FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

Semestrální práce KIV/UPS

Úvod do počítačových sítí

Student: Adam Míka
Osobní číslo: A22B0319P
Email: mikaa@students.zcu.cz
Datum: 20. září 2024

Obsah

Seznam obrázků

Listings

Reference

1 Zadání

Hlavní cíle: Zbytek zadání zde: *PDF*

2 Analýza úlohy

Základem této práce je zrealizovat síťovou hru „Rock-Paper-Scissors“ (kámen, nůžky, papír) mezi dvěma hráči, kteří společně odehrají několik kol (typicky 10). Aplikace se skládá z **TCP serveru** a **klientské aplikace**, přičemž mezi nimi probíhá výměna dat podle jednoduchého vlastního protokolu.

Z hlediska analýzy byly zjištěny následující klíčové požadavky:

- **Stavová logika a sledování průběhu hry:** Každý hráč může být v různých stavech (např. `LOBBY`, `PLAYING`, `RECONNECTING`), přičemž přechod do jiného stavu je řízen příchozími zprávami (`login`, `ready`, `game`, ...).
- **Definovaný formát zpráv:** Veškerá komunikace používá formát s prefixem `RPS` a jednotlivé části zprávy jsou odděleny znakem `|`. Zprávu ukončuje středník (`;`).
- **Jednoduchá kontrola validity:** Server i klient kontrolují, zda je prefix `RPS` přítomen, zda následuje očekávaný typ zprávy (např. `login`, `ping`, `game`) a zda je dle herního stavu tato zpráva povolena. Nevalidní zpráva se okamžitě odmítá.
- **Přesné řízení průběhu herního kola:** Hráči nezávisle odešlou své volby (*rock*, *paper*, *scissors*), server vyhodnotí výsledek a aktualizuje skóre (v rámci `score|x|y;` zprávy).
- **Obsluha odpojení a reconnect:** Aplikace řeší situace, kdy jeden z hráčů ztratí spojení. Server hráče dočasně označí jako `RECONNECTING` a pokud se připojí zpět, hra může pokračovat. V případě trvalého odpojení je druhý hráč přesunut zpět do `LOBBY`.

Součástí implementace je také mechanismus `ping--pong`, který slouží ke zjištění dostupnosti klienta, popř. k detekci neaktivních klientů v důsledku chyb na síti. Celkově tak aplikace zajišťuje robustnost a konzistenci probíhajícího zápasu. Výsledkem je funkční a přehledné řešení pro multiplayerovou hru s možností obsluhy více hráčů a jednoduchou správou hry na úrovni serveru.

3 Popis síťového protokolu

3.1 Základní formát zprávy

Každá zpráva mezi klientem a serverem začíná tzv. *magickým prefixem* `RPS` a má strukturu:

RPS| ⟨příkaz⟩ | ⟨data⟩;

Znak `;` ukončuje jednu logickou zprávu.

3.2 Typy zpráv

V této sekci jsou uvedeny typy zpráv používané v komunikačním protokolu aplikace Rock-Paper-Scissors. Zprávy jsou rozděleny na dvě kategorie: **zprávy odesílané z klienta na server** a **zprávy odesílané ze serveru na klienta**.

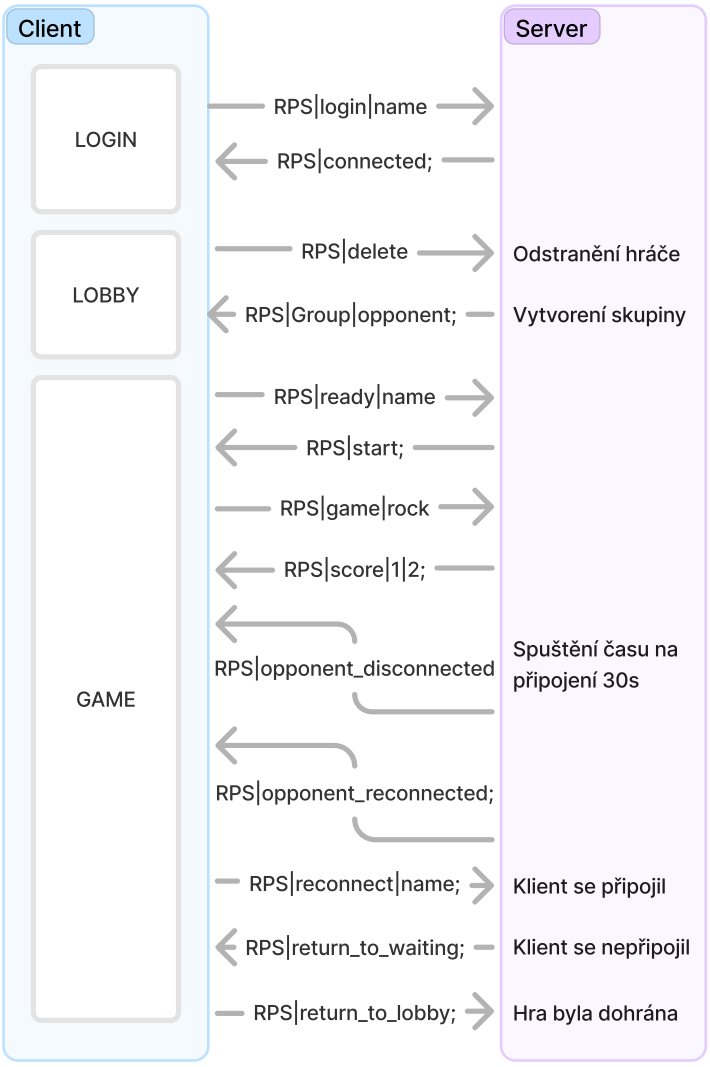
3.2.1 Příklady zpráv z klienta na server

- `RPS|login|Alice` – Klient žádá o přihlášení se jménem „Alice“.
- `RPS|ping` – Klient zasílá dotaz typu *heartbeat*.
- `RPS|ready|Alice` – Klient se hlásí jako připraven zahájit hru.
- `RPS|game|rock` – Klient posílá svou volbu „kámen“.
- `RPS|reconnect|Alice` – Pokud dojde k odpojení klienta, při opětovném připojení klient posílá tuto zprávu na server.
- `RPS|return_to_lobby` – Po ukončení hry klient posílá zprávu na server jako informaci o připojení do lobby pro další hru.
- `RPS|delete` – V případě vypnutí klienta kliknutím na křížek klient posílá tuto zprávu na server.

3.2.2 Příklady zpráv ze serveru na klienta

- `RPS|connected;` – Server potvrzuje, že uživatel „Alice“ se úspěšně přihlásil.
- `RPS|pong;` – Server odpovídá na *ping* zprávu.
- `RPS|Group 1|opponent_name;` – Server oznamuje klientovi, že byl zařazen do skupiny `Group 1` s protivníkem `opponent_name`.
- `RPS|start;` – Server oznamuje začátek hry.

- `RPS|score|1|2;` – Server posílá aktuální stav skóre (`player_score = 1`, `opponent_score = 2`).
- `RPS|opponent_disconnected;` – Pokud dojde k dočasnému odpojení klienta, server posílá zprávu, aby klient počkal na opětovné připojení protivníka.
- `RPS|opponent_reconnected;` – Oznámení, že se klient znovu připojil a hra může pokračovat.
- `RPS|return_to_waiting;` – Pokud vyprší čas na připojení, server informuje klienta, aby se přesunul zpět do lobby pro další hru.
- `RPS|error|Invalid action;` – Server vrací chybový stav klientovi, pokud klient provedl neplatný tah nebo jinou chybu.



Obrázek 1: Diagram komunikace serveru a klienta

3.3 Validace zpráv

Validace zpráv v tomto programu se provádí kontrolou formátu, obsahu a také kontrolou souvislosti se stavem hráče. Zpráva je vždy rozdělena rourou ("—") na několik částí, z nichž první bývá prefix (například "RPS") identifikující protokol, a další části nesou informaci o typu zprávy (jako "login", "ready", "game") a konkrétních datech (např. ID hráče nebo volba "rock/-paper/scissors"). Pokud je formát neúplný, neodpovídá očekávanému prefixu nebo pokud zpráva nedává smysl z hlediska herního stavu hráče, je vyhodnocena jako nevalidní.

Typické příklady chybně formátovaných nebo nevalidních zpráv:

1. `ABCD|login|muj-hrac` - *Prefix "ABCD" je neplatný (chybí "RPS").*
2. `RPS|ping` od hráče, který ještě neprošel `login` - *Server nezná ID daného hráče (stav "Unknown player"), proto je zpráva vyhodnocena jako nevalidní.*
3. `RPSloginmuj-hrac` - *Chybějí oddělovače "—" i prefix "RPS". Je to jeden nepravidelný řetězec.*
4. `RPS||ready` - *Za prefixem "RPS" chybí typ zprávy. Volná roura navíc signalizuje chybějící část.*
5. `RPS|game|scissors` od hráče, který ještě není ve stavu `PLAYING` - *Hráč musí být nejprve* ***`login`****, poté* ***`ready`****, aby mohl posílat herní tah. Server proto zprávu odmítne.*
6. `RPS|ready|` - *Za slovem* ***`ready`*** *chybí ID hráče. Není možné ověřit, kdo tuto zprávu posílá.*
7. `RPS|error|XYZ` - *Pokud server obdrží tento formát od klienta, jedná se o neznámý typ zprávy, nebo je vyžadováno jeho odeslání pouze ze strany serveru, a tudíž ji vyhodnotí jako nevalidní.*

Nevalidní zpráva se obvykle vyřizuje okamžitým zasláním chybové odpovědi (např. ve formě `RPS|error|důvod_chybové_zprávy;`) a následným ukončením spojení, pokud se jedná o závažné porušení formátu nebo stavu.

4 Struktura klienta

Klientská aplikace je zodpovědná za komunikaci se serverem, přijímání zpráv a reakci na ně. Níže jsou popsány nejdůležitější části kódu klienta, které zajišťují tyto funkce.

4.1 ServerListener.listen_to_server

```
def listen_to_server(self):
    buffer = ""
    while True:
        try:
            response = self.client_socket.recv(1024).
               decode("utf-8")
            buffer += response
            while ";" in buffer:
                message, buffer = buffer.split(";", 1)
                if message.startswith("RPS|"):
                    if message == "RPS|pong":
                        self.last_pong_received = time.
                           time()
                    else:
                        self.route_server_message(
                           message[4:])
                else:
                    self.disconnect_client("Invalid
                       message received")
                    return
        except OSError:
            # Zpracování ztráty spojení
            ...
        except Exception as e:
            # Neznámá chyba => odpojení
            ...
```

Listing 1: Naslouchání zprávám od serveru

Popis:

- **Přijímání dat od serveru:** Používá metodu `recv` k přijímání bloků dat od serveru a ukládá je do proměnné `buffer`.
- **Oddělení jednotlivých zpráv:** Hledá znak `;` jako oddělovač mezi logickými zprávami a rozděluje je.

- **Validace prefixu `RPS|`:** Kontroluje, zda zpráva začíná prefixem `RPS|`. Pokud ne, odpojí klienta.
- **Zpracování `pong`:** Aktualizuje čas příjmu `pongu` při obdržení zprávy `RPS|pong`.
- **Směrování ostatních zpráv:** Validní zprávy bez `pong` předává metodě `route_server_message` pro další zpracování.

4.2 `ServerListener.route_server_message`

```
def route_server_message(self, message: str):
    parts = message.split("|")
    if not parts:
        self.disconnect_client("Invalid message received
           ")
        return

    message_type = parts[0]
    if message_type == "score" and len(parts) == 3:
        # Zpracování stavu skóre
        self.update_score(int(parts[1]), int(parts[2]))
    elif message_type == "error" and len(parts) == 2:
        self.disconnect_client(parts[1])
    # Další případy (start, opponent_reconnected, atd.)
```

Listing 2: Směrování routování zpráv podle typu

Popis:

- **Rozdělení zprávy:** Rozděluje zprávu podle znaku `|` do jednotlivých částí.
- **Reakce na typ zprávy:**
  - `score`: Aktualizuje skóre hráčů.
  - `error`: Odpojí klienta s příslušnou chybovou zprávou.
  - `start`, `opponent_reconnected`: Další specifické reakce na různé typy zpráv.
- **Odpojení při neznámém typu zprávy:** Pokud je typ zprávy neznámý nebo nesprávně formátovaný, klient se odpojí.

4.3 ServerListener.disconnect_client

```
def disconnect_client(self, error_message="Connection lost"):
    self.ping_active = False
    try:
        if self.client_socket:
            self.client_socket.close()
    except:
        pass
    # Zobrazení chyby uživateli a návrat na přihlašovací obrazovku
    self.show_error_window(error_message)
    self.return_to_login_screen()
```

Listing 3: Zajištění odpojení klienta

Popis:

- **Zastavení ping mechanismu:** Nastaví `ping_active` na `False`, aby klient přestal odesílat `ping` zprávy.
- **Uzavření socketu:** Pokusí se bezpečně uzavřít socket připojení k serveru.
- **Informování uživatele:** Zobrazuje uživateli chybovou hlášku a vrací jej na přihlašovací obrazovku.

5 Struktura serveru

Serverová aplikace je zodpovědná za správu herního stavu, komunikaci s klienty a zajištění konzistence hry. Níže jsou popsány nejdůležitější části kódu serveru, které zajišťují tyto funkce.

5.1 TCPServer::handleClientData

```
void TCPServer::handleClientData(int fd)
{
    int a2read = 0;
    ioctl(fd, FIONREAD, &a2read);
    if (a2read > 0) {
        std::vector<char> buffer(a2read);
        int bytes_received = recv(fd, buffer.data(),
            a2read, 0);
        std::string message(buffer.begin(), buffer.begin
            () + bytes_received);
        auto parts = split(message, '|');
        if (parts[0] != "RPS") {
            reset_and_remove_player(
                get_player_id_from_socket(fd));
        }
        // Další zpracování zprávy
    }
    // Zpracování případného zavření spojení
}
```

Listing 4: Zpracování dat od klienta

Popis:

- **Kontrola dostupných dat:** Pomocí `ioctl` zjistí, kolik dat je připraveno k přečtení na daném socketu.
- **Přijetí dat:** Pokud jsou data k dispozici, použije `recv` k jejich přečtení do bufferu.
- **Validace zprávy:** Rozdělí zprávu podle znaku `|` a zkontroluje prefix `RPS`. Pokud prefix chybí, hráč je odstraněn ze serveru.
- **Další zpracování:** Pokud je zpráva validní, pokračuje se ve zpracování konkrétního příkazu (např. `login`, `ready`, `game`).

5.2 GameServer::reset_and_remove_player

```
1 void GameServer::reset_and_remove_player(const std::
     string &player_id)
2 {
3     player_queue.erase(player_id);
4     player_last_ping.erase(player_id);
5     player_groups.erase(player_id);
6     notify_player_return_to_lobby(get_opponent(player_id
        ));
7     // Další vyčištění
8 }
```

Listing 5: Resetování a odstranění hráče

Popis:

- **Odstranění z fronty hráčů:** Hráč je odstraněn z fronty čekajících hráčů (`player_queue`).
- **Vymazání pingů:** Odstraní záznam o posledním `pingu` hráče (`player_last_ping`).
- **Odstranění ze skupiny:** Hráč je odstraněn ze své herní skupiny (`player_groups`).
- **Informování protivníka:** Oznámí protivníkovi, aby se vrátil do lobby prostřednictvím `notify_player_return_to_lobby`.

5.3 TCPServer::run

```
1  void TCPServer::run()
2  {
3      while (running) {
4          fd_set tests = client_socks;
5          select(FD_SETSIZE, &tests, nullptr, nullptr,
               nullptr);
6          for (int fd = 3; fd < FD_SETSIZE; ++fd) {
7              if (FD_ISSET(fd, &tests)) {
8                  if (fd == server_socket)
                       handleNewConnection();
9                  else handleClientData(fd);
10             }
11         }
12         check_for_timeouts();
13     }
14     cleanup();
15 }
```

Listing 6: Hlavní smyčka serveru

Popis:

- **Sledování socketů:** Používá `select` k čekání na aktivitu na kterémkoli socketu (nové připojení nebo data od klienta).
- **Zpracování aktivních socketů:** Pro každý aktivní socket zavolá příslušnou metodu – `handleNewConnection` pro nové připojení nebo `handleClientData` pro data od klienta.
- **Kontrola timeoutů:** Pravidelně kontroluje, zda nedošlo k překročení časových limitů pro `ping` zprávy a jiné časově citlivé operace.
- **Čištění po ukončení:** Při ukončení běhu serveru zavolá metodu `cleanup` pro uvolnění všech zdrojů.

5.4 TCPServer::handleNewConnection

```
1  void TCPServer::handleNewConnection()
2  {
3      sockaddr_in peer_addr{};
4      socklen_t len_addr = sizeof(peer_addr);
5      int client_socket = accept(server_socket,
           reinterpret_cast<sockaddr *>(&peer_addr), &
           len_addr);
6      if (client_socket != -1) {
7          FD_SET(client_socket, &client_socks);
8          std::cout << "New client connected: " <<
               client_socket << "\n";
9      }
10 }
```

Listing 7: Zpracování nového připojení

Popis:

- **Přijetí spojení:** Používá `accept` k přijetí nového klienta.
- **Sledování nového socketu:** Přidá nový socket klienta do sady `client_socks`, aby byl sledován na budoucí aktivitu.
- **Logování:** Vypíše informaci o novém připojení pro účely sledování a ladění.

5.5 GameServer::check_for_timeouts

```
void GameServer::check_for_timeouts()
{
    auto now = std::chrono::steady_clock::now();
    for (auto it = player_last_ping.begin(); it !=
       player_last_ping.end(); ) {
        if (std::chrono::duration_cast<std::chrono::
           seconds>(now - it->second).count() > 5) {
            reset_and_remove_player(it->first);
            it = player_last_ping.erase(it);
        } else {
            ++it;
        }
    }
}
```

Listing 8: Kontrola timeoutů hráčů

Popis:

- **Detekce neaktivních hráčů:** Prochází seznam posledních `ping` zpráv a identifikuje hráče, kteří neodpověděli v rámci definovaného časového limitu (např. 5 sekund).
- **Odstranění timeoutovaných hráčů:** Hráči, kteří překročili časový limit, jsou odstraněni ze všech herních struktur pomocí `reset_and_remove_player`.
- **Čištění záznamů:** Vymazání hráčů z `player_last_ping`, aby se zabránilo opakovaným kontrolám.

5.6 TCPServer::cleanup

```
void TCPServer::cleanup()
{
    for (int fd = 0; fd < FD_SETSIZE; ++fd) {
        if (FD_ISSET(fd, &client_socks)) {
            close(fd);
            FD_CLR(fd, &client_socks);
        }
    }
    if (server_socket != -1) {
        close(server_socket);
    }
    game_server.cleanup();
    std::cout << "Server cleanup completed.\n";
}
```

Listing 9: Čištění serverových zdrojů

Popis:

- **Uzavření klientských socketů:** Prochází všechny možné sockety a zavírá ty, které jsou aktivní, čímž uvolňuje systémové zdroje.
- **Uzavření serverového socketu:** Bezpečně zavře hlavní serverový socket, který naslouchá na nová připojení.
- **Vyčištění herních struktur:** Volá metodu `cleanup` na objektu `game_server` pro odstranění všech zbytkových dat a stavů.
- **Logování:** Vypisuje informace o průběhu čištění pro účely sledování a ladění.

5.7 Shrnutí

Tato sekce popisuje klíčové komponenty serverové části aplikace, které zajišťují komunikaci s klienty, správu herního stavu a udržení konzistence hry. Metoda `handleClientData` zpracovává příchozí data od klientů, `reset_and_remove_player` zajišťuje čisté odstranění hráče ze všech herních struktur, `run` představuje hlavní smyčku serveru, která čeká na aktivitu a řídí průběh hry, a `check_for_timeouts` detekuje a odstraňuje neaktivní hráče. Metoda `cleanup` pak zajišťuje bezpečné uvolnění všech zdrojů při ukončení serveru.

6 Závěr

Cílem této práce bylo navrhnout a implementovat jednoduchý, ale robustní komunikační protokol pro hru „Rock-Paper-Scissors“ ve formě klient–server architektury. Dokumentace popsala základní strukturu komunikačních zpráv (prefix `RPS`, oddělovač `|`, ukončovací znak `;`), validaci těchto zpráv na straně klienta i serveru a zpracování herních stavů. Popsané řešení umožňuje přihlášení hráčů, ping–pong monitorování aktivity, správu zápasů, detekci timeoutů i zvládnutí situací, kdy se hráč znovu připojuje po ztrátě spojení.

Hlavním přínosem je srozumitelný a snadno rozšiřitelný protokol, který minimalizuje chybné stavy a umožňuje rychlou identifikaci a odpojení klienta při neplatném formátu nebo vypršení časového limitu. Výsledkem je funkční implementace, která demonstruje jak principy síťové komunikace, tak i správu stavů a synchronizaci dat ve hře pro dva hráče. Tento základ lze dále rozvíjet, například přidáním dalších herních režimů či možností pro více hráčů, aniž by bylo nutné zásadně měnit stávající protokol.