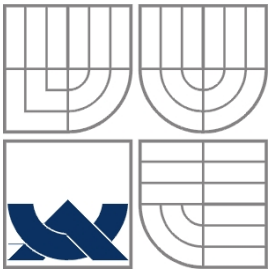
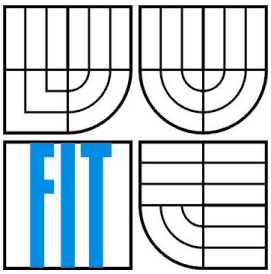


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

HERNÍ SERVER
GAME SERVER

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

MIROSLAV DRBAL

VEDOUCÍ PRÁCE
SUPERVISOR

ING MARTIN HRUBÝ, PH.D.

BRNO 2007

Abstrakt

Tato práce se zabývá konstrukcí herního serveru, sloužícího jako univerzální platforma pro tvorbu a hraní her. Dále technickými požadavky kladenými na herní server, dnešními zástupci herních serverů a to jak z oblasti komerčních tak i volně vyvíjených aplikací. Práce se dále pokouší analyzovat a řešit některé problémy, se kterými se setkáváme u konstrukce herních serverů pro velký počet simultánně připojených hráčů.

Klíčová slova

Herní server, multiplayer, online, MMORPG, síťové jádro, databáze

Abstract

This work is aimed to game server construction which is used as universal platform for creating and playing games. Technical demands which are issued on game server, nowadays game servers and their representants from the open source or commercial pool. This work is trying to analyze and solve some problems which you can challenge when you are trying to construct game server for moderate count simultaneously connected players.

Keywords

Game server, multiplayer, online, MMORPG, netcore, database

Citace

Miroslav Drbal: Herní server, bakalářská práce, Brno, FIT VUT v Brně, 2008

Herní server

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Martina Hrubého, Ph. D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Miroslav Drbal
14.05.2008

Poděkování

Chtěl bych poděkovat Ing. Martinu Hrubému, Ph. D., který mě jako vedoucí práce směřoval ke správnému cíli a vnesl do tohoto projektu mnoho konstruktivních myšlenek.

Obsah

1 Úvod

Jakmile se počítačová technika začala rozšiřovat z čistě laboratorního a vědeckého prostředí do prostých domácností, započal i vývoj počítačových her. S postupným zvyšováním výpočetního výkonu osobních počítačů se začala zlepšovat i kvalita her. Evoluce her probíhala od prvopočátečních, převážně textových a logických her přes všemožné 2D arkády až k dnešním moderním 3D akčním a strategickým hrám, jejichž grafika je v některých případech ne nepodobná realitě.

Nástup Internetu a dostupnost síťových technologií (LAN sítě) běžným uživatelům vnesla do světa her nový rozměr. Nyní se již nesváděly bitvy jen mezi člověkem a počítačem, ale hráč dostal možnost vyzkoušet si svoje dovednosti a reakce i proti jinému člověku.

Moje práce je zaměřená právě na konstrukci herního serveru, který by umožňoval vzájemnou interakci více hráčů prostřednictvím počítačové sítě a zároveň by poskytoval univerzální platformu pro snadnou tvorbu a testování nových her. Hlavní motivací pro tento projekt je právě potřeba dostatečně univerzální platformy pro tvorbu logických problémů – her, na kterých by bylo možné testovat různé druhy umělých inteligencí, které by tento problém řešily.

Druhá kapitola uvádí do problematiky herních serverů. Detailněji zde definuji pojem *herní server* a zabývám se existujícími herními servery. Třetí kapitola obsahuje teoretický návrh mého vlastního herního serveru, čtvrtá kapitola se zabývá vlastní implementací. Následuje kapitola věnovaná implementaci vzorových her. Metodika testování je shrnuta v šesté kapitole. V závěru naleznete zhodnocení dosažených výsledků a návrhy pro další možný vývoj aplikace.

2 Současné herní servery

Pokud se máme zabývat otázkou herní serverů, je důležité pojem *herní server* nějakým způsobem definovat, byť se jedná o pojem intuitivně srozumitelný a podrobněji jej popsat.

Jak tedy nejlépe definovat pojem *herní server*? Herní server by se dal zřejmě nejlépe charakterizovat jako sdílený systém pro společnou komunikaci hráčů, stanovující herní pravidla (čas, prostor, pravidla vzájemné interakce herních objektů atd.), plnící funkci dozorce nad dodržováním těchto pravidel, který se současně stará o konzistenci herních dat a jejich případné ukládání. V neposlední řadě herní server definuje na první pohled neviditelnou, ale naprosto nepostradatelnou věc, kterou si hráči při samotném hraní vůbec nemusí uvědomovat, a tou je komunikační protokol. Jedná se o přesně definovaný formát, kterým server komunikuje s připojenými herními klienty a naopak.

2.1 Klasifikace herních serverů

V předchozí kapitole byl stručně definován pojem herní server. V následujících odstavcích se budu zabývat problematikou klasifikace herních serverů z pohledu počtu připojených hráčů a rozeberu technická specifika jednotlivých skupin.

Rozsáhlou množinu herních serverů lze rozdělit do následujících dvou skupin.

2.1.1 Listen servers (naslouchající servery)

Zde se jedná převážně o herní servery, které jsou přímou součástí herního klienta. To znamená, že hráč má možnost přímo z herního rozhraní hry založit a hostovat hru, ke které se připojují další hráči a sám zakladatel se dané hry aktivně účastní. Tohoto systému se převážně využívá u síťových her, které nekladou vysoké nároky na výpočetní výkon serveru, na kterém jsou provozovány. Tato technologie je oblíbená převážně v malých nebo domácích sítích LAN. Na bezpečnost v takových herních systémech není kladen tak veliký důraz, což se odráží v absenci jakéhokoliv kryptování přenášených dat po sdíleném komunikačním médiu (zde by se určitě našly i výjimky, ale ty pro jednodušší vysvětlení pojmu *listen server* zanedbáme).

Klasickými zástupci v této oblasti her jsou například notoricky známý *Quake* a nebo *Age of Empires* jakožto zástupce strategických her.

2.1.2 Dedicated servers (vyhrazené servery)

Tato skupina serverů se vyznačuje tím, že již nejsou součástí klientské aplikace, ale vystupují jako samostatný program, který ve valné většině případů není distribuován společně s herním klientem. Takové herní systémy jsou charakteristické zejména lepší technologickou propracovaností, která zahrnuje simultánní obsluhu daleko většího počtu hráčů, případnou implementaci zabezpečeného přenosu dat mezi klientem a serverem a centralizované transparentní ukládání stavu hry.

V současnosti nejdokonalejšími vyhrazenými servery jsou servery pro MMORPG[1] hry. Tyto hry poskytují hráčům simulaci virtuálního světa, ať již naprosto fiktivní, jako je tomu u her *World of Warcraft* a *Ultima OnLine*, nebo motivovaný nějakou literární popřípadě filmovou předlohou. Takovouto motivací mohou být například známé Tolkienovy knihy, které jsou do podoby virtuálního světa implementovány ve hře *Lord of the Rings online: Shadows of Angmar*. U těchto typů her, kde vývoj postavy představuje smysl celé hry, jsou na herní servery v oblasti bezpečnosti kladeny vysoké nároky, s čímž souvisí i nutnost používat zabezpečený – kryptovaný přenos dat. Vzhledem k velkému počtu simultánně připojených hráčů (až několik tisíc) je zde také kladen velký důraz na výkonnost síťového jádra a distribuce zátěže mezi větší počet serverů. Nemalé nároky jsou také kladeny na ukládání herních dat, které v drtivě většině případů probíhá transparentně (hráč ve hře vůbec nijak nepozná, že byl uložen jeho herní stav).

1 Massively multiplayer online role-playing game

3 Teoretický návrh herního serveru

Pokud se mám v této kapitole zabývat teoretickým návrhem vlastního herního serveru, je třeba nejprve stanovit požadavky, které budou na tento herní server kladeny a které by měl splňovat.

Jak již bylo předesláno v úvodu, jedná se o implementaci univerzálního herního serveru s možností přidávat další uživatelsky definované hry ve formě dynamicky načítaných knihoven. Herní server by se tedy měl starat o dynamické vytváření instancí jednotlivých her v závislosti na požadavcích od připojených herních klientů. Dále by měl obsahovat kvalitní síťové jádro schopné obsluhovat velký počet simultánně připojených hráčů, rozhraní pro práci s databází, která slouží jako prostředek pro ukládání herních dat. Velice důležitou část pak tvoří objektový návrh stěžejních prvků herního prostředí, který by měl být natolik obsáhlý aby umožňoval snadnou implementaci jak tahových her, tak her hraných v reálném čase.

Filozofie celého serveru vychází z konstrukce MMORPG serverů z kterých jsem čerpal inspiraci. Konkrétně se jedná o opensource projekt implementující serverovou část úspěšné komerční hry *World of Warcraft*, viz *[1]*.

3.1 Síťové jádro

Na úvod této kapitoly zmiňuji volbu transportního protokolu, kterým bude server komunikovat s připojenými klienty. Rozhodovat se zde budu mezi protokoly UDP a TCP, přičemž zde v rychlosti zmíním hlavní rozdíly mezi nimi. Pro podrobnější informace k této tematice bych doporučil literaturu *[3]* a *[4]*.

Protokol TCP na rozdíl od protokolu UDP obsahuje kontrolu doručení odeslané zprávy protistraně, zaručuje konzistenci odesílaných dat a obsahuje informace o stavu spojení. Na druhou stranu komunikace pomocí TCP protokolu vyžaduje větší režijní náklady, které se však v dnešní době dají zanedbat. Proto volím protokol TCP.

Potřeba simultánní obsluhy velkého počtu připojených hráčů klade na síťové jádro nemalé požadavky.

Jedním z těchto požadavků je, aby síťové jádro bylo schopné reagovat na příchozí požadavky od klientů v konstantním čase. Tento požadavek však není obecně realizovatelný. Hlavním problémem se zde stává jak efektivně určit, které síťové sokety z množiny všech obsluhovaných soketů jsou připraveny pro operaci čtení, které jsou připraveny pro operaci zápis a které se nacházejí v chybovém stavu. Současné operační systémy jsou vybaveny metodami, které mají za úkol právě

tyto informace zjišťovat. Jejich efektivita je nastíněna na Obr. 1, který je převzatý z webových stránek projektu libevent *[2]*, a který se zabývá implementací multiplatformní asynchronní síťové knihovny.

Vzhledem k použitému operačnímu systému, pro který jsme se rozhodl herní server implementovat[2], jsem zvolil pro zjišťování stavu soketů metodu *epoll*, která podle uvedeného grafu poskytuje z hlediska časové složitosti velice dobré výsledky i pro velký počet soketů.

Dalším krokem v návrhu síťového jádra je ustanovení obecného komunikačního protokolu, kterým budou server a klient navzájem komunikovat. Jako hlavní cíle jsem si klad snadnou implementovatelnost, jednoduchost dekódování zprávy a formát, který by žádným způsobem nelimitoval vývoj dalších herních modulů. Jako výsledek jsem navrhl následující strukturu komunikačního paketu viz. Obr. 2. Jak je z obrázku patrné herní paket je složen ze dvou základních částí, kterými jsou *Operační kód* a *Specifická data*. Operační kód je bez znaménkové celé 16b číslo, které jednoznačně určuje druh činnosti, která se bude vykonávat a která musí být v podobě výčtového typu specifikována pro každou hru. Na jednu stranu tento systém sebou přináší o něco náročnější implementaci jednotlivých her (musíme vždy definovat množinu signálů a jejich obsluhu) na druhou stranu však tento design poskytuje pro případného vývojáře dalších herních modulů větší flexibilitu a volnost v definici vlastních operačních kódů. Další přínos oproti unifikovanému systému zpráv pro všechny hry vidím v tom, že pokud se změní množina operačních kódů u některé z her z důvodu rozšíření o další vylepšení není nutné provádět žádné další úpravy v ostatních hrách.

3.2 Ukládání dat

Ukládání dat je velice důležitou součástí herních serverů, kde se předpokládá, že hra se bude hrát po delší dobu, ve které může docházet k připojování a odpojování hráčů.

Ukládání herních dat může být realizováno jako serializace instancí herních objektů do specifického binárního souboru, což je častý způsob u herních serverů implementovaných v interpretovaných jazycích jako jsou například .NET nebo Java. Nevylučuje se zde ani použití textového souboru a kódování herních dat do formátu XML. Nevýhoda tohoto způsobu ukládání tkví v omezené manipulaci s těmito soubory pomocí externích aplikací. Většinou je pak potřeba i pro drobné úpravy použít nějaký specializovaný nástroj[3].

2 Volba operačního systému je líže diskutována v kapitole 4 Implementace herního serveru

3 Toto nemusí platit v případě použití XML souboru, jakož to úložiště dat. Zde se však střetáváme s problémem, jak tyto soubory efektivně upravovat, pokud je jejich velikost veliká.

V dnešní době se od toho způsobu ukládání dat upouští a dalo by se říci, že valná většina nově vznikajících herních serverů se spíše orientuje na použití relačních databází, jakými například jsou MySQL[4], PostgreSQL[5] nebo MSSQL[6]. Výhody plynoucí z použití SQL databází jsou ve snadné manipulaci s daty, odpadá nutnost implementovat vyhledávací a indexovací algoritmy, které jsou v těchto databázích již implementované a optimalizované. Další nespornou výhodou je možnost rozdělit zátěž na více databázových strojů (například NDB cluster při použití MySQL).

V implementaci herního serveru se tedy přikloním k variantě s použitím relačních databází, konkrétně k MySQL.

3.3 Základní objekty herního prostředí

Při tvorbě objektů herní logiky jsme postaveni před problém, jak efektivně navrhnout základní třídní strukturu serveru tak, aby plně pokryla požadavky jak pro tahové hry, tak hry hrané v reálném čase. Základní stavební kameny tedy musejí být natolik komplexní aby pokryly požadavky těchto dvou typů her a zároveň se jejich komplexnost nestala případným problémem při implementaci jednoduchých her.

Rozhodl jsem se tedy herní objekty navrhnout spíše jako rozhraní a implementovat v nich pouze nezbytný počet metod, sloužící převážně k rozpoznání typu herního objektu a metody pro generování jednoznačné identifikace objektů v herním světě. Při implementaci konkrétních herních objektů konkrétních her se pak spoléhám na využití dědičnosti a polymorfismu což dohromady umožňuje dokonale oddělit implementaci vlastní herní logiky od čistě systémových vlastností jednotlivých herních objektů. Na tomto místě bych proto rád poukázal na a v kapitole 8 Grafická příloha, na kterých je znázorněna třídní hierarchie a následná agregace jednotlivých tříd.

Pokud vezmeme v úvahu, že herní svět v MMORPG hrách může být neuvěřitelně rozsáhlý, musí nás zákonitě napadnout otázka, jak efektivně pracovat s objekty v herním prostředí, kterých může být až několik desítek tisíc, abychom zbytečně neprováděli aktualizace herních objektů v místech, kde se žádní hráči nevyskytují, a zabránili tak plýtvání systémovými prostředky a následné degradaci výkonu celého herního serveru.

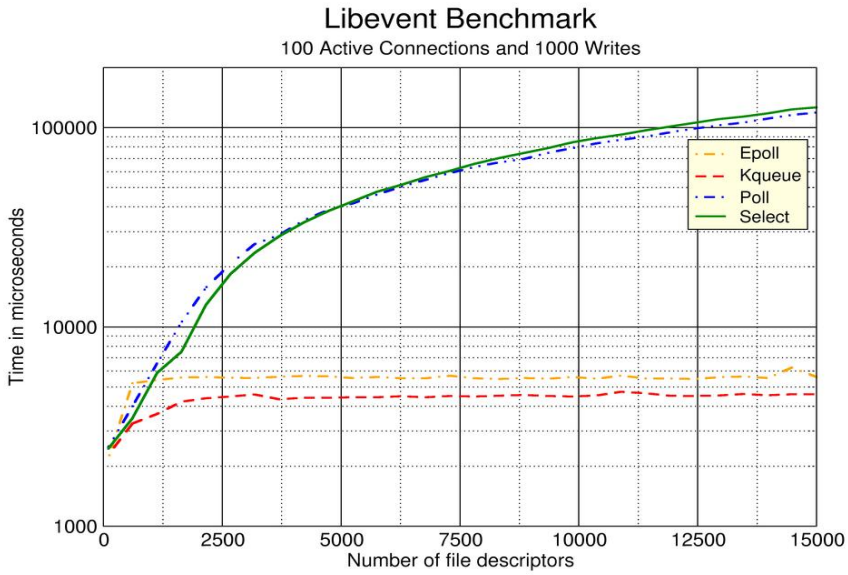
Pro řešení tohoto problému můžeme najít inspiraci v celulárních automatech. Celé herní prostředí tak rozdělíme na pevně stanovený počet buněk, které budeme nazývat *cell*. Tyto buňky budeme na základě pozic hráčů v herním světě aktivovat a deaktivovat a provádět aktualizace pouze

4 Opensource databáze, pro opensource projekty a aplikace je šířena pod GPL licencí; http://www.mysql.com/

5 Opensource databáze šířena pod BSD licencí; http://www.postgresql.org/

6 Komerční databáze od firmy Microsoft; http://www.microsoft.com/sql/default.mspx

nad buňkami, které budou aktivní. Aktivace buněk bude závislá na poloze hráče vůči dané buňce a to tak, že buňka bude aktivní, pokud se hráč vyskytuje přímo v ní nebo pokud se nachází v bezprostředně sousedící buňce. Tím je zajištěno, že pokud se hráč bude pohybovat a překročí hranice své aktuální buňky, vstoupí taktéž do aktivní buňky.



Obr. 1:

Operační kód	Data specifická podle operačního kódu

Obr. 2: Struktura herního paketu

4 Implementace herního serveru

V této kapitole se budu zabývat vlastní implementací herního serveru. Než však s tímto tématem začnu, rád bych se zmínil o použitém programovacím jazyku a cílové platformě.

Pro implementaci jsem zvolil programovací jazyk C++. Vybral jsem si ho z důvodu vysokého výkonu výsledného zkompilovaného kódu, pro svoji platformovou nezávislost, pokud nejsou použity platformově závislé konstrukce, a v neposlední řadě jsem ho zvolil i proto, že v práci s tímto programovacím jazykem mám již dlouhodobější zkušenosti.

Jako cílovou platformu jsem si zvolil operační systém Linux z toho důvodu, že není vázán žádnou komerční licenční smlouvou, je pro něj dostupná široká škála vývojových nástrojů a disponuje kvalitní dokumentací.

Vlastní implementace herního serveru je založena na myšlence, že celé herní jádro se všemi potřebnými stavebními bloky, pro konstrukci vlastních her, je obsaženo v jednom spustitelném souboru a vlastní hry jsou koncipovány jako dynamicky načítané knihovny. Je tedy velmi snadné nějakou hru přidat či naopak odebrat, aniž by se musel celý server znovu překládat. Implementaci jsem rozdělil do několika logických celků[7], které svojí kompozicí tvoří celý herní server. Jedná se o ***síťové jádro, databázové jádro, objekty herní logiky*** (třídy reprezentující objekty, hráče a počítačem řízené protivníky) a ***sdílené prostředky*** (jedná se o třídy a jejich metody, které nesouvisí přímo s vlastní herní logikou a síťovou komunikací, ale poskytují těmto prvkům svoje služby. Převážně se jedná o zprostředkování vláken a třídy pro zpětná volání při výskytu události tzv. *callbacky*).

7 Kompletní návrh tříd je možné nalézt v kapitole 8. Grafická příloha

4.1 Síťové jádro

Základem implementace je knihovna BSD sockets[8] nad kterou jsem vybudoval několik tříd, které reprezentují jednotlivé síťové objekty a zapouzdřují jejich metody. Komunikace klienta se serverem poté probíhá formou zasílání zpráv (paketů), které mají přesně definovanou podobu viz Obr. 2.

Zpracování takovéhoto paketu probíhá nejprve oddělením operačního kódu od zbytku zprávy, jeho následného dekódování a zavolání patřičné obsluhy, které provede deserializaci zbylých dat v paketu, provedení pro daný operační kód specifických akcí a případné zkonstruování paketu s odpovědí a jeho zaslání klientovi.

Při implementování herního serveru jsem narazil na potřebu vytvořit množinu speciálních žádostí odesílaných ne konkrétní obsluze herního prostředí, ale přímo jádru herního serveru, který tato herní prostředí spravuje. Jedná se především o žádosti o získání stavových informací serveru a žádostí k přihlášení se k vybranému typu hry. Pro tyto potřeby jsme zavedl úzus, že operační kódy s hodnotou ***0x1h – 0xFFh*** jsou použity pro přímou komunikaci s herním jádrem a operační kódy v rozsahu ***0x100h – 0xFFFFh*** jsou dány volně k dispozici pro implementaci interní komunikace v rámci jednotlivých her.

4.1.1 ByteBuffer

Tato třída zajišťuje serializování dat základních číselných datových typů a typu *std::string* do bytového pole, které je poté jako proud odesláno na požadovaný soket. Třída také obsahuje metody pro deserializaci takovýchto dat získaných ze síťového soketu. Všechny operace serializace a deserializace dat jsou implementovány jako přetížené operátory << a >>.

Jádro této třídy a zároveň úložiště dat je tvořeno třídou *std::vector*. Tento kontejner jsem zvolil proto, že je interně implementovaný jako obyčejné pole hodnot, což umožňuje kopírovat více bytové položky do a nebo z tohoto pole pomocí metody *memcpy*.

8 Více informací o této knihovně je dostupných na http://en.wikipedia.org/wiki/Berkeley_sockets

```
size_t Read(uint8* data, size_t len)
{
   if (m_rpos + len > m_res.size())          // Ověř zda nechceme číst příliš moc prvků
      len = m_res.size();                    // Pokud ano, koriguj délku
   memcpy (data, &m_res[m_rpos], len);  // Překopíruj obsah do cílového bufferu
   m_rpos += len;                         // Aktualizuj pozici pro čtení
   return len;                            // Vrať počet reálně přečtených dat
}

 size_t Store(const uint8* data, size_t len)
{
    if (m_res.size() < m_wpos + len)          // Ověř zda nechceme zapisovat příliš moc prvků
       m_res.resize(m_wpos + len);            // Pokud ano, rozšiř pole pro zápis
    memcpy (&m_res[m_wpos], data, len);   // Zapiš data do interního vektoru
    m_wpos += len;                         // Aktualizuj pozici pro další zápis
    return len;                            // Vrať počet reálně zapsaných bytů
}
```

Text 1: Implementace metod pro uložení / přečtení daného počtu bytů z ByteBufferu

4.1.2 Socket, TcpSocket, TcpListenSocket

Třída *Socket* tvoří základní stavební kámen pro síťovou komunikaci. Reprezentuje vlastní síťový socket identifikovaný unikátním číslem (deskriptorem), který byl vygenerován operačním systémem. Zároveň slouží jako základní interface, který definuje základní metody pro příjem a odesílaní dat a zjištění stavu ve kterém se soket nachází. Vzhledem k tomu, že komunikace je realizována jako asynchronní přenos, bylo nutné implementovat do třídy *Socket* interní strukturu, která si udržuje stav o procesu odesílání dat vzhledem k tomu, že při asynchronním odesílání dat nemusí být naráz odeslána veškerá požadovaná data. Tuto strukturu jsem pojmenoval DCB (data control block) a její definice je následující.

```
// Send data control block
struct DCB
{
   DCB() : lastIndex(0) {}                  // Inicializační konstruktor
   size_t                        lastIndex;     // Index na konec již odeslaných dat z hlavy fronty
   std::queue<ByteBuffer*> sendQueue;   // Fronta serializovaných dat k odeslání
}m_dcb;
```

Text 2: Implementace DCB odesílaných zpráv

Třída *TcpSocket* rozšiřuje třídu *Socket* o možnost nastavení dalších atributů, jako je například neblokující režim o kterém ještě bude řeč později v souvislosti s třídou *SocketMgr*.

TcpListenSocket je potomkem třídy *TcpSocket*, který umožňuje soketu naslouchat na specifikované adrese a portu a přijímat tak požadavky na nová spojení.

4.1.3 SocketMgr

S narůstajícím počtem soketů, které náhodně vznikají a zanikají (připojování/odpojování klientů), nastává potřeba tyto sokety uchovávat v nějakém kontejneru, testovat jejich stav, zda jsou dostupná data ke čtení, a nebo je-li naopak požadavek na odeslání dat na daném soketu.

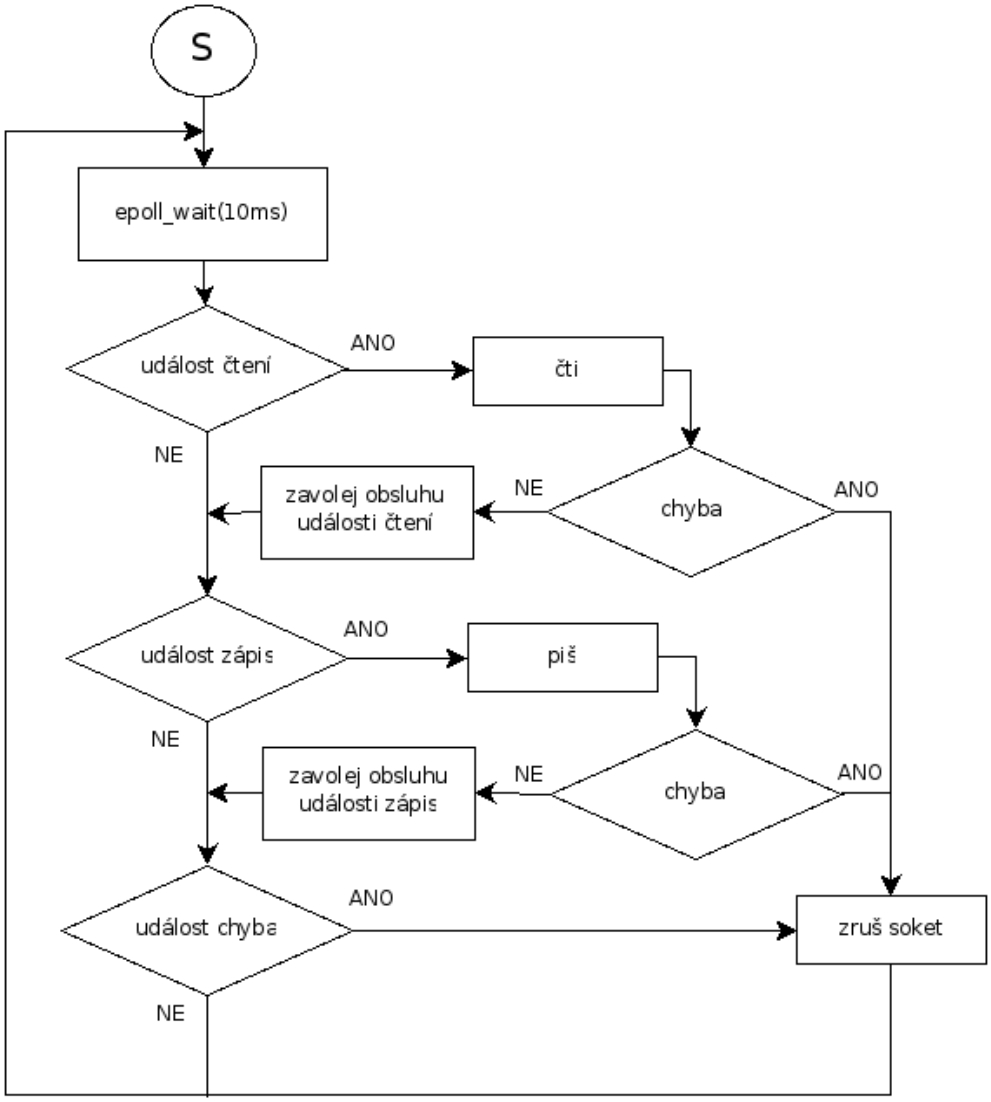
O tuto činnost se stará třída *SocketMgr*. Tato třída byla navržena jako šablonová z toho důvodu, aby se minimalizovala režie způsobená polymorfismem soketů, která by vznikala, kdybychom pro uložení ukazatelů na jednotlivé sokety použili univerzálního předka, třídu *Socket*. Takto je již v době překladu vygenerována specializovaná třída pracující s konkrétními typy soketů a nevzniká tak nutnost přetypovávání za běhu programu.

Všechny klientské sokety jsou uloženy ve standardním poli, které obsahuje ukazatele na příslušné instance. Deskriptor každého soketu je pak indexem do tohoto pole. Využívá se zde toho, že operační systém přiděluje čísla deskriptorům lineárně a to tak, že přidělí vždy nejmenší možné číslo. Nevznikají tedy „hluchá“ místa v poli soketů v případě zániku některých z nich.

Události na soketech jsou testovány každých 10ms pomocí systémového volání *epoll_wait*, pro které je vytvořeno vlastní vlákno. Jak jsem již zmiňoval v teoretickém návrhu, tato metoda zajišťuje vysoký výkon i při velkém množství obsluhovaných soketů. Pro dosažení optimálního výkonu jsou navíc obsluhované sokety přepnuty do neblokujícího režimu, což znamená, že při požadavku na čtení nebo zápis na daném soketu nedochází k blokování programu a čekání na výsledek operace, ale metoda ukončuje svůj běh, jak nejrychleji je to možné.

Pokud dojde k události na soketu, je obsloužena příslušnou metodou objektu, který zajišťuje její obsluhu. Volání této metody je zajištěno pomocí tzv. *callbacku.*

Detailnější pohled na činnost této třídy poskytuje Obr. 3.



Obr. 3: Vývojový diagram obsluhy události na soketu

4.2 Objekty herní logiky

Úvodem této kapitoly bych rád opět poukázal na diagram agregací jednotlivých tříd, který ozřejmí celý návrh serveru a uvede popisované objekty do širších souvislostí a který je možný nalézt v kapitole 8 Grafická příloha.

4.2.1 Server

Jak je již z grafu agregací tříd patrné, tato třída stojí skoro na samém vrcholu celého objektového návrhu serveru a agreguje tak do sebe funkčnost všech podřazených tříd. Pokud opomineme funkčnost všech tříd, které jsou do třídy *Server* agregovány, zjistíme, že tato třída funguje jako most a zároveň směrovač přijatých zpráv od připojených herních klientů a předává tak herní zprávy patřičným instancím her, do kterých jsou hráči připojeni. Zároveň také obsluhuje pakety s výběrem hry a popřípadě odesílá chybový paket, pokud by se hráč pokusil vybrat na serveru neexistující hru.

Aby bylo možné udržovat aktuální informace o tom, kde se jednotlivý herní klienti nacházejí jsou v této třídě implementovány následující datové struktury.

4.2.2 WorldPool

Pro každou hru, jejíž herní logika je načtena ze sdílené knihovny, je na straně serveru vytvořen speciální *WorldPool* identifikovaný unikátním číslem hry, které je specifikováno přímo v herní knihovně. Tento *WorldPool* v sobě poté obsahuje veškeré rozehrané hry, stará se o vytváření nových instancí dané hry, pokud se objeví požadavek na hru a žádná existující instance není k dispozici a dále dokáže vyhledat již existující instanci hry, ke které je možné se připojit.

WorldPool je podobně jako třída *Server* navržen tak, že si udržuje informace o tom, jaké klientské sokety jsou mapovány na konkrétní instanci hry a implementuje metodu, která provádí vyhledání dané herní instance na základě znalosti herního soketu.

```
class WorldPool
{
   ...
   // Definice mapy Socket → World
   typedef HM_NAMESPACE::hash_map<TcpSocket*, World*> Sock2World;
   public:

      World* NewInstance();                          // Vytvoří novou instanci hry
      World* GetFreeInstance();                      // Nalezne první volnou instanci hry
      World* GetWorldForSock(TcpSocket* s);          // Nalezne instanci hry podle soketu
      ...
      void   AddClient(TcpSocket* s, World* world); // Přiřadí soket k dané hře
      void   RemoveClient(TcpSocket* s);             // Odebere mapování soketu na World
      ...
      uint32 GetPoolID() const { return m_poolID; }  // Vrátí jednoznačnou identifikaci typu hry
      void DestroyWorld(World* world);               // Odstraní danou instanci hry
   private:
      Sock2World           m_sockToWorld;            // Mapa Socket → World
      std::vector<World*> m_pool;                    // Vektor všech instancí dané hry
      void*                m_libHandle;              // Handle na knihovnou s implementací hry
      uint32               m_poolID;                 // Jednoznačná identifikace typu hry
};
```

Text 3: Orientační pohled na třídu WorldPool

4.2.3 World, Session

Třída *World* představuje základní interface – kostru, kterou je nutné implementovat v každé nově vytvářené hře. Jedná se především o implementaci reakcí na události připojení, odpojení herního klienta a zpracování události příchodu herní zprávy. Dalším rozšířením této třídy o uživatelské metody, popřípadě agregací dalších tříd, vzniká vlastní herní logika, která je ve výsledku reprezentována jako samotná hra. Aby docházelo k periodickému aktualizování jednotlivých klientských *Sessions* implementuje tato třída speciální vlákno, které periodicky tuto činnost periodicky zajišťuje voláním metody *Update* třídy *World*. Metoda je definována jako virtuální a umožňuje tak její předefinování v poděděných třídách. Toto je velice důležitá vlastnost pomocí níž následně implementuji systém reálného času. Reálný čas je pak jako kvantum kaskádně šířen do všech herních objektů za pomocí jejich vlastních implementací metody *Update*.

```
class World
{
   public:
      ...
      size_t GetConnectedCount() const { m_sessionMap.size(); }  // Vrátí počet připojených hráčů

      virtual void Update(uint32 diff);                          // Periodicky volaná metoda update
                                                                 // z WorldUpdateru

      virtual void OnClientConnect(ucallback data) = 0;          // Čistě virtuální metoda volaná
                                                                 // při připojení hráče
      virtual void OnClientDisconnect(ucallback data);           // Virtuální metoda volaná
                                                                 // při odpojení hráče
      virtual void OnPacketRead(ucallback data);                 // Metoda volaná při přijetí herního paketu

      virtual bool CanPlayerConnect() = 0;                       //  Čistě virtuální metoda udávající zda se
                                                                 // může připojit další hráč

      void Broadcast(ByteBuffer& packet);                        // Odešle paket všem připojeným hráčům
   protected:
      void SessionUpdate();                                      // Aktualizac klientských session, voláno
                                                                 // z metody Update

      SessionMap      m_sessionMap;
      WorldUpdater* m_updater;
};
```

Text 4: Detailní pohled na definici třídy World

```
{
   public:
      // Definice datového typu reprezentujícího mapu jednotlivých Socketu na dané WorldPooly
      typedef HM_NAMESPACE::hash_map<TcpSocket*, uint32> Sock2Pool;
      ...
   private:
      std::set<TcpSocket*>                        m_notAssigned;  // Množina Socketů nepřiřazených k žádné hře
      std::vector<WorldPool*>                     m_pools;        // Pole WorldPoolů jednotlivých her
      SockMgr<TcpSocket, TcpListenSocket> m_sockMgr;      // SocketManager
      Sock2Pool                                   m_sockToPool;   // Mapa Socket → WorldPool
};
```

Text 5: Datové struktury třídy Server

Pro každého připojeného hráče je vytvořena instance třídy *Session*, která zpracovává zprávy přijaté od daného hráče. Dále disponuje statickými metodami, které se starají o reakce na požadavky od hráče identifikované podle operačního kódu. Pro účely snadné deklarace a definice obslužných metod pro jednotlivé operační kódy jsem implementoval jednoduché makro, které podle zadaných parametrů dokáže obslužné metody deklarovat.

```
// EXPAND a EXPAND_I provedou expanzi parametru do zdrojového kódu jako prostý text
#define EXPAND(a) EXPAND_I(,a)
#define EXPAND_I(a,b) a ## b

// HANDLER provede deklarování třídní statické metody jmenéna NAME s předem známými parametry
#define HANDLER(NAME) static void EXPAND(NAME)(Session* session, ByteBuffer& data)

// HANDLER_IMP vytvoří hlavičku metody pro následnou implementoci
#define HANDLER_IMPL(NAME,CLASS) void EXPAND(CLASS)::EXPAND(NAME)(Session* session,
ByteBuffer& data)

// HANDLER_PTR generuje zápis ukazetelu na handler, toto je použito při plnění mapy obslužných metod
#define HANDLER_PTR(NAME,CLASS) &EXPAND(CLASS)::EXPAND(NAME)
```

Text 6: Ukázka maker pro generování odslužných metod

4.2.4 DiscreteMap2D, Cell

Do každé herní logiky neodmyslitelně patří i definice prostoru nebo plochy, na které se daná hra odehrává. Tuto plochu, kterou budu nazývat mapa, reprezentuje šablonová třída DiscreteMap2 (v grafu agregací pojmenována pouze jako Map) jak již název napovídá, jedná se o plochu diskrétních hodnot, které je možné adresovat x-ovou a y-ovou souřadnicí. Šablonovou implementaci jsem zvolil proto, že si myslím, že může být velice užitečné definovat na úrovni deklarace instance třídy jaký typ, bude mít buňka mapy, což se v pozdější implementaci demonstračních her ukázalo jako velice prozíravé řešení.

V souvislosti s implementací mapy jsem vytvořil taktéž šablonovou reprezentaci buňky této mapy, která je schopná v sobě uchovat tři různé kolekce objektů, jejichž typ je opět definován šablonově. Je primárně určena k tomu, aby v sobě uchovávala třídy nebo deriváty tříd *Player, Unit, Object* o kterých zde ještě budu psát.

4.2.5 Player, NPC, Unit, GameObject, Object

Tyto základní herní entity obsahují pouze základní metody převážně pro práci s jejich jednoznačnou identifikací pomocí unikátního čísla tzv. GUID. GUID je 64b číslo jehož horních 32b obsahuje identifikaci typu objektu a dolních 32b jeho pořadové číslo v herním světě.

Každá s těchto herních entit obsahuje virtuální metodu *Update* pro kaskádní předávání informací o uplynulých časových kvantech.

4.3 Databázové jádro

Implementace databázového jádra je postavena pro databázový server MySQL. Celá implementace je postavena na třech třídách. Jedná se o třídy *MySQL*, *QueryResult* a *Field*.

Třída *MySQL* tvoří výchozí interface pro komunikaci s MySQL databází. Obsahuje metody pro ustanovení spojení a pro provádění vlastních SQL dotazů.

Výsledky těchto dotazů jsou zapouzdřeny do třídy *QueryResult*, která představuje kolekci všech přijatých řádků, jejichž jednotlivé sloupce se adresují přetíženým operátorem *[]*.

Jako výsledek této adresace je vrácena instance třídy *Field* jakožto reprezentace daného sloupce v aktuálním řádku. Vzhledem k tomu, že data z databáze MySQL jsou reprezentována jako řetězec znaků(včetně číselných hodnot), disponuje tato třída metodami pro převod tohoto řetězce na požadovaný typ.

4.4 Sdílené prostředky

Do sdílených prostředků náleží všechny třídy, které se přímo nepodílejí na tvorbě logiky herního světa, síťové komunikaci nebo přístupu k databázi, ale poskytují těmto třídám svoji funkčnost bez které by nemohly tyto třídy fungovat.

V první řadě se jedná o třídu implementující vlákna, pojmenovanou *Thread.* Tato třída je postavena nad POSIX threads API[9], které je implementováno snad ve všech unixových systémech. Společně s touto třídou jsou zde definovány i třídy *Mutex* a *MutexGuard* pomocí nichž se řídí zamykání a přístup do kritických sekcí kódu.

Další neméně důležitou třídou je třída *Singleton*, která slouží k vytvoření sdílené instance libovolné třídy. V implementaci je použito dvojí testování na existenci instance třídy, kterou chceme prohlásit za singleton, a jedno zamykání. Tímto návrhovým vzorem zabezpečíme, že i při použití jednoho singletonu více vlákny nedojde k vícenásobnému vytvoření instance.

9 Toto API je definováno v ANSI/IEEE POSIX 1003.1 z roku 1995

```
template <class TYPE>
class Singleton
{
   public:
      static TYPE& Instance()
      {
         if (!m_instance)
         {
            MutexGuard g(m_mutex);
            if (!m_instance)
               m_instance = new TYPE();
         }
         return *m_instance;
      }
   private:
      Singleton(const Singleton&);
      Singleton& operator=(const Singleton&);
      static TYPE* m_instance;
      static Mutex m_mutex;
};

// Makro pro vytvoření instance Singletonu
#define SINGLETON(TYPE) \
   template class Singleton<TYPE>; \
   template <> TYPE* Singleton<TYPE>::m_instance = 0; \
   template <> Mutex Singleton<TYPE>::m_mutex = Mutex();
```

Text 7: Náhled implementace třídy Singleton

Třída *Log* která zajišťuje výstup chybových nebo ladících zpráv na standardní výstup s možností potlačit zobrazování daného typu zpráv.

Velice důležitá je šablonová třída *CallBack*, pomocí níž se vytvářejí zpětná volání třídních metod konkretních instancí tříd nebo klasických C funkcí.

```
// Poviný interface pro všechny typu callbacků
class ICallBack
{
   public:
      virtual void Call(ucallback param) = 0;
};

template <class T, void (T::*Method)(ucallback)>
class CPP_CallBack : public ICallBack
{
   public:
      CPP_CallBack(T& owner) : m_owner(owner) {}
      void Call(ucallback param) { (m_owner.*Method)(param); } // Volání zvolené metody
      T& GetOwner() { return m_owner; }
   private:
      CPP_CallBack();
      T& m_owner;                    //Reference na třídu jejíž metodu budeme volat
};
```

Text 8: Náhled na implementaci třídy pro C++ callback

V neposlední řadě pak zmíním třídu *ConfReader*, která reprezentuje parser a zároveň i úložiště načtených hodnot z konfiguračních souborů, jejichž syntax by se dala popsat následovně.

```
<comment> ::= # {any} <newline>
<valid line> ::= <ident> = <data> ; <newline>
<char>       ::= a..z, A..Z, 0..9
<dot>        ::= .
<ident>      ::= <char> {<ident>}
<data>       ::= <char> {<dot> } {<data> }
```

5 Implementace her

Jak již bylo zmíněno v předchozích kapitolách, vlastní hry jsou reprezentovány dynamicky načítanými knihovnami, z kterých se volají celkem tři základní metody, a které musí nutně každá herní knihovna implementovat. Jedná se o metody:

```
LIB_EXPORT World* InitGame();                // Vytvoření instance hry
LIB_EXPORT void FreeGame(World* world); // Uvolnění instance hry z paměti
LIB_EXPORT uint32 GameTypeID();              // Metoda vrátí unikátní číselný identifikátor hry
```

5.1 Tahová hra

Jako demonstrační implementaci tahové hry jsem zvolil hru piškvorky. Tato hra má jasná a srozumitelná pravidla a je všeobecně známá.

V první řadě bylo potřeba definovat množinu operačních kódů, kterou bude hra i klient znát a pomocí které bude komunikace probíhat. Tato množina je definována jako výčtový typ a vypadá následovně:

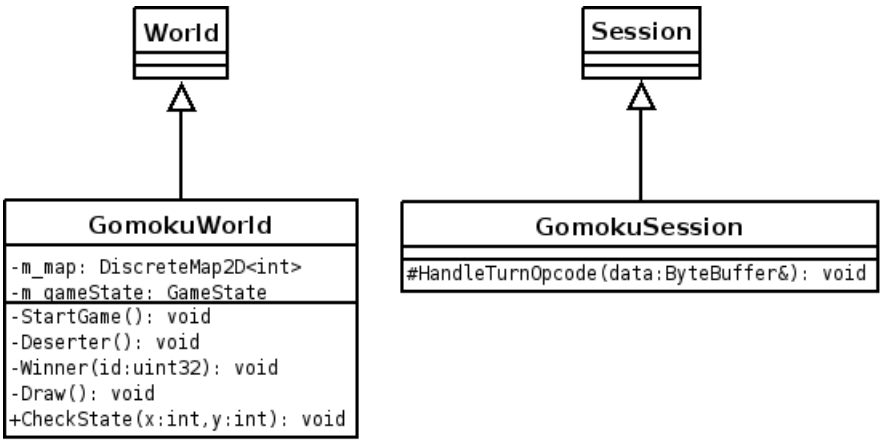
```
enum gomokuopcodes
{
   // Operační kódy odesílané serverem
   SMSG_GAMEBEGIN  = 0x100, // Oznámení o začátku hry
   SMSG_TURN       = 0x101, // Oznámení o tom, že daný hráč je na tahu
   SMSG_PLAYED     = 0x102, // Oznámení o tahu oponenta
   SMSG_GAMEEND    = 0x103, // Oznámení o ukončení hry
   SMSG_ILLEGAL    = 0x104, // Oznámení chybného tahu
   SMSG_SERVERFULL = 0x105, // Oznámení plné kapacity serveru

   // Operační kódy přijímané serverem
   CMSG_TURN       = 0x200, // Oznámení o umístění značky hráčem
};
```

Text 9: Operační kódy hry piškvorky

Prefixem *SMSG_* jsou označeny zprávy odesílané serverem hernímu klientu, zprávy označené prefixem *CMSG_* jsou odesílány herním klientem serveru. Význam jednotlivých operačních kódů je zřejmý z jejich názvů, popřípadě z komentáře, a proto nepovažuji za nutné je zde blíže komentovat. Za povšimnutí možná stojí fakt, že na server od klienta může přijít pouze jeden druh zpráv, na rozdíl od serveru však klient musí být schopen obsloužit o něco větší množinu zpráv.

Druhý krok implementace této hry bylo vytvoření vlastní herní logiky. Toho je dosaženo poděděním základních tříd, implementací ryze virtuálních metod a doplněním metod vlastních jak je možné vidět na Obr. 4.



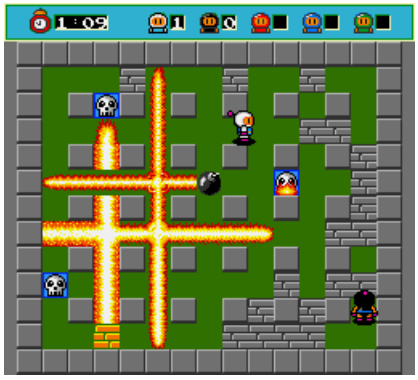
Obr. 4: Odvozené třídy pro hru piškvorky

Základní třída *World* byla rozšířena o metody *StartGame, Deserter, Winner, Draw* sloužící k oznámení začátku hry, opuštění rozehrané partie hráčem před koncem hry, oznámení vítěze a oznámení o nerozhodném výsledku. Metoda *CheckState* má za úkol po každém tahu ověřit, zda se hráč na tahu nestal vítězem partie.

Třída *Session* je v případě hry piškvorky rozšířena pouze o metodu pro zpracování paketu oznámení tahu hráče serveru. Tato metoda má na starosti kontrolu, zda daný hráč je právě na tahu, jestli byl tah veden na korektní souřadnice, případně odesílá hráči informace o tom, že tah byl chybný a z jakých důvodů byl chybný.

5.2 Hra v reálném čase

Při implementaci zástupce kategorie těchto her jsem se nechal inspirovat starou a dobře známou hrou *Dynablaster*[10]. Detailnější představu o této hře si můžete udělat z následujícího obrázku.



Obr. 5: Dynablaster

Pravidla hry jsou následující. Cílem hry je položit bomby tak, aby jejich výbuch zabil protihráče. Počet bomb, které s dají najednou položit a velikost výbuchu se v originální hře modifikuje sbíráním různých bonusů. V mojí implementaci jsou tyto atributy pevně definované. Hraje se na hrací ploše, která obsahuje pevné a zničitelné překážky.

Při implementaci hry jsem nejprve opět nadefinoval množinu operačních kódů, pomocí které server komunikuje s klientem. Tato množina je reprezentována následujícím výčtovým typem.

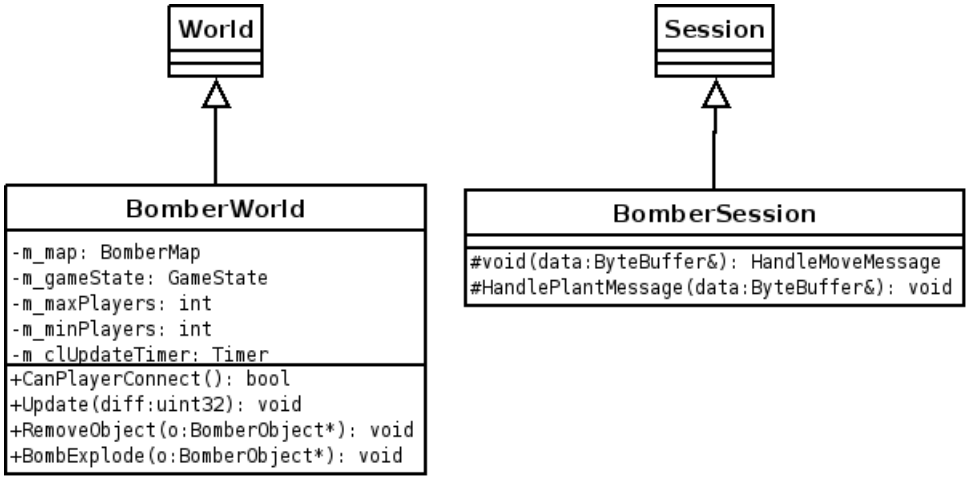
```
enum bomberopcodes
{
   // Operační kódy odesílané serverem
   SMSG_UPDATE = 0x100, // Zpráva obsahuje stavové informace hry
   SMSG_KILL       = 0x101, // Oznámení „ty jsi zabil“
   SMSG_DIE        = 0x102, // Oznámení „byl jsi zabit“

   // Operační kódy přijímané serverem
   CMSG_MOVE    = 0x200, // Požadavek na změnu polohy
   CMSG_PLANT   = 0x201  // Požadavek na položení bomby
};
```

Text 10: Operační kódy hry Bomber

Dalším implementačním krokem je vytvoření herní logiky hry. Toho dosáhneme, stejně jako v předchozí hře, poděděním třídy *World* a *Session,* které rozšíříme o příslušné metody, jak je znázorněno na Obr. 6.

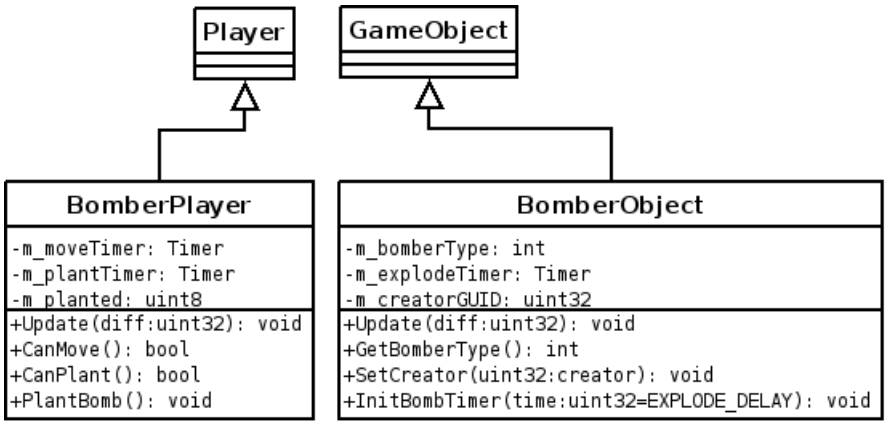
10 Více informací o této hře můžete naleznou na http://en.wikipedia.org/wiki/Bomberman_(TG-16)



Obr. 6: Poděděné třídy World a Session pro hru Bomber

Vzhledem k tomu, že se jedná o systém, ve kterém všechny děje probíhají v závislosti na čase, vzniká tak potřeba časovou informaci distribuovat do všech herních objektů, které s ní budou pracovat. Proto v každém herním objektu implementujeme metodu *Update*, jejíž parametr obsahuje informaci o časovém kvantu, které uplynulo od předchozí aktualizace objektu.

Pro potřeby reprezentace herních objektů jsem vytvořil následující třídy zděděné ze základních tříd herního prostředí.



Obr. 7: Zděděné třídy herních objektů pro hru Bomber

Vzhledem k nutnosti rozlišovat tři základní typy herních objektů (pevnou překážku, zničitelnou překážku a bombu), jsou herní objekty navíc klasifikovány výčtovým typem.

```
enum BomberObjectType
{
   WALL              = (int)'W',
   DESTRUCTABLE = (int)'D',
   BOMB              = (int)'B'
};
```

Text 11: Výčtový typ klasifikace herních objektů hry Bomber

6 Testování

Fázi testování jsem rozdělil na tři části. Část testování jednotlivých tříd nebo třídních celků, testování konkrétně implementovaných her za pomoci jednoduchých konzolových klientů a na závěr testování zprávy paměti pomocí automatického nástroje *valgrind*[11].

Z důvodů testování jednotlivých tříd nebo jejich soustav jsem souběžně s vývojem serveru vyvíjel jednoduché aplikace zaměřené na otestování dílčích funkčních celků. Pomocí nichž jsem testoval správnost parsování konfiguračních souborů, síťovou komunikaci a korektní adresování buněk v implementaci 2D herní mapy.

Pro testování jednotlivých implementovaných her jsem vytvořil jednoduché konzolové klienty. Jejich obsluha se odehrává na úrovní textových příkazů. Grafická reprezentace her je taktéž řešena pouze za pomoci textového výstupu.

Abych odhalil případné nedostatky v práci s pamětí použil jsem na závěr program *valgrind*, který tyto chyby detekuje. Analýzou jsem odhalil následující nedostatky.

```
==22048== LEAK SUMMARY:
==22048==    definitely lost: 60 bytes in 1 blocks.
==22048==      possibly lost: 0 bytes in 0 blocks.
==22048==    still reachable: 4,800 bytes in 25 blocks.
==22048==         suppressed: 0 bytes in 0 blocks.
```

Text 12: Analýza zprávy paměti

Tyto nedostatky jsou způsobeny převážně nedokonalým uvolňováním singletonů z paměti v době ukončování aplikace. Jedná se o chybu, která by neměla nikterak závažně ohrozit chod serveru a ani způsobovat velké úniky paměti.

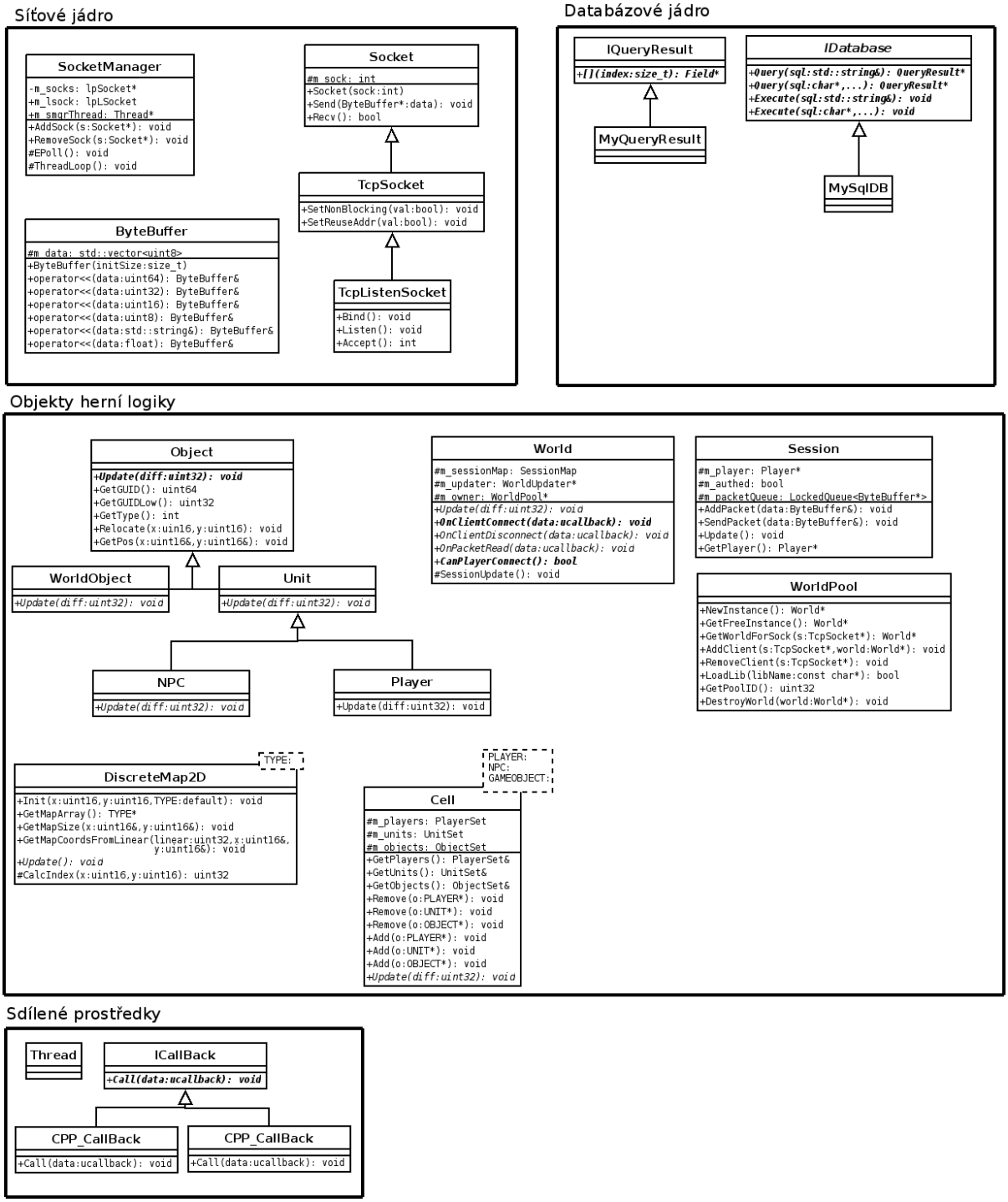
11 Domovskou stránku projektu naleznete na http://valgrind.org/

7 Závěr

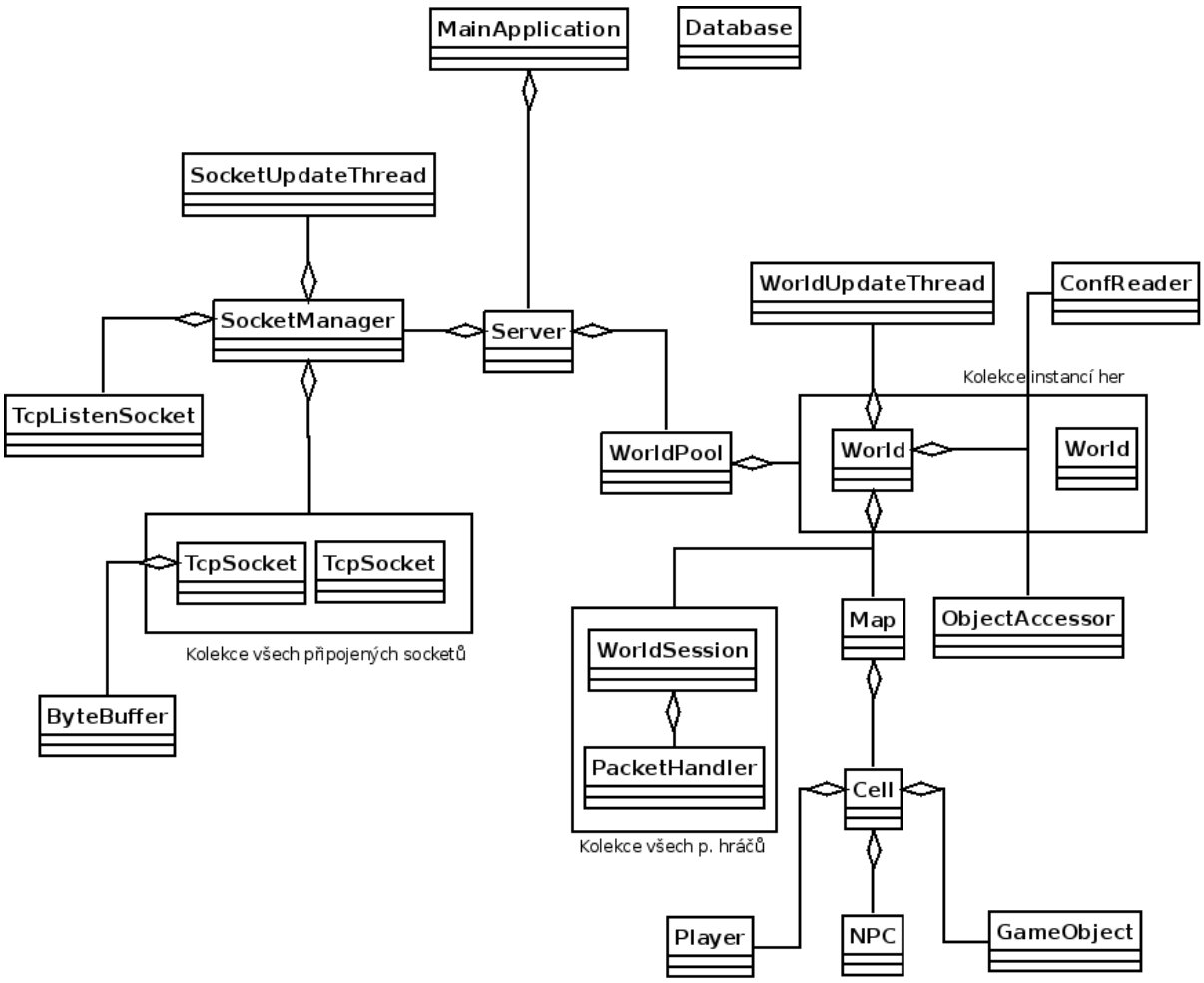
Podařilo se mi vytvořit herní platformu, která umožňuje snadnou implementaci síťových her za pomoci předem připravených základních komponent a poskytuje tak zázemí pro testování nejrůznějších algoritmů umělých inteligencí. Jako přínos vidím možnost testovat výpočetně náročné algoritmy operující v reálném čase na několika různých počítačích zároveň, jejichž simultánní spuštění na jediném výpočetním stroji by bylo například časově příliš náročné.

Zajímavou myšlenkou do budoucna a možným rozšířením je implementace malého vestavěného webového serveru, který by poskytoval statistické informace o dění v jednotlivých hrách, popřípadě by implementoval i jednoduchého webového klienta pro vybrané hry.

8 Grafická příloha



Obr. 8: Objektový návrh



Obr. 9: Graf agregací jednotlivých tříd

Literatura

[1] Open source server pro hru World of Warcraft, http://www.mangosproject.org

[2] Open source projekt asynchonní síťové knihovny, http://www.monkey.org/~provos/libevent

[3] Popis UDP protokolu, http://en.wikipedia.org/wiki/User_Datagram_Protocol

[4] Popis TCP protokolu, http://en.wikipedia.org/wiki/Transmission_Control_Protocol

Seznam příloh