokud mají oba potomci vyšší prioritu, prohodí se uzel s potomkem s nejvyšší prioritou. Algoritmus postupuje interaktivně o úroveň níže, dokud nenarazí na poslední uzel nebo pokud bude mít uzel vyšší prioritu než jeho potomci.

- *Heapify* - pomocná operace, která zkonstruuje v zadaném poli haldu. Postupuje po vrcholech všech podhald a volá se operace *Repair top*. Po ukončení všech iterací reprezentuje dané pole haldu.
- *Enqueue* – tato operace je přesným opakem operace *Repair top*. Prvek se přidává na konec haldy, a tak probublává nahoru do té doby, než má nižší prioritu než jeho rodič. Nebo dokud se neprohodí s kořenovým prvkem a dostane se tak na vrchol haldy.
- *Top* - vrátí hodnotu prvku s nejvyšší prioritou.
- *Dequeue* - vrátí prvek s nejvyšší prioritou a odstraní ho z haldy. Smazání provede tak, že nahradí vrchol posledním prvek haldy a zavolá operaci *Repair top* na aktuální vrchol haldy.
- *Merge* - sloučí dvě haldy do jedné. Vytvoří nové pole, do něhož nakopíruje obsah obou hald a na toto nové pole zavolá operaci *Heapify*.

2.2.2 Mapování haldy na tabulku

Podle většiny zdrojů se prvky binární haldy ukládají buď do seznamu, nebo do pole. Například zde [6] je podrobně popsáno jak ukládat prvky binární haldy do indexovaného pole, kde prvek na indexu nula reprezentuje kořen haldy. Každý vrchol haldy na indexu i má levého potomka na indexu $2 * i + 1$ a pravého potomka na indexu $2 * i + 2$. Index rodiče prvku na indexu i se vypočítá podle vzorce $(i – 1) / 2$. Ukázkové příklady výpočtu vrcholů pomocí těchto vzorců na binární haldě lze vidět na obrázku 1.



Obrázek 1: Aplikace vzorců pro výpočet potomků a rodiče [6]

V této práci se prvky binární haldy ukládají do tabulky v databázi. Princip mapování jednotlivých prvků je stejný jako v případě indexovaného pole od nuly. Tabulka obsahuje záznamy v databázi, kde první záznam má index nula a poslední záznam má index podle vzorce ***celkový počet prvků – 1***. Vzorce pro počítání potomků a rodiče daného vrcholu jsou také stejné. Záznam této tabulky bude obsahovat více atributů a nebude obsahovat přímo číslo reprezentující prioritu daného prvku, ale roli, ze které se bude při operacích nad haldou zjišťovat aktuální priorita. Tato práce tedy ukazuje komplexnější řešení haldy jako prioritní fronty, kde každý prvek v haldě může v průběhu času měnit svou prioritu na základě vnějších okolností.

3 Návrh aplikace

Tato kapitola se zabývá návrhem praktické části práce. Celé řešení je rozděleno na serverovou a klientskou část. Nejprve byly vytvořeny podrobné požadavky na celkovou funkcionalitu systému. To znamená požadavky jak na klientskou aplikaci, tak na program běžící na serveru. Návrh klientské aplikace je realizován pomocí jazyku UML. Aplikace na straně klienta obsahuje pět uživatelských rolí a je popsána use-case diagramem.

3.1 Požadavky na systém

Na základě cílů této práce byly vytvořeny optimální požadavky na klientskou aplikaci, aby bylo možné při testování zjistit všechny výhody i nevýhody adaptivní prioritní fronty. Základem aplikace je pět uživatelských rolí. Jednotlivé role mají přiřazenou v databázi prioritu. Na základě této priority se na serveru určuje, na který klientův požadavek se odpoví dříve. Nepřihlášený uživatel má roli diváka. Tato role může zobrazit seznam závodů podle roku, a zobrazit aktuální výsledky daného závodu. Závodník se již musí přihlásit do systému a oproti divákovi může navíc zobrazit závody, kterých se sám zúčastnil. Také má možnost zobrazit své aktuální výsledky ve vybraném závodě. Uživatelská role zdravotník má možnost v systému zobrazit geografické souřadnice zraněných závodníků. Hasič může zobrazit totéž co zdravotník a zároveň souřadnice všech požárů v daném závodě. Poslední uživatelská role je organizátor (viz níže).

Další požadavek je na straně serveru vytvořit program, který obsahuje několik důležitých částí. První z nich je softwarový webový server, který má za úkol přijímat http požadavky od uživatelů klientské aplikace a tyto požadavky uložit do adaptivní prioritní fronty. Zároveň má za úkol odebírat požadavky s nejvyšší prioritou a odpovídat na ně klientům, kteří dané požadavky odeslali. Webový server ukládá a odebírá požadavky pomocí komponenty s názvem Adaptivní prioritní fronta, která poskytuje algoritmy přidávání, odebírání a třídění prvků ve frontě na základě aktuálně stanovených priorit jednotlivých rolí. Tyto algoritmy jsou popsané v teoretické části této bakalářské práce. Jedná se o algoritmy dynamické datové struktury halda.

Adaptivní prioritní fronta musí přidávat, odebírat a třídit prvky ve frontě na základě aktuálního scénáře priorit. Tento scénář priorit mohou v systému měnit dvě komponenty. První z nich je komponenta nazývající se Robot, která každých deset sekund kontroluje míru

zahlcení serveru požadavky, tedy celkový počet požadavků ve frontě. Podle tohoto počtu mění scénáře priorit pro jednotlivé role. Další komponentou, která má možnost měnit scénáře priorit, je uživatel s rolí organizátor z aplikace na straně klienta. Tato role má volbu změny stavu systému. Jestliže je stav systému nastaven organizátorem na hodnotu *OK*, bude moci komponenta Robot měnit scénáře priorit na základě míry zahlcení systému. Další možné hodnoty jsou *požár* a *zranění*. Při těchto hodnotách stavu systému již nemůže Robot měnit scénáře priorit. Pokud je stav systému nastaven na *požár,* má nejvyšší prioritu role hasič, ostatní role mají oproti hasiči velmi malou prioritu. Podobná situace nastává, jestliže je stav systému nastaven na hodnotu *zranění*. V tomto případě má nejvyšší prioritu zdravotník.

3.2 Návrh scénářů priorit

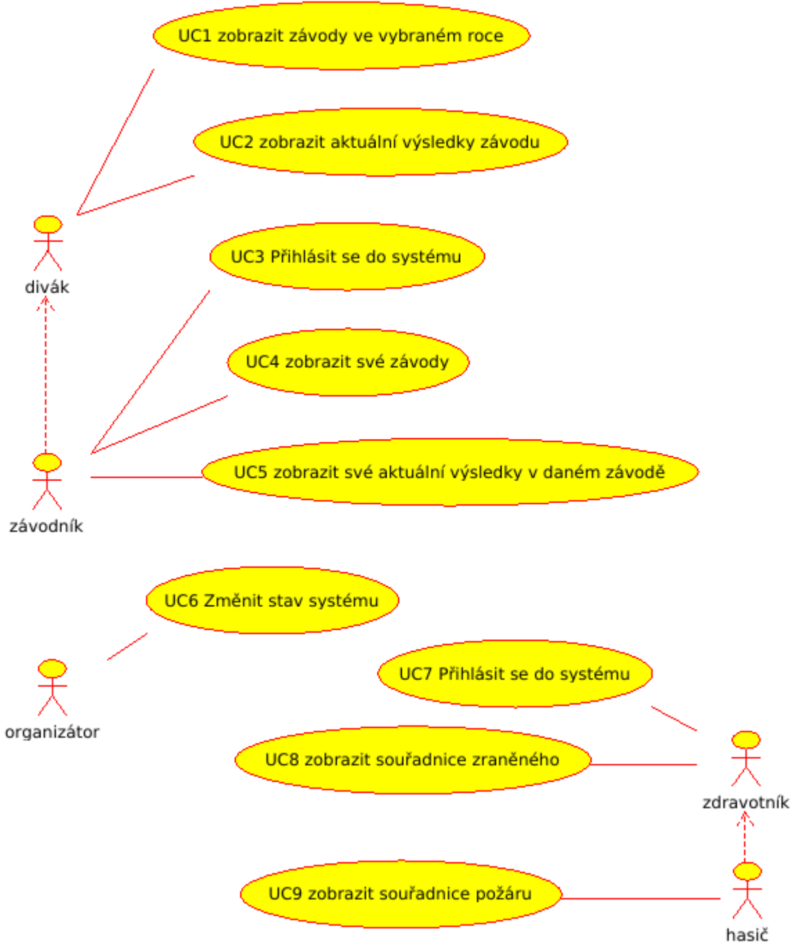
Komponenta Robot mění scénáře priorit podle celkového počtu požadavků ve frontě v případě, že je stav systému nastaven na hodnotu *OK*. Jestliže je stav systému změněn organizátorem na hodnotu *požár* nebo *zranění*, Robot priority měnit nemůže. Pro splnění cílů této práce bylo vytvořeno šest scénářů priorit. Tyto scénáře byly navrženy tak, aby odstranily problém uvíznutí důležitých http požadavků ve frontě. Pro různé úrovně zahlcení systému jsou zvoleny jiné scénáře. Při velkém zahlcení systému požadavky jako je například 110 požadavků ve frontě, budou mít nejvyšší prioritu požadavky, které odeslala role hasič. Robot a organizátor mění scénáře priorit podle tabulky 2.

stav systému	OK	OK	OK	OK	požár	zranění
počet požadavků ve frontě	do 29	30 – 59	60 – 99	100 a více	---	---
divák	5	5	5	5	5	5
závodník	4	4	4	4	5	5
zdravotník	3	2	1	2	4	1
hasič	2	2	1	1	1	4
organizátor	1	1	3	3	5	5
míra zahlcení	žádná	malá	střední	vysoká	---	---

Tabulka 2: Scénáře priorit pro Robota a uživatele organizátor

3.3 Use-case diagram

Use-case diagram znázorňuje požadavky na funkcionalitu systému z pohledu uživatele. V diagramu jsou zobrazeny případy užití (use case). Každý případ užití definuje jednu funkcionalitu, kterou by měl systém umět [7].



Obrázek 2: Use-case diagram klientské aplikace

3.4 Architektura systému

Na základě požadavků na systém byla navržena architektura celého systému. Jde o klasickou síťovou architekturu klient-server, kde celá serverová část je umístěna na jediném počítači a klientská část je umístěna na mobilním zařízení s operačním systémem Android. Komunikace mezi klientem a serverem probíhá síťovým protokolem http.

Na Windows serveru je vytvořen program v programovacím jazyce C#. Program obsahuje komponenty Webový server, Adaptivní prioritní frontu a Robota, jejichž funkcionalita je popsána v kapitole 3.1 Požadavky na systém. Program běží na stroji nepřetržitě. Na serverové části je pro ukládání dat umístěna MySQL databáze na MySQL serveru. Vizualizace architektury systému je vidět na obrázku 3.

Obrázek 3: Architektura systému

4 Implementace

Tato část bakalářské práce zahrnuje realizaci celého systému. Nejprve byla vytvořena serverová část, kde byla po navržení vytvořena databázová struktura umístěná na MySQL serveru. Následovala implementace programu v jazyce C#, která zahrnovala implementaci webového serveru s adaptivním prioritním systémem rozhodování. Nakonec byla vytvořena klientská aplikace v jazyce Java, která běží na operačním systému Android.

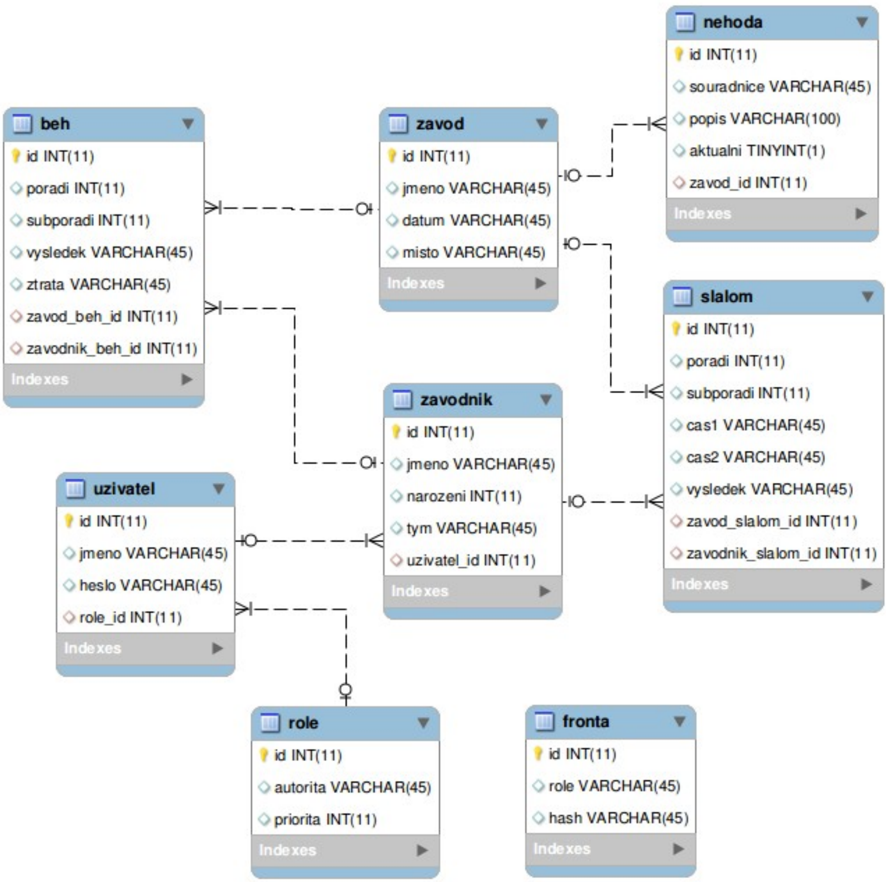
4.1 Databáze

Bylo nutné vybrat rychlou a spolehlivou databázovou strukturu. Proto byla zvolena databáze MySQL díky výhodám uvedeným v této publikaci [8].

- multiplatformní
- vyšší výkon oproti ostatním databázovým strukturám
- bez velkých ztrát výkonu je možnost pojmout velké množství dat
- využití zdarma pro nekomerční účely
- možnost zásahu do zdrojového kódu

4.1.1 Návrh databázové struktury

Pro splnění cílů této bakalářské práce byl vytvořen vhodný návrh databázové struktury, který je na obrázku 4. Závod může obsahovat více závodníků a závodník může být ve více závodech. Vazba mezi těmito entitami je M:N. V jednom závodě může být více nehod. Tabulka *role* představuje scénář priorit, ze kterého se zjišťuje aktuální priorita určité role. Http požadavky od klientů se ukládají do tabulky *fronta*.



Obrázek 4: Návrh databázové struktury

4.1.2 Pomocná třída pro připojení a práci s databází

Byla vytvořena samostatná třída s názvem *DBPripojeni*. Na serveru tuto třídu používají všechny komponenty pro připojení a práci s databází. Ve většině případů je dobré oddělit logiku připojení a práce s databází od ostatních tříd a zapouzdřit tuto logiku do samostatné třídy. Ve zdrojovém kódu to má za následek čistší, znovupoužitelný a čitelný kód [9]. Třída obsahuje řadu pomocných metod pro další třídy. Adaptivní prioritní fronta využívá metod této třídy k třídění, přidávání a odebírání prvků (v tabulce v databázi). Robot zase využívá této třídy ke změnám scénářů priorit, které jsou umístěny v databázi.

4.2 Webový server

Webový server byl vytvořen jako třída napsaná v jazyce C#. V parametru konstruktoru této třídy je uveden formát url v textové podobě. Tento formát url říká, jakým požadavkům bude server naslouchat a dále je ukládat do adaptivní prioritní fronty. Pro testovací účely byl v programu použit formát *'http://*:12345/'*. Do adaptivní prioritní fronty se tedy ukládají pouze http požadavky, které mají v url adrese protokol *http* a port *12345*. Dále je na serveru

realizován validátor, který kontroluje, zda má url adresa za portem řetězec *'/results'*. Ukázka validní url adresy je v příkladu 1.

Každý http požadavek, který webový server odchytí, je v aplikaci reprezentován jako instance třídy *HttpListenerContext*. Pomocí této instance může webový server získat přímo instanci třídy *HttpListenerRequest*, která reprezentuje příchozí http požadavek [10] (obsahuje všechny jeho náležitosti, jako je url, http metoda atd.). Webový server po přijetí http požadavku vytvoří nové vlákno, ve kterém tento požadavek uloží do fronty požadavků. Proces uložení http požadavku do fronty začíná získáním role z url adresy (viz příklad 1). Následně se vytvoří klíč, který bude sloužit jako odkaz na instanci třídy *HttpListenerContext*, která bude uložena v kolekci datového typu *Dictionary*. Do fronty v databázi se vždy uloží hodnoty role a klíč (hash) , jejichž kombinace reprezentuje jeden http požadavek.

```
http://localhost:12345/results?role=hasic
```

Příklad 1: Ukázka url adresy obsažené v http požadavku, který odeslal uživatel role hasič

Webový server přidává http požadavky na vrchol fronty pomocí třídy *AdaptivniPrioritniFronta*, která obsahuje algoritmy založené na operacích dynamické datové struktury halda. Webový server při vkládání požadavku do fronty volá pomocnou metodu s názvem *Enqueue*. Algoritmus této metody byl vysvětlen v teoretické části této práce. Metoda bere jako parametr instanci třídy *HttpListenerContext* (viz příklad 2).

```
fronta.Enqueue(context);
```

Příklad 2: Volání metody Enqueue

Princip odpovídání na http požadavky je realizován na webovém serveru jako jedno nepřetržitě běžící vlákno, které odebírá http požadavky z fronty a následně tyto požadavky zpracovává. Samotné odebrání požadavku z fronty je realizováno metodou *Dequeue* (příklad 3). Tato metoda vrací instanci *HttpListenerContext*, ze které webový server získá instanci třídy *HttpListenerResponse*. Tato instance se při zpracovávání naplní odpovídajícími daty pro uživatele a následně se pomocí metody *Close* této instance odešle odpověď odpovídajícímu koncovému zařízení s klientskou aplikací, která požadavek odeslala.

```
HttpListenerContext context = fronta.Dequeue();
```

Příklad 3: Volání metody Dequeue

4.3 Adaptivní prioritní fronta

Adaptivní prioritní fronta je v programu na straně serveru implementována jako samostatná třída s názvem *AdaptivniPrioritniFronta*. Obsahuje algoritmy pro přidávání, odebírání a třídění prvků v prioritní frontě. Algoritmy jsou použity z dynamické datové struktury halda, které jsou popsané v teoretické části. V této práci je místo klasického ukládání prvků do seznamu nebo do pole použita databázová tabulka.

4.3.1 Základní operace

Třída využívá základní operace haldy ve zdrojovém kódu realizované jako metody, které již byly podrobně popsány v teoretické části. Nejzákladnější jsou metody pro přidávání a odebíraní z fronty požadavků a pro zjištění, zda je fronta prázdná. Přidávání do fronty je implementováno metodou *Enqueue*, tato metoda přijímá jako parametr instanci třídy *HttpListenerContext*. Tedy třídu, která obsahuje http požadavek od klienta a zároveň umožňuje na tento požadavek odpovědět [10].

```
public void Enqueue(HttpListenerContext context)
```

Příklad 4: Hlavička metody Enqueue, pro vkládání do fronty požadavků

Operace odebírání z fronty požadavků je realizována jako metoda s názvem *Dequeue*. Metoda má návratový typ *HttpListenerContext*. Vrací tedy instanci této třídy, která obsahuje http požadavek s nejvyšší prioritou, který byl vyjmut z adaptivní prioritní fronty.

```
public HttpListenerContext Dequeue()
```

Příklad 5: Hlavička metody Dequeue, pro odebrání z fronty požadavků

Poslední metoda ve třídě *AdaptivniPrioritniFronta* s modifikátorem přístupu *public* je metoda s názvem *JePrazdna*. Její návratový typ je *bool*. V případě, že je adaptivní prioritní fronta prázdná, vrací hodnotu *true*. V opačném případě vrací hodnotu *false*.

```
public bool JePrazdna()
```

Příklad 6: Hlavička metody JePrazdna

4.3.2 Implementace metody Enqueue

Při zavolání metody *Enqueue* přijde v parametrech instance třídy *HttpListenerContext*. Z této instance se zjistí role uživatele, který požadavek odeslal a zároveň se vytvoří klíč, který slouží jako ukazatel na umístění instance typu *HttpListenerContext* v kolekci datového typu *Dictionary* (viz níže). Dále se zavolá metoda *VlozPozadavek* v instanci třídy *DBPripojeni* a jako parametry této metody se použijí získaná role a klíč (hash). Metoda uloží požadavek na konec fronty.

Samotná instance třídy *HttpListenerContext* se dále uloží do kolekce typu *Dictionary* společně s klíčem, který bude sloužit jako odkaz k zpětnému získání instance daného požadavku. Po vložení požadavku na konec haldy se zavolá operace *Heapify*. Tato operace bude postupovat od posledního prvku fronty až ke kořenu haldy. Operace je realizována jako metoda s názvem *HeapifyOdKonceNaZacatek* a je třeba tuto metodu zavolat vždy po vložení nového prvku, aby halda byla neustále setříděná podle aktuálních priorit rolí. Ukázka zdrojového kódu je zobrazena v příkladu 7.

```
public void Enqueue(HttpListenerContext context)
{
        String casovaZnamka = DateTime.Now.ToString("yyyyMMddHHmmssffff");

        dbPripojeni.VlozPozadavek(context.Request.QueryString["role"],
        casovaZnamka + context.GetHashCode());
        GlobalData.Instance._httpListenerContexts.Add(casovaZnamka +
        context.GetHashCode(), context);

        HeapifyOdKonceNaZacatek(dbPripojeni.PocetPozadavku() - 1);
}
```

Příklad 7: Implementace metody Enqueue

Po každém nově přidaném prvku do haldy se volá metoda *HeapifyOdKonceNaZacatek,* začínající u posledního prvku a končící u kořenového prvku. Volá se proto, aby halda měla neustále strukturu haldy. Bez zavolání této operace po vložení prvku by halda mohla ztratit správnou strukturu. Základní vlastnost haldy je, že každý prvek, který je na indexu ***i***, má levého potomka na indexu ***2 * i + 1*** a pravého potomka na indexu ***2 * i + 2***. Zároveň rodič tohoto uzlu je na indexu ***(i – 1) / 2***.

Algoritmus této metody zkoumá prvky jeden po druhém. Pokud má aktuálně zkoumaný prvek vyšší prioritu než jeho rodič, oba prvky se mezi sebou prohodí. Proces končí, jestliže priorita rodiče je vyšší než priorita aktuálně zkoumaného prvku, a nebo pokud cyklus dorazil až ke kořenovému prvku. Ukončení algoritmu znamená, že metoda *HeapifyOdKonceNaZacatek* odvedla svou práci a adaptivní prioritní fronta má správnou strukturu podle aktuálního scénáře priorit.

Metoda *HeapifyOdKonceNaZacatek* využívá pomocnou třídu pro práci s databází *DBPripojeni*. SQL dotaz používaný algoritmem pro získání aktuální priority požadavku na určitém indexu je vidět na příkladu 8. Příklad ukazuje situaci, kdy algoritmus vyžaduje od databáze aktuální prioritu požadavku, který je na indexu 2 (v pořadí třetí záznam v tabulce). K získání role, která požadavek odeslala, se využívá vnořený dotaz. Následně se získá z tabulky *role* aktuální priorita role, získaná ve vnořeném dotazu.

```
SELECT priorita FROM role WHERE autorita LIKE (SELECT role FROM fronta
LIMIT 2, 1);";
```

Příklad 8: Získaní aktuální priority daného záznamu na základě indexu (pozice)

4.3.3 Otestování metody Enqueue

K testování metody *Enqueue* se jako nástroj použil webový prohlížeč Chrome, který sloužil jako klient. Scénářem priorit se použil výchozí scénář (stav systému je *OK* a míra zahlcení *žádná*). Nejprve se spustil realizovaný program na serveru a zároveň se spustil webový prohlížeč. V prohlížeči se vytvořilo a odeslalo několik http požadavků. Posloupnost zadaných url adres je vyobrazena v příkladě 9.

```
http://localhost:12345/results?role=divak
http://localhost:12345/results?role=zavodnik
http://localhost:12345/results?role=divak
http://localhost:12345/results?role=zdravotnik
http://localhost:12345/results?role=zavodnik
http://localhost:12345/results?role=hasic
http://localhost:12345/results?role=organizator
http://localhost:12345/results?role=zdravotnik
```

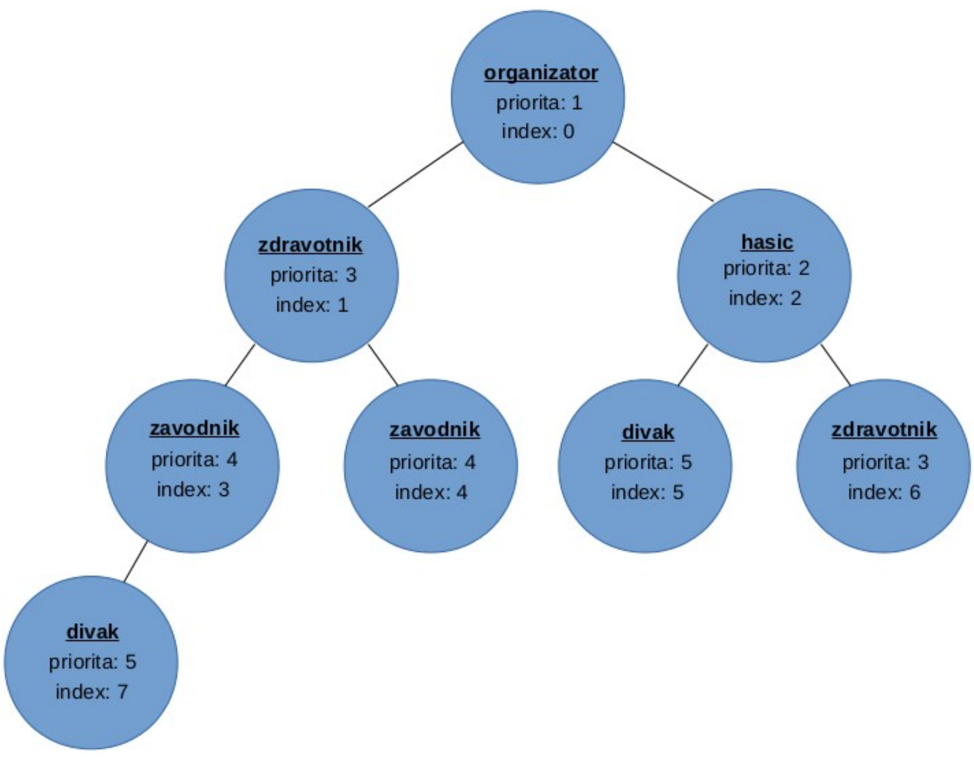
Příklad 9: Seznam url adres, které sloužily k otestování funkcionality metody Enqueue

Po odeslání těchto osmi http požadavků bylo v tabulce osm záznamů (prvků ve frontě). Při každém vložení se zavolala metoda *HeapifyOdKonceNaZacatek*, která setřídila frontu podle aktuálních priorit. Výsledná tabulka s osmi záznamy je vidět na obrázku 5.

id	role	hash
198	organizator	201411171800283761195425394
199	zdravotnik	201411171800336000919122217
200	hasic	201411171800235042284347232
201	zavodnik	201411171800024897675735924
202	zavodnik	201411171800186800175592233
203	divak	201411171800073036920499408
204	zdravotnik	201411171800133842482474336
205	divak	201411171759579087312910286

Obrázek 5: Fronta http požadavků po vložení osmi prvků

Http požadavek organizátora byl první ve frontě a měl tedy index 0. Požadavek jednoho ze dvou diváků byl poslední ve frontě s indexem 7. Poté byla vytvořena grafická podoba haldy z uvedené tabulky. Grafická podobá haldy sloužila ke kontrole, zda měla halda správnou strukturu. Vizualizace této haldy je vidět na obrázku 6.



Obrázek 6: Grafická podoba haldy

Vizualizace haldy názorně dokládá, že algoritmus metody *Enqueue* bere v potaz aktuální scénář priorit, a tak řadí požadavky do fronty správně. Každý vrchol haldy má rodiče s vyšší nebo stejnou prioritou a zároveň levého potomka s nižší nebo stejnou prioritou, než má jeho pravý potomek.

4.3.4 Implementace metody Dequeue

Metoda *Dequeue* je realizována jako veřejná metoda s návratovým typem *HttpListenerContext*. Nejprve se zavolá metoda *VratHashPozadavku* ze třídy *DBPripojeni*, ze které se získá klíč, sloužící k nalezení instance třídy *HttpListenerContext*. Následně se zavolá soukromá metoda *SmazKoren*, která odstraní požadavek z fronty (příklad 10).

Mazání kořenu probíhá prohozením prvního a posledního prvku ve frontě a následně se odstraní prvek, který je nyní poslední (původně první). Nakonec se zavolá metoda *HeapifyOdZacatkuDoKonce*, která bude postupovat od aktuálního kořenového prvku až na konec fronty. Operace *Heapify* od kořenového prvku je zde nutná k tomu, aby halda zachovala svoji správnou strukturu s ohledem na aktuální scénář priorit.

```
ProhoditPozadavky(0, pocetPozadavku - 1);
dbPripojeni.SmazPozadavek(pocetPozadavku - 1);

HeapifyOdZacatkuDoKonce(0);
```

Příklad 10: Mazání prvku z fronty v metodě SmazKoren

Algoritmus metody *HeapifyOdZacatkuDoKonce* má stejný výsledek jako algoritmus metody *HeapifyOdKonceNaZacatek*, a to setřídění fronty do správné struktury. Kontroluje prvky jeden po druhém, od kořenového prvku až k poslednímu a zjišťuje, zda nemá jeden z jeho potomků vyšší prioritu než kontrolovaný prvek sám. Jestliže má jeden z potomků kontrolovaného prvku vyšší prioritu, tak se tyto dva prvky prohodí. Pokud mají oba potomci vyšší prioritu než kontrolovaný prvek, tak se prohodí kontrolovaný prvek s jeho pravým potomkem.

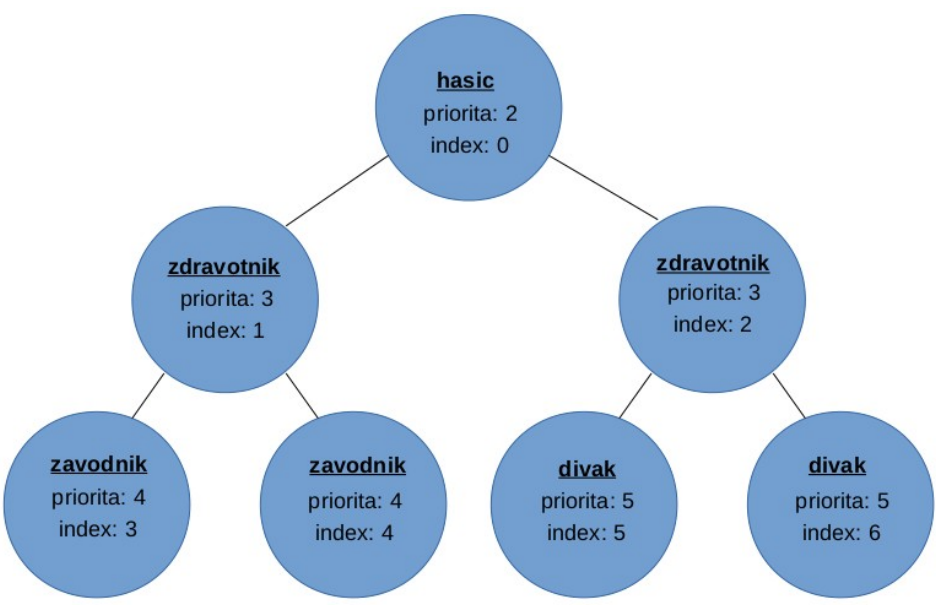
4.3.5 Otestování metody Dequeue

K otestování metody *Dequeue* byly použity záznamy, které se vytvořily v testu metody *Enqueue*. K testu byl použit webový server, ze kterého se jednou zavolala metoda *Dequeue*. Výsledná fronta požadavků po odebrání prvku s nejvyšší prioritou je vidět na obrázku 7.

id	role	hash
198	hasic	201411171800235042284347232
199	zdravotnik	201411171800336000919122217
200	zdravotnik	201411171800133842482474336
201	zavodnik	201411171800024897675735924
202	zavodnik	201411171800186800175592233
203	divak	201411171800073036920499408
204	divak	201411171759579087312910286

Obrázek 7: Fronta http požadavků po odebrání prvku s nejvyšší prioritou

Kořen fronty s indexem 0 byl prohozen s posledním prvek ve frontě na indexu 7. Poté prvek, který byl momentálně na konci fronty, byl odebrán. A nakonec byla zavolána metoda *HeapifyOdZacatkuDoKonce*. Tato metoda nejprve prohodila požadavky diváka a hasiče. Nakonec se prohodily požadavky diváka a zdravotníka. Tímto způsobem se požadavek od diváka dostal na správné místo, a tak halda měla správnou strukturu. Vizualizace haldy po odebrání prvku je vidět na obrázku 8.



Obrázek 8: Vizualizace haldy po odebrání prvku s nejvyšší prioritou

4.4 Robot

Ke splnění hlavního cíle práce bylo nutné vytvořit mechanismus, který bude v průběhu času měnit scénáře priorit na základě zahlcení fronty požadavky. Jako nejvhodnější řešení bylo zvoleno vytvoření třídy s názvem *Robot*, která je umístěna v programu bežícím na serveru společně s webovým serverem a adaptivní prioritní frontou. Třída *Robot* tvoří jádro

adaptivního prioritního systému rozhodování.

4.4.1 Implementace komponenty Robot

Oproti neustále aktivnímu webovému serveru je v programu Robot implementován jako třída, která je pasivní. Robot se aktivuje každých deset sekund. Jestliže se Robot aktivuje a stav systému je nastaven na jinou hodnotu než *OK*, tak Robot zůstává pasivní. Pokud je stav systému nastaven na hodnotu *OK* a zároveň se Robot aktivuje, tak tato komponenta zjišťuje míru zahlcení systému (celkový počet požadavků ve frontě). Na základě míry zahlcení komponenta změní scénář priorit a následně setřídí všechny požadavky ve frontě. Popsaný proces se neprovádí, jestliže je scénář již na aktuální míru zahlcení nastaven. Ukázka zdrojového kódu metody Robota, která se volá každých deset sekund (aktivace pasivního robota) a mění scénáře priorit, je zobrazena na příkladu 11.

```
if ("OK" == GlobalData.Instance.stavSystemu)
{
        if (GlobalData.Instance.miraZahlceni !=
VratMiruZahlceni(pocetPozadavku))
        {
                GlobalData.Instance.miraZahlceni =
VratMiruZahlceni(pocetPozadavku);
                db.ZmenPriorityNaZakladePoctuPozadavku(pocetPozadavku);
                SetriditPozadavkyPodlePriorit ();
        }
}
```

Příklad 11: Úryvek zdrojového kódu metody měnící scénáře priorit

```
else if (pocet >= 60 && pocet < 100)
        {
                dotaz =
                        "UPDATE role SET priorita = 3 WHERE id = 11;" +
                        "UPDATE role SET priorita = 1 WHERE id = 12;" +
                        "UPDATE role SET priorita = 2 WHERE id = 13;" +
                        "UPDATE role SET priorita = 4 WHERE id = 14;" +
                        "UPDATE role SET priorita = 5 WHERE id = 15;";
        }
```

Příklad 12: Ukázka změny scénáře priorit při 60 až 99 http požadavky ve frontě

4.4.2 Otestování komponenty Robot

K úplnému otestování komponenty Robot nestačil pouze webový prohlížeč, jako tomu bylo při testování metody *Enqueue*. Bylo potřeba vytvořit komplexnější řešení klienta, který bude v jednu chvíli odesílat více požadavků najednou, oproti webovému prohlížeči. Takové řešení

klienta bylo nutné k simulaci různých stupňů zahlcení na serveru. Klient byl realizován jako program napsaný v jazyce Java. Program vytvoří nové vlákno pro každé spojení se serverem. Tím je schopen odeslat více požadavků v jeden čas. Ukázka zdrojového kódu je v příkladu 13.

```
for(int i = 0; i < 35; i ++) {
        VlaknoProPozadavek vlaknoProPozadavek = new VlaknoProPozadavek(i,
        "hasic");
        vlaknoProPozadavek.start();
}
for(int i = 0; i < 5; i ++) {
        VlaknoProPozadavek vlaknoProPozadavek = new VlaknoProPozadavek(i,
        "divak");
        vlaknoProPozadavek.start();
}
```

Příklad 13: Ukázka odeslání 35 požadavků role hasič a 5 požadavků role divák

Při každém testu se odeslal určitý počet požadavků na server, který reprezentoval různé stupně zahlcení systému. Zároveň se v průběhu testování pro každý případ měnil stav systému. Výsledkem testování bylo potvrzení správné realizace Robota, který mění scénáře priorit na základě zahlcení systému. Ovšem s ohledem na stav systému (tabulka 3).

stav systému	OK	OK	OK	OK	požár	zranění
počet požadavků ve frontě	6	30	65	120	15	30
divák	5	5	5	5	5	5
závodník	4	4	4	4	5	5
zdravotník	3	2	1	2	4	1
hasič	2	2	1	1	1	4
organizátor	1	1	3	3	5	5
míra zahlcení	žádná	malá	střední	vysoká	---	---

Tabulka 3: Výsledky testování komponenty Robot

4.5 Klientská aplikace

Tato část práce zahrnuje postup implementace klientské aplikace v programovacím jazyce Java. V rámci cílů této práce bylo stanoveno, že aplikace poběží na mobilních zařízeních s operačním systémem Android. Pro vývoj aplikace bylo nutné vybrat vhodné vývojové prostředí. Rozhodování probíhalo mezi vývojovým prostředím IntelliJ IDEA a Eclipse, která jsou pro vývoj aplikací běžících na systémech Android nejznámější. Nakonec bylo vybráno vývojové prostředí IntelliJ IDEA (verze IntelliJ IDEA 13.1.2), protože s ním má autor této

práce bohaté zkušenosti a zároveň byl brán zřetel na zásadní výhodu tohoto prostředí, což je zaměření na vysokou produktivitu vývojářů [11].

Hlavním úkolem aplikace je uživatelům informačního systému zobrazovat data z databáze, s výjimkou uživatelské role organizátor, která může měnit stav systému. Jako nástroj pro testování aplikace byl využit Android emulátor (virtuální mobilní zařízení s operačním systémem Android). Emulátor velmi zrychluje fázi vývoje a pro účely této práce je tento způsob testování dostačující.

4.5.1 Implementace klienta

Prvním bodem implementace klienta bylo vytvoření přihlašovacího okna, které má za úkol identifikovat roli uživatele systému. Ověření zadaných údajů probíhá na straně serveru. Uživatel má možnost se do systému nepřihlašovat, takový uživatel má roli divák. Veškeré možné akce, které mohou uživatelé provádět v rámci klientské aplikace, jsou popsány v návrhu aplikace pomocí use-case diagramu. Na základě akce uživatele, jako je například kliknutí na nějaký rok, odešle klient http požadavek na server. Tento požadavek obsahuje url adresu, která bude mít jako parametr roli uživatele a parametr nesoucí informaci o tom, co uživatel od webového serveru požaduje. U kliknutí na rok, to bude parametr rok, který webový server zpracuje tak, že odešle zpátky klientu seznam závodů v tomto roce (viz obrázek 9).

```
http://10.0.2.2:12345/results?role=divak&rok=2014
```

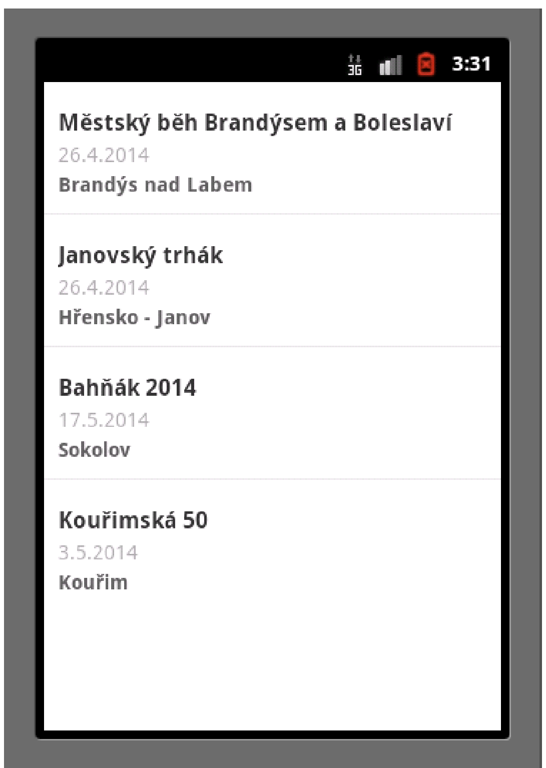
Obrázek 9: Url adresa s parametrem rok

Webový server posílá klientovi data v textové podobě ve formátu JSON - JavaScript Object Notation. Tento formát byl do práce vybrán, protože je jedním z nejpoužívanějších formátů pro výměnu dat na Webu a velmi se hodí při zápisu krátkých strukturovaných dat [12]. Ukázku odpovědi, kterou dostane klient od serveru, pokud si uživatel vyžádá všechny závody v roce 2014, je vidět na obrázku 10.

```
[{"Id": 1, "Jmeno": "Městský běh Brandýsem a Boleslaví",
"Datum": "26.4.2014", "Misto": "Brandýs nad Labem"},{"Id": 2,
"Jmeno": "Janovský trhák", "Datum": "26.4.2014", "Misto":
"Hřensko – Janov"}, {"Id": 3, "Jmeno": "Bahňák 2014", "Datum":
"17.5.2014", "Misto": "Sokolov"}, {"Id": 4, "Jmeno":
"Kouřimská 50", "Datum": "3.5.2014", "Misto": "Kouřim"}]
```

Obrázek 10: Ukázka dat v JSON formátu - seznam závodů v roce 2014

Seznam závodů je zde reprezentován jako pole objektů. Pole je datového typu *JSONArray* a objekty jsou typu *JSONObject*. Tyto objekty obsahují atributy závodů a představují jednotlivé instance závodů. Pomocí balíku *org.json* zdrojový kód klientské aplikace rychle a elegantně zpracovává text v JSON formátu. Výsledná zpracovaná data zobrazí aplikace uživateli v GUI (grafickém uživatelském rozhraní). Uživatelské rozhraní po zpracování seznamu závodů v roce 2014 je vidět na obrázku 11.



Obrázek 11: Seznam závodů v roce 2014 - klientská aplikace

Jakmile klient odešle požadavek v GUI, zobrazí se uživateli kruhový indikátor průběhu, který je obsažen v balíku *android.app*, kde ho reprezentuje třída s názvem *ProgressDialog* [13]. Je tam z toho důvodu, aby uživatel bral na vědomí, že jeho požadavek se právě vyřizuje a zároveň tento indikátor průběhu zabraňuje uživateli odesílat další požadavky na server. Tímto řešením se odstranil problém, který by mohl nastat při zahlcení serveru požadavky, kde by uživatelé, kteří nedostali odpověď od serveru do pár sekund, odesílali své požadavky na server znovu a již zahlcený server by obsahoval navíc více požadavků od jednoho klienta. V seznamu závodů má uživatel možnost kliknout na některý závod. Po kliku se odešle další http požadavek a odpověď od webového serveru bude v podobě výsledků pro tento závod viz obrázek 12.

Obrázek 12: Výsledky závodu – klientská aplikace

5 Testování systému

Testování celého systému probíhalo v několika fázích. Nejprve byla otestována aplikace na straně klienta. Následně proběhlo testování adaptivního prioritního systému rozhodování.

5.1 Testování klientské aplikace

V průběhu vývoje klientské aplikace bylo nepřetržitě uplatňováno ruční testování, zda je část kódu funkční. Na základě požadavků na systém si autor vytvořil seznam úkolů, které bylo potřeba implementovat. Po dokončení každého úkolu následovalo ověření, zda přidaná funkcionalita funguje správně jak má. Toto ověřování probíhalo přímo v grafickém uživatelském rozhraní aplikace. Další testování, které proběhlo v rámci této aplikace, bylo Unit testování. Jedná se o testování tříd a jejich metod [14].

Ruční testování probíhalo na virtuálním zařízení s operačním systémem Android a zároveň na skutečném zařízení s tímto operačním systémem. Nutno podotknout, že cílem práce nebylo nasadit tuto aplikaci do reálného prostředí, ale navrhnout a implementovat celé řešení adaptivního prioritního systému včetně klientské aplikace. Ovšem bez nasazení této aplikace nebylo možné plnohodnotně otestovat adaptivní prioritní systém rozhodování, který je umístěn na serveru. K tomu bylo nutné nasimulovat více klientských požadavků, než mohl programátor poslat z jednoho zařízení. Toto testování je popsáno v další kapitole.

5.2 Testování adaptivní prioritní fronty

Testování adaptivní prioritní fronty probíhalo měřením rychlosti odpovědi od webového serveru (v sekundách). Jako nástroj pro testování byl použit lehce upravený program, který se již používal při testování Robota. V každém testu se odeslalo několik http požadavků na server a ihned po nich požadavek, u kterého se měřil čas příchodu odpovědi od webového serveru. Pro každý případ se použilo řešení webového serveru s adaptivním prioritním systémem a zároveň klasického řešení webového serveru, který přijaté požadavky zpracovává ihned. Tyto dva případy se porovnávaly a následně se vyhodnotilo výhodnější řešení.

Byly testovány všechny případy, ke kterým by mohlo v průběhu nasazení adaptivní prioritní fronty dojít. Nejzajímavější výstupy jsou zobrazeny v tabulce 4. Z tabulky přímo vyplývá, že klasická realizace webového serveru je v naprosté většině případů mnohem rychlejší než

řešení pomocí adaptivní prioritní fronty (viz tabulka 4). Avšak při testování vysokého zahlcení webového serveru požadavky byl potvrzen předpoklad, že uživatelé s aktuálně nejvyšší prioritou (hasič, zdravotník) dostanou odpověď od serveru rychleji než při klasickém řešení webového serveru. Odpověď při zahlceném serveru pro důležité uživatele systému byla vždy přijata do jedné sekundy. Při změně stavu systému na *požár* dostal hasič odpověď velmi rychle, nezávisle na zahlcení serveru požadavky. Stejné výsledky byly naměřeny u uživatele zdravotník při změně stavu systému na *zranění*.

role	divák	divák	hasič	hasič	organizátor	organizátor	závodník	závodník
odezva [s]	0,0194	11,9658	6,0303	0,9952	0,0207	1,028	0,0012	2,9639
počet požadavků	101	101	101	101	66	66	41	41
požadavky s vyšší prioritou	100	100	0	0	15	15	35	35
požadavky s nižší prioritou	0	0	101	101	50	50	5	5
stav systému	OK	OK	OK	OK	OK	OK	požár	požár
míra zahlcení	vysoká	vysoká	vysoká	vysoká	střední	střední	---	---
adaptivní prioritní systém	ne	ano	ne	ano	ne	ano	ne	ano

Tabulka 4: Odezvy webového serveru před a po nasazení adaptivního prioritního systému

6 Závěr

Závěry jsou uvedeny k jednotlivým zadaným cílům:

1. Byl proveden návrh a implementace adaptivního prioritního systému rozhodování podle důležitosti sdělení. Návrh zahrnuje požadavky na systém, architekturu systému a návrh scénářů priorit. Adaptivní prioritní systém je složen z webového serveru a adaptivní prioritní fronty. Webový server přijímá a následně ukládá do fronty požadavky od klientů a zároveň vyřizuje ty s nejvyšší prioritou. Implementace adaptivní prioritní fronty je realizována pomocí algoritmů operací binární haldy. Tyto algoritmy třídí požadavky ve frontě podle scénáře priorit, který se mění na základě zahlcení systému daty a stavu systému.

2. Byla navržena a realizována klientská aplikace, sloužící převážně pro zobrazování dat ze serveru. Tato data jsou sportovního charakteru (seznam závodů, výsledky závodů, souřadnice nehod atd.). Aplikace obsahuje pět uživatelských rolí typických pro sportovní tématiku. Na straně serveru je vytvořen webový server, který přednostně zpracovává požadavky uživatelů s nejvyšší prioritou a následně posílá zpátky klientům odpovědi.

3. Webový server přijímá http požadavky od uživatelů klientské aplikace a ukládá je do adaptivní prioritní fronty společně s rolí uživatele. V adaptivní prioritní frontě se požadavky třídí podle aktuálního scénáře priorit. Webový server přednostně zpracovává požadavky s nejvyšší prioritou, tedy ty na vrcholu fronty.

4. Klientská aplikace běží na operačním systému Android a adaptivní prioritní systém rozhodování je umístěn na Windows serveru. Aplikace byla v této sestavě otestována a funguje podle požadavků na systém.

7 Seznam použité literatury

[1] Dynamické datové struktury. [online]. [cit. 2014-10-06]. Dostupné z: http://www.isd.cz/pascal/6dynprom.html

[2] Úvod do databázových systémů. [online]. [cit. 2014-12-10]. Dostupné z: http://www.fi.muni.cz/~xdohnal/lectures/PB154/czech/zezula13.pdf

[3] Javaalgoritmy.wz.cz: ADT. Javaalgoritmy.wz.cz: Implementace ADT prioritní fronta [online]. 1. vyd. [cit. 2014-10-20]. Dostupné z: http://javaalgoritmy.wz.cz/pfronta.htm

[4] Binární halda. [online]. [cit. 2014-10-22]. Dostupné z: http://www.algoritmy.net/article/15/Binarni-halda

[5] Priority Queues with Binary Heaps. [online]. [cit. 2014-11-12]. Dostupné z: http://interactivepython.org/courselib/static/pythonds/Trees/heap.html

[6] ARRAY-BASED BINARY HEAP INTERNAL REPRESENTATION (Java, C++) | Algorithms and Data Structures. [online]. [cit. 2014-11-13]. Dostupné z: http://www.algolist.net/Data_structures/Binary_heap/Array-based_int_repr

[7] 2. díl - UML - Use Case Diagram. [online]. [cit. 2014-11-15]. Dostupné z: http://www.itnetwork.cz/uml-use-case-diagram

[8] Začínáme s MySQL 1.díl – Živě.cz. [online]. [cit. 2014-11-15]. Dostupné z: http://www.zive.cz/clanky/zaciname-s-mysql-1dil/sc-3-a-102589/

[9] Connect C# to MySQL - CodeProject. [online]. [cit. 2014-11-15]. Dostupné z: http://www.codeproject.com/Articles/43438/Connect-C-to-MySQL

[10] HttpListenerContext – třída (System.Net). [online]. [cit. 2014-11-20]. Dostupné z: http://msdn.microsoft.com/cs-cz/library/system.net.httplistenercontext%28v=vs.110%29.aspx

[11] IntelliJ IDEA. Profesionální prostředí IDE pro Javu. [online]. [cit. 2014-11-22]. Dostupné z: http://www.svetandroida.cz/intellij-idea-java-201406

[12] JSON : jednotný formát pro výměnu dat - Zdroják. [online]. [cit. 2014-12-10]. Dostupné z: http://www.zdrojak.cz/clanky/json-jednotny-format-pro-vymenu-dat/

[13] ProgressDialog | Android Developers. [online]. [cit. 2014-12-10]. Dostupné z:http://developer.android.com/reference/android/app/ProgressDialog.html

[14] Testing Fundamentals | Android Developers. [online]. [cit. 2014-12-10]. Dostupné z:http://developer.android.com/tools/testing/testing_android.html

8 Seznam obrázků

Obrázek 1: Aplikace vzorců pro výpočet potomků a rodiče [6]....5
Obrázek 2: Use-case diagram klientské aplikace....9
Obrázek 3: Architektura systému....10
Obrázek 4: Návrh databázové struktury....12
Obrázek 5: Fronta http požadavků po vložení osmi prvků....17
Obrázek 6: Grafická podoba haldy....17
Obrázek 7: Fronta http požadavků po odebrání prvku s nejvyšší prioritou....19
Obrázek 8: Vizualizace haldy po odebrání prvku s nejvyšší prioritou....19
Obrázek 9: Url adresa s parametrem rok....22
Obrázek 10: Ukázka dat v JSON formátu - seznam závodů v roce 2014....22
Obrázek 11: Seznam závodů v roce 2014 - klientská aplikace....23
Obrázek 12: Výsledky závodu – klientská aplikace....24

9 Seznam tabulek

Tabulka 1: Časové složitosti operací – vkládání a odebírání z fronty, realizace polem a seznamem [3]....4
Tabulka 2: Scénáře priorit pro Robota a uživatele organizátor....8
Tabulka 3: Výsledky testování komponenty Robot....21
Tabulka 4: Odezvy webového serveru před a po nasazení adaptivního prioritního systému. 25

10 Seznam příkladů

Příklad 1: Ukázka url adresy obsažené v http požadavku, který odeslal uživatel role hasič. .13
Příklad 2: Volání metody Enqueue....13
Příklad 3: Volání metody Dequeue....14
Příklad 4: Hlavička metody Enqueue, pro vkládání do fronty požadavků....14
Příklad 5: Hlavička metody Dequeue, pro odebrání z fronty požadavků....14
Příklad 6: Hlavička metody JePrazdna....15
Příklad 7: Implementace metody Enqueue....15
Příklad 8: Získaní aktuální priority daného záznamu na základě indexu (pozice)....16
Příklad 9: Seznam url adres, které sloužily k otestování funkcionality metody Enqueue......16
Příklad 10: Mazání prvku z fronty v metodě SmazKoren....18
Příklad 11: Úryvek zdrojového kódu metody měnící scénáře priorit....20
Příklad 12: Ukázka změny scénáře priorit při 60 až 99 http požadavky ve frontě....20
Příklad 13: Ukázka odeslání 35 požadavků role hasič a 5 požadavků role divák....21