VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

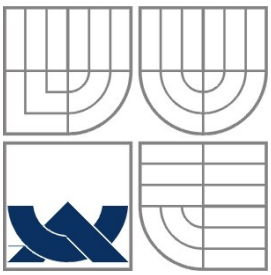
SÍŤOVÁ KOMUNIKACE V J2ME/J2SE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

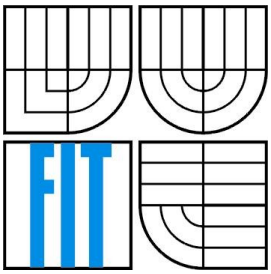
AUTOR PRÁCE
AUTHOR

PETR PŘEROVSKÝ

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SÍŤOVÁ KOMUNIKACE J2ME/J2SE

J2ME/J2SE NETWORK COMMUNICATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR PŘEROVSKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ALEŠ LÁNÍK

BRNO 2009

Abstrakt

V první části se práce zabývá popisem platformy Java Micro Edition s cílem vyzdvihnout její hlavní rysy. Druhá část uvádí možnosti implementace síťové komunikace v prostředí mobilních telefonů a síťové možnosti platformy Java 2 Standard Edition se zaměřením na multiplexní, neblokující server. Třetí část je zaměřena na návrh a implementaci demonstrační aplikace společně s testy výsledného řešení.

Abstract

The first part of this bachelor study is engaged in description of platform Java Micro Edition with the goal to punctuate its main features. The second part presents the possibilities of implementation of networking communication in surroundings of mobile phones and networking posibilities of platform Java 2 Standart Edition with target the multiplex and non-blocking server. The third part is focused on project and implementation of exemplary application with tests of resulting solutions together.

Klíčová slova

J2ME, J2SE, Java New I/O, CLDC, CDC, KVM, CVM, MIDP, MIDlet, Profil, Konfigurace, Virtuální stroj, Soket, Kanál, Datagram, Mobilní telefon

Keywords

J2ME, J2SE, Java New I/O, CLDC, CDC, KVM, CVM, MIDP, MIDlet, Profile, Configuration, Virtual machine, Socket, Channel, Datagram, Mobile phone

Citace

Petr Přerovský: Síťová komunikace J2ME/J2SE, bakalářská práce, Brno, FIT VUT v Brně, 2009

Síťová komunikace J2ME/J2SE

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Aleše Láníka.

.....
Petr Přerovský
2009

Poděkování

Rád bych zde poděkoval Ing. Aleši Láníkovi za jeho odbornou pomoc a cenné rady při zpracování práce. Také bych rád poděkoval své rodině za podporu a přítelkyni za čas věnovaný čtení této práce.

Obsah

1 Úvod

V dnešní době jen zřídka potkáme člověka, který by nevlastnil mobilní telefon. Současný trend vývoje aplikací pro mobilní telefony se zaměřuje spíše na vývoj pro konkrétní mobilní zařízení s operačním systémem (Symbian OS, Windows Mobile, PalmOS, ...). Hlavní nevýhody tohoto trendu spočívají v tom, že vyvíjená aplikace je závislá na příslušné platformě a tím pádem velmi zužuje spektrum uživatelů. Chceme-li, aby naše aplikace byla dostupná široké veřejnosti, je třeba sáhnout po Javě, přesněji po platformě Java Micro Edition (dále jen J2ME), ta je standardně obsažena v hojném množství mobilních telefonů. Aplikace, psané tímto programovacím jazykem se nazývají MIDlety (Mobile Information Device Profile) a umožňují uživateli, s trochou nadsázky, prakticky neomezené rozšiřování funkcí mobilního telefonu.

Tato práce se zabývá tvorbou J2ME aplikací se zaměřením na síťovou komunikaci v prostředí mobilních telefonů. Jsou zde také probírány možnosti a implementace neblokujícího serveru s využitím nové I/O knihovny (dále jen New I/O Api). Obsahem druhé kapitoly práce je představení platformy J2ME a teoretický základ popisující její hlavní rysy, nutné ke zvládnutí dané problematiky.

Ve třetí a čtvrté kapitole jsou obdobným způsobem probrány možnosti implementace síťového rozhraní v platformách J2ME a J2SE (Java 2 Standart Edition). Jsou zde mimo jiné popsány rozdíly mezi datagramovou a soketovou komunikací, serverovým a klientským soketem. Čtvrtá kapitola podrobně popisuje možnosti knihovny New I/O, důležité pro tvorbu serverové části práce.

Pátá kapitola pojednává stručně o analýze problému, specifikaci požadavků a definuje minimální požadavky na mobilní zařízení.

Návrhy nejdůležitějších implementačních částí práce zaměřené převážně na přenosový protokol, tvoří obsah šesté kapitoly. Na tuto kapitolu hned navazuje kapitola další, která popisuje implementaci hlavních částí aplikací, které vycházejí z návrhu v předchozí kapitole.

Předposlední kapitolu tvoří prováděné zátěžové a síťové testy zaměřené převážně na dobu odezvy od serveru. Na konci kapitoly je provedeno shrnutí a zhodnocení provedených testů.

Poslední kapitola obsahuje zhodnocení práce, její přínosy a možnosti jejího rozšíření.

2 J2ME

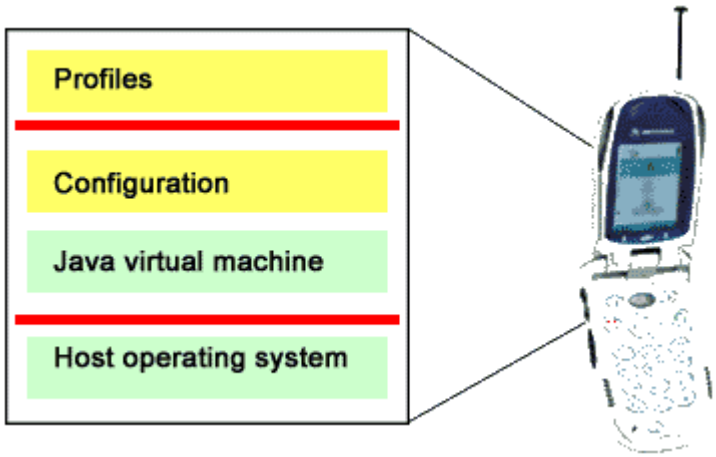
J2ME (Java 2, Micro Edition) je vysoce optimalizovaná verze Javy společnosti Sun Microsystems určená pro malá elektronická zařízení, jakými jsou např. PDA (Personal Digital Assistant), pagery, mobilní telefony, inteligentní doplňkové zařízení STB (Set-Top-Box) a další. J2ME poskytuje kompletní řešení moderních síťových aplikací pro malá zařízení. Kromě jiného Sun Microsystems uvedl celkem 3 různé balíčky Javy:

- Java 2 Standard Edition (J2SE)
- Java 2 Enterprise Edition (J2EE)
- Java 2 Mirco Edition (J2ME)

Tyto typy Javy se liší svým zaměřením a službami, které nabízí. Každý výše zmíněný balík pokrývá jinou oblast, která se ještě dále dělí. Oproti standardní J2SE musí J2ME pracovat s nízkými výkony procesoru, extrémně nízkou pamětí, malým nebo velmi malým displejem. Musí umět obsloužit nestandardní vstupy, jako jsou například dotykové displeje.

2.1 Obecný pohled

Z obecného pohledu je J2ME tvořena v duchu modulární architektury. Obrázek 2.1 podává přehled vztahů jednotlivých modulů pracovního prostředí J2ME. Uprostřed stojí javovský virtuální stroj, který pracuje na hostitelském operačním systému mobilního zařízení. Nad ním je specifická konfigurace J2ME, jenž se skládá z programovacích knihoven, zajišťujících základní funkce daného zařízení. Vrchol tvoří J2ME profily, které obsahují doplňkové programovací knihovny, využívající příbuzných funkcí v podobných zařízeních.



Obrázek 2.1: Architektura pracovního prostředí J2ME [7]

2.2 Konfigurace

Mobilní telefony, PDA, pagery a další malá zařízení se liší svými funkcemi a vlastnostmi. Většinou však používají podobné procesory a mají podobné množství paměti. Proto byly vytvořeny konfigurace. Konfigurace je specifikace softwarového prostředí pro vybranou skupinu zařízení, která je určena sadou charakteristik, na něž se specifikace spoléhá. Většinou jsou to tyto charakteristiky:

- Typ a velikost dostupné paměti.
- Typ a frekvence procesoru.
- Typ síťového připojení, které je na zařízení k dispozici.

Konfigurace má reprezentovat minimální platformu pro dané cílové zařízení. Konfigurace nedefinuje žádné volitelné funkce. Mají-li se vývojáři aplikací spoléhat na konzistentní programovací prostředí a vyvíjet tak hardwarově nezávislé aplikace, jsou výrobci hardwaru povinni danou specifikaci plně implementovat.

V současné době existují dvě standardní konfigurace pro J2ME: CLDC (Connected Limited Device Configuration) a CDC (Connected Device Configuration).

CLDC

Konfigurace CLDC je určena pro nízkoúrovňovou oblast spotřební elektroniky. Tato konfigurace je obvyklejší pro J2ME, specifikuje mnohem menší nároky na zařízení než CDC. Pro CLDC je typickou platformou mobilní telefon nebo PDA s přibližně 512kB volné paměti.

Požadavky pro J2ME CLDC dle oficiální specifikace J2ME [8] (standard JSR-30)

- 128kB až 512kB celkové paměti, z toho <= 256kB ROM/Flash a <= 256kB RAM. Ve většině případů budou mít zařízení více paměti ROM než RAM nebo Flash.
- Omezené zdroje energie, nejčastěji provoz na baterii.
- Připojení k některému typu sítě s omezenou šířkou pásma (9600/bps nebo méně).
- Uživatelské rozhraní s různou mírou propracovanosti.

CLDC nijak nezasahuje do uživatelských rozhraní. Neovlivňuje ani správu událostí či interakce mezi uživatelem a aplikací. To vše spadá do režie rozšiřujících profilů, jako je například MIDP, které přispívají k funkčnosti CLDC.

CLDC zařízení musí podporovat:

- Soubor základních rysů Javy a virtuálního stroje
- Soubor základních knihoven Javy (`java.lang`, `java.util`)
- Základní vstup/výstup (`java.io`)
- Základní podporu sítí (`javax.microedition.io`)
- Zabezpečení

CLDC ve verzi 1.0 neobsahovala podporu plovoucí řádové čárky, proto aplikace založené na CLDC 1.0 neobsahovaly datové typy `float` ani `double`.

CLDC 1.1 přináší větší podporu zařízení s většími zdroji a také větší podporu hardwaru pro práci s plovoucí řádovou čárkou. Přestože CLDC 1.1 podporuje mnoho, co CLDC 1.0 zakazovala, existuje několik dalších funkcí Javy, které budou aplikace v CLDC postrádat. Důvodem je snížení hardwarových a paměťových nároků oproti standardní Javě.

Hlavní vynechané funkce:

- **Finalizace** - (finalizace) nabízí málo na úkor velké složitosti virtuálního stroje, proto CLDC neobsahuje metodu `Object.finalize();`. Není tedy možné provádět závěrečné úklidové operace s objekty dokud není objekt zrušen GC (Garbage Collector).
- **Vlákna** - CLDC samotná vlákna poskytuje, ale není možné vytvořit démonovská vlákna (tj. taková vlákna, která pro ukončení všech nedémonovských vláken automaticky ukončí svůj běh) a ani skupiny vláken.
- **Omezené možnosti zpracování chyb** - CLDC neobsahuje většinu tříd oproti J2SE, které „chyby“ reprezentují. Nastane-li nespecifikovaná „chyba“, zodpovídá za další vhodnou akci samo zařízení, což může znamenat, že je aplikace ukončena nebo dojde k resetování zařízení.
- **Chybí uživatelem definované zavaděče tříd** – z důvodu zabezpečení KVM (Kilo Virtual Machine) neobsahuje uživatelem definované zavaděče tříd. Virtuální stroj, který podporuje CLDC musí mít svůj vlastní zavaděč tříd, který uživatel nemůže nijak odstranit, ani překrýt.

Knihovny tříd CLDC:

CLDC je implementovaná do celé řady platforem, které nemají dostatečné paměťové prostředky pro podporu kompletní J2SE. Proto balíčky a třídy obsažené v CLDC musí mít nízké nároky na systémové prostředky. CLDC je konfigurací a tedy neobsahuje žádné volitelné funkce, ty jsou otázkou profilů. V následujícím oddílu se budeme zabývat již jen CLDC 1.1 a popisem hlavních balíčků konfigurace.

CLDC 1.1	Třídy
java.io	ByteArrayInputStream, ByteArrayOutputStream, DataInputStream, DataOutputStream, InputStream, InputStreamReader, OutputStream, OutputStreamWriter, PrintStream, Reader, Writer
java.lang	Boolean, Byte, Character, Class, Double, Float, Integer, Long, Math, Object, Runtime, Short, String, StringBuffer, System, Thread, Throwable
java.util	Calendar, Date, Hashtable, Random, Stack, TimeZone, Vector
javax.microedition.io	Connector

Tabulka 2.1: Balíčky a třídy CLDC 1.1 [12]

Balíček java.io

Balíček `java.io` v CLDC obsahuje jen část rozsáhlého `java.io` z J2SE. Za zmínku stojí pouze vstupní a výstupní proudy, které lze připojit na zdroj nebo cíl dat. Jsou to `ByteArray-InputStream` resp. `ByteArrayOutputStream`. Zabalíme-li tyto proudy pomocí `DataInputStream` resp. `DataOutputStream`, mohou poskytovat způsob, jak přenášet primitivní datové typy Javy. Ke všem ostatním zdrojům dat lze získat přístup primitivními implementacemi tříd ImputStream resp. OutputStream.

Balíček java.lang

Balíček `java.lang` obsahuje zhruba polovinu tříd svého protějšku `java.lang` J2SE. Některé z obsažených tříd nejsou úplnými implementacemi. Jejich omezení asi nejlépe vystihují body (Viz str. 5).

Balíček java.util

Balíček `java.util` obsahuje třídy týkající se kolekcí a třídy týkající se časových a datových údajů.

Z kolekcí jsou to třídy `Hashtable`, `Stack`, `Vector` a `Enumeration`. Jedná se o podmnožinu kolekcí, které byly dostupné v JDK 1.1. Díky omezeným hardwarovým prostředkům není v CLDC k dispozici systém kolekcí z Java 2.

Z datových tříd se jedná o třídy `Date`, `TimeZone`, `Calendar`. Všechna data v Javě jsou reprezentována pomocí odchylky od 00:00 GTM (Greenwich Mean Time) dne 1. ledna 1970. Třída `Date` je pouze obalem dlouhé hodnoty, reprezentující datum a čas z odchylky zmíněné výše.

Balíček javax.microedition.io

Balíček `javax.microedition.io` nemá žádný svůj ekvivalent v J2SE. Cílem je, aby ho profily na bázi CLDC využívaly jako společný mechanismus pro přístup k prostředkům sítě, které lze adresovat názvem a mohou odesílat, resp. přijímat data pomocí `OutputSteam` resp. `Input-Stream`. Typickým příkladem jsou webové stránky identifikovatelné pomocí URL (Uniform Resource Locator).

CDC

Konfigurace CDC je určena pro výkonná zařízení, která leží mezi CLDC a úplnými stolními systémy, na nichž běží platforma J2SE. Tato zařízení disponují větší pamětí (obvykle 2MB a více) a také výkonnějšími procesory. Mohou proto podporovat úplnější softwarové prostředí Javy. Patří mezi ně například navigační systémy pro vozidla, webové telefony, inteligentní doplňkové zařízení STB (Set-Top Box), domácí spotřebiče a další.

Požadavky pro J2ME CDC dle oficiální specifikace J2ME [9] (standard JSR-36)

- 512kB minimum paměti ROM
- 256kB minimum paměti RAM
- Připojení k nějakému typu sítě

- Podpora kompletní implementace JVM (Java Virtual Machine), jak jsou definovány ve specifikaci *Java Virtual Machine, 2nd Edition* [10]
- Uživatelské rozhraní s různou mírou rozpracovanosti

Obrázek 2.2: Rozdělení dle konfigurace [11]

Na obrázku 2.2 je znázorněno rozdělení produktů spadajících do jedné ze dvou konfigurací. Na obrázku si lze všimnout, že ačkoli jsou obě skupiny produktů podporovány odlišnými konfiguracemi, hranice mezi nimi není naprosto jednoznačná. Je tedy nutné mít na paměti, že hlavní rozdíl mezi CLDC a CDC je především ve velikosti paměti jednotlivých zařízení, v používání baterie, v přítomnosti či nepřítomnosti uživatelského rozhraní. Tyto rozdíly se díky technickým pokrokům značně eliminují.

2.3 Virtuální stroje

Je vhodné podotknout, že konfigurační specifikace nevyžadují, aby implementace použily určitý virtuální stroj. Výrobci hardware si mohou vytvořit vlastní VM (Virtual Machine) nebo si na VM opatřit licenci třetí strany, musí ale splňovat minimální požadavky specifikace. Společnost Sun Microsystems poskytuje vhodný VM ke každé z obou konfigurací. Důvody jsou zřejmé, rozdílné požadavky konfigurací CDC a CLDC na paměť, různé rozlišovací schopnosti displejů, používání baterií a také přítomnost či nepřítomnost uživatelského rozhraní. Virtuálním stroj pro CLDC se nazývá KVM (Kilo Virtual Machine) a pro CLD nese název CVM.

KVM

KVM je kompletní, vysoce přenositelné javovské pracovní prostředí určené speciálně pro malá zařízení s omezenými zdroji a s několika stovkami kilobajtů celkové paměti. Jedná se o skutečný JVM podle specifikace pro virtuální stroje Javy až na některé odchylky nutné ke správnému fungování na malých zařízeních. Tato zařízení obvykle obsahují 16- nebo 32-bitové procesory [11].

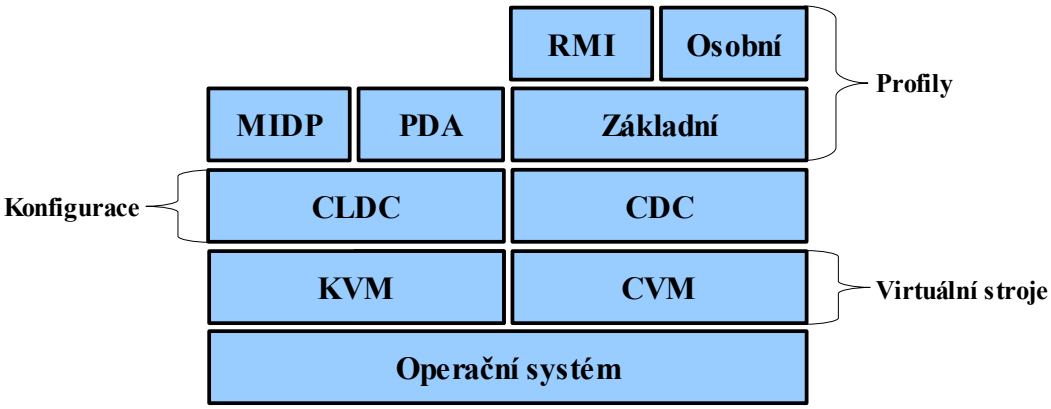
CVM

CVM je vytvořeno pro větší spotřební zařízení. Jedná se o plně funkční Java Virtual Machine podporující všechny rysy a knihovny vyrtuálního stroje Java 2 v oblasti bezpečnosti.

Zkratka C v názvu CVM původně znamenala „Compact“ Virtual Macine. Sun Microsystems později zjistil, že díky nepozornosti prodavačů se výraz „compact“ ve zkratce CVM může snadno zaměnit s K ve zkratce KVM. Proto v současnosti C ve zkratce CVM nemá žádný význam.

2.4 Profily

Konfigurace CLDC nebo CDC poskytují jen základní vrstvu, na níž lze aplikovat řadu profilů poskytujících další programové prostředky. Profily tedy doplňují konfiguraci přidáním další nadstavby tříd, které zajišťují různé funkce, vhodné pro specifický druh zařízení. V současnosti je asi nejznámějším profilem MIDP.



Obrázek 2.3: Celkový pohled na technologie J2ME

MIDP

Profil MIDP (Mobile Information Device Profile) je navržen pro práci s konfigurací CLDC a je určen pro mobilní zařízení, jakými jsou například mobilní telefony a obousměrné pagery. MIDP přináší do CLDC síťové služby, práci s daty a uživatelské rozhraní. Dále také obsahuje standardizované prostředí, umožňující přidávat do koncového zařízení nové aplikace. Aplikace, pracující na bázi MIDP se označují jako MIDlety. V tuto chvíli existují dvě verze MIDP 1.0 a MIDP 2.0. Rozdíly mezi MIDP 1.0 a novinkami v MIDP 2.0 budou zmíněny (Viz podkapitola 2.5).

Profil PDA

Profil PDA (PDAP) je určen pro zařízení disponující většími rozměry displeje a větší pamětí něž mobilní telefony. Je určen pro osobní organizéry PDA. Stejně jako MIDP je profil PDA založen na konfiguraci CLDC a poskytuje API (Application Programming Interface) pro uživatelská rozhraní a také pro ukládání dat v externích zařízeních.

Základní profil

Základní profil (Fundation profile) rozšiřuje CDC o všechny standardní knihovny Javy. Neobsahuje žádné API pro uživatelské rozhraní. Důvodem je typ zařízení, pro které je CDC určeno (např. Set_Top Box, net TV, ...). Jedná se o profil, na kterém se mohou stavět ostatní profily CDC.

Osobní profil

Osobní profil (Personal profile) rozšiřuje základní profil o grafické uživatelské rozhraní. Jde o prostředí s plnou podporou AWT (Abstract Window Toolkit).

RMI profil

RMI profil (Remote Method Invocation) je dalším rozšířením Základního profilu o knihovny pro vzdálené vyvolání metod J2SE. RMI profil navazuje na Základní profil a je určen pro CDC/Základní profil, ne pro CLDC/MIDP.

2.5 MIDP 1.0

Jak již bylo zmíněno dříve, MIDP profil je nadstavbou CLDC, tvoří její vrchol. MIDP 1.0 byl prvním dokončeným profilem pro J2ME a setkáváme se s ním v naprosté většině mobilních telefonů podporujících J2ME.

	Minimální požadavky MIDP 1.0
Požadavky na Displej:	**velikost displeje minimálně 95x54 pixelů** **barevná hloubka minimálně 1 bit = 2 barvy** **poměr stran pixelů přibližně 1:1**
Vstupní zařízení:	**ITU-T telefonní klávesnice 0-9, QUERTY klávesnice, dotykový displej**
Požadavky na paměť:	**128 kB trvalé paměti pro MIDP komponenty** **8 kB trvalé paměti pro data aplikace** **32 kB pro běh aplikace (Java heap)**
Síťové požadavky:	**obousměrný síťový provoz s omezenou rychlostí**

Tabulka 2.2: Hardwarové požadavky MIDP 1.0

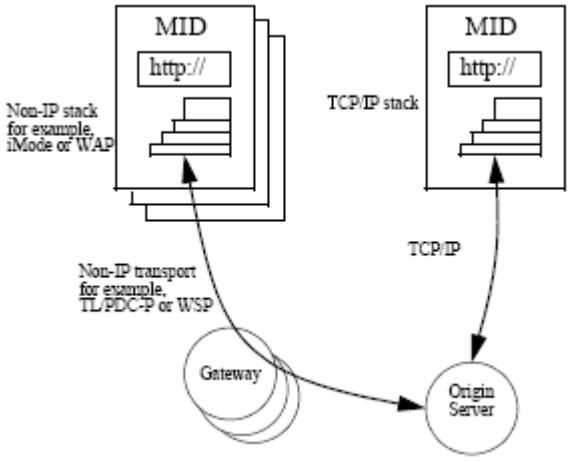
Hlavní části MIDP 1.0 rozšiřující CLDC:

Časovače (Timers)

MIDP rozšiřuje CLDC o časovače (Timers), jedná se o třídy `java.util.Timer` a `java.util.TimerTask`, které umožňují spouštět určité činnosti jednorázově i opakovaně s pevnou frekvencí nebo pevným intervalem. Aby bylo možné časovač použít, musí být MIDlet spuštěn. Jelikož MIDlety nedokáží běžet na pozadí, nelze vytvořit takovou aplikaci, která by nás každé ráno budila při běžně spuštěném zařízení.

Síť (networking)

V balíčku `javax.microedition.io` přibyla podpora HTTP (Hypertext Transfer Protocol) protokolu, který může být implementován IP protokoly (např. TCP/IP), ale také non-IP protokoly jako například WAP (Wireless Application Protocol) či i-mode, využívajících gateway. (Viz Obrázek 2.4).



Obrázek 2.4: HTTP Network Connection [13]

Persistentní ukládání dat v zařízení

MIDP přináší nový balíček `javax.microedition.rms`, který umožňuje aplikacím data trvale ukládat. Po vypnutí a znovu zapnutí zařízení lze data opět číst. Systém se nazývá RMS (Record Management systém) a zavádí databázový záznamový systém. Při odinstalování aplikace ze zařízení musí být aplikace odstraněna úplně a to i včetně veškerých uložených dat, které aplikace měla.

Uživatelské rozhraní

MIDP obsahuje nový balíček `javax.microedition.lcdui`, který zahrnuje třídy a rozhraní k tvorbě uživatelského prostředí v MIDletech. Grafické objekty, tvořící uživatelské prostředí dělíme do dvou kategorií tzv. vysokoúrovňové (high-level API) a nízkoúrovňové (low-level API).

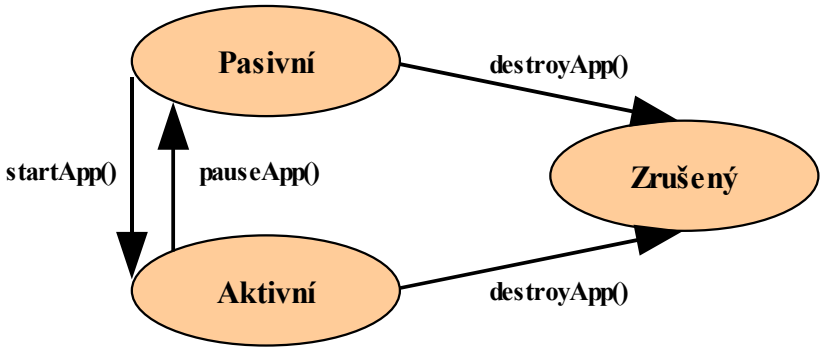
- **vysokoúrovňové API**
  Programátor, využívající objekty vysokoúrovňového API má jen malou kontrolu nad jejich vlastním vzhledem. Není ani zaručeno, že objekty budou na každém zařízení vypadat stejně. Každé zařízení si toto rozhraní může implementovat jinak. Aplikace, využívající vysokoúrovňové API nemají ani přístup ke konkrétním vstupním klávesám. Vysokoúrovňové API se používá hlavně tam, kde je třeba vytvořit nenáročnou, vysoce přenositelnou aplikaci.
- **nízkoúrovňové API**
  Přístup k obrazovce je realizován na úrovni pixelů, to nabízí programátorovi maximální kontrolu nad vzhledem aplikace, ale je třeba vynaložit mnohem větší úsilí při její tvorbě. Nízkoúrovňové API také obsahuje přístup ke konkrétním klávesám. Aplikace tedy může reagovat na různé stisky kláves numerické klávesnice mobilního zařízení. Nízkoúrovňové API se používá v aplikacích, ve kterých je třeba maximálního zachování vzhledu aplikace např. hry apod. Nevýhodou tohoto API je závislost na rozměrech displeje, tudíž nezaručuje tak vysokou přenositelnost jako vysokoúrovňové API.

Vysokoúrovňové API	**Alert, ChoiceGroup, DataField, Form, Gauge, Item, List, StringItem, TextBox, TextField, Ticker**
Nízkoúrovňové API	**Canvas, Graphic, Image**

Tabulka 2.3: Rozdělení High a Low-level API [13]

Správa průběhu aplikací

MIDP obsahuje balíček `javax.microedition.midlet`, zahrnující abstraktní třídu `MIDlet`. Aplikace, běžící v profilu MIDP, jsou známé pod pojmem midlety. Midlet obsahuje alespoň jednu třídu, která musí být potomkem abstraktní třídy `javax.microedition.midlet.MIDlet`. Musí implementovat abstraktní metody `startApp()`, `pauseApp()`, `destroyApp()`, které řídí životní cyklus aplikace. Aplikace se může nacházet v jednom ze tří stavů, které reprezentuje obrázek.



Obrázek 2.5: Životní cyklus Midletu

Když je MIDlet vytvořen, nachází se ve stavu *pasivní*. Jakmile je MIDlet připraven, kontrolní program ho přivede do stavu *aktivní*. V tomto stavu MIDlet běží a je možná interakce s uživatelem.

Aplikaci může přerušit sám program, nebo systém MIDP. Do stavu *zrušený* se může MIDlet přesunout z obou předchozích stavů. Jakmile je MIDlet ve stavu zrušený, měl by systému uvolnit všechny svoje obsazené zdroje.

2.6 MIDP 2.0

MIDP 2.0 přináší celou řadu novinek, zaměřených především na vývoj her. MIDP 2.0 je zpětně kompatibilní s MIDP 1.0. Obsahuje téměř jednou tolik tříd jako jeho předchůdce a proto se také zvýšily požadavky na hardwarové prostředky. MIDP 2.0 nevyžaduje žádnou konkrétní konfiguraci CLDC, ale předpokládá se, že většina zařízení s MDP2.0 bude obsahovat konfiguraci CLDC ve verzi 1.1. Tabulka 2.4 reprezentuje minimální požadavky na hardware s rozdílem oproti MIDP 1.0 (rozdíl uveden v závorkách).

	Minimální požadavky MIDP 2.0 vs. (MIDP 1.0)
Požadavky na Displej:	**velikost displeje minimálně 95x54 pixelů**
	barevná hloubka minimálně 1 bit = 2 barvy
	poměr stran pixelů přibližně 1:1
Vstupní zařízení:	**ITU-T telefonní klávesnice 0-9, QUERTY klávesnice, dotykový displej**
Požadavky na paměť:	**256kB (původně 128 kB) trvalé paměti pro MIDP komponenty**
	8 kB trvalé paměti pro data aplikace
	128kB (původně 32 kB) pro běh aplikace (Java heap)
Síťové požadavky:	**obousměrný síťový provoz, s omezenou rychlostí**
Zvuk:	**schopnost přehrávat tóny, ať už hardwarově nebo softwarově**

Tabulka 2.4: Hardwarové požadavky MIDP 2.0 vs. (MIDP 1.0)

Hlavní novinky obsažené v MIDP 2.0

Sítě (networking)

Balíček `javax.microedition.io` byl rozšířen o rozhraní pro podporu dalších protokolů. MIDP 2.0 v současnosti podporuje tyto protokoly:

- http
- https
- datagram
- datagram server
- soket
- server soket
- ssl
- comm

Herní rozhraní

Širokou oblastí využití J2ME je i psaní her pro mobilní telefony. Z toho důvodu je MIDP 2.0 obohaceno o balík `javax.microedition.lcdui.game`, určený speciálně k vývoji her pro mobilní zařízení. Na vývoji tohoto balíku se podílela řada předních výrobců mobilních telefonů (např. Nokia, Siemens apod.).

Hlavní myšlenkou herního rozhraní je skládání grafiky z několika vrstev grafických objektů. Například u scény znázorňující lyžaře sjíždějícího sjezdovku by jednu vrstvu tvořilo pozadí sjezdovky, druhou vrstvu například plápolající praporek a třetí vrstvou by byl sám lyžař.

GameCanvas

Jedná se o hlavní třídu herního rozhraní. Oproti klasické třídě `Canvas` obsahuje třída `GameCanvas` (off-screen) buffer, který umožňuje kreslení všech grafických operací nejprve v bufferu mimo obrazovku. Teprve po zavolání metody `flushGraphics()` je obsah bufferu vykreslen na obrazovku. Jelikož grafické operace jsou časově velmi náročné, je používání bufferu při více grafických operacích naprostou nutností. Dále třída `GameCanvas` poskytuje metodu `getKeyStates()` ke zjištění stavu tlačítek. Tato metoda vrací `int`, kde ke každé herní akci odpovídá jeden bit.

Konstanty, reprezentující herní akce, obsažené ve třídě `GameCanvas`:

- `FIRE-PRESSED`
- `DOWN-PRESSED`
- `LEFT-PRESSED`
- `GAME_A_PRESSED`
- `GAME_B_PRESSED`
- `GAME_C_PRESSED`
- `GAME_D_PRESSED`
- `RIGHT-PRESSED`
- `UP-PRESSED`

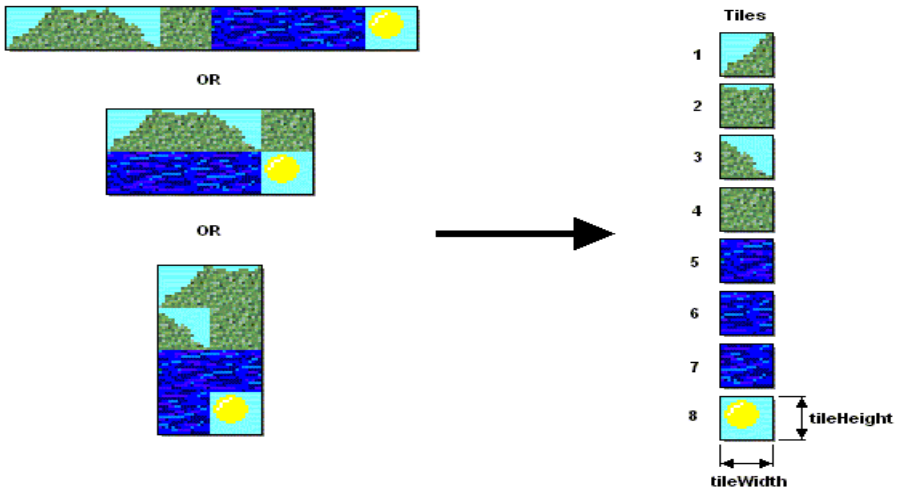
Layer (vrstva)

Třída `Layer` implementuje, jak již bylo výše zmíněno, systém vrstev. Vrstva je viditelný objekt s několika atributy. Každá vrstva má svoji pozici (souřadnice levého horního rohu, defaultně v bodě [0,0]) vzhledem k aktuálnímu souřadnicovému systému, výšku, šířku, viditelnost (viditelná resp. neviditelná). K vykreslení vrstvy je třeba volat její vlastní metodu `paint(Graphics)` v metodě `paint(Graphics)` třídy `GameCanvas`.

TiledLayer

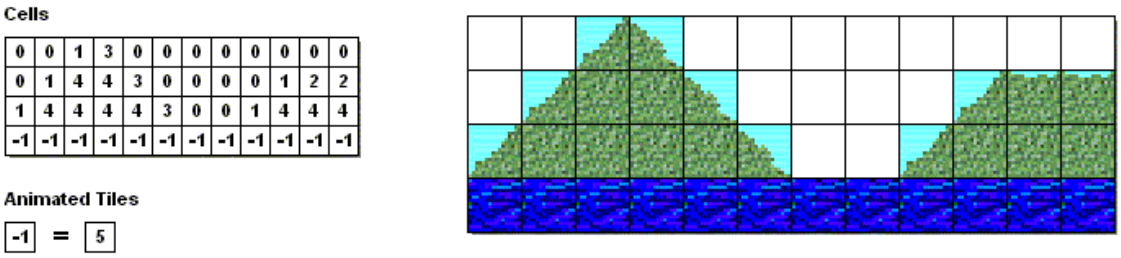
Třída `TiledLayer` se využívá především ke tvorbě rastrové vrstvy pozadí z jednotlivých dlaždic, které se opakují. Dlaždice se zadávají v jednom obrázku, který se pak předhodí konstruktoru třídy `TiledLayer`. Ten pak podle zadaných rozměrů jedné dlaždice automaticky rozřeže obrázek na sadu rámečků (dlaždic) o stejném rozměru. Dochází tak ke značným úsporám paměti tím, že není nutné

používat extrémně velké obrázky k vytvoření pozadí. Každé dlaždici je přiřazen unikátní index podle její polohy v obrázku. Indexy jsou číslovány od 1 (Viz. Obrázek 2.6).



Obrázek 2.6: TiledLayer, princip přidělování jedinečného indexu dlaždicím [14]

Obrázek 2.6 zobrazuje princip přidělování jedinečného indexu jednotlivým dlaždicím. Dlaždice umístěná v levém horním rohu obrázku má index 1. Zbývající dlaždice jsou pak číslovány v řádku podle pořadí (indexy jsou přiřazovány nejprve v rámci prvního řádku, pak druhého, třetího a tak dále).



Obrázek 2.7: Pozadí vytvořené pomocí TieldLayer [14]

Obrázek 2.7 vyobrazuje příklad výsledného sestavení pozadí z dlaždic podle jejich indexů. Index 0 v objektu třídy `TiledLayer` reprezentuje vždy index prázdného místa. Záporný index značí animovanou dlaždici.

Sprite

Třída `Sprite` je potomkem třídy `Layer`. Používá se pro animaci obrázků, jednoduché transformace např. různé rotace, překlápění a zrcadlení, umožňuje také kontrolu kolizí s jinou vrstvou. I zde jsou všechny rámečky v jednom zdrojovém obrázku a práce s nimi je obdobná jako u třídy `TiledLayer`. Zde index rámečku nezačíná od 1, nýbrž od 0. Třída `Sprite` nemá prázdnou

pozici. Pořadí v jakém se mají rámečky na displeji zobrazovat, určuje posloupnost dostupných rámečků ze zdrojového obrázku. Defaultně je posloupnost nastavena na {0, 1, 2, až počet dostupných rámečků – 1}. Pro animaci lze metodou `setFrameSequence (int[] novaPosloupnost)` nastavit novou posloupnost rámečků. Tuto posloupnost lze procházet za pomoci cyklických metod `nextFrame()` a `prevFrame()`. Metodu `setFrame(int indexRamecku)` použijeme pro přechod přímo na index rámečku v posloupnosti.

LayerManager (správce vrstev)

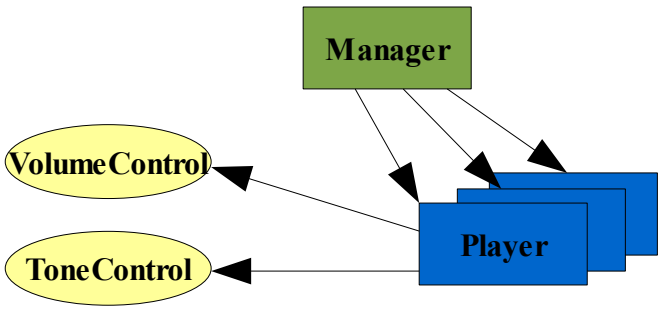
Správce vrstev udržuje setříděný seznam skupiny vrstev, jejichž pořadí určuje hodnota z-index a ulehčuje práci s těmito vrstvami. Jednotlivé vrstvy se do správce vkládají pouze jednou.

PushRegistry

Velkou novinkou v MIDP 2.0 jsou PushRegisty, které umožňují spustit aplikaci automaticky bez interakce uživatele. Spuštění aplikace můžeme buď naplánovat na určitý čas nebo iniciovat přes síťové rozhraní. Spuštění aplikace probíhá klasicky, voláním metody `MIDlet.startApp()`.

Zvuk

V MIDP 1.0 si ovládání zvuku řešil každý výrobce po svém. Nová verze MIDP 2.0 s sebou přináší balíčky `javax.microedition.media` a `javax.microedition.media.control`, které umožňují nenáročnou práci se zvukem v malých zařízeních pracujících s profilem MIDP. Obrázek 2.9 zobrazuje tři základní kameny (`Manager`, `Player` a `Contorl`) architektury rozhraní pro práci se zvukem



Obrázek 2.8: Rozhraní pro práci se zvukem v MIDP 2.0

3 J2ME síťová komunikace

Užitečnost mobilních zařízení, na kterých je J2ME implementována se zakládá především na jejich komunikaci s okolním světem. Tato kapitola se zabývá pouze zařízeními založenými na konfiguraci CLDC a hostující profil MIDP.

CLDC specifikace v J2ME oproti J2SE definuje zcela nový rámcový systém (framework) síťové komunikace tvořící základnu veškeré vnější konektivity, kterou mají dodávat profily založené na CLDC. Tento framework nese zkratku GCF (Generic Connection Framework) a je obsažen v balíčku `javax.microedition.io`. Jeho základ tvoří rozhraní `Connection`, které obsahuje jen jedinou metodu `close()`, sloužící k uzavření spojení. Rozhraní `Connection` nelze použít k vytvoření spojení, proto neobsahuje metodu `open()`. Spojení lze získat pomocí třídy Connector. Tato třída obsahuje tři statické metody `open()` vytvářející `Connection`.

Jelikož mobilní zařízení musí být schopno nepřetržitě reagovat na vstupní požadavky (stisky kláves, příchozí hovor, apod.), je nutné veškerou síťovou komunikaci implementovat ve vlákně určeném extra pro síťovou komunikaci.

3.1 Třída Connector

Třída `Connector` je jakousi továrnou vytvářející objekty typu `Connection`. Jak již bylo zmíněno výše, `Connector` obsahuje tři statické metody `open()`:

- `public static Connection open(String name)`
- `public static Connection open(String name, int mode)`
- `public static Connection open(String name, int mode, boolean timeout)`

Parametr `name` zde určuje adresu URL obvykle ve tvaru **schéma:adresa;parametry**. *Schéma* určuje typ používaného protokolu, *adresa* je identifikátor zařízení, ke kterému se má připojit a p*arametry* určují doplňující informace, potřebné k úspěšnému otevření spojení. Pro představivost uvedu dva příklady použití:

- `socket://192.168.1.155:10997`
- `http://www.seznam.cz/index.html`

První příklad vytváří spojení typu `socket` na IP adresu zařízení a port, na kterém naslouchá. Druhý příklad realizuje spojení typu `http` s použitím doménového jména.

Parametr `mode` nastavuje přístupový režim. Jedná se o volitelný parametr, který může nabývat jedné z těchto tří hodnot `Connector.READ` (pouze čtení), `Connector.WRITE` (pouze zápis) nebo kombinaci obou hodnot `Connector.READ_WRITE` (čtení i zápis současně). Defaultně je parametr `mode` nastaven na `Connector.READ_WRITE`. Je třeba si uvědomit, že existují zařízení, u kterých nelze použít defaultní nastavení, jelikož jej nepodporují. Mezi taková zařízení patří

například některé tiskárny, které nepodporují čtecí režim. Použitím režimu, který zařízení nepodporuje, způsobí vyvolání výjimky `IllegalArgumetException`.

Parametr `timeout` indikuje povolení používat časové limity (jsou-li podporovány) pro operace čtení a zápis. Jsou-li časové limity podporovány, pak každá operace čtení resp. zápis má svůj vnitřní časový limit na provedení operace. Pokud je `timeout` nastaven na `true` a časový limit konkrétní metody vyprší, je vyvolána výjimka `InteruptedIOException`. Časové limity se využívají zejména k odstranění zbytečného, příliš dlouhého blokování zařízení operacemi, pokoušející se o čtení nebo zápis. Aplikaci není nikterak umožněno měnit nastavení časových limitů.

3.2 Druhy připojení

Schopnost získat vstupní a výstupní proudy k zařízení, ke kterému jsme připojeni, definují dvě rozhraní `InputConnection` a `OutputConnection`. Obě dvě jsou odvozena od rozhraní `Connection` a každé z nich obsahuje po dvou druzích metod, umožňujících otevřít vstupní resp. výstupní proud. Rozhraní InputConnection k otevření vstupního proudu používá následující dvě metody:

- `public InputStream openInputStream() throw IOException`
- `public DataInputStream openDataInputStream() throw IOException`

Rozhraní `OutputConnection` k otevření výstupního proudu používá následující vlastní dvě metody:

- `public OutputStream openOutputStream() throw IOException`
- `public DataOutputStream openDataOutputStream() throw IOException`

Rozhraní InputStream resp. OutputStream umožňují přístup k proudům na úrovni bajtů. Jinak je tomu u DataInputStream resp. DataOutputStream, ty umožňující přenášet primitivní datové typy jako jsou např. int, char, String.

Je-li podporována obousměrná komunikace, je možné spojit metody rozhraní InputStream a OutputStream do jediného rozhraní StreamConnection.

3.3 Sokety

Sokety poskytují API, které zajišťuje přímý přístup k nižší úrovni síťové komunikace. Umožňují komunikaci dvou aplikací na různých zařízeních pomocí protokolu TCP/IP. K realizaci spojení mezi dvěma aplikacemi jsou zapotřebí dva odlišné sokety. První čekající na příchozí spojení (serverový) a druhý, který musí spojení vytvořit (klientský).

Klientské proudové sokety

K vytvoření klientského soketového spojení můžeme v J2ME použít dva způsoby. První způsob nabízí využití soketové implementace v konfiguraci CLDC 1.0 za pomocí rozhraní `StreamConnection`. Druhý způsob využívá rozhraní `SocketConnection` v profilu MIDP 2.0. Ať už se rozhodneme pro první či druhý způsob, postup při vyváření spojení je vždy obdobný popisu níže a při experimentech s implementací jsem nenarazil na žádné podstatné rozdíly.

Vytvoření klientského spojení:

- Aby se klient mohl připojit, musí nejdřív vytvořit adresu ve tvaru:
  `socket://adresa:port`
- Adresu předáme jako parametr `name` statické metodě `open()` třídy `Connector`. Společně s adresou nastavíme i parametr `mode` na hodnotu `Connector.READ_WRITE`. Tímto získáme vstupně-výstupní spojení.
- Ze získaného spojení vytvoříme StreamConnection nebo SocketConnection rozhraní.
- Pomocí metod openInputStream() a openOutputStream() otevřeme vstupní a výstupní proudy síťové komunikace.
- Po dokončení veškeré komunikace uzavřeme oba proudy pomocí metody `close()` rozhraní `Connection`.

Serverové proudové sokety

Činnost serverového soketu se od klientského liší především tím, že se nepřipojuje na žádné jiné zařízení, ale permanentně naslouchá na portu hostitele. Je tedy zřejmé, že parametr `name` metody `open()` nepotřebuje adresu cílového zařízení a neobsahuje ani adresu hostitele. Chceme-li například vytvořit server, naslouchající na portu 80, pak parametr `name` bude vypadat následovně:

- `socket://:80`

Další podstatný rozdíl, oproti klientskému soketu je, že metoda `open()` třídy `Connector` nevrací rozhraní typu `StreamConnection`, nýbrž `StreamConnectionNotifier`. Toto rozhraní obsahuje, kromě metody `close()` zděděné od `Connection`, také metodu `acceptAndOpen()`. Jedná se o metodu, keterá čeká až se na server připojí klient. Jakmile se tak stane, vrací objekt `StreamConnection`. Tento objekt již funguje jako klasický klientský soket na straně serveru. Pokud chceme, aby server obsloužil víc než jednoho klienta, je nutné metodu `acceptAndOpen()` volat ve smyčce.

Obvyklým způsobem pro obsloužení více klientů současně, je pro každého nově příchozího klienta vytvořit vlákno. Tím se dá vyhnout zpoždění, způsobující čekání na obsluhu jednotlivých klientů, ale na úkor velkých systémových nároků spojených s přepínáním mezi jednotlivými vlákny.

3.4 Datagramy

V CLDC je krom proudových soketů i podpora pro datagramy. Datagramy využívají ke komunikaci *nespolehlivý protokol UDP a oproti proudovým soketům se liší v několika pohledech.

- **Proud vs. zprávy** – Proudový soket posílá spojitý proud dat od odesilatele k příjemci. Datagramy žádné spojité proudy dat neposílají, místo toho zasílají data v oddělených paketech.
- **Připojení** – U proudových soketů se mezi odesilatelem a příjemcem vytvoří spojení ve kterém tečou veškeré proudy dat. Data jsou tedy posílána resp. přijímána jen jedním odesilatelem resp. příjemcem a nemůže se stát, že by data putovala k jinému cíli. U datagramu nic takového není, každá zpráva je adresována individuálně a může být zaslána různým cílům a stejně tak zprávu můžeme obdržet od více zdrojů.
- **Spolehlivost** – Nedojde-li k přerušení spojení, je přenos dat u proudových soketů zaručen. Stejně tak je zaručeno odeslání a příjem dat ve správném pořadí bez duplicit. U datagramu žádné takové záruky nejsou.
- **Náklady na spojení –** Aby proudové sokety mohly poskytovat takové záruky na přenosy dat po síti jaké poskytují, vyžadují se proto mnohem větší náklady na sestavení spojení než u datagramu. Z tohoto pohledu jsou proudové sokety ve značné nevýhodě oproti datagramům.

Parametr `name` metody `open(String name)` třídy `Connector` má stejné tvary jako proudové sokety.

- **Výstupní**
  `datagram://cil:port`
- **Vstupní**
  `datagram://:port`

Volání metody `open()` s jedním z těchto dvou typů parametru `name` vrací objekt `Dtagram-Connection`, který je odvozen od rozhraní `Connection`. Znamená to tedy, že nejsou k dispozici metody vracející vstupní resp. výstupní proudy. Namísto toho rozhraní `DatagramConnection` obsahuje čtyři metody vytvářející objekt `Datagram`:

- `public Datagram newDatagram(int size)`
- `public Datagram newDatagram(int size, String address)`
- `public Datagram newDatagram(byte[] buf, int size)`
- `public Datagram newDatagram(byte[] buf, int size, String address)`

* „*nespolehlivý*“ je zavádějící označení. Protokol UDP nezaručuje doručení datagramu, doručení ve správném pořadí a nemůžeme se spoléhat ani na to, že daný datagram nepřijde vícekrát.

První dvě metody přidělují datagramu vyrovnávací paměť o dané velikosti. Další dvě vytvářejí datagram s již přidělenou vyrovnávací pamětí. Parametr `address` slouží k překrytí parametru `name` metody `open()` a `size` u dvou posledních metod určuje, kolik je datagram schopný přijmout dat do vyrovnávací paměti.

Odeslání datagramu můžeme provést například takto:

- vytvoříme výstupní objekty `DatagramConnection` a `Datagram`,
- pomocí metody `setData(byte[] buffer, int offset, int len)` do datagramu umístíme odesílaná data,
- zavoláme metodu `send(Datagram dgram)` objektu `DatagramConnection` a předáme ji datagram k odeslání.

Příjem datagramu je o něco složitější v tom, že je potřeba dopředu znát velikost přijímaných dat k alokování dostatečně velké vyrovnávací paměti pro očekávaná data. Příjem datagramu:

- vytvoříme vstupní objekt `DatagramConnection`,
- alokujeme dostatečnou vyrovnávací paměť pro očekávaná data,
- vytvoříme objekt `Datagram` a přidělíme mu alokovanou paměť s nastavením maximální velikost přijímaných dat,
- voláním metody `receive(Datagram dgram)` objektu `Datagram-Connection` čekáme na příjetí datagramu.

4 Síťová komunikace J2SE a knihovna Java New I/O

Předpokládá se, že čtenář má alespoň základní zkušenosti s programováním v jazyce Java. Nebudu zde proto popisovat celou J2SE, ale zaměřím se pouze na její síťové možnosti při tvorbě proudového serveru.

Následující kapitola je silně spjata s předchozí kapitolou, ve které jsou mimo jiné vysvětleny principy síťové komunikace, datagramy a proudové sokety. Proto zde již popsané skutečnosti a principy nebudu dále popisovat.

4.1 Sokety

Princip soketů je stejný jako v předchozí kapitole (Viz Kapitola 3). Proto zmíním pouze rozdíly oproti implementaci soketů v J2ME.

Soketové API je v J2SE umístěno v balíku `java.net`. K vytvoření klientského soketu se používá instance třídy `Socket` a k vytvoření serverového soketu instance třídy `ServerSocket`.

Aby se klientský soket mohl připojit, musí se konstruktoru třídy `Socket` předat **adresa** a **port** cíle. Adresa může být instancí třídy `String` a nebo instancí třídy `InetAddress`. Adresa typu `InetAddress` v sobě zapouzdřuje IP adresu cíle. Instance třídy `InetAddress` se získá pomocí statické metody `getByName(String host)` třídy `InetAddress`. Tato metoda na vstupu očekává buď IP nebo URL adresu cíle.

Klientský soket (Socket)

Jak již bylo zmíněno (Viz Kapitola 3), klientský soket zastupuje pouze jednu stranu spojení. Při vytváření klientského soketu je možné použít tyto dva konstruktory:

- `Socket(InetAddress address, int port)`
- `Socket(String address, int port)`

Spojení se vytváří tak, že konstruktoru třídy `Socket` je předána adresa a port cíle. Získáme tak instanci třídy `Socket`, které je automaticky přidělen jeden z volných portů počítače. Jakmile dojde k vytvoření spojení, můžeme použít odpovídající metody instance třídy `Socket` k získání vstupního a výstupního proudu `java.io.InputStream` a `java.io.OutputStream`. Tyto proudy obsahují metody `read(byte[] b)` pro načtení a `write(byte[] b)` pro odeslání bajtového pole.

Serverový soket (ServerSocket)

Před vytvořením serverového soketu je potřeba určit hostitelský port, na kterém bude soket očekávat nově příchozí klienty. Serverový soket tvoří instance třídy `ServerSocket` a je ji možné vytvořit

voláním konstruktoru `ServerSocket(int port)`, kterému předáme předem určený hostitelský port. Předáme-li jako port (0), pak se hostitelský port vygeneruje automaticky.

Nově příchozího klienta přijímáme pomocí metody `accept()` objektu `ServerSocket`. Tato metoda blokuje program dokud se někdo nepřipojí. Metoda `accept()` při příchodu nového klienta vytváří instanci třídy `Socket` reprezentující soket příchozího klienta a po jeho vytvoření řízení programu vrací. Pomocí nově vytvořeného objektu `Socket`, je možné s klientem komunikovat.

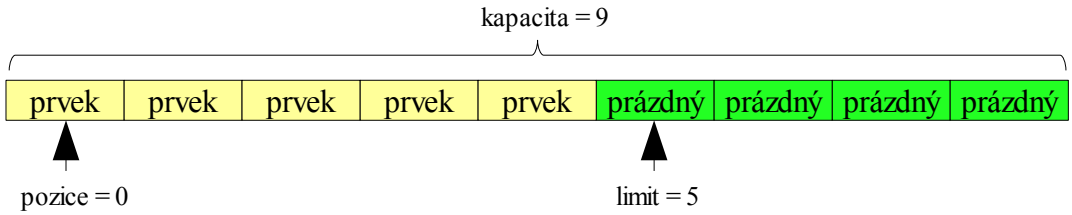
4.2 Java New I/O knihovna

Knihovna New I/O přináší pět balíků z nichž pouze dva byly použity pro tuto práci. Jedná se o balíky pro práci s buffery, síťovými kanály a selektory:

- `java.nio`
- `java.nio.channels`

Buffery

Balík `java.nio` obsahuje deset tříd pro práci s buffery, z toho osm přímo pro práci s buffery primitivních datových typů, kromě typu `boolean`. Jedná se o sofistikovanější a rychlejší řešení než klasická javová pole. Hlavními prvky každého bufferu je jeho kapacita, pozice a mez. Kapacita udává maximální počet prvků, které buffer může obsahovat. Pozice je index aktuálního prvku, který má být čten nebo zapisován. Mez označuje index první pozice, kterou ještě nelze přečíst, nebo na kterou nelze zapsat.



Obrázek 4.1: Nastavení kapacity, pozice a meze bufferu

Všechny buffery jsou zděděny od společné třídy `Buffer`. Ta poskytuje metody pro základní práci s buffery:

- `clear()` - nastaví mez na kapacitu bufferu a pozici na 0, tím umožní zápis do bufferu od jeho začátku.
- `flip()` - nastaví mez na aktuální pozici a pozici na 0. Používá se k ohraničení dat v bufferu. Zajišťuje, aby se při přenosech nepřenesla nějaká data navíc.
- `rewind()` - pozici nastavuje na 0 a mez nechává nezměněnou. Umožňuje opětovné načítání dat.

Alokace bufferu se provádí metodou `alocateDirect(int kapacita)` pro přímé buffery a `alocate(int kapacita)` pro nepřímé buffery. Přímé buffery nabízejí vyšší výkon, nepřímé se využívají tam, kde je potřeba častá realokace.

Kanály

Balík `java.nio.channels` obsahuje několik kanálů pro práci se sítí. Jsou to například kanály pro práci s datagramy, se sokety, roury a další. Princip datagramů byl vysvětlen výše (Viz Kapitola 3). Kanály společně s buffery z New I/O mohou nabízet totožnou funkčnost jako třídy `InputStream` a `OutputStream`.

V implementaci práce byly použity kanály typu `SocketChannel` a `ServerSocket-Channel`, proto se dále zmíním podrobněji jen o této dvojici tříd.

Klientský kanál (SocketChannel)

Tato třída je odvozena od mnoha rozhraní, ostatně jako všechny kanály obsažené v balíku `java.nio.channels`. Kanál typu `SocketChannel` v sobě zapouzdřuje klientský soket a jeho využití je stejné jako instance třídy `Socket`. Společně s použitím Selektoru (třída `Selecto` bude zmíněna níže) a objektu `ServerSocketChannel` (třída bude taktéž popsána níže) umožňují vytvořit neblokující, bezvláknový server.

Hlavní metody třídy SocketChannel

- `public static SocketChannel open(SocketAddress remote)`
- `connect(SocketAddres remote)`
- `public void close()`
- `public int read(ByteBuffer dst)`
- `public int write(ByteBuffer dst)`

Metoda `open(SocketChannel remote)` vytvoří instanci třídy `SocketChannel`, otevře kanál soketu a připojí se na adresu zadanou v parametru `remote`. Zavoláme-li `open()` s prázdným parametrem, musíme později k připojení soketu použít metodu `connect(Socket-Addres remote)` objektu `SocketChannel`. Třída `SocketAddress` je rodičovskou třídou třídy `InetAddress`.

Metoda `close()` uzavře kanál.

Metoda read(ByteBuffer dst) načítá data z kanálu do bufferu. Vrací počet načtených dat. Vrátí-li hodnotu -1, kanál byl uzavřen.

Metoda write(ByteBuffer dst) zapisuje data z bufferu na kanál. Před jakýmkoli zápisem bufferu je třeba volat metodu `flip()`, která ohraničí data v bufferu. Metoda vrací počet zapsaných dat.

Serverový kanál (ServerSocketChannel)

Třída `ServerSocketChannel` obsahuje metodu `open()` vracející její instanci. Objekt vytvořený touto metodou ještě není plnohodnotný serverový kanál. Aby byl kanál kompletní a byl schopný naslouchat, je zapotřebí k němu přidružit objekt `ServerSocket` získaný metodou `socket()` a zavoláním jeho metody `bind(SocketAddress address, int port)`.

Přijetí nového kanálu příchozího klienta je realizováno metodou `accept()` třídy `ServerSocektChannel`.

Pomocí metody `configureBlocking(boolean block)` je vhodné nastavit serverovému kanálu neblokující mód. Volba módu zásadně ovlivňuje metodu `accept()`. Bude-li nastaven neblokující mód, pak metoda `accept()` vrací buď `SocketChannel` klienta nebo null a nijak neblokuje běh aplikace. Při blokujícím módu metoda `accept()` blokuje síťové vlákno do doby, dokud se na server někdo nepřipojí.

Po ukončení práce se serverem je zapotřebí uzavřít `ServerSocket` jeho vlastní metodou `close()` a `ServerSocketChannel` také jeho vlastní metodou `close()`.

4.2.1 Selektor

Selektor je nejdůležitější částí multiplexního, neblokujícího, bezvláknového serveru. Selektor je jednoduše řečeno správce skupiny kanálů, u kterých je možné nastavit příznak aktivity. Díky tomuto příznaku lze zjistit, který kanál je aktivní.

Selektor je obsažen v balíku `java.nio.channels`. Jeho instanci tvoříme metodou `open()` třídy `Selector`. Nově vytvořený selektor je prázdný a neobsahuje žádné kanály. Abychom do selektoru dostali nějaký kanál, musíme jej do něj zaregistrovat.

Registrace kanálu

Dříve než kanál zaregistrujeme do selektoru, je třeba jej přepnout do neblokujícího módu. Registrace do selektoru se provádí metodou `register(Selector sel, int ops)` příslušného kanálu. První parametr očekává odkaz na selektor, kam se má kanál zaregistrovat. Druhý parametr očekává hodnotu reprezentující příznak aktivity. Pokud kanál tuto metodu neobsahuje, není selektorem podporován. Jako příznak aktivity může být nastavena jedna z konstant třídy `SelectionKey` nebo bitové součty více konstant. Konstanty třídy `SelectionKey`:

- OP_ACCEPT
- OP_CONNECT
- OP_READ
- OP_WRITE

Registrace kanálu s příznakem např. OP_READ zajistí, aby se selektor zajímal o kanál pouze v případě, že z něj nějaká data přicházejí na server. Návratovou hodnotou metody `register()` je instance třídy `SelectionKey`. Ta reprezentuje zaregistrovaný kanál v selektoru a umožňuje mimo jiné tento kanál ze selektoru odregistrovat metodou `cancel()`.

Výběr aktivních kanálů ze selektoru

K výběru slouží tři druhy metod `select()`:

- `select()` - blokuje síťové vlákno, dokud nějaký zaregistrovaný kanál neprojeví aktivitu
- `select(long timeout)` - blokuje síťová vlákno po dobu `timeout` v milisekundách
- `selectNow()` - nijak neblokuje vlákno, okamžitě se vrací bez ohledu na to kolik kanálů je aktivních

Metody `select()` a `SelectNow()` vracejí počet aktivních kanálů a pomocí metody `SelectedKeys()` je možné získat množinu `Set<SelectionKey>` těchto kanálů. Objekt `SelectionKey` lze převést zpět na kanál metodou `channel()`. Dále už je práce s kanálem typická dle typu kanálu.

5 Analýza

Následující kapitola je zaměřená především na analýzu práce a na specifikaci požadavků, které byly kladeny hned na začátku vývoje aplikace.

Dle zadání má být cílem této práce vytvořit netriviální aplikaci reprezentující zvládnutí klient/server problematiky. Pro demonstraci klientské části byla zvolena jednoduchá hra pro mobilní telefon inspirovaná vzorem legendární hry Dyna Blaster. Druhou část tvoří samotný herní server pro tuto aplikaci.

5.1 Požadavky na funkčnost

Server

Hlavní požadavky na server jsou specifikovány v následující bodech:

- Server musí být schopný přijmout a obsluhovat velké množství klientů.
- Server musí pracovat jako neblokující a multiplexní. Výhradně nemá být implementován jako vícevláknový server.
- Veškeré akce, které budou klienti generovat proběhnou na straně serveru, výsledky budou distribuovány všem klientům.
- Kolize související s pohybem klientů bude řešit server.

Klient

- Klientská aplikace má být realizována jako tenký klient, veškeré akce uživatele budou vykonávány na serveru.
- Klient musí být schopný zobrazit sebe i všechny ostatní klienty připojené na serveru.
- Klient musí podporovat interakci uživatelů, například možnost pohybu.
- Klient by měl být schopný pracovat na zařízení s minimální velikostí obrazovky 176*220 pixelů a dokázat se pohybovat i po rozsáhlé mapě větších rozměrů než jsou rozměry displeje.

5.2 Minimální požadavky na mobilní zařízení

Aplikace předpokládá, že mobilní zařízení bude splňovat tyto minimální požadavky.

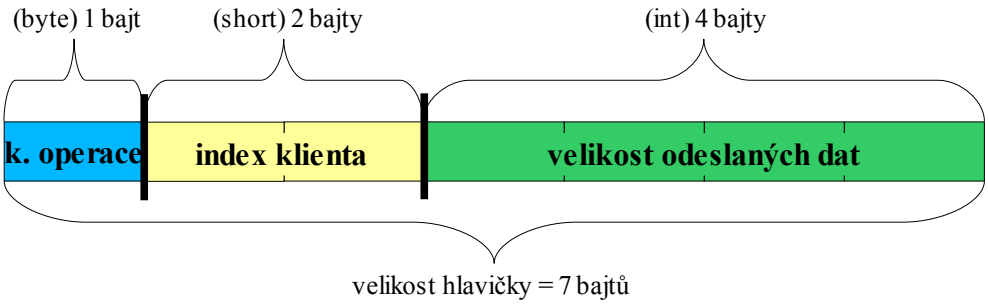
- Konfigurace CLDC 1.1.
- Profil MIDP 2.0.
- Minimální velikost obrazovky 176*220 pixelů.
- Přístup k datové síti alespoň GPRS, ale silně doporučeno WiFi z důvodu nižší odezvy a velkého počtu přenášených paketů při hře s více hráči. Na sítích 3G/UMTS netestováno, ale předpokladem je podstatné zlepšení hratelnosti oproti GPRS.

6 Návrh implementace

6.1 Hlavička

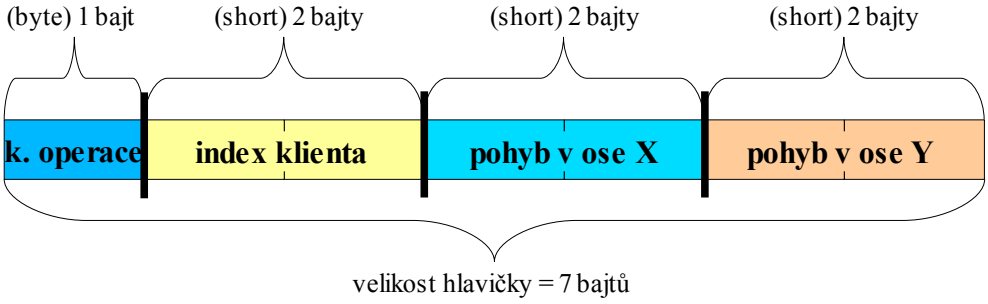
Každou informaci putující ze serveru ke klientovi a naopak je třeba nějak přenést. K tomu je určena hlavička, která slouží jako univerzální nosič informací, a která jako celek informuje o provedené operaci.

Hlavičkou je blok o velikosti sedm bajtů. K vytvoření klientské a serverové aplikace je zapotřebí dvou druhů hlaviček (klientskou a serverovou). Díky pohybovým operacím musí klientská hlavička nést o jednu informaci navíc než hlavička putující ze serveru ke klientovi a to při zachováni stejné velikosti hlavičky.



Obrázek 6.1: Serverová odchozí hlavička

Obrázek 6.1 reprezentuje návrh hlavičky posílané ze serveru směrem ke klientům. Tato hlavička zapouzdřuje trojici různých datových typů číselných informací. První část hlavičky (1 bajt) obsahuje číselný kód operace, kterou hlavička jako celek reprezentuje. Druhou částí jsou 2 bajty obvykle nesoucí informaci o identifikaci klienta, který operaci vyvolal. Jako poslední částí jsou 4 bajty datového typu `int`, které informují o velikosti aktualizačních dat odeslaných těsně za hlavičkou.



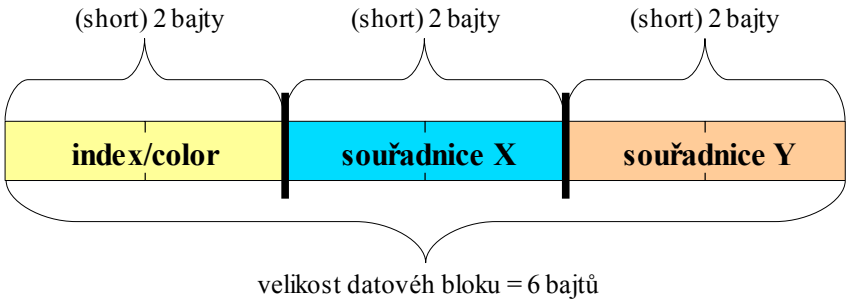
Obrázek 6.2: Klientská odchozí hlavička

Na obrázku 6.2 je vyobrazena struktura hlavičky posílané z klientské aplikace na server. Hlavička zapouzdřuje čtveřici různých informací o dvou datových typech. První dvě části hlavičky

mají totožný význam jako u serverové hlavičky. Poslední dvě reprezentují souřadnice pohybu klienta po mapě.

6.2 Datový blok

Datové bloky se posílají jen jedním směrem a to od serveru ke klientovi. Data musí být předcházeny hlavičkou informující klienta, že mu budou zaslána aktualizační data a také kolik dat má klient na vstupu očekávat. Datový blok v sobě musí zapouzdřovat informace o jednom objektu (klient, bomba nebo prázdný blok). Jeho využití je především k aktualizaci dat, zajišťující stejné a konzistentní datové vektory klienta a serveru. Následují obrázek vyobrazuje strukturu jednoho datového bloku.

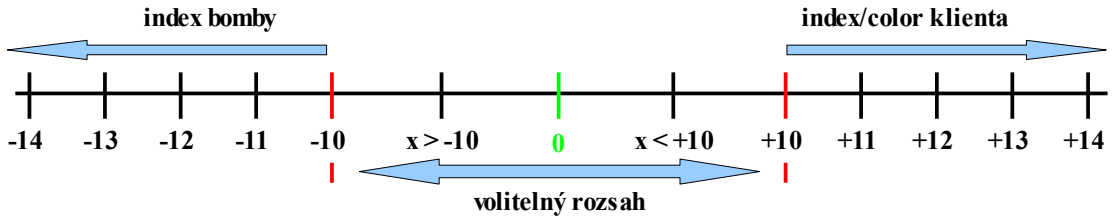


Obrázek 6.3: Struktura datového bloku

Jak již bylo řečeno, datový blok reprezentuje objekt (klient, bomba a nebo prázdný blok). První část datového bloku objektu typu klient obsahuje ve většině případů index klienta, ale může také obsahovat číslo barvy klienta a to jen v případě, když se klient na server nově připojí. Jedná-li se o datový blok bomby, pak první část bloku bude reprezentovat index bomby. Prázdný blok bude obsahovat na všech pozicích hodnotu 0. Poslední dvě části bloku reprezentují vždy souřadnice daného objektu na mapě.

6.3 Odlišení různých objektů pomocí indexů

Jelikož datové bloky mají reprezentovat data různých objektů, je třeba je od sebe nějakým způsobem odlišit. K rozlišení dat nejlépe vyhovuje číselná hodnota první části datového bloku.



Obrázek 6.4: Rozlišení a rozsah indexů

Obrázek 6.4 zobrazuje způsob řešení, jak rozlišit různé druhy dat. Hodnota 0 byla vybrána pro identifikaci prázdného bloku, který slouží jen k vyplnění prázdného místa v datovém vektoru. Použití

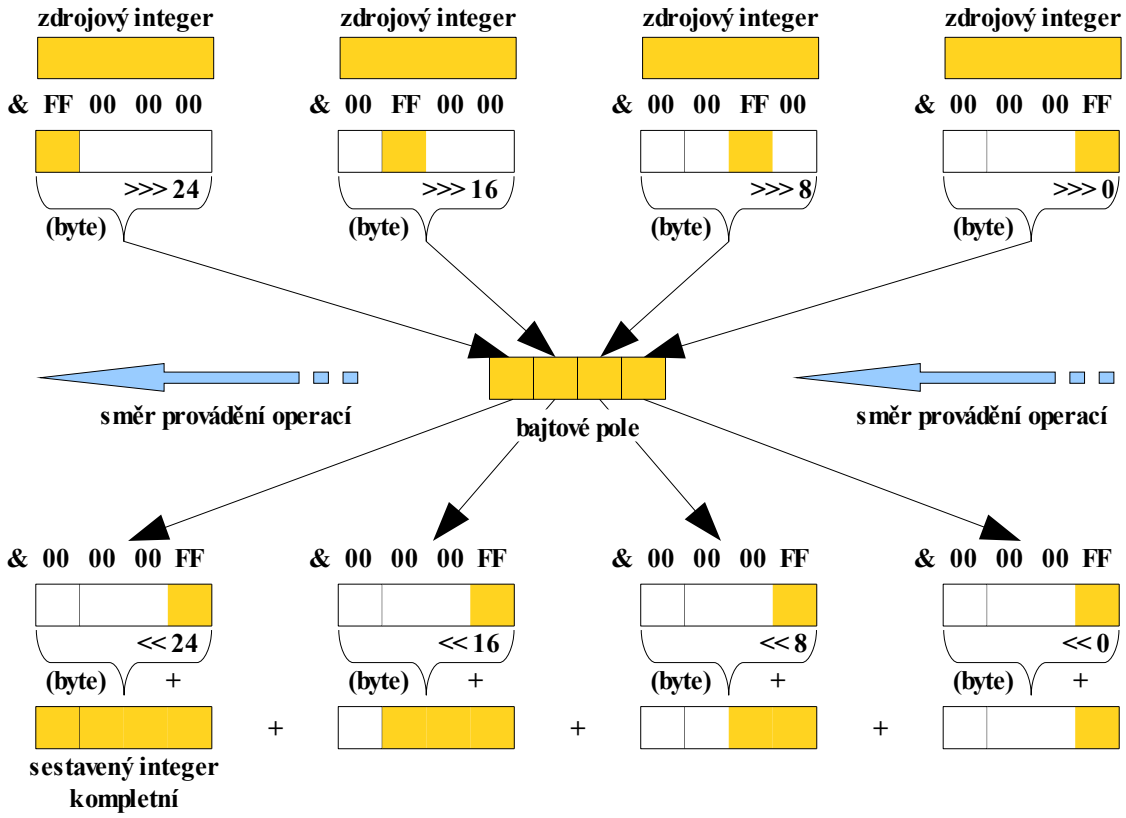
prázdného bloku je nutné proto, aby se zamezilo rozsypání indexů při odpojení klienta uvnitř vektoru klientů či výbuchu bomby uložené někde uprostřed vektoru bomb. Výbuchem bomby nebo odpojení klienta se rozumí jejich odstranění z příslušného datového vektoru.

Existence prázdného bloku přináší problém, jak od sebe rozeznat tři různé nuly. Jádrem problému je, že první přihlášený klient má index 0 a první položená bomba má také index 0, k tomu ještě připočtěme prázdný blok reprezentovaný také hodnotou 0. Řešením tohoto problému je posunutí indexu do záporného či do kladného směru o konstantní hodnotu 10. Tímto se zajistí rozlišení klientské nuly, nultého indexu bomby a hodnoty 0 reprezentující prázdný blok. Vznikne také okolí hodnot kolem hodnoty 0 a to v rozsah <-9;0) a (0;+9>. Tyto hodnoty nemají přiděleny žádné významy a proto zůstanou nevyužity jako volitelný rozsah.

Další problém spočívá v tom, jak od sebe rozlišit klientská data a data bomb. Jako řešení bylo zvoleno rozlišení kladnou a zápornou hodnotou (Viz Obrázek 6.4).

6.4 Převody celočíselných datových typů

Hlavička a datový blok jsou tvořeny polem bajtů o velikosti 7 bajtů pro hlavičku a 6 bajtů pro datový blok. Pole jsou rozděleny do jednotlivých částí, přičemž každá část reprezentuje informaci v podobě nějaké hodnoty celočíselného datového typu (`byte`, `short` nebo `int`). Aby bylo možné hlavičku nebo blok sestavit, je zapotřebí vytvořit převodní metody, které dokážou převést primitivní datové typy `short` a `int` na pole bajtů a naopak. Návrh převodní metody pro typ `int` je na obrázku 6.5.



Obrázek 6.5: Návrh převodu datového typu integer do bajtového pole a zpět

Obrázek 6.5 vyobrazuje návrh převodní metody na kterou jsou kladeny požadavky na rychlost a jednoduchost převodu. Metoda si vystačí jen se základními operacemi jako jsou posuvy bitů, přetypování a sčítání.

6.5 Komunikační protokol

Komunikace je založená na výměně zpráv, které reprezentují hlavičky (Viz podkapitola 6.1) a přenosu dat reprezentovaných datovými bloky (Viz podkapitola 6.2). Komunikace vždy začíná startovací zprávou ze serveru, kdy server se klientovi představí pozdravem a očekává klientovu odpověď. V aplikaci klient/server komunikaci vždy inicializuje klient a v rámci TCP protokolu server přijímá spojení.

Význam každé zprávy reprezentuje kód operace v prvním bajtu zprávy. Následující tabulka obsahuje všechny použité kódy operací a jejich význam v protokolu. První sloupec tabulky udává kód operace, druhý obsahuje název zprávy reprezentující operaci v programu. Ve třetím sloupci je uvedena strana, která zprávu odesílá, kde C-klient, S-server a Sb-server broadcast.

Kód operace	Zpráva	Strana	Popis
120	HELLO	S,C	Pozdrav server/klient, server registruje klienta.
100	ACCEPT	S	Akceptování klienta serverem, hlavička obsahuje identifikační údaje klienta.
-100	NON_ACCEPT	S	Server zahlcen, klient nebyl akceptován.
-121	LOGOUT	C	Odpojení klienta ze serveru.
-122	CLIENT_LOGOUT	Sb	Informace klientům, že se některý klient odpojil.
112	PING	C,S	Udržování spojení mezi klientem a serverem.
101	DATA_REFRESH	Sb	Zpráva o aktualizaci dat, zpráva obsahuje informaci o velikosti dat zasílaných za zprávou.
55	PUT_BOMB	C	Požadavek na položení bomby.
-55	EXPLOSION_BOMBS	Sb	Zpráva o výbuchu bomby nebo sady bomb na serveru, zpráva obsahuje informaci o velikosti dat výbuchu, zasílaných za zprávou.
-50	BOMB_EXPLOSE	S	Zpráva o výbuchu klientovi bomby.
33	PUTTING	S	Potvrzení položení klientovi bomby.
-33	NOT_PUTTING	S	Položení bomby na danou pozici není možné.
66	KILLER	S	Zpráva klientovi o počtu zabitých nepřátel jeho bombou.
-66	KILLED	S	Zpráva klientovi o jeho zabití.
10	MOVE	C	Požadavek pohybu na souřadnice [x,y].

Tabulka 6.1: Kódy operací komunikačního protokolu

6.5.1 Sémantika komunikačního protokolu

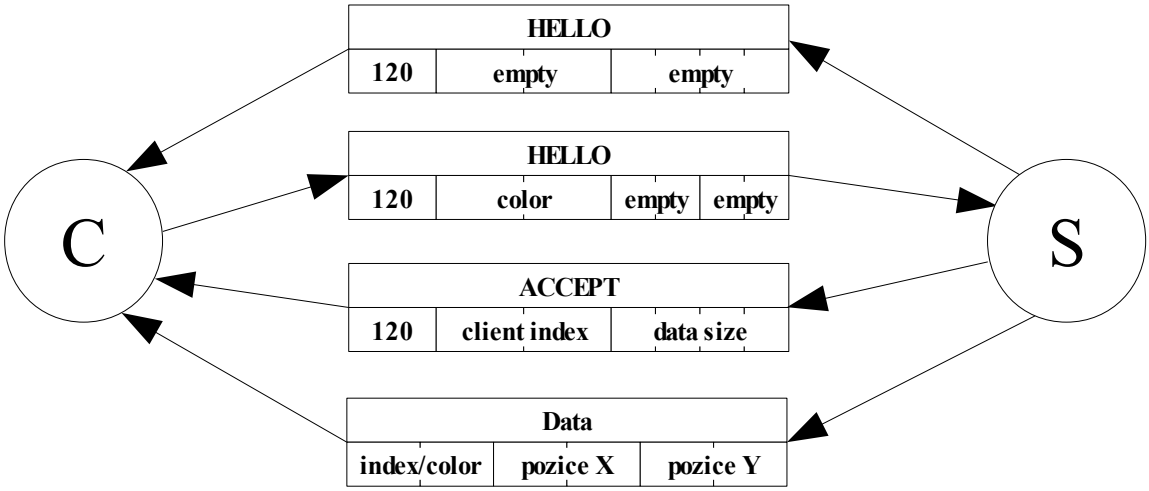
Sémantiku komunikačního protokolu udává v jakém pořadí mají být a nebo musí být zprávy posílány. Celou sémantiku lze shrnout do třech hlavních částí:

- Navázání spojení.
- Výměna zpráv a refresh dat.
- Ukončení spojení.

Komunikační zprávy reprezentují hlavičky, obsahující informace potřebné k realizaci požadované operace na serveru. Každá odchozí zpráva z klienta, kromě zprávy **HELLO**, musí obsahovat identifikátor (**index**) klienta. Identifikátor klientovi přidělí server a zašle mu jej až v druhé zprávě **ACCEPT**.

Navázání spojení

Navázání spojení je inicializováno vždy z klientovy strany. Server zahajuje komunikaci zasláním uvítací zprávy **HELLO**, na kterou klient musí odpovědět taktéž uvítací zprávou **HELLO**, ve které serveru předává informaci o zvolené barvě postavičky. Server čeká na zprávu **HELLO** od klienta a po jejím obdržení zašle klientovi zprávu **ACCEPT,** obsahující identifikátor klienta (**index**) a velikost inicializačních dat, zasílaná hned za zprávou **ACCEPT**. Obdrží-li server při navazování spojení s klientem jinou zprávu, než klientskou **HELLO** spojení odepře.



Obrázek 6.6: Navázání spojení s obsahem hlaviček

Výměna zpráv, refresh dat

Veškerou komunikaci obstarávají zprávy uvedené v tabulce (Viz Tabulka 6.1) s výjimkou zpráv **HELLO**, **ACCEPT** a **LOGOUT**. Tato trojice zpráv je určena jen a pouze k navázání a ukončení spojení, v komunikaci klienta se serverem se dále nijak nepoužívají. Komunikaci může probudit jedna ze čtyř možných akcí.

Situace probouzející komunikaci:

1) Klient pošle požadavek na pohyb nebo položení bomby
2) Výbuch bomby/bomb
3) Udržování spojení
4) Odpojení klienta

situace	Zprávy
1	MOVE, PUT_BOMB, DATA_REFRESH
2	BOMB_EXPLOSE, KILLER, KILLED, EXPLOSION_BOMBS, DATA_REFRESH
3	PING
4	LOGOUT, DATA_REFRESH, CLIENT_LOGOUT

Tabulka 6.2: Rozdělení zpráv dle situace

Komunikace mezi serverem a klientem v případě první situace probíhá tak, že klient posílá požadavek na vykonání operace na server a výsledek této operace je klientovy zaslán společně s výsledky operací ostatních klientů v nejbližší možné *aktualizaci dat. Datová aktualizace probíhá ve většině případů jednou za cca 20ms, ale může být předčasně vyvolána odpojením některého klienta a příchodem zprávy **LOGOUT** nebo také výbuchem bomby.

Druhou situaci vyvolává informace o výbuchu jedné nebo více bomb naráz v jednom okamžiku na serveru. Informování o výbuchu vyvolá odeslání několika zpráv (Viz Tabulka 6.2). Za zprávou **EXPLOSION_BOMBS** se odesílají data potřebná k vykreslení výbuchu na klientech. Počet zpráv **BOMB_EXPLOSE** a **KILLER** je proměnlivý a je závislý na počtu výbuchem zasažených bomb a klientů.

Udržování spojení je iniciováno klientem a probíhá v případě, kdy po dobu 30s ze serveru nepřicházejí žádná data. Nepřijde-li do 30s ze serveru žádná zpráva, klient odešle zprávu **PING** a dalších 30s čeká na odpověď. Neodpoví-li server do 30s, klient ukončí spojení se serverem.

Při odpojování zasílá klient serveru zprávu **LOGOUT** a uzavře spojení, server zprávu příjme a s klientem spojení zruší. Server aktualizuje data (**DATA_REFRESH**) a všem klientům rozešle informaci o odpojení daného klienta, zprávu **CLIENT_LOGOUT**.

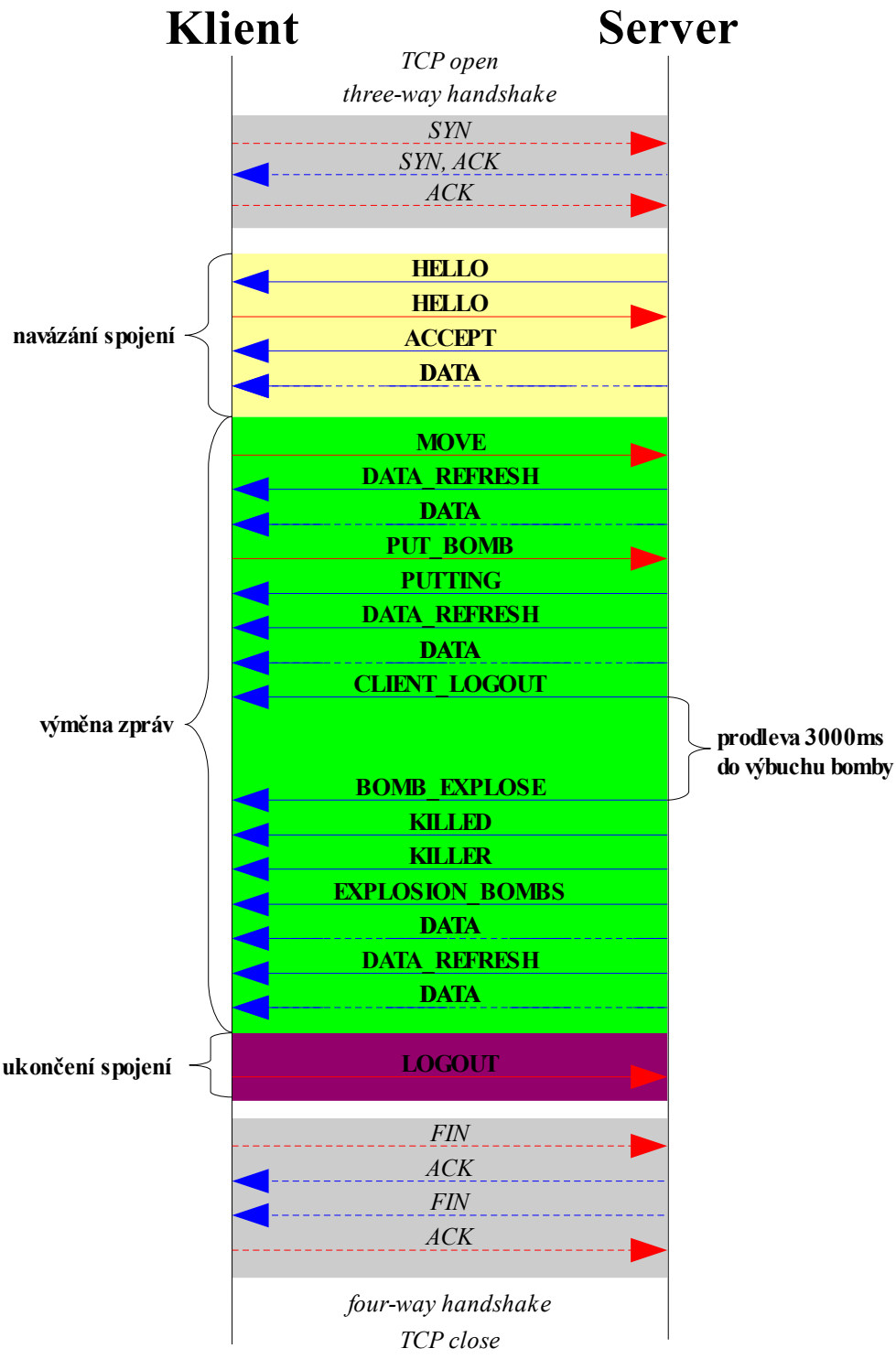
Ukončení spojení

Komunikaci může ukončit kterýkoliv z klientů zasláním zprávy **LOGOUT**. Po jejím odeslání klient uzavře spojení se serverem a server po jejím obdržení uzavírá spojení s klientem na svojí straně.

Ukončení spojení může také nastat v případě příjmu špatné hlavičky, jejíž formát se neshoduje s očekávaným formátem hlavičky (Viz podkapitola 6.1), jejíž kód operace se neshoduje s kódem v tabulce kódů operací (Viz Tabulka 6.1). Ukončení může také způsobit zahlcení serveru nebo rozpad spojení způsobený vnějšími vlivy nebo jakékoliv neočekávané situace, mající vliv na rozpad či ztrátu spojení.

* Aktualizace dat, jedná se pouze o rozdílovou aktualizaci dat. Pokud nejsou žádná aktualizační (rozdílová) data k dispozici, zpráva typu DATA_REFRESH společně s daty se neposílá.

Příklad komunikace



Obrázek 6.7: Komunikační protokol, příklad komunikace

Obrázek 6.7 reprezentuje příklad komunikačního protokolu dle následujícího scénáře: Klient se připojí na server, server jej registruje a akceptuje. Klient se jednou pohne a pokládá bombu, mezi tím

došlo k odpojení jiného klienta ze serveru. Klient setrvává na pozici položené bomby až do chvíle jejího výbuchu. Bomba po 3s vybuchuje a zabíjí klienta. Klient je informován o výbuchu jeho položené bomby. Klient obdrží zprávu, že bomba zabila nějakého klienta a také, že on sám byl zabit. Dále se odešlou data určená k vykreslení výbuchu. Poté se klient ze serveru odpojuje. Komunikace končí.

7 Implementace

V této kapitole budou popsány hlavní implementační detaily, nejdůležitější třídy a zajímavé funkce, obsahující výsledné aplikace. Implementace vychází z výše probrané teorie a návrhu (Viz Kapitola 2 až Kapitola 4 a Kapitola 6). Obecné principy popsané výše se v této kapitole již nebudou opakovat.

7.1 Server.java

Server je založený na principu neblokujícího, multiplexního serveru. K jeho implementaci byly využity technologie poskytující knihovna Java New I/O (Viz podkapitola 4.2). Konkrétně se jedná o třídy `Selector`, `ByteBuffer`, `SocketChannel` a `ServerSocketChannels`.

Selektor je jádrem celého serveru, zajišťuje veškerou komunikaci mezi serverem a připojenými klienty mimo přijímání spojení. Jakmile se spustí server a vytvoří síťové vlákno, dojde k otevření serverového kanálu metodou `openServerSocket()`. Serverový kanál slouží pouze pro příjem nového spojení a naslouchá na lokální adrese hostitele, kterou zjistí pomocí statické metody `getLocalHost()` třídy `java.net.InetAddress`.

Síťové vlákno

- **run()**
  Tato metoda vytváří síťového vlákno, uvnitř kterého běží nekonečná smyčka se zpožděním 20ms. Smyčka v sobě volá tři hlavní metody serveru `acceptNewConnections()`, `readIncomingMessages()` a `testBombExplosion()`. Dohromady tyto metody zprostředkovávají kompletní obsluhu všech požadavků, které klienti iniciují.

- **acceptNewConnections()**
  Metoda vyřizuje žádosti o připojení na server, nově připojené klienty pak registruje do selektoru. Metoda blokuje běh programu dokud neobslouží celou množinu připojujících se klientů.

- **readIncomingMessages()**
  Metoda pomocí selektoru a jeho metod ke zjištění aktivních kanálů, vytvoří množinu kanálů, posílajících požadavky na server. Tuto množinu pak prochází a každý aktivní kanál obslouží. Obsluha aktivního kanálu spočívá v načtení klientem zasílané hlavičky, rozebrání hlavičky na jednotlivé části a dle jejího kódu provedení požadované operace. Metoda běží dokud neobslouží všechny aktivní klienty, poté zavolá metodu `dataRefresh()`, ta rozešle rozdílovou aktualizaci dat všem klientům.

- **testBombExplosion()**
  Prochází vektor uložených bomb a testuje jednotlivé bomby na výbuch. Tato metoda se ve smyčce provádí jako úplně poslední. To proto, aby klienti měli vždy před jakýmkoli

výbuchem všechna data aktualizovaná a bylo zřetelné, koho výbuch zasáhl. Metoda prochází jednotlivé bomby ve vektoru položených bomb a porovnává jejich detonační čas s aktuálním. Výbuch bomby je spojen s prohledáváním okolí bomby, zda se v její bezprostřední blízkosti nenachází další bomba, která bude výbuchem zasažena a také odpálena a nebo klient, kterého výbuch zabije. Vybuchlá bomba pak odešle informaci svému autorovi o tom, že explodovala a zasažený klient obdrží zprávu o jeho zabití. Po provedení testu se všem klientům odesílají data, která se použijí k vykreslení výbuchu (Viz Obrázek 6.7: příklad komunikace) a nová aktualizační data, která obsahují nové pozice zabitých klientů.

Vektory

Data přihlášených klientů a položených bomb se ukládají na server do datových vektorů.

Login klienta

Klient, který se na server připojí je nejdříve registrován v selektoru, ale stále není přihlášen. K přihlášení klienta dojde až ve chvíli, kdy server obdrží od klienta zprávu HELLO. Poté je vytvořena instance třídy `Client`, obsahující základní informace o klientovi doručené zprávou HELLO. Nově vytvořený objekt klienta je uložen do vektoru klientů, je mu přiřazen identifikátor jeho pozice ve vektoru. Klientovi jsou zaslány jeho identifikační údaje na serveru společně s daty v datových vektorech. Až v tuto chvíli je klient na serveru přihlášen a může komunikovat.

Proudy dat, výměny zpráv, aktualizace

Server provádí aktualizaci síťového vlákna jednou za 20 ms, po tuto dobu se požadavky na server staví do fronty a jsou obslouženy až ve chvíli budoucí aktualizace. Je-li ve frontě více než jeden požadavek, pak ve chvíli aktualizace jsou všechny hromadně obslouženy a každá změna dat je uložena na buffer. V době dokončení obsluhy všech požadavků jsou data z bufferu odeslána všem klientům v podobě aktualizačních dat a buffer je vymazán.

Odhlášení klienta

Odhlášením klienta je klient odstraněn z datového vektoru, ostatním klientům je rozeslána informace o odpojení jednoho z nich.

7.2 Move.java

Třída `Move` obsahuje globální mapu rozmístění veškerých objektů umístěných na serveru. Jedná se o dvourozměrné pole typu `int`. Využívá se především k detekci kolize objektu s objektem např. při pohybu. Jakmile je na server doručen požadavek klienta na nějakou akci související s pohybem po mapě, volá se metoda `clientMove()` instance třídy `Move`. Metodě se do parametrů předají veškeré informace o klientovi (spojení s klientem v podobě `SocketChannel`, index klienta a souřadnice pohybu). Dále už se o veškeré akce s vyřízením požadavku postará metoda sama.

Mimo metody související s pohybem po mapě jsou k dispozici i metody pro generování nové, volné pozice na mapě. Tyto metody pracují tak, že nejprve vygenerují pseudonáhodné číslo, poté jej převedou do souřadnicového systému globální mapy a testují danou pozici zda je prázdná či nikoli. Narazí-li na prázdné místo, s hledáním končí a vrací pozici volného místa. Pokud se tak nestane, opakují celý proces znovu až do nalezení prázdného místa. Jedná se o tyto dvě metody:

- `public int generNewPositionX()`
- `public int generNewPositionY()`

Využívají se při přihlášení nového klienta, kdy je potřeba index klienta přiřadit na volnou pozici v globální mapě, ale také v případě klientova zabití.

K odlišení objektů umístěných na mapě existují 4 druhy indexů. Indexy reprezentující bomby a klienty, index prázdného místa a index statického políčka mapy. Index prázdného místa je vždy 0 a index mapy vždy -1. K odlišení bomb a klientů je použit stejný princip popsaný výše (Viz podkapitola 6.3).

7.3 Convert.java

Společná třída pro server i klienta. Obsahuje důležité statické metody pro převody datových typů `int` a `short` do bajtových polí a naopak (Viz podkapitola 6.4).

7.4 MultiBomber.java

Jedná se o `MIDlet` klientské aplikace. Třída `MultiBomber` v sobě zastřešuje obsluhu síťového vlákna, správu jednotlivých obrazovek aplikace a také správu veškerých commands (tlačítek), které aplikace obsahuje. V této třídě je také implementována kompletní tabulka kódů operací síťového protokolu (Viz Tabulka 6.1).

Třída obsahuje trojici metod pro správu aplikace (Viz str. 12). Metoda `startApp()` ve svém těle obsahuje uzamykatelný kód, který se spustí jen při prvním zavolání metody `startApp()`. Je tomu tak proto, aby nedocházelo znova k inicializace všech proměnných při probouzení aplikace nacházející se ve stavu *Pasivní* (Viz Obrázek 2.5).

Jak již bylo zmíněno, síť běží ve vlákně. Obsluha sítě musí běžet v externím vlákně zejména proto, že `MIDlet` nesmí být nijak blokován. Důvodů je hned několik, např. obsluha klávesnice, obsluha příchozího hovoru, sms. Tyto akce nesmějí být nikdy blokovány jinou akcí.

Síťové vlákno

- **run()**
  Metoda při startu síťového vlákna nejdříve inicializuje všechny herní datové vektory, aby bylo možné data ze sítě ukládat. Teprve poté vytváří spojeni mezi klientem a serverem (Viz str. 19). Je-li spojeni navázáno, vstoupí se do nekonečné smyčky, která je zpožďována o stejný timeout jako u serveru (20 ms). Uvnitř smyčky jsou volány dvě hlavní metody třídy `MultiBomber` obsluhující veškerá akce s příjmem a zpracováním hlavičky.

- **readHeadAndConvert(InputStream clientInput)**
  Tato metoda načte a rozebere příchozí hlavičku na jednotlivé informace v ní obsažené (Viz Obrázek 6.1). Ty jsou pak v celé třídě dostupné pomocí metod `iGetOp()`, `iGetIndex()`, `iGetSize()`. Informace, které tyto metody vracejí jsou dostupné do okamžiku nového načtení hlavičky.

- **IncomingMessageProccessing()**
  Metoda je volána po každé načtené hlavičce a zabezpečuje provedení patřičných akcí odpovídajících kódu operace příchozí hlavičky.

7.5 MyGameCanvas.java

Tato třída je potomkem třídy `GameCanvas` (Viz. str. 14) a obstarává herní rozhraní celé klientské aplikace. K implementaci rozhraní bylo použito základních tříd pro tvorbu herních rozhraní v profilu MIDP 2.0 (Viz podkapitola 2.6). Jedná se o třídy `Sprite` a `TiledLayer`, které se starají o vykreslování grafických objektů na displeji.

Zajímavostí této třídy je implementace řešení posouvání obrazovky po obrazu, který je příliš velký a celý na obrazovce nelze zobrazit. Řešení poskytuje metoda `translate(int x, int y)` třídy `Graphics`, která umožňuje posun počátečního bodu obrazovky. Posuneme-li počáteční bod proti směru plánovaného pohybu, navodíme tím tak dojem posunutí obrazovky v požadovaném směru po velkém obraze.

7.6 StatusLine.java

Jak název sám napovídá, jedná se o implementaci stavového řádku klientské aplikace viditelného na spodní straně obrazovky. Aby stavový řádek byl na každém zařízení na stejné pozici, je třída `StatusLine` založená pouze na možnostech nízkoúrovňového API. Šířka stavového řádku bude vždy maximum šířky obrazovky. Výška je rovna 1/4 výšky obrazovky zařízení, na kterém je aplikace spuštěna.

Třída `StatusLine` implementuje lokátor aktuální polohy uživatele na mapě. Podle této polohy dokáže vykreslit v pravém horním rohu žlutý rámeček, reprezentující aktuálně zobrazenou plochu rozsáhlé mapy.

7.7 PreloadCanvas.java

Třída je potomkem třídy `Canvas` a je součástí klientské aplikace. Obrazovka vytvořená touto třídou slouží k tomu, aby uživatele informovala o stavu aktuálně prováděné operace. Příkladem implementace je preloader běžící při startování hry, který informuje o stavu připojování. Metoda `increase()` třídy `PreloadCanvas` slouží k inkrementaci stavového políčka.

8 Testování odezvy serveru a obsluhy požadavků

Mezi nejdůležitější kritéria serveru patří doba odezvy. Odezva je časový okamžik začínající odesláním dotazu a končící příjmem odpovědi na daný dotaz. Další kritérium, které ovlivňuje odezvu je čas, po který je server schopen obsloužit určitý požadavek. Testování serveru a klientské aplikace se týká především těchto dvou kritérií.

Tato kapitola bude obsahovat návrh a implementaci tří druhů testování společně s jejich výsledky. Budou zde zmíněny jednak nástroje použité pro testování, emulátory **Java(TM) Wireless Toolkit** a **MicroEmulator** a také zařízení, na kterých testování probíhalo. K monitorování vytížení procesoru a alokované paměti byl použit program **JConsole** obsažený v J2SE od verze 5.0.

Testování bylo prováděno za pomoci deseti aktivně komunikujících klientských aplikací připojených na server. Každá aplikace zasílá požadavek s periodou 250 ms. Při testování byly vyhodnocovány výsledky související s dobou odezvy a obsloužením požadavku.

8.1 Testovací nástroje

Sun Java(TM) Wireless Toolkit 2.5.2 for CLDC (emulátor)

Wireless Toolkit obsahuje sadu nástrojů a technologií od firmy Sun Microsystems, určených pro vývoj aplikací pro malá zařízení jakými jsou mobilní telefony. Tento balík nástrojů obsahuje emulátor, který slouží především k testování aplikací bez nutnosti jejich instalace přímo do paměti mobilního telefonu. Tento emulátor nabízí užitečné nástroje vhodné k testování, mezi nejužitečnější patří monitor alokované paměti a monitor síťového provozu.

MicroEmulator

Jedná se o Java emulátor Java 2 ME (J2ME) psaný v jazyce Java 2 SE (J2SE) týmem nezávislých vývojářů. Oproti Wireless Toolkit je tento emulátor daleko rychlejší, což je zřetelné i z výsledků testů níže.

Tato práce byla testována v aplikaci *MicroEmulator verze 2.0.4. Starší verze 2.0.3 měla potíže s odchytáváním kláves přes metodu `getKeyStates()` třídy `GameCanvas`. Novější verze tímto nedostatkem již netrpí.

JConsole

Aplikace JConsole umožňuje monitorovat využití systémových prostředků jedné nebo více aplikací najednou. Slouží především k monitorování zátěže CPU či alokované paměti sledované aplikace. Samozřejmostí je také možnost ukládání všech výsledků monitorování. JConsole je produkt

* V době psaní této zprávy byla poslední verze programu dostupná na stránce: http://www.microemu.org/

firmy Sun Microsystems, je tedy velmi pravděpodobné, že výsledky poskytované touto aplikací jsou věrohodné.

8.2 Testovací zařízení

Testy byly prováděny na dvou počítačových sestavách a třech mobilních telefonech:

Sestava:

Zařízení: Notebook
Procesor: Intel Pentium M 1,73GHz
Operační paměť: 2x1024MB
Grafická karta: Intel Media Graphics Accelerator 900 (sdílená paměť 128MB)
Konektivita: LAN, WiFi
Operační systém: Windows XP Professional, SP 2

Zařízení: Desktop
Procesor: Intel Pentium 4 2.00GHz
Operační paměť: 1x512MB + 1x256MB
Grafická karta: NVIDIA GeForce4 MX 440 64MB
Konektivita: LAN
Operační systém: Windows XP Professional, SP 2

Mobilní telefon:

Telefon: Sony Ericsson G900
Procesor: 208MHz
Paměť: 128MB RAM, 160MB ROM
Displej: 240x320 px, 262 tisíc barev
Konektivita: GPRS, WiFi
Operační systém: Symbian 9.1 UIQ 3.0

Telefon: Sony Ericsson P1i
Procesor: 180MHz
Paměť: 128MB RAM, 160MB ROM
Displej: 240x320 px, 262 tisíc barev
Konektivita: GPRS, WiFi
Operační systém: Symbian 9.1 UIQ 3.0

Telefon: Sony Ericsson K750i
Procesor: 110MHz
Paměť: 34MB
Displej: 176x220 px, 262 tisíc barev
Konektivita: GPRS

8.3 Návrh a implementace testování

Testování bylo prováděno na malé lokální síti skládající se z následujících zařízení a jejích rolí v testování:

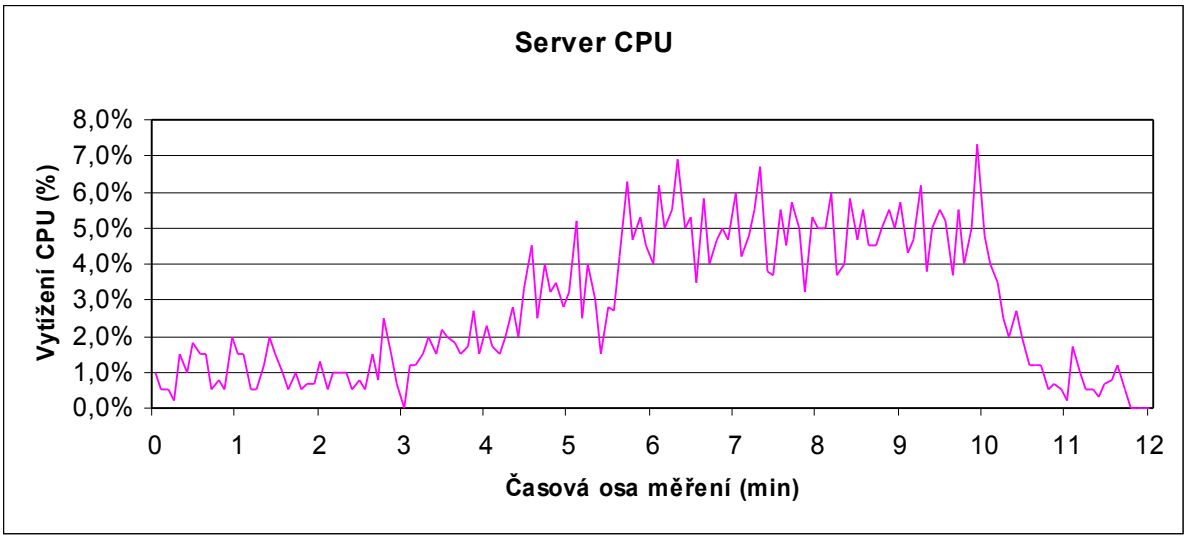
- Desktop PC: server.
- Notebook: zastupující strana deseti klientů.
- Router obsahující integrovaný bezdrátový přístupový bod.
- Mobilní telefon: nezávislý klient

8.4 Vytížení serveru při obsluze klientů

Tento test si klade za cíl provést statistiku vytížení serveru při připojení a hromadné obsluze množiny deseti aktivních klientů. Aktivita klienta je založena na náhodném pohybu do 4 stran v periodách 250 ms a položením jedné bomby každých 3000ms.

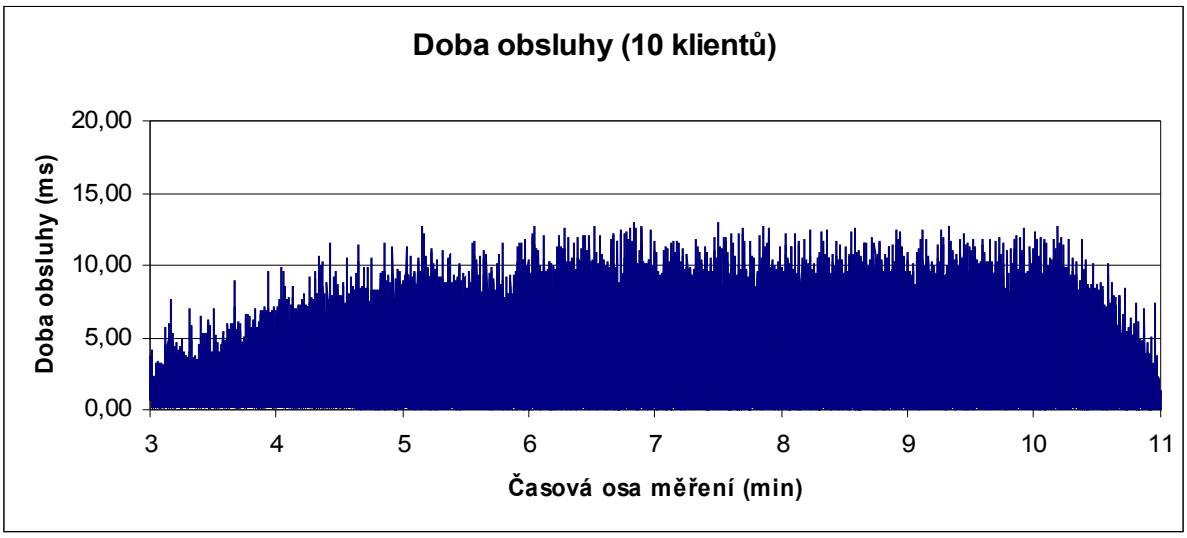
Test probíhal po dobu 12 minut, kdy v prvních 3 minutách server běžel na prázdno a až poté se klienti postupně připojovali. Na serveru byly monitorovány 3 hlavní hodnoty:

- vytížení procesoru
- doba hromadné obsluhy všech aktivních klientů v daném časovém okamžiku
- alokovaná paměť



Obrázek 8.1: Vytížení CPU serveru při aktivních klientech

Z grafu je patrné, jak server běží první 3 minuty naprázdno, klienti se začínají postupně připojovat v rozmezí třetí a páté minuty testu. Po dobu 4 minut je server vytížen požadavky od všech připojených klientů. Od desáté minuty testu dochází k postupnému odpojování všech klientů.



Obrázek 8.2: Doba hromadné obsluhy aktivní klientů

Na obrázku 8.3 je znázorněna rostoucí doba obsluhy při zatížení serveru. Každý záznam reprezentuje dobu obsluhy všech aktivních klientů v daném časovém okamžiku. Doba obsluhy je závislá na množství přijatých požadavků v daném okamžiku. Časová osa reprezentuje započetí měření prvním připojeným klientem a ukončení měření posledním odpojeným klientem. Průměrná doba obsluhy mezi šestou a desátou minutou, kdy server vytěžovalo deset klientů je 5.7 ms.

Doba obsluhy zde reprezentuje časový okamžik mezi přijetím všech zpráv od aktivních klientů, jejich zpracování a hromadné rozeslání výsledků všem klientům.

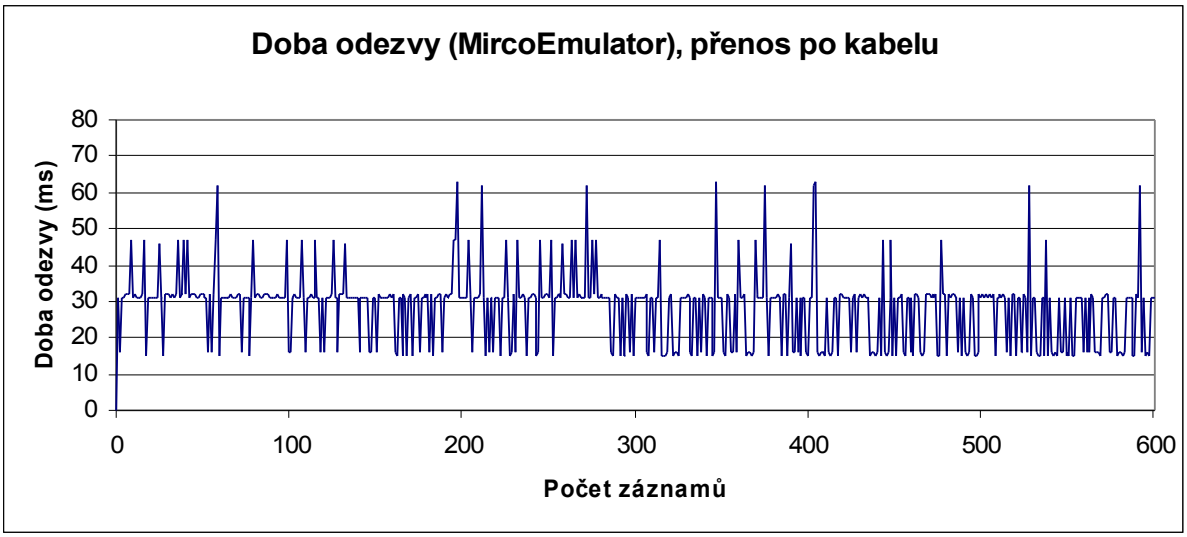
Paměť, která byla alokována v průběhu testu nepřesáhla hodnotu 2Mb, v průměru bylo alokováno 1.4 Mb.

8.5 Doba odezvy, přenos po kabelu

V tomto testu je hlavním cílem změřit průměrný čas potřebný k odeslání požadavku na server, jeho zpracování serverem a přijetí odpovědi klientem. Jako přenosové médium bylo zvoleno propojení kabelem.

K provedení testu bylo použito deset klientů nastavených tak, aby zasílali požadavky na server každých 250 ms. K monitorování byl použit referenční klient se záznamem rozdílu mezi časovými údaji odeslání požadavku a příjmu odpovědi. Server v tomto testu neodesílá klientům hromadnou odpověď, ale každému klientovi odpovídá zvlášť. Test byl vyhotoven dvakrát, jednou za použití nástroje **MicroEmulator** a podruhé s použitím **Wireless Toolkit** emulátoru.

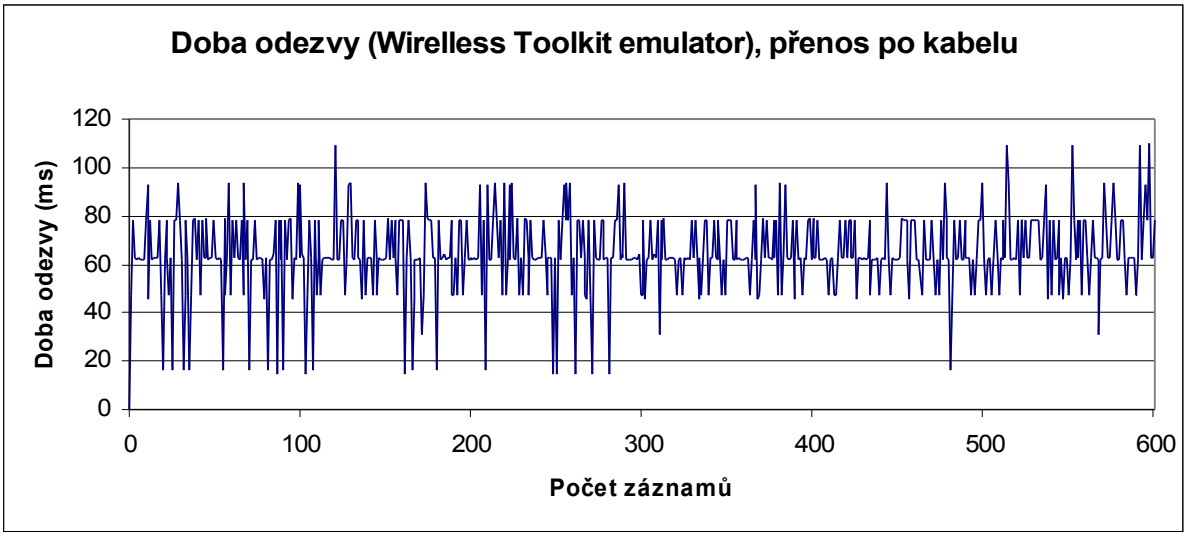
Popis infrastruktury sítě v tomto testu: Zařízení jsou připojena k serveru pomocí kabelu, avšak v cestě jim stojí router s integrovaným switchem, do kterého je kabelem připojen server i notebook jako klientská stanice.



Obrázek 8.3: Odezva serveru, MicroEmulator, přenos po kabelu

Z obrázku 8.4 jsou patrné jisté výchylky v odezvě, ale je jasně vidět, že průměrná odezva se pohybuje okolo 30 ms. Tomu odpovídá i vypočtená průměrná odezva z naměřených 600 hodnot, ta je rovna 28,30 ms.

Druhý testovaný emulátor má pro svůj provoz podstatně větší požadavky na systémové zdroje, než výše zmíněný MicroEmulátor. V případě nadměrného vytížení stroje, na kterém emulátor běží, dochází k velkému zkreslení výsledků. Z tohoto důvodu byli ostatní klienti spuštěni pod nástrojem MicroEmulátor, který má znatelně menší systémové nároky než Wireless Toolkit emulátor.



Obrázek 8.4: Doba odezvy, Wireless Toolkit, přenos po kabelu

Obrázek 8.5 reprezentuje výsledky naměřené odezvy na druhém emulátoru. Z výsledků je jasně vidět, že ačkoli se jedná o totožný test, je průměrná doba odezvy emulátor Wireless Toolkit asi o polovinu vyšší než u MicroEmulatoru. Domnívám se, že zpoždění způsobuje sám emulátor

Wireless Toolkit implementací ladících nástrojů a síťové komunikace. Průměrná doba odezvy tohoto testu vypočtená ze 600 záznamů je 64,05 ms.

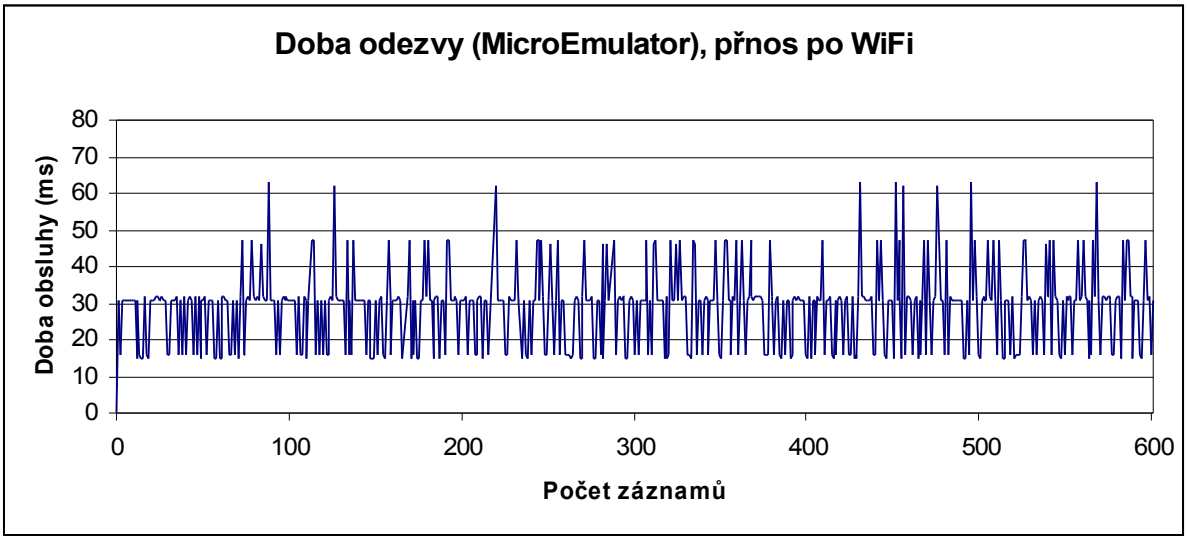
8.6 Doba odezvy, přenos po WiFi

Tento test je ve všech bodech scénáře totožný s předchozím, výše uvedeným testem. Přenosové médium v tomto testu tvoří vzduch a testovací zařízení jsou k serveru připojena pomocí WiFi. Jedním z připojených zařízení je mobilní telefon, na kterém běží současně i výpočet průměrné doby odezvy z naměřených hodnot v časovém intervalu jedné minuty.

Cílem tohoto testu je zjistit dobu odezvy při bezdrátové komunikaci oproti předešlému testu a ověřit si předpoklad, že bezdrátová síť způsobí zvýšení odezvy.

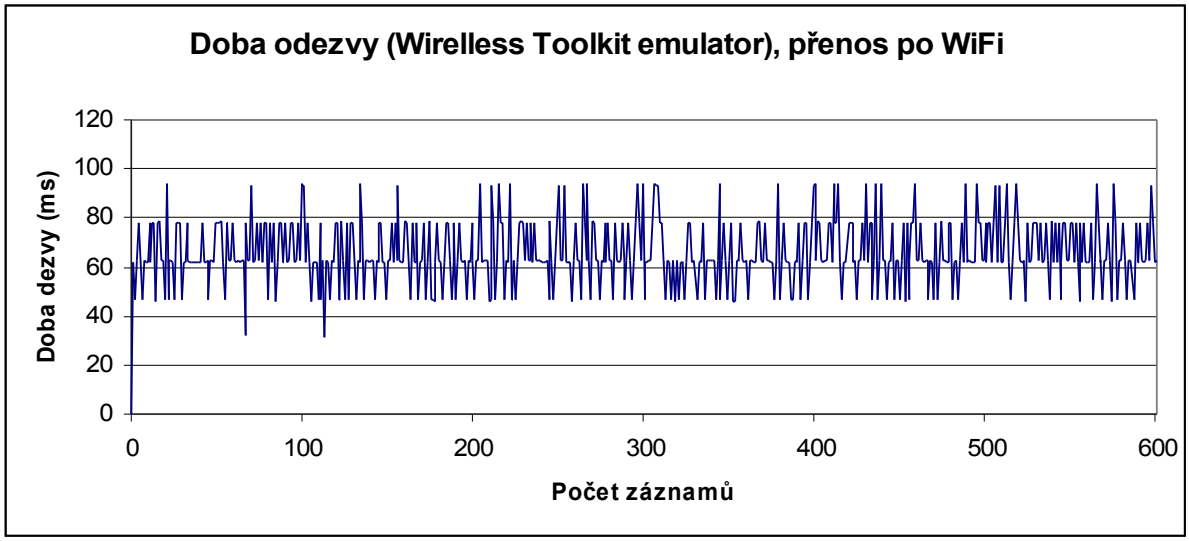
Popis infrastruktury sítě: Zařízení jsou k serveru připojená pomocí WiFi sítě. V cestě jim však stojí router, který pracuje jako Access Point, a který je se serverem propojen kabelem.

Stejně jako předchozí test, je i tento vyhotoven dvakrát a to pro **MicroEmulator** a **Wireless Toolkit** emulator.



Obrázek 8.5: Doba odezvy, MicroEmulator, přenos po WiFi

Předpoklad vyšší odezvy pro 10 klientů, způsobené přechodem na bezdrátovou síť se nijak nepotvrdil. Průměrná doba odezvy pro MicroEmulátor zůstává stejná. Průměrná doba odezvy vypočtená z 600 vzorků se rovná 30,29 ms.



Obrázek 8.6: Doba odezvy, Wireless Toolkit, přenos po WiFi

Doba odezvy pro Wireless Toolkit emulátor zůstává taktéž téměř nezměněna. Průměrná doba odezvy vypočítaná z hodnot 600 vzorků se rovná 65,38 ms.

Průměrná doba odezvy vypočítaná z naměřených hodnot na mobilním telefonu se rovná 88,47 ms.

8.7 Závěr a shrnutí testů

Díky výše provedeným testům si dovolím tvrdit, že rozdíly ve zpoždění odezvy způsobuje implementace bezdrátového síťového rozhraní samotného mobilního telefonu a také další různé aspekty jako například špatný nebo vůbec žádný signál, více WiFi AP (Access Point) v okolí způsobujících rušení signálu vysíláním na stejných kanálech. Nasvědčuje tomu i fakt, kdy při orientační zkoušce WiFi spojení byl proveden test odezvy od samotného routeru, programem **Pocket Ping** na mobilním zařízení s operačním systémem Windows Mobile. Tento test spojení vykázal průměrnou dobu odezvy cca 150 ms s výkyvy 300 ms, místy i 500 ms. Tento test byl jen zkouškou spojení v zarušených prostorách CVT FIT a nebyl nijak dokumentován. Uvádím jej pouze jen jako příklad. Celkovou dobu odezvy ovlivňuje i kvalita použitých stavebních prvků počítačové sítě. V případě testování této práce je dalším místem, kde mohou nastávat zpoždění, router SMC WBR14-G obsahující integrovaný switch a AP.

Testování se skupinou deseti aktivních klientských aplikaci hodnotím jako úspěšné a průměrnou odezvu 30 ms pro MicroEmulator jako velmi dobrou, zohledním-li testování na malé síti složené ze standardních síťových prvků. Neopomenu také zmínit, že výběr aktivních kanálů na serveru probíhá v intervalech 20ms a tento interval samozřejmě zasahuje i do výsledné odezvy serveru.

Shrnutí průměrné doby odezvy serveru:

odezva (ms)	test
28,30	MicroEmulator, přenosové médiu kabel
64,05	Wireless Toolkit emulátor, přenosové médium kabel
30,29	MicroEmulator, přenos pomocí WiFi
65,38	Wireless Toolkit emulátor, přenos pomocí WiFi
88,47	Mobilní telefon Sonny Ericsson G900, přenos pomocí WiFi
295,75	Mobilní telefon Sonny Ericsson G900, přenos pomocí GPRS
307,17	Mobilní telefon Sonny Ericsson K750i, přenos pomocí GPRS

Tabulka 8.1: Průměrná doba odezvy u jednotlivých zařízení

9 Závěr

9.1 Zhodnocení výsledků

Ke splnění požadavků zadání a demonstraci zvládnuté problematiky byla navržena ukázková aplikace hry pro více účastníků v jazyce Java Micro Edition. Inspirací při tvorbě klientské aplikace mi byla legendární hra Dyna Blaster. Serverová část práce byla založena na knihovně Java New I/O, která přináší metody a technologie k tvorbě neblokujícího serveru, schopného provozu v jednom vlákně.

Testování výsledné aplikace prokázalo uspokojivé výsledky v časech odezvy serveru. Jelikož je klientská aplikace navržena jako tenký klient a překreslování scény souvisí s příchozími daty od serveru, má odezva serveru pro plynulý běh aplikace absolutní význam.

Práce pro mě byla zajímavá a v mnoha případech zábavná. Přinesla mi mnoho nových zkušeností a poznatků. Jako přínos vidím využití této práce jako příklad použití New I/O knihovny.

9.2 Možnosti rozšíření práce

Tato práce byla pojata jako příklad síťové komunikace mezi mobilním telefonem a počítačovým serverem. Z tohoto hlediska jsou vhodné následující rozšíření.

- Rozdíly v technologii. Z hlediska srovnání dvou různých technologií může být serverová aplikace rozšířena o více vláknovou verzi s možností sledování rozdílů mezi oběma technologiemi.

- Dotykové obrazovky. V rámci uživatelského rozhraní může být klientská aplikace rozšířena o plnou podporu dotykového displeje se zaměřením na možnosti takto ovládat herní akce.

- Podpora více her a více map. V rámci demonstrace síťové komunikace je v současnosti implementována podpora pouze jedné statické mapy pro všechny klienty. Bylo by vhodné server rozšířit o podporu více map a více současně běžících her, ze kterých by si nově připojení klienti mohli vybírat.

9.3 Optimalizace

Jako optimalizaci navrhuji především zefektivnění síťového protokolu sloučením více současně zasílaných zpráv do jedné jediné zprávy. Každá zasílaná zpráva vyžaduje akce spojené s oživením spojení a odesláním zprávy. Tyto akce zaberou určité systémové zdroje i čas. Zefektivněním síťového protokolu se ulehčí jednak síťovému provozu, ale také se výrazně posílí rychlost odezvy serveru při velkém množství komunikujících klientů.

Literatura

Monografie

[1] TOPLEY, Kim. *J2ME v kostce : Pohotová referenční příručka.* 1. vyd. Praha : Grada Publishing, 2004. 536 s. ISBN 80-247-0426-9.

[2] MAHMUD, Qusay, H. *Naučte se Java 2 Micro Edition : Pohotová příručka.* 1. vyd. Praha : Grada Publishing, 2004. 246 s. ISBN 80-247-0444-7.

[3] HORTON, Ivor. *Java 5.* Praha : Neortex, 2005. 1443 s. ISBN 80-86330-12-5.

[4] HEROUT, Pavel. *Učebnice jazyka Java.* 3. vyd. České Budějovice : KOPP, 2007. 381 s. ISBN 978-80-7232-323-4.

[5] SPELL, Brett. *Java : Programujeme profesionálně.* Praha : Computer Press, 2002. 1022 s. ISBN 80-7226-667-5.

Elektronické zdroje

[6] *Co umožňuje Java v mobilních telefonech?* [online]. Arti s.r.o. [cit. 03.05.2009]. Dostupné na http://www.arti.cz/zajimavosti/java/co_umi_java.htm

[7] *Dive into the J2ME configuration layer* [online]. Tech republic.[cit. 03.05.2009]. Dostupné na http://articles.techrepublic.com.com/5100-10878_11-1046647.html

[8] *JSR30. Connected, Limited Device Configuration* [online]. Java Community Process [cit. 05.05.2009]. Dostupné na http://www.jcp.org/en/jsr/detail?id=30

[9] *JSR36. Connected Device Configuration* [online]. Java Community Process [cit. 08.05.2009]. Dostupné na http://www.jcp.org/en/jsr/detail?id=36

[10] *Java Virtual Machine Specification* [online]. Sun Developer Network [cit. 08.05.2009]. Dostupné na http://java.sun.com/docs/books/jvms/

[11] *J2ME Building Blocks for Mobile Devices* [online]. Sun Microsystems [cit. 11.05.2009]. Dostupné na http://java.sun.com/products/cldc/wp/KVMwp.pdf

[12] *CLDC 1.1* [online]. [cit. 12.05.2009]. Dostupné na http://java.sun.com/javame/reference/apis/jsr139/

[13] *JSR-000037 Mobile Information Device Profile (MIDP)* [online]. Java Community Process [cit. 12.05.2009]. Dostupné na http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html

[14] *MIDP 1.0 specifikace* [online]. Java Communiy Process [cit. 13.05.2009]. Dostupné na http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html

[15] *Java ME* [online]. FI WIKIMUNI [cit. 13.05.2009]. Dostupné na http://kore.fi.muni.cz:5080/wiki/index.php/Java_ME

[16] *Java 2 Platform Standard Edition 5.0 API Specification* [online]. APIs and Documentation on Sun Developer Network [cit. 13.05.2009]. Dostupné na http://java.sun.com/j2se/1.5.0/docs/api/

[17] *Síťování v Javě : New I/O* [online]. Root.cz [cit. 14.05.2009]. Dostupné na http://www.root.cz/clanky/sitovani-v-jave-new-io/

[18] BITTNEROVÁ, Lucie, Rút. *Seriál o J2ME* [online]. Interval.cz [cit. 14.05.2009]. Dostupné na http://interval.cz/vyvoj-aplikaci/j2me/

A Obsah CD

Obsah přiloženého CD:

- soubor: *bp.pdf*
  obsahuje text technické zprávy bakalářské práce

- adresář: *multiBomber\client*
  obsahuje přeložený a spustitelný kód výsledné klientské aplikace (.jar)

- adresář: *multiBomber\server*
  obsahuje přeložený a spustitelný kód serverové části (.jar)

- adresář: *soorce\BomberClient*
  obsahuje zdrojové kódy klientské části (J2ME), včetně projektových souborů vývojového prostředí NetBeans

- adresář: *soorce\BomberServer*
  obsahuje zdrojové kódy serverové části (J2SE), včetně projektových souborů vývojového prostředí NetBeans

- adresář: *soorce\doku*
  obsahuje zdrojový soubor technické zprávy (OpenOffice 2.4)

- adresář: *javadoc\client*
  obsahuje dokumentaci vygenerovanou zdrojového kódu klienta vygenerovanou progamem Javadoc

- adresář: *javadoc\server*
  obsahuje dokumentaci vygenerovanou zdrojového kódu serveru vygenerovanou progamem Javadoc

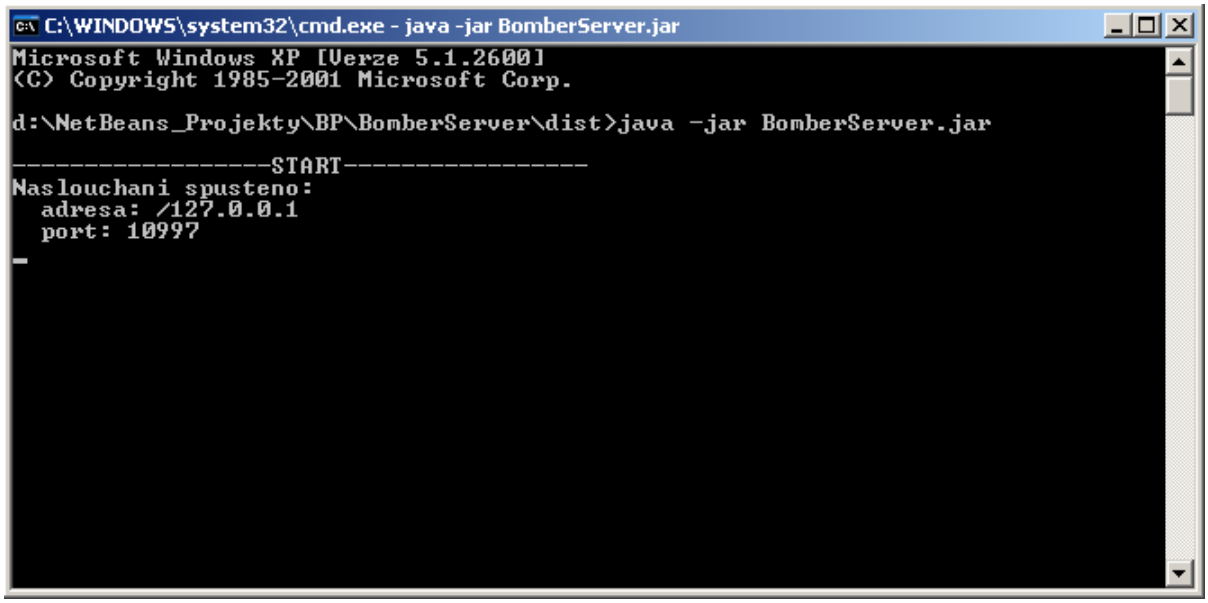
- adresář: *poster*
  obsahuje prezentační plakátek k aplikaci MultiBomber

B Návod

Tento návod předpokládá, že na počítači, kde se aplikace bude spouštět je nainstalován operační systém Windows XP a nainstalována Java.
Na přiloženém CD v adresáři *multiBomber\client* nalezneme přeloženou a spustitelnou klientskou aplikaci určenou pro instalaci na mobilní telefon, nebo pro spuštění v emulátoru. Před započetím práce doporučuji stáhnout si alespoň jeden ze softwarových emulátorů J2ME např. [1]MicroEmulátor a nebo spouštět aplikaci v prostředí NetBeans s použitím Sun Java Wireless Toolkit 2.5.2.

B.1 Spuštění serveru

Prvním krokem, než spustíme klientskou aplikace je spustit server a nastavit IP adresu a port, na kterých má očekávat příchozí požadavky na spojení. Server je umístěn v adresáři *multiBomber\server* a spouští pomocí příkazové řádky dávkou **java -jar BomberServer.jar <IP-adresa> <port>**. Spustíme-li server bez parametru, automaticky naslouchá na drese 127.0.0.1 a portu 10997. Stejně tak zadáme-li jeden nebo více než dva parametry.



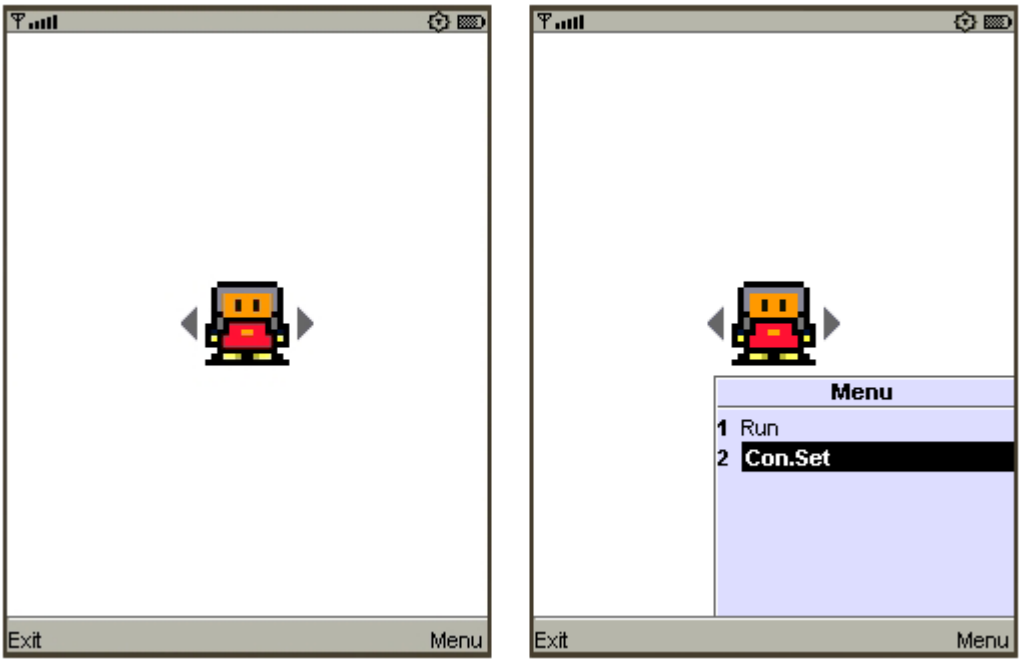
Obrázek B.1: Příklad spuštění serveru v příkazovém řádku.

B.2 Spuštění klienta

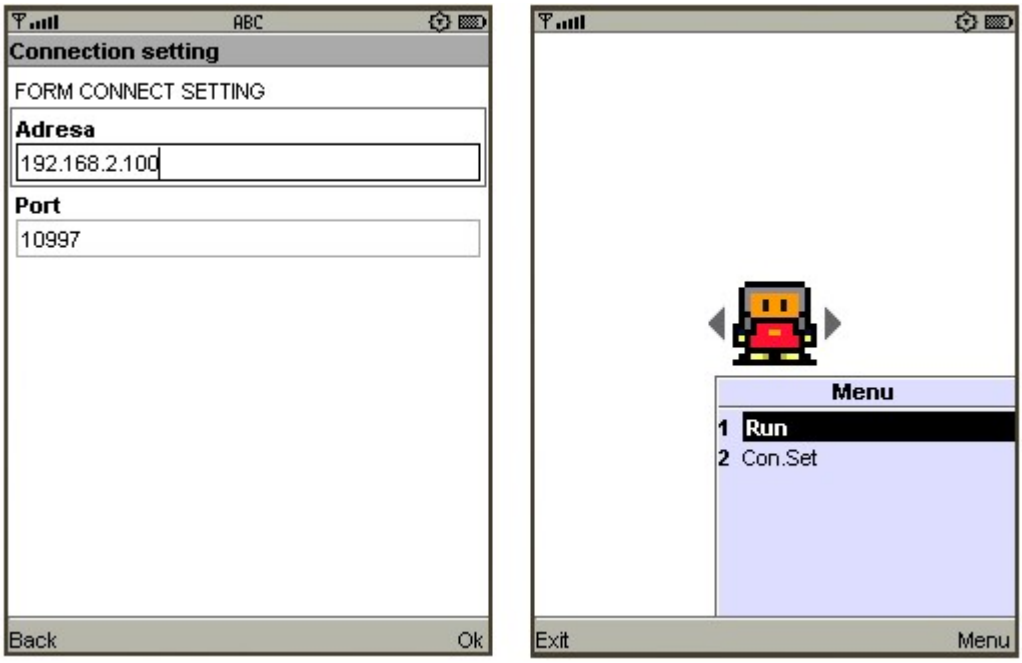
Před spuštěním klienta je důležité, aby server běžel, jinak se nebudeme moci nikam připojit. Po naběhnutí klientské aplikace se zobrazí uvítací obrazovka s logem hry, tu potvrdíme tlačítkem **Play**, abychom se dostali do nastavení hry.

1 V době psaní této práce byla poslední verze aplikace MicroEmulátor dostupná ke stažení na stránkách:
http://snapshot-2x.microemu.org/microemulator/download/

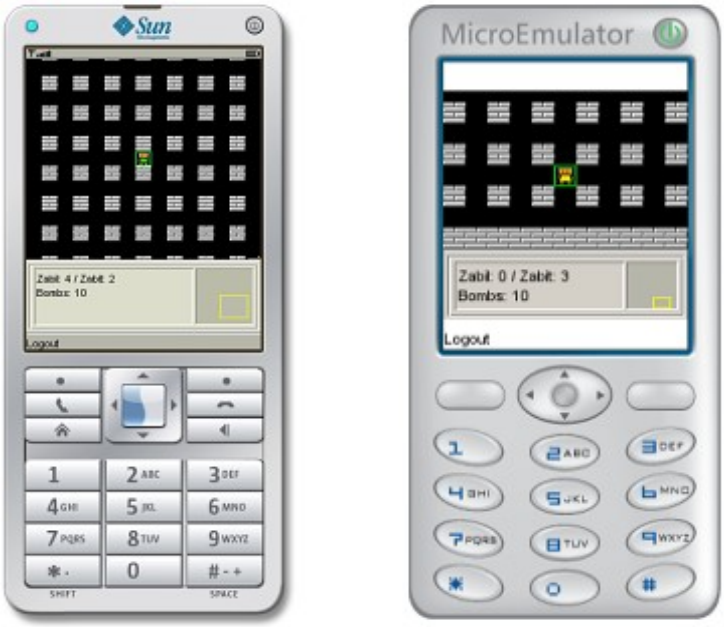
Po potvrzení obrazovky se dostáváme do sekce, kde si můžeme zvolíme barvu naší postavičky. Akce šipek jsou mapovány na klávesy **doleva 4** a **doprava 6**. Přes tlačítko **Menu** rozvineme okénko se dvěma možnostmi. Pomocí tlačítka **Run** můžeme hru přímo spustit a nebo přes **Con.Set** přejít do nastavení spojení. V nastavení spojení můžeme změnit IP-adresu a port. Defaultně je IP nastavena na 127.0.0.1 a port na 10997. Potvrzením tlačítka **OK** se dostaneme zpět do nastavení postavičky. Odtud už přes menu, tlačítkem **Run** spustíme hru.



Obrázek B.2: Výběr postavičky a nastavení spojení



Obrázek B.3: Nastavení spojení a spuštění hry



Obrázek B.4: Výsledná aplikace hry MultiBomber

Na obrázku B.4 je vidět výsledná aplikace hry MultiBomber. Herní akce, jsou mapována na tlačítka herních akcí, tak jak je specifikuje sám výrobce zařízení. V převážné většině jsou to ale klávesy **4-pohyb dolevy**, **6-pohyb doprava**, **8-pohyb nahoru**, **0-pohyb dolů** a **5-akce FIRE**. Pokud má telefon herní klávesy mapovány i na střední joystick, pak i ten reprezentuje jednotlivé herní akce.

C Seznam zkratek

J2ME - Java 2 Micro Edition
J2SE - Java 2 Standar Edition
J2EE - Java 2 Enterprise Edition
PDA - Personal Digital Assistant
STB - Set-Top-Box
CLDC - Connected Limited Device Cpnfiguration
CDC - Connected Device Configuration
VM - Virtual Machine
GB - Garbage Collector
KVM - Kilo Virtual Machine
CVM - „C“ Virtual Machine
JVM - Java Virtual Machine
GTM - Greenwitch Mean Time
MIDP - Mobile Information Device Profile
API - Application Programming Interface
AWT - Abstract Window Toolkit
RMI - Remote Method Invocation
URL - Uniform Resource Locator
HTTP - Hypertext Transfer Protocol
WAP - Wireless Application Protocol
TCP - Transmission Control Protocol
IP - Internet Protocol
GPRS - General Packet Radio Service
UMTS - Universal Mobile Telecommunications systém
WiFi - Wireless LAN
LAN - Local Area Netwokr
AP - Access Point