Implementace serveru a klienta pro dámu po síti

Ondřej Vaic

A17B0385P
ondra.vaic@gmail.com

Seminární práce pro předmět
KIV/UPS

Fakulta aplikovaných věd
Západočeská Univerzita
14/01/2020

Zadání semestrální práce z předmětu KIV/UPS

- síťová hra pro více hráčů, architektura server-klient (1:N), PC
- server: C/C++
- klient: Java, C# (i např. Unity), Kotlin nebo jiný vysokoúrovňový jazyk (musí schválit cvičící)

Varianty zadání:

- tahová hra (prší, mariáš, šachy, piškvorky, ...)
- real-time hra (různé skákačky a střílečky, tanky, "Bulánci", ...)
- pseudo-real-time (Pong, Arkanoid, ...)

Požadavky na protokol:

- textový protokol nad transportním protokolem TCP
- bez šifrování
- využijte znalostí ze cvičení při návrhu (např. transparentnost při přenosu dat, apod.)
- když se nic neděje (žádný hráč nic nedělá), nic se neposílá
    - výjimkou může být občasná ping zpráva
- na každý požadavek přijde nějaká reakce (byť by šlo pouze o jednoznakové potvrzení, že se operace podařila)

Legenda:

- 🔴 červeně jsou označeny body, jejichž nesplnění automaticky vede k vrácení práce
- 🟡 oranžově jsou označeny body, které nemusí vést k vrácení práce (stále jsou ale povinné)
- 🔵 modře jsou označeny body, kvůli kterým práce již nebude vracena

Požadavky na aplikace:

- 🔴 aplikace jsou při odevzdání přeloženy standardním nástrojem pro automatický překlad (make, ant, maven, scons, ...; nikoliv bash skript, ani ručně gcc/javac, ani přes IDE)
    - pokud potřebujete nějakou knihovnu, verzi Javy, .. před prezentací si ji zajistěte a nainstalujte
- 🔴 je zakázáno využití jakékoliv knihovny pro síťovou komunikaci a serializace zpráv - to řeší váš kód sám pouze s BSD sockety (server) a nativní podporou ve standardní knihovně (klient; Java, C#, ..); není dovoleno použít ani C++20 networking rozhraní
- 🟡 kód aplikací je vhodně strukturovaný do modulů, resp. tříd
- 🟡 kód je dostatečně a rozumně dokumentovaný komentáři
- 🔴 aplikace (server i klient) jsou stabilní, nepadají na segfaultu (a jiných), všechny výjimky jsou ošetřené, aplikace se nezasekávají (např. deadlock)
- 🔴 počet hráčů ve hře je omezen pouze pravidly dané hry; vždy by však měla jít dohrát ve 2 lidech (abychom ji mohli testovat)
- 🟡 po vstupu se hráč dostane do "lobby" s místnostmi, hráč má možnost si vybrat místnost a vstoupit do ní (pokud nepřesahuje limit hráčů); případně je zařazen do fronty a čeká na naplnění herní místnosti
- 🔴 hra umožňuje zotavení po výpadku způsobené nečekaným ukončením klienta, krátkodobou nebo dlouhodobou síťovou nedostupností
    - dle pravidel hry se pak buď:
        - čeká na návrat hráče (hra se pozastaví)

        - nečeká na návrat hráče, hra pokračuje a po obnovení se hráč připojí do následujícího kola hry (ale hráči je stále po připojení obnoven stav)
        - nečeká na návrat hráče, hra pokračuje (ale hráči je stále po připojení obnoven stav)
    - krátkodobá nedostupnost nesmí nutit hráče k manuálnímu pokusu o připojení (klient vše provede automaticky)
    - dlouhodobá nedostupnost už by naopak měla (i s příslušnou zprávou hráči)
    - všichni hráči v dané hře musí vědět o výpadku protihráče (krátkodobém, dlouhodobém)
    - hráč, který je nedostupný dlouhodobě, je odebrán ze hry; hra pak může místnost ukončit (dle pravidel, většinou to ani jinak nejde) a vrátit aktivního protihráče zpět do lobby
- ● hráči jsou po skončení hry přesunuti zpět do "lobby"
- ● obě aplikace musí běžet bez nutnosti je restartovat (např. po několika odehraných hrách)
- ● obě aplikace ošetřují nevalidní síťové zprávy; protistranu odpojí při chybě
    - náhodná data nevyhovující protokolu (např. z /dev/urandom)
    - zprávy, které formátu protokolu vyhovují, ale obsahují očividně neplatná data (např. tah figurky na pole -1)
    - zprávy ve špatném stavu hry (např. tah, když hráč není ve hře/na tahu, apod.)
    - zprávy s nevalidními vstupy dle pravidel hry (např. šachy – diagonální tah věží)
    - ● aplikace můžou obsahovat počítadlo nevalidních zpráv a neodpojovat hned po první nevalidní zprávě, až po např. třech
- ● obě aplikace mají nějakou formu záznamu (log)
    - zaznamenávají se informace o stavech hráčů, her, popř. chybové hlášení, apod.

Server:

- ● server je schopen paralelně obsluhovat několik herních místností, aniž by se navzájem ovlivňovaly (jak ve smyslu hry, tak např. synchronizace)
- ● počet místností (limit) je nastavitelný při spouštění serveru, popř. v nějakém konfiguračním souboru
- ● celkový limit hráčů (dohromady ve hře a v lobby) je omezen; rovněž se dá nastavit při spuštění serveru nebo konfigurací
- ● stejně tak lze nastavit IP adresu a port, na které bude server naslouchat (parametr nebo konfigurační soubor; ne hardcoded)

Klient:

- ● klient implementuje grafické uživatelské rozhraní (Swing, JavaFX, Unity, popř. jiné dle možností zvoleného jazyka a prostředí) (ne konzole)
- ● klient umožní zadání adresy (IP nebo hostname) a portu pro připojení k serveru
- ● uživatelské rozhraní není závislé na odezvě protistrany - nezasekává se v průběhu např. připojení na server nebo odesílání zprávy/čekání na odpověď
- ● hráč a klient je jednoznačně identifikovaný přezdívkou (neřešíme kolize)
    - [nepovinné] chcete-li, můžete implementovat i jednoduchou registraci (přezdívka + heslo), aby se kolize vyřešily
- ● všechny uživatelské vstupy jsou ošetřeny na nevalidní hodnoty
    - totéž platí pro např. ošetření tahů ve hře (např. šachy, aby věž nemohla diagonálně, apod.)

- ● klient vždy ukazuje aktuální stav hry - aktuální hrací pole, přezdívky ostatních hráčů, kdo je na tahu, zda není nějaký hráč nedostupný, atp.
- ● klient viditelně informuje o nedostupnosti serveru - při startu hry, v lobby, ve hře
- ● klient viditelně informuje o nedostupnosti protihráče - ve hře

Dokumentace obsahuje:

- základní zkrácený popis hry, ve variantě, ve které jste se rozhodli ji implementovat
- popis protokolu dostatečný pro implementaci alternativního klienta/serveru:
    - formát zpráv
    - přenášené struktury, datové typy
    - význam přenášených dat a kódů
    - omezení vstupních hodnot a validaci dat (omezení na hodnotu, apod.)
    - návaznost zpráv, např. formou stavového diagramu
    - chybové stavy a jejich hlášení (kdy, co znamenají)
- popis implementace klienta a serveru (programátorská dokumentace)
    - dekompozice do modulů/tříd
    - rozvrstvení aplikace
    - použité knihovny, verze prostředí (Java), apod.
    - metoda paralelizace (select, vlákna, procesy)
- požadavky na překlad, spuštění a běh aplikace (verze Javy, gcc, ...)
- postup překladu
- závěr, zhodnocení dosažených výsledků

Průběžné odevzdání během semestru za bonusové body:

1) (cca 4. týden) popis protokolu - stavový diagram a přenášené zprávy (formát, apod.)
2) (cca 8. týden) kostra serveru, volitelně i klienta - připojení, elementární výměna zpráv, např. vylistování seznamu místností a možnost založit novou
3) (cca 12. týden) vyrovnání serveru/klienta s nevalidními stavy, řešení výpadků

Závěrečné odevzdání:

- minimálně je nutné získat alespoň 15 bodů, maximálně je možné získat až 30 bodů
- do termínu stanoveného cvičícím lze získat plný počet bodů (obvykle polovina ledna); poté je za každý den zpoždění (vč. sobot a nedělí) odečten 1 bod z celkového hodnocení práce
- odevzdání musí proběhnout nejpozději do mezního data stanoveného cvičícím (obvykle konec ledna)
- student předvede funkčnost řešení na PC v laboratoři UC-326
- server je spuštěn na jednom PC s GNU/Linux, klient na jednom PC s GNU/Linux a druhém PC s MS Windows
- před odevzdáním si student připraví prostředí, aby předvádění mělo hladký průběh - ověří, zda se obě aplikace úspěšně přeloží na obou prostředích, zda je lze spustit a propojit
    - laboratoř je vám k dispozici, pokud v ní zrovna neprobíhá výuka, zkouška nebo jiná akce
- průběh odevzdání bude určitě zahrnovat (v režii studenta):
    - překlad klienta a serveru
    - spuštění s různými parametry
    - odehrání jedné celé hry bez výpadků (jejich simulace) a bez nevalidních dat

- schopnost reagovat na výpadky (obě aplikace; dle zadání)
- schopnost vyrovnat se s nevalidními daty (obě aplikace; dle zadání)
- ověření náročnosti na systémové prostředky

Užitečné příkazy, tipy a triky:

- server, co naslouchá na 127.0.0.1:10000 a produkuje náhodná data:
  - cat /dev/urandom | nc -l 127.0.0.1 -p 10000
- klient, co se připojí na 127.0.0.1:10000 a produkuje náhodná data:
  - cat /dev/urandom | nc 127.0.0.1 10000
- simulace výpadku klienta/serveru (pouze vzdáleně):
  - stylem DROP (zahazuje pakety):
    - iptables -A INPUT -p tcp --dport 10000 -j DROP
  - stylem REJECT (odmítá pakety a odpovídá patřičnou ICMP zprávou):
    - iptables -A INPUT -p tcp --dport 10000 -j REJECT
  - odebrání pravidel - záměna -A za -D
- je vhodné (nikoliv povinné) zkusit, zda neuniká na serveru nějaká paměť (valgrind)
  - aby se zobrazila čísla řádku, kompilujte (gcc, clang) s přepínačem –g
  - opravením chyb, které valgrind vypíše, můžete výrazně snížit riziko „náhodných“ pádů a chyb v době odevzdání
- dodatečné verze Javy a knihoven neinstalujte do svého home – hrozí vyčerpání místa na vašem diskovém prostoru v AFS; místo toho použijte na Linuxu např. lokální složku /tmp
- pokud server padá na cílovém prostředí (Linux) a není jasné kde, přeložte rovněž s přepínačem -g a před jeho spuštěním použijte příkaz "ulimit -c unlimited" (popř. místo ‘unlimited‘ použijte nějaké rozumně velké množství paměti)
  - po pádu se vygeneruje soubor s názvem "core" - ten lze analyzovat nástrojem gdb, např. "gdb -c core muj-server" (za předpokladu, že zkompilovaná binárka serveru se jmenuje "muj-server")
  - v gdb můžete vypsat aktuální zanoření (zde v době pádu) příkazem "bt" ("backtrace")
  - případně můžete rovnou spouštět server uvnitř GDB ("gdb muj-server", příkaz "r")
  - podrobnější dokumentace nástroje GDB a jeho ovládání je například zde: http://sourceware.org/gdb/current/onlinedocs/gdb/

1 Server

Server je realizován v jazyce c++. Podporuje připojení více klientů pomocí funkce `select()`. Server se chová jako konečný automat. Server odpojí klienta při přijmutí zprávy která nezpůsobí validní přechod. Dále budou popsány hlavní části implementace serveru a formát zpráv.

1.1 Formát přijímaných zpráv

První písmeno zprávy určuje co se má vykonat. Následuje číslo zprávy odeslané klientem. Poté čárka, po ní datová část ukončena symbolem |. Například tedy i1,Ondrej|
Touto zprávou by se uživatel identifikoval.

1.2 Formát odesílaných zpráv

Stejný jako formát přijímaných zpráv s tím rozdílem že server nečísluje zprávy.

1.3 Popis řešení

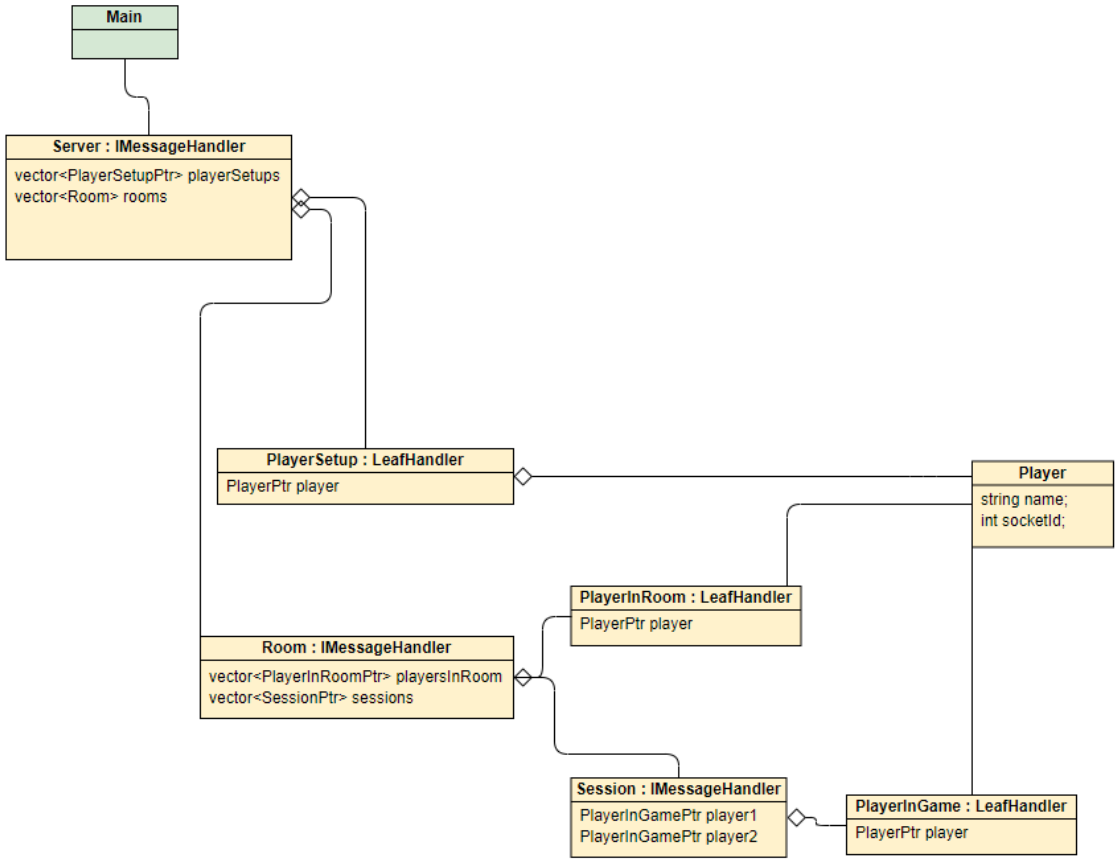
1.3.1 Main.cpp

Vstupním bodem aplikace je Main.cpp. V metodě `int main()` jsou načteny konfigurace serveru ze souboru ServerConfig.con. Dále se vytvoří instance objektu Server se specifikovanými parametry, a zavolá se nad ní metoda `MainLoop()`. Po doběhnutí `MainLoop()` program končí.

1.3.2 Datové struktury

Celý server můžeme chápat jako strom tvořený z implementací rozhraní `IMessageHandler`, listy jsou také implementace virtuální třídy `LeafHandler`. Zodpovědnost za zpracování zprávy prochází jednotlivými vrstvami až do listů kde jsou zprávy zpracovány. Zpracování může změnit stav instance `LeafHandler`. To může zapříčinit smazání této instance, a vytvoření jiné implementace `LeafHandler` která je poté předána do jiné implementace `IMessageHandler`. [1] Tento diagram znázorňuje tuto datovou strukturu.

[1] Je důležité zmínit že v celé aplikaci jsou využity smart pointery, aby se zamezilo chybám způsobeným neuvolněním paměti.



1.3.3 LeafHandler.h a IMessageHandler.cpp

Pro pochopení toku programu je důležitá třída `LeafHandler`. Je to virtuální třída která zpracovává požadavky klienta. Implementují ji třídy `PlayerInRoom`, `PlayerInGame`, `PlayerSetup` a `PlayerInGame`. Na druhou stranu instance třídy `IMessageHandler` se starají o kolekce typů `LeafHandler` nebo `IMessageHandler`. Instance `LeafHandler` jsou listy ve kterých se přijímají zprávy.

LeafHandler je značná úroveň abstrakce, která ale zamezí opakování kódu usnadní rozšiřování protokolu. Je to virtuální generická třída která se pro svou implementaci potřebuje znát typ `T`[2] a `S`[3], a implementaci funkce `init`. Jejím cílem je umožnit implementovat funkci `init()` podtřídama tak, aby bylo možné následující. Ukázka z třídy `PlayerSetup`:

1

[2] Enum stavu
[3] Typ který třídu implementuje

```
2  void PlayerSetup::init(){
3  commands[SETTING_NAME][NAME_IDENTIFY] = bind(&PlayerSetup::setName, this, _1);
4  commands[SETTING_NAME][EXIT_GAME] = bind(&PlayerSetup::exitGame, this, _1);
5  commands[SETTING_NAME][TRY_RECONNECT] = bind(&PlayerSetup::tryReconnect, this, _1);
6
7  commands[CHOOSING_ROOM][ROOM_ENTER] = bind(&PlayerSetup::setRoom, this, _1);
8  commands[CHOOSING_ROOM][BACK] = bind(&PlayerSetup::backToChoosingName, this, _1);
9  }
```

`SETTING_NAME` je `enum` který určuje stav klienta, `NAME_IDENTIFY` je konstanta typu `char` která určuje jaká zpráva byla přijata. `commands` je hashmapa hashmap které mají odkaz na funkci která se má spustit. Je-li klient ve stavu `state`, a přijme zprávu `message` jednoduše se zavolá commands[state][m− >GetIdentifier()](m). Všechny funkce odpovídají kontraktu: vrací boolean (značí jestli byla funkce provedena v pořádku) a přijímá jeden parametr typu `Message`. [4]

Tímto se tedy eliminuje velké množství if statementů, jednotlivé implementace se zkrátí, a mohou se soustředit především na výkonný kód. Před tímto zavoláním se samozřejmě musí zkontrolovat jestli existuje `commands[state][identifier]`, pokud neexistuje, klient je odpojen, protože se nejedná o validní zprávu. Tato kontrola je provedena pouze na jednom místě, není tedy možné na tuto kontrolu zapomenout. Další kontrola spočívá v tom, že i přesto že přijde zpráva která podle stavu a identifikátoru může vypadat validně, obsah nemusí být validní. Proto vrací všechny funkce které zpracovávají zprávu `boolean` hodnotu, která indikuje jestli proběhlo vše v pořádku.

Dále je uvedena implementace `ResolveMessage()` v `LeafHandler`. Lze vidět výhoda generického kódu pro přijímání a zpracování zpráv.

```
1  void ResolveMessage(fd_set* sockets) override {
2
3          if(!initialized){
4              init();
5              initialized = true;
6          }
7
8          if(!FD_ISSET(player->GetSocketId(), sockets) || player->IsRead()){
9              return;
10         }
11         NetworkManager::GetResponse(player);
12         player->CreateMessages();
13
14         vector<string> splitMessages = player->GetReadyMessages();
15         player->SetRead(true);
16
17         if(player->IsDisconnected()){
18             return;
19         }
```

[4] `Message` je jednoduchá třída která obsahuje číslo zprávy, identifikátor, a datovou část. Tyto pole se inicializují z přijmuté zprávy.

```
20
21          for (const string& message : splitMessages) {
22              MessagePtr m = make_shared<Message>(message, player);
23
24              if(m->GetIdentifier() == PING){
25                  player->Ping();
26                  NetworkManager::SendConfirm(player, m->GetMessageNumber());
27                  continue;
28              }
29
30              if(player->IsCheating() || player->IsDisconnected())
31                  return;
32
33              if (commands.find(state) != commands.end())
34              {
35                  if (commands[state].find(m->GetIdentifier()) != commands[state].end())
36                  {
37                      bool validFormat = commands[state][m->GetIdentifier()](m);
38
39                      if(validFormat){
40                          NetworkManager::SendConfirm(player, m->GetMessageNumber());
41                          continue;
42                      }
43                      cout << " invalid format or information " << endl;
44                  }
45                  cout << "incorrect identifier" << endl;
46              }
47              cout << "incorrect state" << endl;
48
49              //if state doesnt exist or identifier doesnt exit or is not valid format
50              player->SetCheating();
51              return;
52          }
53          player->ClearReadyMessages();
54      }
```

Dále bude uvedena blíže popsána implementace třídy `Server` a `Session`, tedy implementace `IMessageHandler` a implementace třídy `PlayerInRoom`, tedy implementace `LeafHandler`. Ostatní implementace `IMessageHandler` a `LeafHandler` fungují na podobném principu a tudíž si myslím že není nutné je popisovat detailně.

1.3.4 Server.cpp

Instance třídy server se stará o běh celého serveru. Hlavní je funkce `MainLoop`. Ta vytvoří na nové vlákno v kterém běží funkce která kontroluje stav klientů, jestli má být nějaký z nich odpojen.

Dále se v nekonečné smyčce pomocí funkce `select()` čeká na aktivitu některého z klientů. V případě připojení nového klienta je zavolána funkce `handleNewPlayer()`,

ta vytvoří novou instanci třídy `Player` s příslušným file descriptorem. Instance třídy `Player` reprezentuje klienta a je předávána dále protokolem. Nejdříve je s touto instancí inicializována instance třídy `PlayerSetup`, ta je uložena do vectoru `playerSetups`.

Dále je z funkce `MainLoop` zavolána funkce `ResolveMessage()`. Jako parametr jsou ji předány `fd_set sockets`, to jsou sockety na kterých se něco stalo. V třídě `Server` zpracování znamená, že se projde vector `playerSetups`, nad instancemi třídy `PlayerSetup` se znovu zavolá `ResolveMessage()`. Podle jejich stavu se buď nic neděje, klient je odpojen, nebo pokud si klient už vybral místnost tak je přidán do místnosti.

Ukázka kódu, podobně vypadají všechny další implementace `IMessageHandler` které nejsou implementace třídy `LeafHandler`.

```
1  void Server::resolveSetUps(){
2      for(auto& playerSetUp : playerSetups){
3          playerSetUp->ResolveMessage(&sockets);
4
5          if(playerSetUp->IsPlayerToLeave()){
6              close(playerSetUp->GetPlayer()->GetSocketId());
7              continue;
8          }
9
10         //add player to room
11         if(playerSetUp->IsPlayerInitialized()){
12             PlayerPtr player = playerSetUp->GetPlayer();
13             rooms[player->GetRoom()]->SetPlayer(player);
14         }
15     }
16
17     //remove added players from setups
18     Utils::RemoveIf(playerSetups,[](const PlayerSetupPtr& p){
19         return p->IsPlayerInitialized() ||
20         p->GetPlayer()->IsDisconnected() ||
21         p->GetPlayer()->IsCheating() ||
22         p->HasReconnected() || p->IsPlayerToLeave();
23     });
24 }
```

Podobně vypadá i funkce `resolveRooms()`. Z místnosti se může klient vrátit zpět do výběru místností. Prakticky to znamená že je znovu vytvořena instance třídy `PlayerSetup` s instací třídy `Player` a ta je vložena do vectoru `playerSetups`. Samozřejmě uvnitř instance `Room` jsou smazány všechny příslušné reference z vectorů které je drží.

1.3.5 Session.cpp

Instance `Session` reprezentují hru mezi dvěma hráči. Ty jsou vytvořeny v `Room`. V Session jsou dvě instance třídy `PlayerInGame`. Nejdůležitější je znovu

`ResolveMessage()`.

```
1  void Session::ResolveMessage(fd_set* sockets){
2
3      player1->ResolveMessage(sockets);
4      player2->ResolveMessage(sockets);
5
6      if(player1->IsPlaying()){
7          cout << "player 1 playing " << player1->GetPlayer()->GetSocketId() << " " << pl
8          resolveGameState(player1, player2);
9      }else{
10         resolveGameState(player2, player1);
11     }
12
13     resolveCheaterLeaver(player1, player2);
14     resolveCheaterLeaver(player2, player1);
15
16     if(player1->GetPlayer()->IsDisconnected() && player2->GetPlayer()->IsDisconnected()
17         state = SESSION_ENDED;
18     }
19 }
```

Nejdříve se zpracují zprávy od obou hráčů, a poté se nejdříve vyřeší stav hry. Ten se řeší funkcí `resolveGameState()`, jestli někdo vyhrál, pošlou se příslušné zprávy hráčům a stav se nastaví na `SESSION_ENDED`. Stav je následně kontrolován v `Room` a hráči jsou vráceny do místnosti. Dále se v `resolveGameState()` rozhodne o tom, jestli už skončil tah. Pokud ano, pošlou se příslušné zprávy oběma hráčům, a je jim změněn stav z `WAITING` na `PLAYING` a naopak. Poté je v `ResolveMessage()` zavolána funkce `resolveCheaterLeaver`. Pokud se jeden z hráčů odpojil, upraví se stavy `Session` a hráčů, a rozešlou se příslušné zprávy. Do `Session` je možné připojit hráče zpět po odpojení funkcí `TryReconnect`.

1.3.6 PlayerInRoom.cpp

Instance třídy `PlayerInRoom` reprezentují hráče který je v místnosti. Hráč může být ve dvou stavech, čekající na hru `WANTS_TO_JOIN` nebo může pouze být v místnosti v tom případě je ve stavu `CHILLING`.

```
1  void PlayerInRoom::init(){
2      commands[CHILLING][WAIT_GAME] = bind(&PlayerInRoom::setWantsToJoin, this, _1);
3      commands[CHILLING][BACK] = bind(&PlayerInRoom::backToChoosingRoom, this, _1);
4
5      commands[WANTS_TO_JOIN][STOP_WAIT_GAME] = bind(&PlayerInRoom::stopWaitingGame,
6      this, _1);
7      commands[WANTS_TO_JOIN][BACK] = bind(&PlayerInRoom::backToChoosingRoom, this, _1);
8  }
```

Funkce `setWantsToJoin` přepne stav z `CHILLING` na `WANTS_TO_JOIN`, `stopWaitingGame` udělá opak. Dále lze z obou stavů přejít do stavu `WANTS_TO_LEAVE` funkcí

`backToChoosingRoom`. Pokud je v nějaké instanci `Room` hráč ve stavu `WANTS_TO_LEAVE`, je z ní odebrán a vložen do `playerSetUps` v `Server`. Narozdíl od hráče který se akorát připojil je inicializován se stavem `CHOOSING_ROOM`.

1.3.7 Ostatní třídy

Game.cpp Reprezentuje hru a validuje tahy.

Board.cpp Reprezentuje herní desku.

Vector2D.cpp Reprezentuje bod v x a y jsou integery.

Utils.cpp Obsahuje statické pomocné funkce. Většina z nich je volána z `Game.cpp`.

NetworkManager.cpp Obsahuje statické funkce které přijímají a odesílají zprávy.

NetworkManager.h Obsahuje konstanty identifikátorů zpráv.

2 Klient

Klient je vytvořen v herním enginu Unity, pomocí programovacího jazyku c#.

2.1 Popis řešení

Stav na klientu je reprezentován aktuální scénou. V každé scéně je jeden `GameObject` který dědí od `NetworkManager`, tyto scripty končí příponou `Network` například `LoginNetwork`. Dále je v každé scéně jeden `Manager` ty končí příponou `Manager` například `GameManager`.

`NetworkManager` příjímá zprávy a na jejich základě volá funkce z objektu `Manager`. `Manager` zase zpracovává vstupy od uživatele, a podle toho odesílá příslušné zprávy.

Prakticky to může vypadat následovně. Tlačítko confirm v úvodní obrazovce spustí funkci `TryLogin()` v `LoginManager`. V té se zjistí jméno, zvaliduje, tlačítko je zablokováno a nakonec je pomocí instance `LoginNetwork` odelána zpráva s jménem. Tato zpráva je na serveru přijata a je na ní odeslána odpověď confirm nebo deny. Tato zpráva je přijata na klientu v `LoginNetwork`, ten ji pouze předá `LoginManager`, a ten buď požádá o jiné jméno, nebo načte další scénu, tedy tu kde si hráč vybere místnost. Obdobně to vypadá u ostatních zpráv.

2.1.1 NetworkManager

Nejzajímavější třídou je `NetworkManager`. Umožňuje podobně jako `LeafHandler` na serveru inicializovat funkce do hashmapy. Jak už bylo zmíněno dědí od ní ostatní třídy které jsou poté velmi jednoduché, ukázka dále.

Ve funkci `Awake()` (funkce Unity) se inicializuje spojení a spustí se vlákno které přijímá zprávy. Když je přijata celá zpráva, to znamená zpráva zakončená |, je zařazena do fronty přijatých zpráv `commandsToInvoke`. Zároveň běží `Update()` (funkce Unity, několikrát za sekundu), v které jsou tyto zprávy odebírány z `commandsToInvoke` a zpracovány pomocí `resolveCommand()`. Pokud nějaké zpracování zprávy způsobí výjimku, klient je odpojen, a je načtena scéna ve které se může znovu připojit. Synchronizace těchto dvou vláken je zajištěna použitím datové struktury `ConcurrentQueue`.

Jeden z problémů vývoje v Unity je udržení spojení mezi jednotlivými scénami. Unity typicky smaže všechny objekty při načtení nové scény. Tento problém je řešen pomocí třídy `ScriptableObject`, ta umožňuje uchovat objekty mezi scénami. `NetworkManager` si udržuje odkaz na `SocketObject` který je potomkem `ScriptableObject`. V něm je uložen socket, informace o číslování zpráv, a zprávy které byli přijaty v předchozí scéně před přepnutím do jiné scény[5]. Tyto zprávy jsou v `Awake()` v nové scéně zařazeny do fronty jako první.

Před přepnutím do jiné scény je hlavně důležité zastavit vlákno které přijímá zprávy, a uložit nezpracované zprávy. O to se stará funkce `ShutDown()`.

Pingování pro rozpoznání odpojení je implementováno tak, že klient posílá jednou za dvě sekundy ping. Na serveru se měří čas od poslední zprávy od klienta. Server na každou zprávu ping odpovídá potvrzením. Na klientu se měří čas od přijmutí těchto potvrzení.

2.1.2 LoginNetwork

Jako příklad podtřídy `NetworkManager` je zde uvedena třída `LoginNetwork`. Jelikož je krátká, je zde uvedena celá. Lze vidět jak je jednoduchá a tudíž by bylo snadné rozšířit hru o další scény.

```
1  public class LoginNetwork : NetworkManager
2  {
3      [SerializeField] private LoginManager loginManager;
4
5      protected override void initializeActions()
6      {
7          base.initializeActions();
8          actions.Add(Consts.NAME_CONFIRM, resolveNameConfirm);
9          actions.Add(Consts.NAME_DENY, resolveNameDeny);
10         actions.Add(Consts.RECONNECT, resolveReconnect);
11     }
```

[5] K tomu může dojít pokud server odešle více zpráv najednou, ale jedna z nich způsobí přepnutí přechod do jiné scény.

```
12
13      public void SendReconnectName(string name)
14      {
15          send(Consts.TRY_RECONNECT, name);
16      }
17
18      public void SendProposeName(string name)
19      {
20          send(Consts.NAME_IDENTIFY, name);
21      }
22
23      public void SendExit()
24      {
25          send(Consts.EXIT_GAME, "");
26      }
27
28      private void resolveReconnect(string message)
29      {
30          loginManager.Reconnect();
31      }
32
33      private void resolveNameConfirm(string message)
34      {
35          loginManager.ConfirmName();
36      }
37
38      private void resolveNameDeny(string message)
39      {
40          loginManager.DenyName();
41      }
42  }
```

2.1.3 LoginManager

Třída `LoginManager` také není nijak složitá. Takto může vypadat například funkce která se zavolá po kliknutí na tlačítko confirm.

```
1   public void TryLogin()
2   {
3       if(waitingForConfirm)
4           return;
5
6       waitingForConfirm = true;
7
8       string text = inputField.text;
9
10      if (text.Contains(",") || text.Contains("|"))
11      {
12          showHint();
```

```
13            waitingForConfirm = false;
14            return;
15        }
16
17        nameToSet = text;
18        networkManager.SendProposeName(text);
19    }
```

Po příchodu potvrzení jména je zavolána tato funkce. Uloží jméno, ukončí `NetworkManager` a poté načte další scénu.

```
1  public void ConfirmName()
2  {
3      PlayerPrefs.SetString("Name", nameToSet);
4      networkManager.ShutDown();
5      SceneManager.LoadScene(1);
6  }
```

2.1.4 Ostatní třídy

Consts Stejné jako `Identifiers.h` na serveru.

Figurine Reprezentuje hrací figurku.

Board Reprezentuje herní pole.

3 Uživatelská dokumentace

3.1 Server

Server lze přeložit nástrojem `make`. Stačí spustit příkaz `make` v jeho kořenovém adresáři. Server se spouští z příkazové řádky příkazem `./server` . Některé parametry serveru se dají změnit pomocí konfiguračního souboru `ServerConfig.con`, ten je uložen v kořenovém adresáři. Takto vypadá formát konfiguračního souboru:

maxNumPlayers 10
maxPendingConnections 10
port 9001
numberOfRooms 2

Tento formát je nutné dodržet a je nutné specifikovat všechny 4 parametry

3.2 Klient

Klient musí být přeložen pomocí Unity. Klienta je možné zkompilovat pro Windows i Linux bez jakýchkoliv změn. Klienta lze spustit dvojklikem na obou

operačních systémech. Na Windows `Dama.exe` na Linux `LinuxBuildFinal.x86_64` oba v kořenovém adresáři po rozbalení.

3.2.1 Scény

Klient obsahuje tyto scény. Scény jsou uvedeny v pořadí očekávaného průchodu (kromě výpadku sítě).

SetIp Hráč zde může nastavit IP adresu a port. První scéna při prvním spuštění.

Login Zde se uživatel přihlásí.

Rooms Uživatel zde vidí seznam místností a může si vybrat do které se chce připojit. Zároveň pro každou místnost vidí stejné informace o počtech hráčů jako když je v místnosti.

Room Hráč si zde může vybrat jestli chce čekat na hru, a zároveň vidí kolik lidí je ve hře, kolik lidí čeká na hru a kolik lidí je v místnosti kteří na hru nečekají.

Game Hlavní scéna, odehrává se zde hra. Hráč může táhnout pokud je na tahu, nebo jedním z tlačítek zvolit netáhnout, a nebo se vzdát.

TryReconnect Scéna která se načte při krátkodobé nedostupnosti. Klient se se zkouší opakovaně připojit, po šedesáti sekundách přejde hra do scény `Error`.

Error Zobrazí se při dlouhodobém odpojení. Uživatel musí kliknout na tlačítko Reconnect pro další pokus o připojení. Druhé tlačítko SetIp načte scénu `SetIp`.