České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

BAKALÁŘSKÁ PRÁCE

2007 Radek Vláčil

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra matematiky
Obor: Inženýrská informatika
Zaměření: Softwarové inženýrství

Vývoj rozhraní pro vzdálené ovládání systému mainframe
Development of the system for the remote control of a mainframe

BAKALÁŘSKÁ PRÁCE

Vypracoval: Radek Vláčil
Vedoucí práce: Ing. Tomáš Oberhuber
Rok: 2007

Prohlášení

Prohlašuji, že tuto bakalářskou práci jsem vypracoval samostatně. Při tvorbě bakalářské práce jsem čerpal pouze z literatury, uvedené v seznamu použité literatury.

V Praze dne _____ _____

podpis

Poděkování

Rád bych poděkoval zejména vedoucímu bakalářské práce Ing. Tomáši Oberhuberovi za jeho rady a věcné připomínky. Dále také zaměstnancům firmy CA za jejich vstřícnost a ochotu.

Název práce: **Vývoj rozhraní pro vzdálené ovládání systému mainframe**

Autor: Radek Vláčil

Obor: Inženýrská informatika

Druh práce: Bakalářská práce

Vedoucí práce: Ing. Tomáš Oberhuber Katedra matematiky, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze.

Abstrakt: Hlavním cílem této práce je položit základ aplikace, která umožní vyhnout se pracování s TSO/ISPF při vývoji na mainframe. Tím značně zjednodušit jak samotný vývoj, tak i zapracovávání nových programátorů. Myšlenka projektu je taková, že bude kompletní IDE na klientské straně. Na straně serveru pak bude jen aplikace, která bude přijímat příkazy od klienta a vykonávat je. Vzhledem k nutnosti efektivní komunikace bude server s klienty komunikovat pomocí TCP/IP soketů. V práci jsou tedy shrnuty základní znalosti jak o mainframe, tak o tom, jak pracovat se sokety standardu POSIX. Nakonec je popsán základní návrh aplikace, kterou jsme vytvořili a kterou budeme dále rozvíjet.

Klíčová slova: Mainframe, C/C++, vzdálený, rozhraní, server

Title: **Development of the system for the remote control of a mainframe**

Author: Radek Vláčil

Abstract: Main goal of this project is to develop an application which would be able to replace the necessity of interacting with TSO/ISPF during the development of other projects. That will simplify not only the development but training of new developers as well. The idea is to have complete IDE on the client side. On the server side there will be only an application receiving, answering and processing demands from the client. Due to requirement of effective communication the server and the client will communicate by TCP/IP sockets. In this work there are included basic knowledge of mainframe as well as TCP/IP sockets POSIX compliant. At the end there is described design of our application, that we created and that will be further developed.

Key words: Mainframe, C/C++, remote, interface, server

Obsah

Úvod

Hlavní cíle

Jak už název práce napovídá, naším úkolem bude vyvinout aplikaci, která bude umožňovat vzdáleně ovládat systém mainframe. Vzhledem k tomu, že tento úkol je značně náročný, je jasné, že v této práci půjde především o nastudování problematiky a nakonec alespoň o základní návrh, který jsme vytvořili. Aplikace se vyvíjí především pro to, aby zjednodušila proces vývoje dalších aplikací. Budeme se soustředit na to, aby se programátor mohl vyhnout práci s TSO/ISPF, kde TSO je obdoba příkazové řádky a ISPF je textové uživatelské rozhraní, a mohl pracovat se svým oblíbeným vývojovým prostředím (IDE). Hlavní myšlenkou tedy bude mít kompletní IDE na pracovní stanici. Na straně serveru, pak bude pouze aplikace zpracovávající požadavky od klienta. Přenášený objem dat může být při vývoji větších projektů značný, a proto se budeme snažit, aby byla komunikace efektivní. Rozdělili jsme si průběh vývoje do několika částí

1. Vytvoření spojení mezi klientem a serverem
2. Autorizace klienta
3. Schopnost prohlížet datové sady (obdoba souborů na mainframe)
4. Schopnost vytvořit a mazat datové sady
5. Schopnost editovat datové sady
6. Schopnost spouštět úkoly a prohlížet jejich výstup
7. Schopnost monitorování úkolů, jejich ukončování
8. Schopnost debugování kódu

Způsob implementace

Aplikace bude vyvíjena v C/C++. Vzhledem k potřebě efektivity komunikace jsme se rozhodli, že navrhneme způsob komunikace od základů, abychom si ho mohli lépe uzpůsobit našim požadavkům. Jedním z dalších požadavků je i přenositelnost klientské části. To vylučuje použití například RPC, u kterého by mohly vznikat kolize vzhledem k různým implementacím. Síťová komunikace proto bude implementována pomocí TCP/IP soketů.

Implementace serveru

Na straně serveru budou, kromě již jmenovaného, použita vlákna. Vzhledem k tomu, že programy pro mainframe jsou z velké části psané v assembleru, nevyhneme se tomu ani my a bude nutné pro některé požadavky použít assemblerovská makra.

Implementace klienta

Klientská strana se bude skládat z jedné knihovny, která bude obsahovat všechna volání umožňující komunikaci se serverem. Dále pak to budou jednotlivé klientské aplikace linkované s touto knihovnou a využívající její funkce (API). Mohou mít formu rozšiřujícího modulu do již existujícího IDE, nebo úplně samostatné aplikace. Vzhledem k potřebě přenositelnosti klientské strany by se nemělo API klientské knihovny lišit na jednotlivých systémech.

K tomu, abychom mohli být při vývoji úspěšní, budeme potřebovat znalosti jak o mainframe tak o samotné síťové komunikaci. Proto se nejdříve zaměříme na potřebnou teorii a pak na popis struktury navržené aplikace.

1 Mainframe

V této části si popíšeme mainframe, uvedeme jejich vlastnosti a seznámíme se se základními nástroji, které budeme pro vývoj na mainframe potřebovat.

1.1 Úvod do problematiky

1.1.1 Historie

V roce 1964 poprvé představila firma IBM rodinu pěti vysoce výkonných strojů nazvaných System/360 (S360). Ty znamenaly počátek éry mainframe. Tehdy to byly jediné počítače, které se dalo pořídit. Počítače S/360 pomáhaly například Apollu 11 při přistání na Měsíci. IBM uvedla na trh zhruba každých 10 let nový model. V roce 1970 to byly S370. Jejich hlavní výhodou bylo, že mohly používat více procesorů, a proto se jejich výkon dal podle potřeby zvýšit. Poprvé se objevila technologie virtuální paměti. V roce 1988 přišly S/370XA, které přinesly 31-bitové adresování (dříve 24-bitů). Pro ně byla ve stejném roce uvedena na trh aplikace DB2 pro správu databází. Ta se používá na mainframe dodnes.

Devadesátá léta byla ve znamení nástupu osobních počítačů (PC). Tehdy prožily mainframe největší krizi. PC byly malé a stále výkonnější, mnohým se zdálo, že jim mainframe již nebudou schopny konkurovat. Ukázalo se však, že díky své spolehlivosti a bezpečnosti, hrají mainframe neotřesitelnou roli na poli informačních technologií (IT). V roce 1990 byla představena nová řada mainframe S/390. Tyto mainframe byly úplnou novinkou, byly daleko menší a levnější. Obsahovaly nejen nové a výkonnější procesory, ale i jiné technologie například tzv. paralelní sysplex. Tato technologie umožňuje více počítačům S/390 pracovat dohromady jako jeden systém, umožňuje bezpečné sdílení zdrojů a také sofistikované řízení zátěže. Posledním modelem vydaným v roce 2000 jsou tzv. z/Series. Jejich nejdůležitější změnou je 64-bitová adresace a plná podpora GNU/Linuxu.

1.1.2 Využití

I když si to možná ani neuvědomujete, je velice pravděpodobné, že už jste mainframe použili. Některé banky mají totiž síť bankovních automatů napojenou právě na mainframe, který zpracovává všechna potřebná data ve vaší bance.

V dnešní době hrají mainframe velice důležitou roli především tam, kde je potřeba velký výpočetní výkon, nebo obsluha mnoha požadavků a to vše za pokud možno stálého běhu. To znamená, že mainframe jsou nasazovány především do finančnictví, zdravotnictví nebo i vládních struktur.

1.2 z/OS

Operační systém je skupina programů, které se starají o vnitřní běh počítače. Operační systém z/OS je schopen spravovat souběžně mnoho procesů. Má schopnost přistupovat k velkému množství dat, a starat se o jejich přesuny a zpracování. Tím je ideální právě pro nasazení na mainframe.

Operační systém z/OS byl vydán stejně jako mainframe typu z/Series v roce 2000, v současné době je naprosto dominantní. K jeho nejdůležitějším činnostem patří

- Správa dat, jejich skladování, načítání
- Péče o zabezpečení, kontrola přístupu k datům
- Péče o využití hardwarových prostředků, jejich přidělování běžícím aplikacím
- Poskytuje síťové služby
- Poskytuje služby pro vývoj nových aplikací
- Podpora Unixu
- Podpora tiskových služeb

Páteří systému je aplikace nazvaná BCP (*Base Control Program*), ta zprostředkovává nejzákladnější služby, například přiděluje aplikaci procesorový čas prostřednictvím WLM (*Workload Manager*). Toto nejsou jediné části z/OS, jen těch základních je přes třicet.

Kromě z/OS se na mainframe používají i jiné operační systémy. Za zmínku jistě stojí z/VM a Linux for zSeries.

z/VM (*z/Virtual Machine*) dokáže na jednom počítači spustit více jiných operačních systémů, přičemž každému systému vytvoří jeho vlastní virtuální počítač. Spuštěný systém tak vůbec nepozná, že sdílí hardwarové prostředky s ostatními systémy.

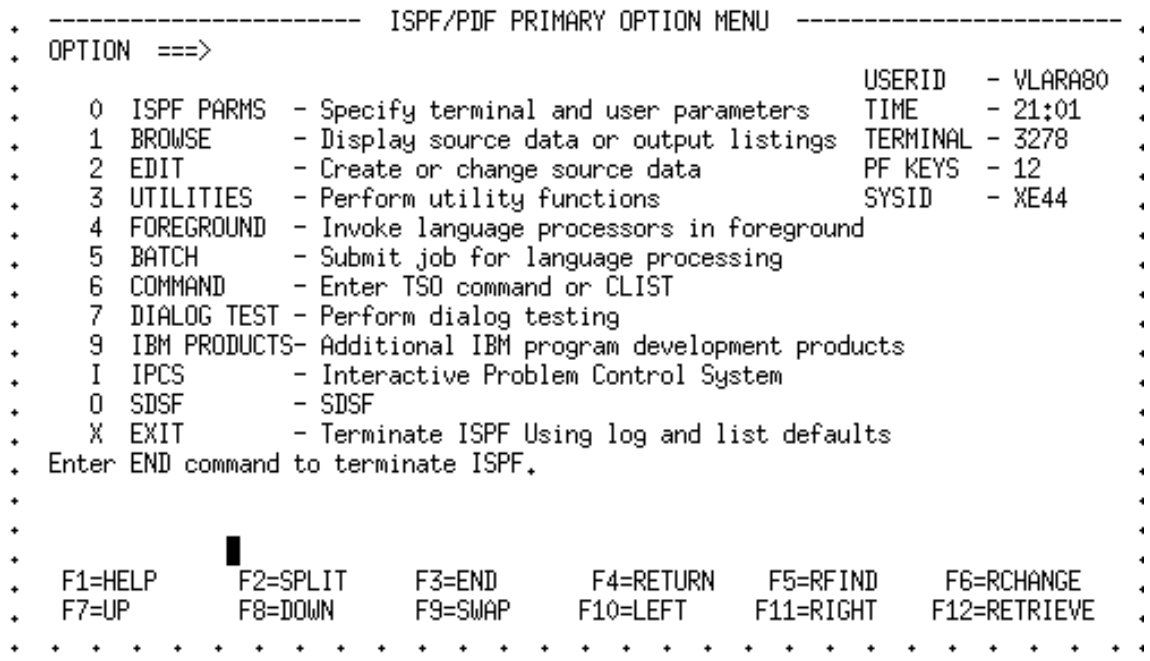
Linux for zSeries je linuxová distribuce upravená pro mainframe. Je velice oblíbená, protože umožňuje využít kvality mainframe a spojit je s širokou nabídkou aplikací pro jiné linuxové distribuce.

Díky architektuře mainframe, je takřka pravidlem, že na mainframe neběží pouze jeden operační systém. Právě využitím virtualizace se může klidně stát, že vedle sebe běží i desítky operačních systémů na jednom počítači.

1.3 ISPF

Pro z/OS existuje řada aplikací, jednou z nich je i ISPF (*Interactive System Productivity Facility*). Je to textové uživatelské prostředí, pomocí kterého můžeme pracovat se systémem z/OS. Pro práci s z/OS můžeme také použít například USS (*Unix System Services*) nebo TSO/E (*Time Sharing Option/Extensions*). TSO je vlastně příkazová řádka. USS je certifikovaná implementace UNIXu, optimalizovaná pro z/OS. Díky integraci USS s TSO umožňuje USS využívat kromě obvyklých služeb UNIXu i příkazů z TSO. Tím umožní zpracovávat UNIXové soubory pomocí ISPF.

ISPF se poprvé objevilo v roce 1975, od té doby se stále používá. Slouží jak programátorům k vývoji nových aplikací, tak uživatelům k práci s těmito aplikacemi. Slouží ale i administrátorům ke komunikaci s operačním systémem. Sestává ze čtyř hlavních komponent. Správce dialogu (DM, *Dialog Manager*) má na starost



Obrázek 1: Základní menu ISPF

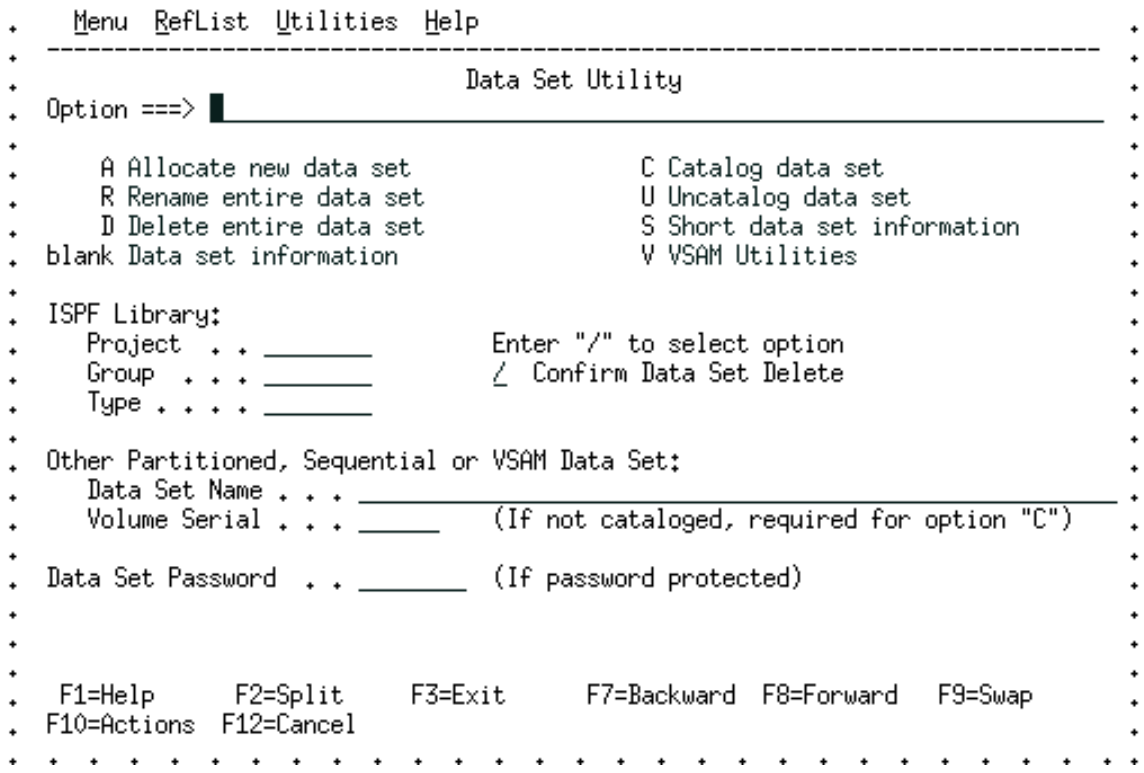
komunikaci mezi člověkem a počítačem. Přijímá příkazy, zobrazuje zprávy a stará se o panely. Další částí je nástroj pro vývoj (PDF, *Program Development Facility*), správce softwarové konfigurace knihoven (SCLM, *Software Configuration Manager*) a agent pracovní stanice (WA, *Workstation Agent*). WA umožňuje funkci ISPF na vzdálených terminálech.

1.3.1 Základní menu

Jak vypadá základní obrazovka ISPF, s kterým jsme pracovali, ukazuje obrázek 1. Příkazy se zadávají na řádek nadepsaný OPTION ===> (občas také přímo u položek). Zde zadáme číslo nebo znak určující položku menu. Pokud již víme, jaká nabídka bude následující a jakou volbu na ní chceme zvolit, můžeme použít zkráceného příkazu. V něm jsou jednotlivé volby za sebou vypsány a odděleny tečkou. Například 3.4 znamená položku 3 z aktuálního menu a 4 z následujícího. Dále ovládáme ISPF funkčními klávesami. Jejich konkrétní význam je popsán vždy ve spodní části panelu. Z těch důležitějších je to určitě F3, klávesa pro návrat. Má podobný význam jako klávesa Esc na PC. Další velice užitečná klávesa je F2, ta rozdělí obrazovku na dvě a tím vytvoří dvě na sobě nezávislá okna. Klávesa F9 pak mezi nimi přepíná.

1.3.2 Práce s datovými sadami

V ISPF je mnoho nástrojů. V dalším textu se ale zaměříme pouze na ty, které byly pro nás nejdůležitější. Byl to nástroj *Data Set Utility*, který umožňuje správu datových sad. Dále pak editor, kterým jsme tyto datové sady upravovali.



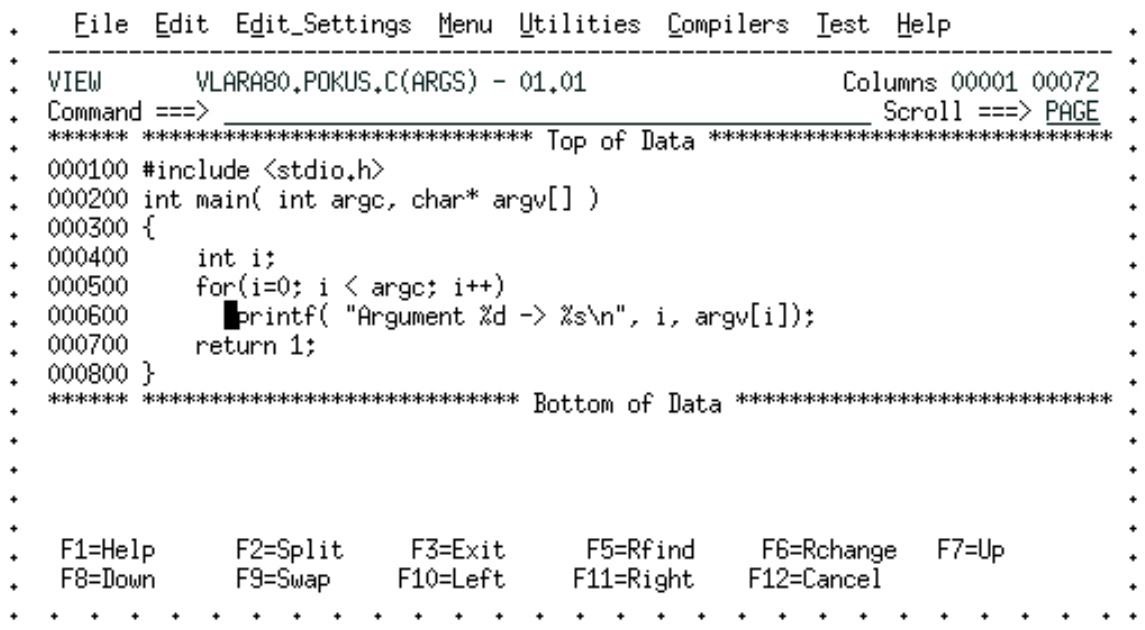
Obrázek 2: Menu Data Set Utility

V ISPF vybereme položku 3 - `UTILITIES` a následně 2 - `Data set` (zkráceně 3.2). Objeví se nám menu, které můžete vidět na obrázku 2. Je to menu *Data Set Utility*. Pomocí tohoto menu získáváme přístup k nástroji, který nám umožní provádět s datovými sadami širokou paletu operací. Z těch nejdůležitějších je to jejich vytváření a mazání. V řádcích nadepsaných PROJECT, GROUP a TYPE vyplníme postupně jméno datové sady (např. VLARA80, SOURCES, C). Na příkazovém řádku již jen vybereme, jestli chceme datovou sadu tohoto jména vytvořit nebo například smazat.

1.3.3 Editor

Když už máme vytvořenou datovou sadu, chceme do ní něco také zapsat. K tomu nám poslouží editor obsažený v ISPF. Dostat se do něj můžeme několika cestami. Nejpřímější je to přes volbu 1 - `BROWSE` nebo 2 - `EDIT` v základním menu ISPF, podle toho jestli chceme datovou sadu pouze prohlížet, nebo ji i měnit. Objeví se nové menu, ve kterém do podobných polí jako při vytváření, vyplníme jméno datové sady a stiskneme klávesu `Enter`. Další možností je například přes volbu 3.4. Dostaneme se do menu *Data Set List Utility*, která umožňuje zobrazit všechny datové sady se stejnou maskou jména. Po odeslání enterem se zobrazí seznam datových sad. Teď již stačí se dostat kurzorem vlevo vedle požadovaného jména a napsat *e* pro editaci nebo *v* pro prohlížení, odeslat enterem a jsme opět v editoru.

Obrazovku s editorem ukazuje obrázek 3. Tu si popíšeme detailněji. Úplně na-



Obrázek 3: Editor v ISPF

hoře je klasické menu, které známe i z PC. V položce File si můžeme například uložit změny. Položkou, na kterou bych ale rád upozornil, je v Edit položka Hilite. Ta nastavuje zvýrazňování syntaxe. Námi nejpoužívanější byl mód JCL a C. V levém horním rohu pak vidíme, jestli je datová sada otevřena pouze pro prohlížení (VIEW) nebo pro editaci (EDIT). Vedle tohoto je pak celé jméno otevřené datové sady. Zde se také budou hodit funkční klávesy a to hlavně ty pro posouvání. Jsou to F7/F8 pro posun nahoru/dolů a F10/F11 pro posun vpravo/vlevo. Jejich kompletní seznam i přiřazenou akci najdeme v dolní části obrazovky.

Teď již ke konkrétní úpravě souborů. Příkazy pro editor by se daly rozdělit do dvou skupin. Jedna skupina příkazů se píše na příkazový řádek nadepsaný Command ===>. Druhou skupinou jsou příkazy, které se píší na jednotlivé řádky místo čísel řádků vlevo. Začneme s druhou skupinou. Před řádek můžeme tedy napsat následující příkazy:

- i - vloží nový řádek za řádek aktuální
- d - smaže aktuální řádek
- r - zopakuje aktuální řádek
- c - příkaz pro kopírování, v pravém rohu editoru se objeví: MOVE/COPY is pending. To znamená, že jsme ještě nedokončili příkaz, řádek pro kopírování je pouze označen musíme ho ještě vložit. Vložení provedeme příkazem a nebo b (viz níže).
- m - příkaz pro přesun, pro dokončení příkazu platí to samé co pro kopírování.

- a - příkaz pro vložení. Tento příkaz provede ukončení kopírování nebo přesunu. Označený řádek se vloží za řádek aktuální.
- b - stejné jako a, pouze se označený řádek vloží před aktuální řádek.
- víceřádkové varianty dd, cc, mm, rr - pomocí těchto příkazů můžeme provádět operace s více řádky. Například při kopírování napíšeme cc na první řádek, od kterého chceme kopírovat, a stejné cc na poslední řádek kopírované oblasti. Pak již stačí stejně jako při jednořádkové variantě pouze vybrat pomocí a nebo b, kam se má vybraná oblast vložit.
- varianty s čísly - c5, d2 příkazy můžeme kombinovat ještě s čísly. Jeho hodnota stanoví, na kolik řádků chceme příslušnou operaci aplikovat. Například když chceme smazat pět řádků, stačí použít příkaz d5.

Když náhodou chceme příkaz zrušit (např. probíhající kopírování), stačí přepsat zadaný příkaz mezerami a editor se vrátí do základního módu. Můžeme použít i RESET popsaný níže.
Teď již k příkazům pro příkazovou řádku.

- SAVE - příkaz sloužící k uložení změn
- CANCEL - umožní ukončení editace bez uložení změn
- RECOVERY ON/OFF - zpřístupní funkci UNDO
- UNDO - vrátí poslední provedenou akci
- CAPS ON/OF - po odeslání řádku, se celý převede na velká písmena (výhodné pro JCL)
- PROFILE - zobrazí aktuální nastavení editoru
- RESET - provede vyčištění obrazovky editoru, smažou se všechny speciální řádky (=COLS>, =BNDS>, =MASK>, ==MSG>), provede se také odstranění zvýraznění po použití FIND a zruší se všechny nedokončené příkazy.
- FIND text - nalezne požadovaný text. Po více výskytech se můžeme pohybovat funkční klávesou F5.
- HELP - zobrazí nápovědu. Všechny řádkové i primární příkazy najdete v položce 12 a 13.
- SUBMIT - spustí aktuálně otevřený JCL úkol.

Když se vám povede zažít si všechny tyto příkazy, je psaní v editoru hračkou. Někomu, kdo je zvyklý pracovat například s editorem Vim, by nemělo činit používání tohoto editoru žádné potíže. V tomto krátkém přehledu jsme sepsali pouze základní funkce. Tomu, kdo bude tento editor používat více a chce se v něm zdokonalit, můžu jen doporučit již zmiňovaný příkaz HELP, z něj pak hlavně položky 12, 13 a 14.

1.4 JCL

Mainframe obvykle provádí dva typy prací. Jsou to dávkové úkoly a transakce v reálném čase. Dávkové úkoly (dále jen úkoly) jsou typem práce, která probíhá bez zásahu uživatele. Uživatel pouze na začátku oznámí systému, co chce provádět pomocí příkazů jazyka JCL (*Job Control Language*). Pak už jen zažádá o jejich provedení a čeká na výsledek. Jakmile příkazy odešle, ztrácí nad prováděním kontrolu. Kontrolu přebírá část systému z/OS, která se jmenuje JES (*Job entry subsystem*). Cílem JES je spouštění úloh co nejvíce automatizovat, co nejideálněji využít systémové prostředky a postarat se o konflikty při sdílení souborů. Proto jsou úkoly zadávány pomocí jazyka JCL, který je navržen tak, aby podal systému maximální informace o tom co má být provedeno, kolik paměti bude potřeba, kolik výpočetního času úkol zabere a také s jakými datovými sadami se bude pracovat, aby se zajistilo jejich případné zamčení.

Je důležité upozornit, že datová sada pro JCL musí mít přesný formát (kvůli zpětné kompatibilitě). Musí mít pevnou délku záznamů 80 znaků. Strukturu každého JCL úkolu přehledně ukazuje obrázek 4. Ačkoliv příkazů JCL je velké množství v praxi jsou nejpoužívanější následující tři.

- **JOB** používá se k pojmenování příslušného úkolu, dále může nastavovat vlastnosti celého úkolu.
- **EXEC** poskytuje informace o tom, co se má spustit. Je to buď binární program nebo procedura JCL. V jednom úkolu může být příkazů EXEC více. Jednotlivým částem s příkazem EXEC se říká krok úkolu.
- **DD** *data definition* poskytuje vstupy a výstupy příkazu EXEC. Tento příkaz spojuje datovou sadu, jiné I/O zařízení nebo funkci k DDNAME v programu, který spouštíme. DDNAME je jméno, které popisuje jednotlivé vstupy a výstupy. DDNAME jsou napsány přímo v aplikaci, kterou budeme spouštět a která pomocí nich k I/O přistupuje. Přirovnat by se daly k cout nebo cin, které jsou v C++, které ukazují na standardní vstup a výstup, ale dají se přesměrovat. Příkazy DD vždy přísluší k jednomu kroku úkolu.

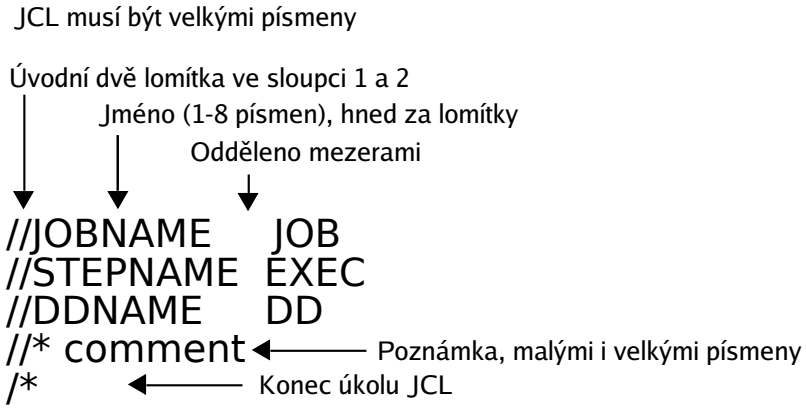
Z parametrů příkazů, které zde uvedeme, jsou vybrány jen ty důležitější. Jejich kompletní seznam a další detaily o těch vyjmenovaných najdete například v literatuře [5].

1.4.1 JOB

Uvedeme a popíšeme si některé parametry příkazu JOB, které jsme používali.

```
//VLARA80A   JOB (90300000),'VLARA80',CLASS=A,REGION=4096K,
//           MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
```

- VLARA80A je jméno úkolu pro systém. Bývá zvykem použít jméno uživatele, který úkol spouští.



Obrázek 4: Syntaxe JCL

- (90300000) určuje bezpečnostní klasifikaci a identifikační informaci o úkolu. Je u každého systému jiná. Uživatel si ji může zjistit od zkušenějších kolegů.
- 'VLARA80' jméno uživatele. Úkol bude mít stejná přístupová práva jako tento uživatel.
- CLASS definuje požadavky na systémové prostředky, příslušnou hodnotu je opět potřeba zjistit.
- REGION určuje kolik paměti bude úkolu přiděleno. Pro běžné úkoly 4096K stačí.
- MSGLEVEL určuje množství systémových hlášení, která budou přijímána.
- MSGCLASS směruje odchycené systémové zprávy. Toto je opět závislé na systému.
- NOTIFY říká, kam má být poslána informace o ukončení úkolu. Hodnota `&SYSUID` je nahrazena naším uživatelským jménem, to znamená, že informace se zobrazí nám.

1.4.2 EXEC

Příkaz EXEC má stejnou strukturu jako příkaz JOB. Většinou má jen jeden parametr. Je to buď PROC nebo PGM. Ty určují, zda se bude volat spustitelný program, nebo další JCL procedura. Když použijeme PROC, jsou v parametrech příkazu EXEC parametry volané procedury. Běžné parametry přítomny u příkazu EXEC PGM= jsou

- PARAMS nastavuje parametry spouštěného programu
- COND podmínka pro podmíněné spouštění
- TIME definuje časový limit pro zpracování

Příklad volání procedury může vypadat například takto:

```
//STEP1  EXEC PROC=SETRID,VSTUP=VLARA80.TRID.DATA(DATA),
//       VYSTUP=VLARA80.TRID.DATA(VYSTUP)
```

1.4.3 DD

Příkaz DD má velké množství parametrů. Vyjmenujeme jen ty nejdůležitější.

- DSN specifikuje jméno datové sady. Může to být i dočasná datová sada, nebo reference na datovou sadu.
- DISP nastavuje, co se má s datovou sadou dále dělat. Více v další sekci 1.4.4.
- SPACE definuje, kolik místa bude potřeba pro novou datovou sadu
- SYSOUT definuje umístění systémového výstupu (může být i datová sada)
- VOL SER specifikují jméno disku či pásky
- DCB specifikuje další vlastnosti datové sady. Má mnoho vlastních parametrů
  - LRECL logická délka záznamu, množství bytů/znaků v jednom záznamu
  - RECFM formát záznamu, např. s fixní délkou, s proměnou délkou
  - BLOCKSIZE určuje délku bloku, v kterém jsou jednotlivé záznamy. Typicky je to násobek LRECL.
  - DSORG určuje typ datové sady, např. zda je sekvenční, nebo je to knihovna
- LABEL jméno pásky
- DUMMY znamená prázdný vstup, nebo zahodit výstup (podobné jako /dev/null)
- * říká, že data budou následovat přímo v JCL úkolu

1.4.4 Parametr DISP

Tento parametr příkazu DD je jeden z nejdůležitějších. Mimo další použití totiž informuje systém, jak se zachovat k datovým sadám, aby se zabránilo konfliktům s dalšími bežícími úkoly. Kompletní parametr má tyto tři pole

```
DISP=(stav, normální konec, výjimečný konec)
```

Platné jsou i zkrácené zápisy

```
DISP=(stav, normální konec)
DISP=stav
```

- Pole stav
  - NEW ukazuje, že bude vytvořena nová datová sada. Tento úkol pak bude mít výhradní práva pro přístup k vytvořené datové sadě při svém běhu. Datová sada stejného jména nesmí existovat.
  - OLD říká, že datová sada už existuje. Tento úkol bude mít výhradní práva pro přístup k datové sadě při svém běhu.
  - SHR znamená, že datová sada už existuje. Úkol bude tuto datovou sadu sdílet s jinými právě bežícími úkoly. Tyto úkoly musí mít též nastaveno SHR.
  - MOD znamená, že datová sada už existuje. Tento úkol bude mít výhradní práva pro přístup k datové sadě. Vše, co bude zapsáno do datové sady, bude připojeno za poslední záznam.
- Pole normální konec
  Nastavení tohoto pole určuje, co se má s datovou sadou dělat, pokud krok úkolu proběhne bez chyb.
  - DELETE smaže datovou sadu.
  - KEEP ponechá datovou sadu, ale neuloží ji do katalogu.
  - CATLG ponechá datovou sadu a uloží ji do katalogu.
  - UNCATLG ponechá datovou sadu, ale vyjme ji z katalogu.
  - PASS říká, že co se má s datovou sadou udělat, je stanoveno v dalším kroku.
- Pole výjimečný konec
  Toto pole určí, co se má s datovou sadou udělat, pokud nastane v kroku chyba. To, co se do tohoto pole vyplňuje, je naprosto shodné s předchozí položkou. Můžeme tedy datovou sadu smazat, ponechat atd.

1.4.5 Alokace nové datové sady

Pokud zadáme parametr DISP roven NEW, je nutné specifikovat, jaké vlastnosti má mít nová datová sada. Používá se k tomu parametr SPACE nebo LIKE. Parametr `LIKE=jméno_datové_sady` znamená, že budou použity stejné vlastnosti, jako má zadaná datová sada. Parametr SPACE má formát:

```
SPACE=(Jednotky,(Primární, Sekundární, Adresářové bloky))
```

- Jednotky určují v jakých jednotkách budou čísla v závorce (TRK, CYL, KB, MB, REC).
- Primární udává s jakou velikostí se datová sada vytvoří.
- Sekundární udává, o kolik se datová sada může zvětšit, když je naplněna.

- Adresářové bloky udávají velikost oblasti pro popis úsekových datových sad.

```
//  DD DISP=(NEW,CATLG,DELETE),SPACE=(CYL,(10,5)),
//     DSN=VLARA80.INTRO.DATA(SORTOUT)
```

V kroku s tímto řádkem bude vytvořena nová datová sada o velikosti 10 cylindrů, s možností rozšíření o 5 cylindrů.

1.4.6 Závěr s příklady

Když už jsme si napsali nějaký úkol, chceme ho i spustit. To uděláme buď přímo z editoru nebo z TSO. V editoru v řádku pro příkazy napíšeme `SUB` nebo `SUBMIT`. V TSO je to pak `SUBMIT 'jméno datové sady s úkolem'`. Nakonec si ukážeme několik příkladů, které detailně vysvětlíme.

```
//VLARA80A   JOB (90300000),'VLARA80',CLASS=A,REGION=4096K,
//           MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
//MYSORT     EXEC PGM=SORT
//SORTIN     DD DISP=SHR,DSN=VLARA80.SORT.DATA
//SORTOUT    DD SYSOUT=*
//SYSOUT     DD SYSOUT=*
//SYSIN      DD *
   SORT FIELDS=(1,3,CH,A)
/*
```

- VLARA80A je jméno úkolu
- MYSORT jméno kroku, ve kterém se spustí program SORT
- SORTIN toto je DDNAME, přímo použito v programu SORT pro jeho vstup. Vidíme, že pro vstup, je použita datová sada se jménem `VLARA80.SORT.DATA`, která bude otevřena v stavu SHR.
- SORTOUT další DDNAME. V programu SORT použito pro jeho výstup. Výstup z programu SORT je nastaven tak, aby se zařadil k ostatním zprávám z úkolu.
- SYSOUT nastavení výstupu pro zprávy úkolu. Hvězdička znamená že je odeslán do JES na jeho standardní výstup.
- SYSIN je vstup do úkolu. V tomto případě jsou to parametry programu SORT, které nastavují jak má třídit (1-3 sloupec, podle znaků, vzestupně).

```
//OBETO80J    JOB (90300000),'OBETO80',CLASS=A,REGION=4096K,
//            MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
//STEP1   EXEC PGM=SORT
//STEPLIB DD DSN=SYS1.SICELINK,DISP=SHR
//  DD DSN=SYS1.SORTLPA,DISP=SHR
```

```
//SORTIN DD DISP=SHR,DSN=OBETO80.INTRO.DATA(PLANETS)
//SORTOUT DD DISP=(NEW,CATLG,DELETE),SPACE=(TRK,(10,5,10)),
//  DSN=OBET80.INTRO.DATA(SORTOUT)
//SYSOUT DD SYSOUT=*
//SYSIN DD *
  SORT FIELDS=(1,3,CH,A)
/*
```

- STEP1 opět spouštíme program SORT.
- STEPLIB připojí knihovnu s programy. Tato knihovna bude přístupná pouze v kroku STEP1. Příkaz DD nastaví jméno knihovny a že může být datová sada sdílená. Jelikož se pouze čte, nemůže vzniknout žádný konflikt mezi běžícími úkoly.
- SORTIN nastavení vstupu pro program SORT, opět datová sada použita jako sdílená.
- SORTOUT říká, že se pro výstup má vytvořit nová datová sada. Když krok proběhne bez chyb, má se nová datová sada uložit do katalogu, když nastane chybový stav, má se datová sada smazat. Vytvoří se datová sada o velikosti 10 stop, může se zvětšit o 5, adresář bude mít 10 bloků. DSN pak specifikuje, jaké jméno se přidělí nové datové sadě.
- Ostatní je stejné jako v předchozím příkladu.

```
//VLARA80A  JOB (90300000),'VLARA80',CLASS=A,REGION=48M,
//          MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
//MYLIB     JCLLIB ORDER=('CEE.SCEEPROC','CBC.SCBCPRC')
//COMPPRC   EXEC PROC=EDCCB,
//   CPARM='SO LIST',
//   INFILE='VLARA80.POKUS.C(SERVER)'
//   OUTFILE='VLARA80.POKUS.LOAD(SERVER),DISP=SHR'
```

Tento úkol je motivací pro příští kapitolu, překládá zdrojový kód pomocí procedury EDCCB.

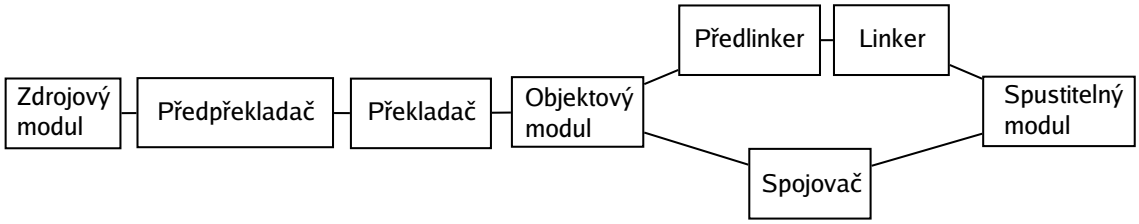
- MYLIB definuje jaké knihovny se mají připojit, JCLLIB má platnost pro celý úkol. Do závorky za parametr ORDER jich můžeme napsat více a oddělit čárkami.
- COMPPRC spustí se procedura EDCCB, která nám přeloží a spojí program uložený v datové sadě definované v parametru INFILE. Výsledek uloží do datové sady v OUTFILE, která již existuje. V parametru CPARM jsou předány parametry pro překladač.

1.5 Překlad programů v C/C++

V poslední části kapitoly si ukážeme, jak překládat zdrojové kódy C/C++ na mainframe. Na začátek si krátce vysvětlíme, jak vůbec překlad probíhá. Nakonec pak ukážeme několik příkladů.

1.5.1 Cesta zdrojového kódu

Průběh vytváření spustitelného programu je ve všech jazycích podobný. Ve všech jazycích se používá modulárního stylu. To znamená, že se každý soubor se zdrojovým kódem přeloží zvlášť a pak se vytvořené objektové moduly spojí dohromady. Výhodou tohoto stylu je, že může být každý modul v jiném jazyce. Tento modulární styl také velice šetří čas, při vývoji aplikace. Když totiž měníme některý modul, stačí přeložit pouze ten, který jsme změnili. Ty nezměněné už znovu překládat nemusíme. Celý průběh přehledně ukazuje obrázek 5.



Obrázek 5: Vytvoření spustitelného modulu

První co přijde na řadu po vytvoření zdrojového kódu, je předpřekladač. Jeho úkolem je zpracovat části zdrojového kódu, které ale nejsou částí programovacího jazyka. Například EXEC SQL či EXEC CICS. Tyto musí být nejdříve převedeny do příkazů příslušného programovacího jazyka.

Další přichází na řadu překladač. Ten vytvoří objektový modul. Tento modul není zatím spustitelný, jsou v něm totiž zatím nevyřešené odkazy na vnější funkce. Ty se vyřeší až v následující fázi.

Dále, jak je vidět z obrázku, jsou možné dvě cesty. Buď použijeme předlinker a linker, nebo použijeme spojovač. Úkolem linkeru je vyřešit jednotlivé odkazy mezi objektovými moduly a vytvořit z nich jeden spustitelný modul. Toto je ale starší a méně účinný postup. Na rozdíl od spojovače linker nezvládá zpracovávat například objektové moduly s dlouhými jmény 64-bitové moduly a jiné. Pro odstranění některých těchto nedostatků se používá předlinker.

Právě kvůli těmto nedostatkům je doporučováno používat spojovač. Spojovač dokáže to samé co linker. Navíc dokáže spojit nejen objektové, ale i spustitelné moduly. Můžeme tak z několika spustitelných modulů udělat jeden.

1.5.2 Základní příklady

K překladům vždy budeme používat pomocné procedury, které mají jednak nastavenou většinu cest k potřebným souborům a také spojují několik úkonů dohromady.

Jejich kompletní seznam najdete v použité literatuře například v [1]. Nyní si ukážeme úplně základní příklad. Mějme tento jednoduchý zdrojový kód napsaný v jazyce C. Vzniklý program by měl vypsat součet proměnných x a y.

```
VLARA80.BP.SOURCES(FIRST)

#include <stdio.h>

int main() {
   int x=1;
   int y=2;
   printf("Vysledek je: %d\n", x+y);
}
```

Tento zdrojový kód přeložíme, spojíme a spustíme následujícím úkolem.

```
//VLARA80A JOB (90300000),'VLARA80',NOTIFY=&SYSUID,
//          MSGCLASS=H,CLASS=A,MSGLEVEL=(1,1),REGION=0M
//MYLIB JCLLIB ORDER=(CBC.SCCNPRC)
//*
//COMPPRC  EXEC PROC=EDCCBG,
// INFILE='VLARA80.BP.SOURCES(FIRST)'
/*
```

- MYLIB tento řádek načte knihovnu CBC.SCCNPRC. V ní je procedura EDCCBG, kterou použijeme.
- COMPPRC v tomto kroku je spuštěna procedura EDCCBG. Z posledních třech písmen jména procedury můžeme poznat, co vlastně dělá (C-compile, B-bind, G-go). Tato procedura nám tedy zdrojový kód přeloží, spojí a spustí.
- INFILE udává vstupní soubor se zdrojovým kódem.

Tento jednoduchý příklad funguje na mainframe, který jsme měli k dispozici. Je ale asi jistější připsat, kde se mají hledat standardní hlavičky, jakou je `stdio.h`. Tyto hlavičky se nachází v CEE.SCEEH.H. Nastavíme tedy proceduře DDNAME COMPILE.SYSLIB, jak je vidět níže. Tento řádek připíšeme na konec úkolu.

```
//COMPILE.SYSLIB DD DSN=CEE.SCEEH.H,DISP=SHR
```

Další jednoduchý příklad, který si ukážeme bude napsán v C++ a bude obsahovat jeden hlavičkový soubor.

```
VLARA80.BP.HEADERS(EXAMPLE)

class A {
private:
```

```
   int a,b;
public:
   A(int x, int y) : a(x),b(y) {}
   int sum() {return a+b;}
};

VLARA80.BP.SOURCES(SECOND)

#include "example.h"
int main() {
   A a(3,4);
   printf("Vysledek je: %d\n", a.sum());
}
```

Příslušný úkol pro překlad pak bude vypadat:

```
//VLARA80A JOB (90300000),'VLARA80',NOTIFY=&SYSUID,
//          MSGCLASS=H,CLASS=A,MSGLEVEL=(1,1),REGION=0M
//MYLIB JCLLIB ORDER=(CBC.SCCNPRC)
//*
//COMPPRC  EXEC PROC=CBCCBG,
// CPARM='OPTFILE(DD:OPTFILE)',
// INFILE='VLARA80.BP.SOURCE(SECOND)'
//COMPILE.SYSLIB DD DSN=CEE.SCEEH.H,DISP=SHR
//COMPILE.OPTFILE DD *
   LSEARCH('VLARA80.BP.HEADERS')
/*
```

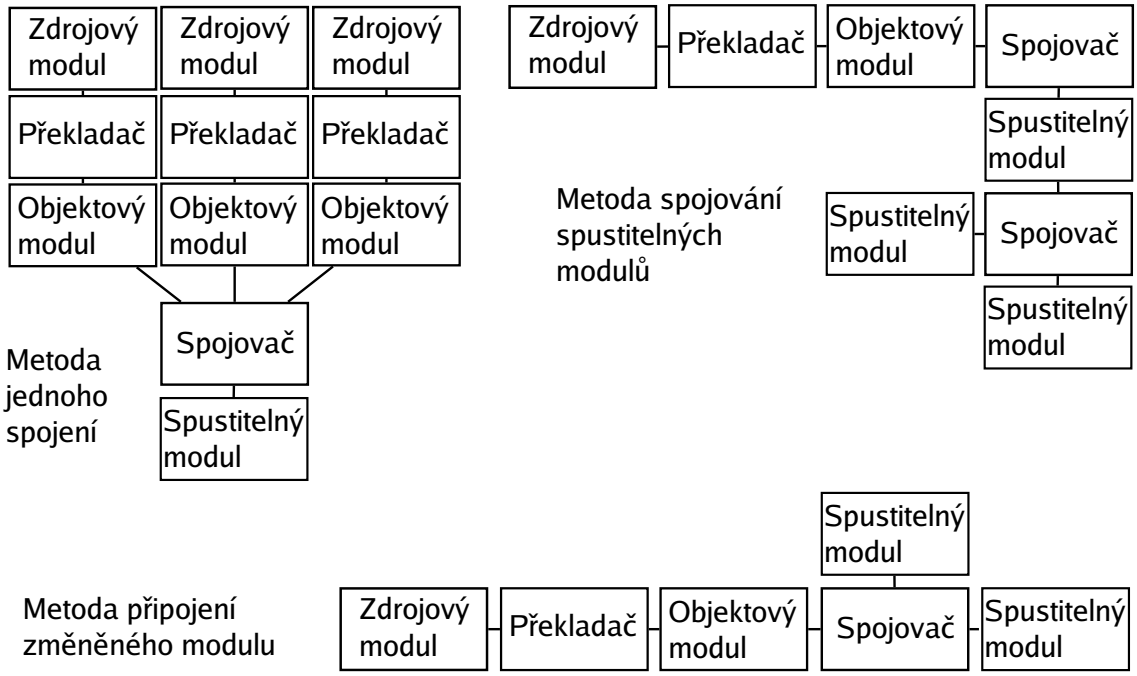
- MYLIB opět musíme otevřít knihovnu, ve které je procedura CBCCBG.
- COMPPRC spustíme proceduru CBCCBG, ta nám přeloží, spojí a spustí program psaný v C++.
- CPARM nastavíme parametry překladače. Parametry budou přiřazeny ze souboru OPTFILE.
- INFILE nastavení vstupního souboru.
- COMPILE.SYSLIB nastavuje cestu ke standardním hlavičkám.
- COMPILE.OPTFILE obsahuje námi nastavené parametry překladače.
- LSEARCH říká překladači, kde má hledat naše hlavičkové soubory.

1.5.3 Jak spojovat

Spojit program můžeme několika cestami. Přehledně to ukazuje obrázek 6. První metoda přímého spojení, je klasickou metodou, kterou známe i z PC. Protože na PC není podobný nástroj jako je spojovač musí se pouze linkovat. To znamená, že každé kompilování na PC probíhá první cestou. Nejdříve se vytvoří jednotlivé objektové moduly a pak se dohromady slinkují. Když se změní jeden objektový modul musí se opět linkovat s ostatními nezměněnými moduly. Na mainframe je tato cesta jen jedna z možností.

Spojovač totiž dokáže spojovat i jednotlivé spustitelné moduly. To ukazuje metoda spojování spustitelných modulů. Tyto moduly fakticky ještě spustitelné nejsou, protože nemusejí mít vyřešeny všechny odkazy.

To, že spojovač dokáže spojit i spustitelné moduly, otevřelo cestu k poslední metodě. Tou je metoda připojení změněného spustitelného modulu. Ta šetří při vývoji nejvíce času. Když při vývoji dojde ke změně jednoho zdrojového modulu, přeložíme ho a spojíme. Tento modul pak spojíme s původním spustitelným modulem starší verze. Spojovač si dokáže vyhledat část kódu starší verze a tu změnit za verzi novou.



Obrázek 6: Metody spojování

1.5.4 Složitější příklad

Na závěr si uvedeme složitější příklad na překlad a následné spojení. Bude to program skládající se ze dvou zdrojových modulů a jedné hlavičky. Hlavička obsahuje definici jednoduché třídy, jejíž datové složky jsou dvě proměnné typu `int`. Třída dále obsahuje sérii metod, které provádějí s datovými složkami aritmetické operace.

Ve vedlejším souboru (VLARA80.BP.SOURCES(OPERATE)) pak implementujeme jednotlivé metody vytvořené třídy.

```
VLARA80.BP.HEADERS(OPERATE)

class Operations {
private:
   int x;
   int y;
public:
   Trida(int a, int b) : x(a),y(b) {}
   void setXY(int, int);
   int add();
   int subtract();
   int multiply();
};
```

```
VLARA80.BP.SOURCES(OPERATE)

#include "operate.h"
void Operations::setXY(int a, int b) {
   x=a;
   y=b;
}
int Operations::add() {
  return x+y;
}
int Operations::subtract() {
  return x-y;
}
int Operations::multiply() {
  return x*y;
}
```

Druhý zdrojový modul obsahuje funkci `main()`. Do tohoto modulu vložíme hlavičku `operate.h`. Modul si vytvoří instanci naší třídy a použije její metody. Nejdříve vytvoří instanci s hodnotami 4 a 5 a vytiskne jejich součin. Po té přenastaví datové složky na 3 a 2 a vytiskne jejich rozdíl.

```
VLARA80.BP.SOURCES(THIRD)

#include <stdio.h>
#include "trida.h"
int main() {
   Operations op(4,5);
   printf("Vysledek je: %d\n", op.mulitply());
```

```
   t.setXY(3,2);
   printf("Vysledek je: %d\n", op.subtract());
}
```

Zdrojový kód je celkem zřejmý, přejděme ale k úkolu, kterým toto přeložíme.

```
//VLARA80A JOB (90300000),'VLARA80',NOTIFY=&SYSUID,
//          MSGCLASS=H,CLASS=A,MSGLEVEL=(1,1),REGION=0M
//MYLIB JCLLIB ORDER=(CBC.SCCNPRC)
//*
//COMPCL   EXEC PROC=CBCCB,
// CPARM='OPTFILE(DD:OPTFILE)',
// INFILE='VLARA80.BP.SOURCES(OPERATE)',
// OUTFILE='VLARA80.BP.LOAD(OPERATE),DISP=SHR'
//COMPILE.SYSLIB DD DSN=CEE.SCEEH.H,DISP=SHR
//COMPILE.OPTFILE DD *
   LSEARCH('VLARA80.BP.HEADERS')
/*
//COMPM   EXEC PROC=CBCCB,
// CPARM='OPTFILE(DD:OPTFILE)',
// INFILE='VLARA80.BP.SOURCES(THIRD)',
// OUTFILE='VLARA80.BP.LOAD(THIRD),DISP=SHR'
//COMPILE.SYSLIB DD DSN=CEE.SCEEH.H,DISP=SHR
//COMPILE.OPTFILE DD *
   LSEARCH('VLARA80.BP.HEADERS')
/*
//BIND.SYSIN  DD *
  INCLUDE INOBJ(OPERATE)
/*
//BIND.INOBJ DD DSN=VLARA80.BP.LOAD,DISP=SHR
//
```

- MYLIB nečte knihovnu s procedurou CBCCB, kterou budeme používat.
- COMPCL v tomto kroku přeložíme a spojíme pomocí procedury CBCCB zdrojový kód naší třídy.
- COMPILE.OPTFILE musí nastavit cestu k hlavičce, kterou vkládáme.
- COMPM toto je další krok úkolu. Spouští opět proceduru CBCCB, která přeloží a spojí hlavní zdrojový modul.
- COMPILE.OPTFILE opět nastavuje cestu k hlavičce, která je vložená i v tomto zdrojovém modulu.
- BIND.SYSIN zde můžeme poslat příkazy spojovači.

- INCLUDE toto je příkaz spojovači, aby spojil hlavní modul s dalším modulem, který je členem knihovny INOBJ.
- BIND.INOBJ přiřadí DDNAME INOBJ datovou sadu, do které jsme si uložili vytvořený modul s třídou.

2 Sokety

2.1 Úvod

Sokety umožňují komunikaci mezi dvěma procesy a to jak na lokální úrovni, tak přes internet. Proto bychom měli na úvod říci něco o počítačových sítích a vysvětlit si pojmy, které budeme dále používat. Budeme používat TCP/IP sokety, mluvme proto o těchto protokolech TCP/IP. TCP/IP je navržen k přenášení dat takzvanými paketově přepínanými sítěmi. Paket je blok dat, který si sebou nese informace, kam má být doručen. Paketově přepínaná síť pak čte tyto informace a postupně přeposílá paket blíže k místu dodání. Termíny paket a datagram jsou často zaměňovány, tyto termíny se ale liší. Paket je jakýkoliv blok dat, který obsahuje informace o místě doručení. Datagram má pak konkrétní strukturu definovanou v protokolu IP. V současné době jsou používány dvě verze protokolu IP, jsou označovány IPv4 a IPv6. Datagramy mají v těchto verzích jinak definovanou hlavičku. IPv6 se snaží řešit dnešní problém nedostatku IP adres, v IPv4 jsou na adresu pouze 4 byty, kdežto v IPv6 je to bytů 16. Rozdíl není samozřejmě jen v množství adres, i když to byl jeden z hlavních důvodů zavedení nového protokolu, ale i v bezpečnosti, rychlosti atd.

2.2 Architektura TCP/IP

Protokol je množina pravidel, které určují syntaxi a význam jednotlivých zpráv při komunikaci. Protokolová architektura TCP/IP je pak sadou protokolů pro komunikaci v počítačové síti. Vzhledem k velké složitosti a komplexnosti problému je síťové komunikace rozdělena do několika vrstev. Jednotlivé vrstvy mezi sebou spolupracují. Každá vrstva pak využívá služeb vrstvy nižší a naopak poskytuje své služby vrstvě vyšší.

Komunikační protokol definuje komunikaci mezi stejnými vrstvami dvou různých počítačů. Spojení mezi těmito vrstvami je vytvořeno sousední nižší vrstvou. Použitím modelu s několika vrstvami dosáhneme toho, že výměna protokolů jedné vrstvy nemá dopad na ostatní. Příkladem je jistě komunikace pomocí různých síťových architektur - ethernet, token ring, sériová linka.

2.2.1 Popis vrstev

Architektura TCP/IP je rozdělena do čtyř vrstev.

1. **Vrstva síťového rozhraní** (*Network interface layer*)
   Je to nejnižší vrstva, která umožňuje přístup přímo k fyzickému médiu (např. síťová karta). Její konkrétní implementace závisí na fyzické realizaci sítě.

2. **Síťová vrstva** (*Network layer*)
   Zajišťuje především síťovou adresaci, předávání a směrování datagramů. Tato vrstva je implementována jak ve směrovacích tak v koncových prvcích sítě.

3. **Transportní vrstva** (*Trasnport layer*)
   Tato vrstva je implementována až v koncových zařízeních a umožňuje tak při-

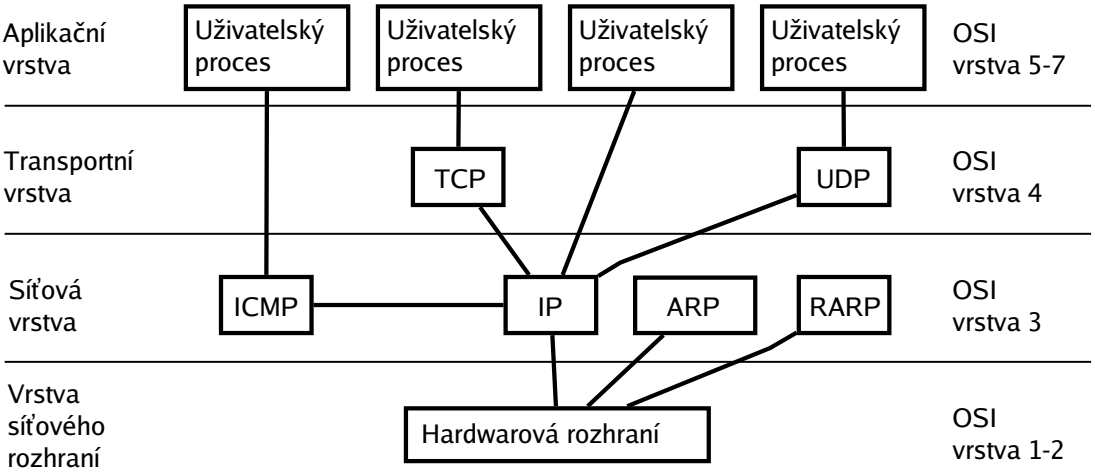
způsobit chování sítě potřebám aplikace. Poskytuje spojované či nespojované transportní služby (viz. protokoly TCP, UDP).

4. **Aplikační vrstva** (*Application layer*)
   To už jsou konkrétní aplikace, které využívají datových přenosů ke konkrétním službám pro uživatele. Např. FTP, HTTP, DNS.

2.2.2 Protokoly

Každá vrstva nabízí určité protokoly, přehledně to zobrazuje obrázek 7.

- **IP** (*Internet Protocol*) je to základní protokol, který provádí posílání datagramů na základě IP adres. Není zaručeno, že datagram dojde tak, jak byl poslán, nebo zda vůbec dojde. To už musí zaručit protokoly z vyšších vrstev.
- **ICMP** (*Internet Control Message Protocol*) se používá k přenosu řídících hlášení, které se týkají chybových stavů a zvláštních okolností při přenosu dat. Používá se například při zjištění dostupnosti počítače pomocí příkazu `ping`.
- **ARP** (*Address Resolution Procol*) se používá k mapování IP adres na adresy hardwarové. V lokálních sítích se hardwarové adresy nazývají MAC adresy (*Media Access Control address*).
- **RARP** (*Reverse Address Resolution Procol*) je přesným opakem ARP. Tento protokol přiřadí MAC adrese konkrétní IP adresu.
- **TCP** (*Transport Control Protocol*) je to transportní protokol, který poskytuje plně duplexní a spolehlivý přenos dat. Poskytuje spojované služby. To znamená, že před začátkem přenosu aplikačních dat se dvě strany komunikace nejdříve domluví - vytvoří spojení. Komunikace je také pomalejší proto, protože se ověřuje, jestli došly datagramy tak, jak byly poslány.
- **UDP** (*User Datagram Protocol*) je transportní protokol, který poskytuje nespolehlivé spojení, pro aplikace, které spolehlivé spojení nepotřebují. Nemá fázi navazování ani ukončování spojení a už první segment UDP obsahuje aplikační data. Tento protokol poskytuje nespojované služby. Spojení se nevytváří a data neověřují, komunikace je proto rychlejší. Například u internetové televize celkem nevadí, když datagram nedojde a obraz přeskočí. Naopak je vyžadována rychlost a proto je UDP vhodnou volbou.
- **HTTP** (*HyperText Transfer Protocol*) toto je jeden z protokolů aplikační vrstvy definující způsob přenosu webových stránek.
- **FTP** (*File Transfer Protocol*) další z protokolů aplikační vrstvy, který je navržen pro přenos souborů přes internet.
- **DNS** (*Domain Name Service*) poslední z protokolů aplikační vrstvy, který si uvedeme. DNS slouží k překladu doménových jmen (adresy typu www.aabb.cz) na IP adresy.



Obrázek 7: Architektura TCP/IP

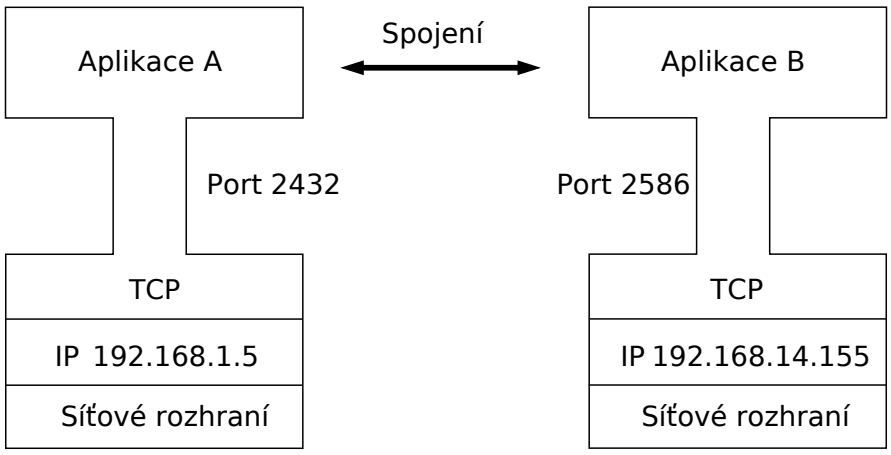
2.3 Charakteristika soketů

Samotných soketů je velké množství druhů. Jejich společnou vlastností je, že nám umožňují pomocí file deskriptorů, komunikovat s ostatními programy. Protože my budeme komunikovat výhradně po síti, budeme se zabývat jedním druhem soketů nazývající se Internetové sokety (*Internet Sockets*). Internetový soket je pak uspořádaná trojice (protokol, IP adresa, port), která jednoznačně identifikuje konce spojení mezi dvěma aplikačními porty.

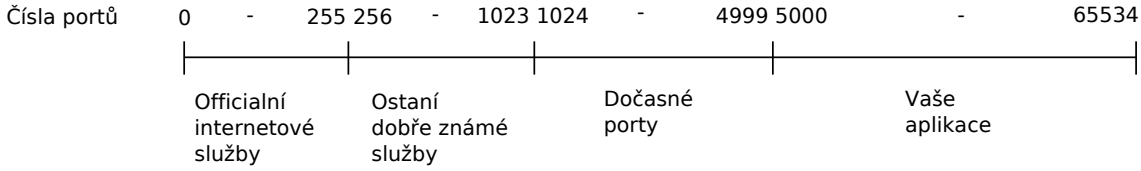
Port reprezentuje aplikaci na TCP/IP hostiteli. Samotná hodnota ale neurčuje, jaký protokol je použit. Existuje však dohoda, která vyhrazuje určitá čísla portů pro konkrétní služby a jejich protokoly z aplikační vrstvy. Je to například port 80, který je vyhrazen pro HTTP, nebo 21, který je vyhrazen pro FTP. Služby, které jsou takto přiřazeny k číslům portů, mohou využívat jak TCP tak i UDP, to není dohodou určeno. Konkrétní implementace těchto služeb však mohou jednu z možností vylučovat. Například není možné, aby bylo SSH implementováno přes UDP. Jak je vidět na obrázku 9, čísla do 1023 jsou vyhrazena pro oficiální služby. Čísla od 1024 do 4999 jsou vyhrazena pro dočasná spojení. Klient sám většinou nepotřebuje vědět jaký mu byl přidělen port. Čísla těchto portů jsou pak generována automaticky a přidělována klientovi při vytváření spojení. Čísla nad 5000 už zůstávají většinou volná a to je místo pro naše aplikace.
Internetové sokety se rozdělují podle protokolu, který využívají, na:

- Stream – využívá TCP
- Datagram – využívá UDP
- Raw – používá protokoly ze síťové vrstvy

Jejich vlastnosti se pak odvíjejí právě od protokolu, který využívají. *Stream soket* je spojovaný a zajišťuje, že data, která byla poslána, přijdou k příjemci všechna



Obrázek 8: Koncept soketů



Obrázek 9: Rozložení portů

a ve stejném pořadí, v jakém byla poslána. *Datagram soket* pak spojovaný není a nezajišťuje ani doručení dat. *Raw soket* také není spojovaný a umožňuje přímý přístup k síťové vrstvě, můžeme tedy poslat námi vytvořený ICMP paket, kde sami můžeme vyplnit například příjemce i odesilatele.

Pro naši aplikaci jsou samozřejmě nejvýhodnější *Stream sokety*. Nebudeme se muset starat o kontrolu, zda přišlo vše a v jakém pořadí, což značně zjednoduší náš kód.

2.4 Identifikační struktury

Na mainframe jsou k dispozici dvě knihovny s implementovanými sokety, Native TCP/IP (implementovaná v TCP/IP v z/OS Communications Server) a UNIX (implementována v z/OS UNIX System Services). Vzhledem k tomu, že Native TCP/IP nesplňuje standardy POSIX, není doporučováno vyvíjet nové programy s touto knihovnou. Použijeme tedy sokety z knihovny UNIX. Z toho, že splňuje tyto standardy, je zřejmé, že programy psané v C/C++ na mainframe, budou velmi podobné s těmi, co bychom psali pro ostatní systémy z rodiny UNIX.

2.4.1 Obecná struktura adresy

Jak tedy adresujeme soket? Obecná struktura pro adresaci má následující schéma:

```
struct sockaddr {
    unigned short   sa_family;
    char            sa_data[14];
}
```

Vzhledem k tomu, že `unsigned short` jsou 2 byty a `char` je 1 byte, můžeme si snadno spočítat, že celá struktura zabírá 16 bytů. V proměnné `sa_family` je uložena informace o rodině adres. Pole `sa_data` je pak místo vyhrazené pro další data potřebná k adresaci, jejichž tvar je však závislý na adresové rodině.

2.4.2 Rodiny adres

Rodiny adres nám definují různé styly adresování. Všichni členové jedné rodiny pak používají stejné schéma adresování koncových bodů soketů. Dnešní TCP/IP podporuje dva druhy adresových rodin `AF_INET` a `AF_INET6`. Již z názvů je trochu vidět, že `AF_INET` znamená protokol IPv4, `AF_INET6` pak IPv6.

2.4.3 Adresa v rodině `AF_INET`

Pro použití v rodině `AF_INET` je vytvořena specializovaná struktura `sockaddr_in`, na GNU/Linuxu ji lze najít v hlavičce `netinet/in.h`, na mainframe pak v `in.h`.

```
struct sockaddr_in {
    short int       sin_family;
    usigned short   sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];

}
```

V této struktuře je 14 bytové pole `sa_data` rozděleno na `sin_port`, `sin_addr` a menší pole `sin_zero`. Můžeme si ověřit, že díky poli `sin_zero` má struktura opět velikost 16 bytů. K plnému pochopení si musíme, ale ukázat jak vypadá struktura `in_addr`.

```
struct in_addr {
    uint32_t s_addr;
}
```

V této struktuře je uložena IP adresa. Použitý typ `uint32_t`, je vlastně pouze `unsigned int`, který má ale délku 32 bitů tedy 4 byty[1]. To nám přesně vychází pro použití 4 bytové IP adresy adresové rodiny `AF_INET`.

[1]Musíme explicitně vypsat 32 bitovou délku, jelikož standard jazyka C/C++ nezajišťuje `int` o velikosti 4 byty.

2.4.4 Bytový pořádek

S touto problematikou identifikačních struktur úzce souvisí i problematika bytového pořádku čísel. Ne na všech architekturách jsou totiž čísla ukládána stejně. Rozdílný je pořádek bytů u více bytových číselných typů. V praxi jsou nejrozšířenější dva druhy:

- **Little-Endian** – s rostoucí adresou roste i číselná významnost obsahu. U `int` to znamená, že jeho 4 byty jsou uspořádány tak, že jednotky jsou v prvním a miliardy v posledním. Tento typ používají hlavně procesory s architekturou x86.
- **Big-Endian** – zde je to přesně naopak, tedy nejvíce číselně významné byty jsou první, s rostoucí adresou pak klesá číselný význam obsahu. Tento typ používají například procesory Motorola, PowerPC, ale i mainframe řady S/390 nebo zSeries.

Budeme-li tedy chtít posílat čísla po síti mezi dvěma stanicemi, s neznámými pořádky bytů, budeme je muset nejdříve upravit, aby měly námi dohodnutý bytový pořádek. Standardem je ale tzv. síťový bytový pořádek (*Network byte order*). Za síťový bytový pořádek byl zvolen Big-Endian.

Musíme tedy používat funkce, které nám zajistí převedení čísla z bytového pořádku našeho počítače na síťový a zpět. Tyto funkce se na GNU/Linuxu nalézají v hlavičce `arpa/inet.h`, na mainframe v `in.h`. `Short` je typ o dvou bytech `long` má byty čtyři. Musíme proto při převodu dávat pozor jakou funkci pro jaký typ používáme.

`htons()`	**h**ost **to** **n**etwork **s**hort
`htonl()`	**h**ost **to** **n**etwork **l**ong
`ntohs()`	**n**etwork **to** **h**ost **s**hort
`ntohl()`	**n**etwork **to** **h**ost **l**ong

Tabulka 1: Funkce na převod pořadí bytů

2.4.5 Funkce pro práci s IP

Dalšími funkcemi, které nám usnadní práci s identifikací, jsou funkce, které dokáží pracovat s IP adresami. Uvedeme si jen ty nejdůležitější. Tyto funkce se nalézají na GNU/Linuxu v hlavičce `arpa/inet.h`, na mainframe v `inet.h`.

`inet_addr()`	IP **addr**ess conversion
`inet_aton()`	**a**scii **to** **n**etwork
`inet_ntoa()`	**n**etwork **to** **a**scii

Tabulka 2: Funkce na převod IP adres

Řekněme, že máme `struct sockaddr_in ina`, pak použití funkce `inet_addr()` bude následující.

```
ina.sin_addr.s_addr = inet_addr("10.11.12.13");
```

Tato funkce již vrací převedené číslo v síťovém pořadí bytů. Nemusíme tedy vrácené číslo ještě převádět pomocí `htonl()`. Nevýhodou této funkce ale je, že při chybě převodu vrací -1. To se při vložení do typu `uint32_t`, který je `unsigned` (bez znaménka), přemění na číslo největší možné. Takové číslo pak odpovídá maximální IP adrese 255.255.255.255, která má smysl, je to broadcast. Začala se proto používat funkce `inet_aton()`. Při úspěchu vrací nenulovou hodnotu, při neúspěchu vrací nulu. Vzhledem k tomu, že struktura, kterou chceme vyplnit, se předává jako parametr, nemůže se stát, že by se chybové hlášení překrývalo s nějakou IP adresou. Tato funkce ale zatím není implementována na všech platformách. Takovou platformou je i mainframe. Budeme proto muset použít starší funkci `inet_addr()`. Prototyp funkce `inet_aton` vypadá:

```
int inet_aton(const char *cp, struct in_addr *inp);
```

Uvedeme si ještě příklad na celkové vyplnění struktury `sockaddr_in`.

```
struct sockaddr_in ina;

ina.sin_family = AF_INET;          // lokální bytový pořádek
ina.sin_port = htons(6789);        // převedeme na síťový bytový pořádek
// inet_aton("10.11.12.13", &(ina.sin_addr));
ina.sin_addr.s_addr = inet_addr("10.11.12.13");
memset(ina.sin_zero, '\0', sizeof ina.sin_zero);
```

Proměnná `sin_family` je vyplněna v host byte order. Tato proměnná totiž obsahuje pouze informaci pro náš operační systém. Z bezpečnostních důvodů je doporučováno nastavit pole `sin_zero` na samé nuly pomocí funkce `memset()`, jak ukazuje příklad.

Jako poslední nám zůstává funkce `inet_ntoa()`. Tato funkce, jak už ze jména plyne, převádí IP adresu z čísla na text, který má klasickou strukturu čtyř čísel oddělených tečkami.

```
char* inet_ntoa(struct in_addr in);
```

Tato funkce vrací pointer na `char`. Pole, na které však tato funkce vrací pointer, je staticky uloženo ve funkci a proto pokaždé, když tuto funkci zavoláte, je pole přepsáno novou IP adresou.

```
char *a1, *a2;

a1 = inet_ntoa(ina1.sin_addr);  // tato je 10.11.12.13
a2 = inet_ntoa(ina2.sin_addr);  // tato je 10.11.12.14
printf("adresa 1: %s\n",a1);
printf("adresa 2: %s\n",a2);
```

Tento příklad vytiskne:

```
address 1: 10.11.12.14
address 2: 10.11.12.14
```

Proto, když chceme převádět více IP adres, musíme si nejdříve pomocí funkce `strcpy()` zkopírovat text do našeho vlastního pole.

2.5 C/C++ Soket API

V této část se zaměříme na výčet a popis nejdůležitějších funkcí, které poskytuje C/C++ Soket API (*Application Programming interface*) a které budeme potřebovat v naší aplikaci. Začneme funkcemi potřebnými pro vytvoření serveru, který bude naslouchat na námi zvoleném portu. Potom si ukážeme jaké funkce potřebuje klient, aby se k takovému serveru mohl připojit.

2.5.1 `socket()`

První co musíme udělat, je alokovat soket pomocí funkce `socket()`. Tato funkce vrací deskriptor k novému soketu. Když vytváříme více soketů, funkce zajistí, aby tato čísla byla pro jeden proces unikátní. Při proběhnutí bez chyb je to tedy kladné číslo větší než dvě. Nula, jedna a dva jsou totiž rezervovány pro standardní vstup, výstup a pro chybový výstup. Její prototyp vypadá:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol)
```

V případě nějaké chyby vrací funkce číslo záporné a nastavuje proměnnou `errno` na hodnoty uvedené v tabulce 8 v příloze A.

V našem případě komunikace po síti pomocí protokolu TCP budeme funkci `socket()` volat s následujícími parametry:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0)
```

2.5.2 `bind()`

Další funkcí, kterou si uvedeme, je funkce `bind()`. Tato funkce se používá pro přiřazení soketu ke konkrétnímu portu a konkrétnímu síťovému rozhraní.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

Proměnná `sockfd` je file deskriptor soketu, který chceme použít pro `bind()`, `my_addr` je struktura, o které jsme psali v minulé kapitole, `addrlen` je velikost datového typu `struct sockaddr`. Vše si shrneme v krátkém příkladu. Nesmíme ale zapomenout na kontrolu, zda volání funkce proběhlo bez chyb. Tuto kontrolu příklad neobsahuje[2].

```
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(7777);
my_addr.sin_addr.s_addr = inet_addr("10.11.12.13");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
```

Ve volání se přetypovává ze `sockaddr_in*` na `sockaddr*`, že je toto přetypování korektní jsme si uvedli v minulé kapitole. Když proběhne funkce bez chyby vrátí 0, jinak -1 a nastaví proměnnou `errno` na hodnoty uvedené v tabulce 9 v příloze A. Musíme mít na paměti to, co ukazuje obrázek 9. Čísla portů nižší jak 1024 jsou rezervována. Volat funkci `bind()` s těmito porty může jen proces se superuživatelskými právy.

```
my_addr.sin_addr.s_addr = INADDR_ANY;
```

Když chceme připojení soketu na síťový interface automatizovat, stačí vložit místo IP adresy konstantu `INADDR_ANY`. Tím zajistíme připojení na všechna dostupná rozhraní. Funkce `bind()` už si příslušná rozhraní a jejich IP adresy vyhledá sama.

2.5.3 `listen()`

Když už jsme se připojili k portu a rozhraní, musíme operačnímu systému říci, aby na tomto portu přijímal požadavky na připojení. K tomu nám slouží funkce `listen()`.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Parametr `sockfd` je opět deskriptor soketu, `backlog` udává délku fronty, do které se budou ukládat informace o požadavcích na spojeních než budou zpracovány. U běžných serverů se tato fronta nastavuje na délku asi 20 požadavků. Funkce v případě úspěchu vrací 0, jinak -1 a nastaví `errno`. Přesný popis chybových konstant opět najdete v příloze A v tabulce 10.

[2]To platí pro všechny příklady v této kapitole, ošetření chyby už je jen zkontrolování vrácené hodnoty, což by příklady jen znepřehledňovalo

Pořadí volání těchto funkcí v programu musí být takové, jaké je zde uvedeno. Je jasné, že bez soketu nemůžeme volat `bind()`. Kdybychom ale volali `listen()` před `bind()`, naslouchal by náš server na náhodném portu, který byl jádrem procesu přidělen. Server, který naslouchá při každém spuštění na jiném portu, je víceméně nepoužitelný.

2.5.4 `accept()`

K tomu abychom přijali klienta, který se k serveru snaží připojit, slouží funkce `accept()`.

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Tato funkce vrací číslo nového deskriptoru, na kterém můžeme komunikovat s přijatým klientem. Ten původní předaný funkci v parametru `sockfd` stále existuje a naslouchá novým připojením. Funkci musíme ještě předat adresu námi vytvořené struktury `sockaddr`, kam nám funkce uloží informace o přijatém klientovi. Použití funkce `accept()`:

```
int sockfd, new_fd, size;
struct sockaddr_in their_addr;

size = sizeof(struct sockaddr_in);
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &size);
```

Když se klienta nezdaří přijmout vrátí funkce -1 a nastaví `errno`. Popis některých hodnot opět naleznete v příloze A v tabulce 11.

Teď jsme si vyjmenovali všechny funkce potřebné pro vytvoření serveru, který bude naslouchat na námi zvoleném portu. Pro klienta, který se bude na takový server připojovat, musíme postupovat trochu odlišně. Vytvoření soketu je stejné a nutné jak pro klient tak pro server. Klient už ale nemusí použít funkci `bind()`, číslo portu se přiřadí automaticky. O jeho hodnotě se dočtete v minulé kapitole. Ani volání funkce `listen()` a `accept()` není potřebné, ke klientovi se totiž nikdo připojovat nebude.

2.5.5 `connect()`

Připojení na server se provede voláním jediné funkce:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int  addrlen);
```

Parametr `sockfd` je deskriptor soketu, který si klient vytvořil, `*serv_addr` je pointer na strukturu obsahující adresu serveru. V parametru `addrlen` se pak předává délka této adresy. Je-li spojení navázáno, vrací funkce 0, jinak -1 a opět nastavuje `errno` na hodnoty v tabulce 12, příloha A.

2.5.6 `send()`

Když už máme vytvořené spojení, chceme posílat data. K tomu nám slouží funkce `send()`.

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int sockfd, const void *msg, int len, unsigned int flags);
```

Kde `sockfd` je opět deskriptor. Ukazatel `*msg` obsahuje adresu začátku dat v paměti, která budeme posílat. Parametr `len` znamená kolik bytů se pošle. Poslední parametr `flags` je pro nás nepotřebný a budeme ho nastavovat na nulu. Funkce vrací počet opravdu odeslaných bytů nebo -1 a nastaví hodnotu `errno`. Její hodnoty opět najdete v příloze A, tabulka 13.

2.5.7 `recv()`

Tato funkce je naopak určena pro příjem dat.

```
#include <sys/types.h>
#include <sys/socket.h>

int recv(int sockfd, void *buf, int len, int flags);
```

Parametry jsou shodné s předchozí funkcí. Funkce vyčte ze soketu `len` bytů dat a uloží je ve stejném pořadí do paměti začínající na adrese `buf`. Funkce vrací počet opravdu přečtených bytů, 0 když je spojení na druhé straně ukončeno a -1 při chybě. Samozřejmě opět nastaví proměnnou `errno`. Některé hodnoty najdete v tabulce 14, v příloze A. Za zmínku ještě stojí některé hodnoty parametru `flags`. Když funkci předáme 0, funkce bude čekat, dokud nepřijdou alespoň nějaká data a pak je vyčte. Předání konstanty `MSG_PEEK` zapříčiní, že data budou uložena do paměti na námi zadané místo, nebudou ale smazána z fronty čekající na vyčtení. To nám umožní doručená data zkontrolovat před kompletním přečtením. Další a poslední konstanty o kterých se zmíníme jsou `MSG_WAITALL` a `MSG_DONTWAIT`. Ty mění vlastnosti čekání funkce `recv()`. První zapříčiní, že funkce bude čekat, dokud nepřijde počet bytů v `len`, nebo není spojení ukončeno. Druhá pak zapříčiní, že se nebude čekat vůbec. Přečte se co, je doposud ve frontě. Když je fronta prázdná je vrácena -1 a `errno` je nastavena na `EAGAIN`.

2.5.8 `close()`

Poslední funkce, na kterou rozhodně nesmíme zapomínat, je funkce na uzavření soketu. To umožní uvolnit systémové prostředky, které jsou pro jednotlivé sokety vyhrazeny. Ať už je to hodnota v tabulce deskriptorů, nebo jsou to buffery na dočasné ukládání příchozích dat čekajících na vyčtení.

```
#include <unistd.h>

int close(int fd);
```

Použití této funkce je poměrně přímočaré. Stačí ji zavolat s jediným parametrem a tím je deskriptor. Funkce si již sama rozhodne, jaký druh soketu byl vytvořen, a podle toho vše uvolní. Vrací 0 při úspěchu a -1 při chybě.

Příklad, který shrne všechny námi zde popsané funkce a názorně ukáže návrh serveru a klienta, najdete v příloze B.

3 Popis návrhu naší aplikace

V této části si detailně představíme projekt, jehož vývoj je cílem nejen této bakalářské práce, ale zároveň i úkolem do dalších let. Nejdříve si popíšeme jednotlivé části, následně pak vysvětlíme, jak dohromady fungují. Celá aplikace je rozdělena do tří celků.

3.1 Knihovna zpráv

První částí je knihovna zpráv. Knihovna definuje pravidla komunikace. Je proto ve stejné podobě jak na straně serveru, tak na straně klienta. Skládá se z těchto souborů:

`protocol.h`
`input.h, .cpp`
`outputSocket.h`
`message.h, .cpp`
`messenger.h, .cpp`

Tabulka 3: Soubory knihovny zpráv

3.1.1 `protocol.h`

Tento soubor obsahuje základní datové typy a konstanty definující pravidla komunikace mezi klientem a serverem. Je to hlavně definice hlavičky každé zprávy:

```
typedef long Command;

struct Header {
    Command cmd;
    long dataSize;
    long lineSize;
};
```

Komunikace pak probíhá následovně. Klient se připojí na server pomocí námi popsaných soketových funkcí. Vyplní tuto strukturu, pošle ji a za ní data. Server si vyčte ze soketu tuto hlavičku. Z ní pak jednak zjistí kolik dat za hlavičkou je a pak také co s nimi má provést. Parametr `lineSize` je víceméně informativní. Je plánováno, že při přenosu souborů bude udávat délku řádky. Na mainframe mají totiž soubory často pevnou délku řádky. Tím bychom mohli optimalizovat jejich přenos.

Je vidět, že abychom hlavičku mohli odeslat, musíme vědět, jak dlouhou zprávu za ní posíláme. To je nevýhoda tohoto způsobu komunikace. Další možností by bylo posílání dat po blocích s konstantní délkou, přičemž bychom četli tak dlouho, dokud bychom nenarazili na blok označený jako poslední. Výhoda tohoto způsobu je, že

dopředu nemusíme znát délku zprávy, nevýhoda pak, že musíme kontrolovat každý blok, jestli už náhodou není poslední. Tento protokol je i složitější na implementaci. Proto byl prozatím zvolen druh první. Když se však ukáže potřeba znalosti délky zprávy dopředu jako velký problém, půjde snadno způsob komunikace změnit.

Dále jsou v této hlavičce definovány číselné konstanty, které budeme vkládat do proměnné `cmd`. Přiřadíme tak jednotlivým číslům význam. Zatím máme nadefinovány tyto konstanty:

- `CMD_EMPTY 0` pomocná hodnota
- `CMD_LOGIN 1` příkaz pro zalogování k serveru, data za hlavičkou obsahují uživatelské jméno a heslo
- `CMD_CLOSE_CONNECTION 2` příkaz pro uzavření spojení
- `CMD_ECHO 3` testovací příkaz echo, server odpoví tím, co jsme mu poslali

3.1.2 `message.h, .cpp`

Je jasné, že načíst celou zprávu do paměti i s daty, a pak ji předávat ke zpracování, není vhodné řešení. Bylo proto potřeba vytvořit třídu, která by v sobě zahrnovala nejenom kolik dat přišlo, kolik zbývá vyčíst, ale také odkud se vůbec tyto data mají číst. Předávat se pak bude instance této třídy a data tak nikdy nemusí být celá v paměti. Proto byla vytvořena třída `Message`. Její důležitá metoda a datové složky jsou:

```
class Message {
    Header hdr;
    InputObject* in;
    long dataRemain;

public:
    ...
    int dataRead(void* buf, int n);
};
```

V objektu `Message` je uložena hlavička příchozí zprávy, kolik dat ještě zbývá vyčíst a taky ukazatel na speciální objekt `InputObject`, v kterém je obsaženo odkud se má číst. Metoda `dataRead()` volá tu samou metodu objektu `InputObject` a odečítá množství vyčtených dat z proměnné `dataRemain`.

3.1.3 `input.h, .cpp`

Tyto dva soubory obsahují definici třídy, o které jsme se už zmínili. Je to třída `InputObject`. Třída je ryze virtuální. Definuje nám důležitou virtuální metodu `dataRead`, za pomoci níž budeme moci číst z různých míst jednou metodou.

```
class InputObject {
    ...
    virtual int dataRead(void* buf, int n)=0;
};
```

Od této třídy jsme odvodili třídy `InputMemory`,`InputFile` a `InputSocket`. Jak už je z jednotlivých jmen asi zřejmé, v odvozených třídách jsme vždy předefinovali metodu `dataRead()`, podle toho jestli čteme z paměti, souboru nebo soketu. Jediným ukazatelem na `InputObject` ve třídě `Message` a voláním metody `dataRead()`, tak dostáváme data z různých míst.

3.1.4 `outputSocket.h`

Tato třída je velice podobná těm ze souboru `input.h`. Je ale uzpůsobena ne pro načítání dat, ale k jejich odesílání. Vzhledem k tomu, že předávání informace kam se data mají zapsat, zatím nebylo potřeba, nebyla vytvořena stejná hierarchie s abstraktní třídou jako u vstupu.

3.1.5 `messenger.h, .cpp`

Poslední třída z knihovny zpráv je třída `Messenger`. Navržena je pro posílání a příjem zpráv soketem. S každým alokovaným soketem se alokuje i jeden objekt této třídy.

```
class Messenger {

    InputSocket* in;
    OutputSocket* out;

public:
    ...
    int sendMessage(Message& msg);
    int recvMessage(Message& msg);
};
```

Pomocí metod `sendMessage()` a `recvMessage()`, snadno zajistíme odeslání vytvořené zprávy. Zde také probíhá konverze číselných typů ve struktuře `Header` na síťové bytové uspořádání.

3.2 Klientská část

Klientská část se skládá z knihovny a z jednoho ukázkového klienta, který volá funkce této knihovny. Celá klientská strana je navrhována tak, aby mohla být pro různé systémy zaměněna pouze knihovna a implementace klienta zůstala stejná.

client.cpp
clientLib.h, .cpp

Tabulka 4: Soubory klientské části

3.2.1 Klientská knihovna

Zde jsou implementována volání pro komunikaci se serverem. Zatím jsou implementovány pouze základní funkce. Jsou to funkce pro vytvoření a uzavření spojení, přihlášení a funkce pro testování komunikace, která vrací to samé co, bylo posláno.

```
int createConnection(Messenger*& messng, char* ip, short port);
   int cmdLogin(Messenger* messng, char* user, char* pass);
          int cmdCloseConnection(Messenger* messng);
        Message cmdEcho(Messenger* messng, char* ch);
```

Tabulka 5: Funkce klienského API

- **createConnection()** funkce na základě předané IP adresy a portu vytvoří soket spojený se serverem. Dále je vytvořen objekt třídy `Messenger`. Ten, pak bude reprezentovat spojení v našem klientovi. Ukazatel na vytvořený objekt je uložen do předané proměnné `messng`. Funkce vrací 1 při úspěchu, 0 při chybě.
- **cmdLogin()** vezme předané jméno a heslo, vloží je do zprávy a pošle ji na server. Počká na odpověď serveru. Pak vrátí konstantu z hlavičky `protocol.h`, podle toho jestli byla autorizace úspěšná či nikoliv.
- **cmdClose()** dotáže se serveru, zda je možno ukončit spojení. Tím je vytvořena možnost pro zamítnutí ukončení spojení například kvůli otevřeným souborům. Ty mohly být změněny a je potřeba změny buď uložit, nebo je zrušit.

3.2.2 Jednoduchý klient

Je to ukázková implementace klienta, který pracuje na příkazové řádce. Nejdříve vytvoří spojení, poté se dotáže na přihlašovací jméno a heslo a zavolá knihovní funkci pro přihlášení. V cyklu pak čte ze standardního vstupu a posílá přečtená data na server s příkazem echo. Server vrátí stejný text a klient ho vypíše na obrazovku.

3.3 Serverová část

Serverová část je rozdělena na tři složky. První z nich je `serverThread`. Je to kostra serveru, obsahuje cyklus, který naslouchá na příslušném portu a zpracovává požadavky na spojení od klientů. Pro každé příchozí spojení vytvoří vlákno, které se už postará o jednotlivé klienty. Dále je to `cmdExecutor`. Tento objekt obsahuje pole

ukazatelů na jednotlivé funkce, které bude server vykonávat, a umožňuje volání příslušných funkcí na základě indexu v tomto poli. Poslední částí je pak samotná implementace funkcí zpracovávající jednotlivé příkazy od klienta.

`cmdExec.h, .cpp`
`cmdFuncs.h, .cpp`
`serverThread.cpp`

Tabulka 6: Soubory serverové části

3.3.1 cmdFuncs

Tyto soubory obsahují implementaci funkcí, které zpracovávají příchozí zprávy od klienta a vytvářejí na ně odpovědi. Všechny jejich prototypy odpovídají tvaru:

```
int cmd[Name](Message& recvMsg, Message& sentMsg)
```

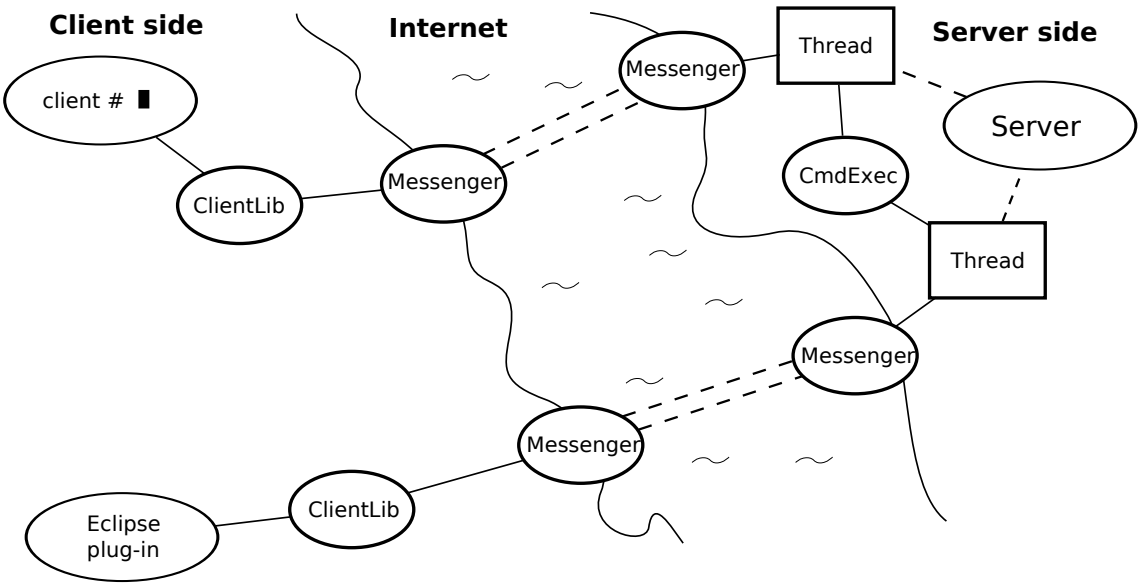
- Návratová hodnota je typu `int`, obsahuje informaci o tom jak zpracovávání zprávy dopadlo.
- Name odpovídá jménu příkazu, který funkce vykonává.
- `recvMsg` je zpráva přijatá serverem od klienta.
- `sentMsg` je zpráva připravená k odeslání obsahující výsledek požadavku klienta.

Prozatím jsou implementovány tyto funkce:

`cmdLogin(Message& recvMsg, Message& sentMsg)`
`cmdEcho(Message& recvMsg, Message& sentMsg)`
`cmdCloseConnection(Message& recvMsg, Message& sentMsg)`

Tabulka 7: Funkce zpracovávající požadavky klienta

- `cmdLogin()` vyčte z přijaté zprávy strukturu `Identification` (definovaná v `protocol.h`) a podle jejího obsahu se pokusí klienta přihlásit.
- `cmdEcho()` nedělá nic jiného než, že předá do zprávy k odeslání zprávu přijatou. Objekt `Messenger` pak při posílání vyčte celou přijatou zprávu a zároveň ji pošle zpět.
- `cmdCloseConnection()` zkontroluje, zda je možno ukončit spojení mezi serverem a klientem, aniž by došlo ke ztrátě dat. Například při otevřených a neuložených souborech. Odhlášení v odpovědi povolí nebo zamítne. Prozatím povoluje odhlášení vždy.



Obrázek 10: Struktura návrhu

3.3.2 cmdExecutor

Objekt slouží k rychlejšímu volání funkcí, které zpracovávají požadavky klienta. Obsahuje pole ukazatelů na tyto funkce. V poli jsou funkce uloženy podle konstant definovaných v `protocol.h`. Instance objektu je pro server vytvořena globálně a je pouze jedna. Jednotlivá vlákna pak přes tento objekt lehce přistupují k požadovaným funkcím. Mají k dispozici následující dvě metody

- `execCmd(Message& recvMsg, Message& sentMsg)` metoda si z předané zprávy `recvMsg` vyčte její hlavičku a rozhodne se jakou funkci pro zpracování má zavolat. Výsledek je pak vrácen v `sentMsg` a připraven k odeslání pomocí objektu `Messenger`.
- `execCmdLogin(Message& recvMsg, Message& sentMsg)` tato metoda zkontroluje zda je v `recvMsg` zpráva požadující přihlášení tzn. volání funkce `cmdLogin()` a když ano zavolá ji. Funkce `cmdLogin()` je totiž zvláštní tím, že je volána pouze jednou a to vždy těsně po vytvoření spojení. Proto pro ni byla vytvořena speciální metoda. Vlákno pak přijme první zprávu a předá ji této metodě. Tím je zajištěna kontrola, že první zpráva od klienta bude přihlašovací.

3.3.3 serverThread

Jak již jméno napovídá, server je naprogramován tak, aby v cyklu naslouchal příchozím spojením a pro každé vytvořil vlákno. To pak zpracovává jednotlivé požadavky klienta. Server obsahuje jednu globální instanci objektu `cmdExecutor`. Ta při inicializaci serveru naplní pole s ukazateli na jednotlivé funkce příkazů. Když se vlákno spustí, počká na první zprávu a zavolá metodu `cmdExecutor::execCmdLogin()`.

Tím se provede autorizace klienta. Pak už se v cyklu přijímají zprávy od klienta, zpracovávají se voláním metody `cmdExecutor::execCmd()` a posílají se zpět odpovědi, dokud není ukončeno spojení.

3.4 Shrnutí

To, jak vše funguje, přehledně shrnuje obrázek 10. Vidíme zde dvě strany klientskou a serverovou spojenou pomocí soketů přes internet. První co je z obrázku vidět, že klient může být reprezentován například programem na příkazové řádce, nebo plug-inem do IDE Eclipse. Tyto různé programy pomocí volání funkcí z klientské knihovny vytvoří spojení se serverem. Zprávy se pak posílají pomocí objektu `Messenger`. Server si pro každé nové spojení vytvoří vlákno a objekt `Messenger` spojený s příslušným soketem. Tento objekt si zprávu od klienta přečte a předá ji do vlákna. Vlákno ji hned předává do objektu `CmdExecutor` ke zpracování. Zpráva vytvořena jako odpověď je opět předána objektu `Messenger` k odeslání zpět klientovi. Klient si ji přečte a může na ni reagovat. Například dalším posláním nějaké zprávy.

Závěr

V této práci jsme shrnuli základní informace o systémech mainframe, které jsme potřebovali znát pro zahájení vývoje naší aplikace. Dále jsme pak sepsali věci potřebné k implementaci komunikace přes TCP/IP pomocí soketů. Nakonec jsme položili základy projektu, kterým se budeme zabývat v dalších letech. V příloze B jsou dva jednoduché programy (server a klient), pro komunikaci s mainframe. Bohužel na mainframe, na kterém jsme pracovali, jsme z bezpečnostních důvodů neměli práva tyto programy řádně otestovat. Otevření portu a spuštění aplikace, která na něm bude naslouchat a která jistě obsahuje bezpečnostní mezery, představuje takové riziko, jaké nejsou majitelé z pochopitelných důvodů ochotni podstoupit. Vzhledem k tomu, že je ale implementace (až na hlavičky) víceméně stejná jako na GNU/Linux, stačilo k pochopení principů komunikace a k testování použití systému GNU/Linux a běžného PC.

Vzhledem k tomu, že vývoj aplikace takového rozsahu je velice časově náročný, je jasné, že práce je teprve na začátku. Většinu času jsme se zabývali spíše studiem problematiky. Přesto se nám povedlo vytvořit základní návrh budoucí struktury aplikace. Program, který je zatím k dispozici, je vyvinut pouze pro testování a je spíše ilustrací směru, jakým se bude práce dále ubírat. Byl testován pouze na GNU/Linux. Na mainframe jsme zatím neměli potřebná práva ke spuštění. Dále zůstal nedořešen problém překladu serverové části na mainframe. Server totiž využívá vlákna. Tuto technologii jsme zde neprobrali a její hlubší nastudování bude předmětem budoucí práce. Nejsou v něm prozatím ani ošetřeny různé výjimečné situace, které by mohly nastat například přerušením spojení (z různého důvodu) nebo tím, že jedna z komunikujících stran nepošle zprávu ve tvaru, jaký očekáváme atd.

Další vývoj proto nebude spočívat pouze v přidávání další funkcionality, ale také v refaktoringu stávajícího kódu a v ošetření oněch výjimek. Z bodů, které jsme si sepsali v úvodu, jsme se propracovali k bodu číslo dva. Funkce `cmdLogin()`, která by měla provádět autentifikaci, zatím pouze kontroluje, zda uživatelské jméno a heslo odpovídá dvěma stringům, které jsou v programu přímo vypsány. Zde bude v budoucnu použito assemblerovské makro, které už ale není předmětem této práce.

Funkce, která by měla být přidána v nejbližší době, je určitě otevření souborů a odeslání obsahu klientovi. Odeslání samotného obsahu nebude obtížné, problém může ale nastat se zalamováním řádků, které je na různých systémech různé. Dále pak, když má více uživatelů přístup ke stejným souborům, mohou vznikat problémy při jejich souběžné editaci. Jedním z dalších úkolů by mělo proto být nastudování způsobu zamykání souborů a problematiky s tím spojené.

Reference

[1] Jan Hofta: Použití systému mainframe pro zpracování dat, bakalářská práce, 2006

[2] ABCs of z/OS System Programming Volume 1,2,3,4

[3] Richard Stones, Neil Matthew, Linux začínáme programovat, Computer Press 2000

[4] Dave Elder-Vass, MVS Systems Programming, iUniverse 2000

[5] Donna Kelly, Jim Harding, MVS JCL in Plain English, Xlibris Corporation 2002

[6] Kevin McQuillen, Anne Prince, MVS Assembler Language, Mike Murach and Associates 1987

Seznam obrázků

Seznam tabulek

A Hodnoty proměnné errno

EPROTONOSUPPORT	Protokol není podporován jmenným prostorem
EMFILE	Tabulka deskriptorů je zaplněna
EACCESS	Nemáte právo vytvořit soket
ENOBUFS	Nedostatek vyrovnávací paměti

Tabulka 8: Hodnoty proměnné errno nastavené funkcí socket()

EBADF	sockfd není platným deskriptorem
EINVAL	soket už má přidělenu adresu
EACCES	adresa je chráněna a uživatel není superuživatelem
ENOTSOCK	argument je souborovým deskriptorem, není to soket

Tabulka 9: Hodnoty proměnné errno nastavené funkcí bind()

EBADF	sockfd není platným deskriptorem
ENOTSOCK	sockfd není deskriptorem soketu
EOPNOTSUPP	typ soketu není podporován voláním listen()

Tabulka 10: Hodnoty proměnné errno nastavené funkcí listen()

EAGAIN, EWOULDBLOCK	soket je non-blocking a fronta na připojení je prázdná
EBADF	deskriptor je neplatný
ECONNABORTED	spojení bylo přerušeno
EINTR	volání bylo přerušeno předtím než se někdo připojil
EINVAL	soket nenaslouchá, nebo addrlen je neplatné
EMFILE	vyčerpán limit otevřených deskriptorů na jeden proces
ENFILE	vyčerpán limit otevřených deskriptorů pro celý systém
ENOTSOCK	deskriptor ukazuje na otevřený soubor, ne soket
EOPNOTSUPP	soket není typu SOCK_STREAM

Tabulka 11: Hodnoty proměnné errno nastavené funkcí accept()

`EBADF`	špatný deskriptor
`EFAULT`	adresa soketu je mimo adresový prostor procesu
`ENOTSOCK`	deskriptor není platným deskriptorem soketu
`EISCONN`	soket je již spojen
`ECONNREFUSED`	spojení bylo serverem odmítnuto
`ETIMEDOUT`	časový limit pro spojeni vypršel
`ENETUNREACH`	síť není dosažitelná
`EADDRINUSE`	adresa je již používána

Tabulka 12: Hodnoty proměnné `errno` nastavené funkcí `connect()`

`EBADF`	špatný deskriptor
`EFAULT`	adresa soketu je mimo adresový prostor procesu
`ENOTSOCK`	deskriptor není platným deskriptorem soketu
`EMSGSIZE`	soket požaduje, aby tato zpráva byla poslána atomicky, ale velikost zprávy toto znemožňuje
`EWOULDBLOCK`	soket je označen jako neblokující a požadovaná operace by blokovala
`ENOBUFS`	systém není schopen alokovat interní buffery, operace může být úspěšná až budou buffery k dispozici
`ENOBUFS`	výstupní fronta pro síťové rozhraní je zaplněna, to obvykle znamená, že rozhraní přestalo posílat, ale může to být také způsobeno občasným přetížením rozhraní

Tabulka 13: Hodnoty proměnné `errno` nastavené funkcí `send()`

`EAGAIN`	soket je označen non-blocking a tato operace by zablokovala, nebo vypršel časový limit, který byl nastaven
`EBADF`	`sockfd` není platný deskriptor
`ECONNREFUSED`	vzdálený host odmítl síťové spojení
`EFAULT`	pointer na přijímací buffer ukazuje mimo adresový prostor procesu
`EINTR`	příjem dat byl přerušen doručením signálu, před jakýmikoliv daty
`EINVAL`	předán neplatný argument
`ENOTCONN`	soket, který by měl být spojen, spojen není
`ENOTSOCK`	argument `sockfd` neukazuje na soket

Tabulka 14: Hodnoty proměnné `errno` nastavené funkcí `recv()`

B Příklad implementace serveru a klienta

Nejdříve si ukážeme příklad serveru. Ten počká na připojení jednoho klienta a odpoví mu.

```
/*
 * #include <manifest.h>  // hlavičky pro překlad na mainframe
 * #include <bsdtypes.h>
 * #include <socket.h>
 * #include <in.h>
 * #include <netdb.h>
 * #include <stdio.h>
 * #include <string.h>
 */

#include <sys/types.h>    // hlavičky pro překlad na linuxu
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char buf[12];              // buffer na příjem a posílání zpráv
    struct sockaddr_in server; // adresa serveru
    struct sockaddr_in client; // adresa klienta
    int server_socket;         // soket serveru
    int client_socket;         // nový soket pro klienta
    int addrlen;               // délka adresy
    int result;                // výsledky voláni

    server_socket = socket(AF_INET, SOCK_STREAM, 0);
        // vytvoření soketu
    if (server_socket < 0) {
        fprintf(stderr, "Nepodarilo se vytvorit soket\n");
        exit(1);
    }

    server.sin_family      = AF_INET;
    server.sin_port        = htons(7777);
    server.sin_addr.s_addr = INADDR_ANY;

    result = bind(server_socket, (struct sockaddr *)&server,
sizeof(server));
        // připojení serverového soketu ke konkrétnímu portu
    if (result < 0) {
        fprintf(stderr, "Nepodarilo se pripojit k portu\n");
        exit(2);
```

```
    }

    result = listen(server_socket, 1);
        // nastavení, aby server na soketu naslouchal, s délkou fronty 1
    if (result != 0) {
        fprintf(stderr, "Chyba v nastaveni naslouchani\n");
        exit(3);
    }

    addrlen = sizeof(client);
    client_socket = accept(server_socket, (struct sockaddr *)&client,
&addrlen);
        // přijetí nového spojení od klienta
    if (client_socket  == -1) {
        fprintf(stderr, "Nezdarilo se prijeti noveho spojeni\n");
        exit(4);
    }

    result = recv(client_socket, buf, sizeof(buf)-1, 0);
        // přijetí zprávy od klienta
    if (result == -1) {
        fprintf(stderr, "Prijeti zpravy se nezdarilo\n");
        exit(5);
    }
    if (result > 0) {
        buf[result]='\0'; // přidaní znaku ukončující string
        printf("Prijata zprava je: %s\n", buf); // tisk zprávy
    }

    strcpy(buf, "tady server"); // překopírovaní zprávy do bufferu

    result = send(client_socket, buf, strlen(buf), 0);
        // odeslání zprávy zpět
    if ( result < 0) {
        fprintf(stderr, "Nepodarilo se odeslat zpravu\n");
        exit(6);
    }

    close(client_socket); // zavření soketu klienta
    close(server_socket); // zavření soketu serveru

    printf("Server uspesne skoncil\n");
    exit(0);
}
```

Teď už příklad klienta, který se připojí na server, pošle zprávu a přijme odpověď.

```
/*
 * #include <manifest.h>   // hlavičky pro překlad na mainframe
 * #include <bsdtypes.h>
 * #include <in.h>
 * #include <socket.h>
 * #include <netdb.h>
 * #include <stdio.h>
 * #include <string.h>
 */

#include <sys/socket.h>   // hlavičky pro překlad na linuxu
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {

    char buf[12];              // buffer na příjem a posílaní zpráv
    struct sockaddr_in server; // adresa serveru
    int sockfd;                // soket klienta
    int result;                // výsledky volání

    strcpy(buf, "tady klient"); // překopírování zprávy do bufferu

    server.sin_family      = AF_INET; // adresová rodina
    server.sin_port        = htons(7777); // port, na který se připojíme
    server.sin_addr.s_addr = inet_addr("10.11.12.13");
                                          // IP adresa serveru

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
        // vytvoření soketu
    if (sockfd < 0) {
        fprintf(stderr, "Nepodarilo se vytvorit soket\n");
        exit(1);
    }

    result = connect(sockfd, (struct sockaddr *)&server, sizeof(server));
        // připojení na server
    if (result < 0) {
        fprintf(stderr, "Nepodarilo se pripojit na server\n");
        exit(2);
    }

    result = send(sockfd, buf, strlen(buf), 0);
```

```
        // odeslání zprávy
    if (result < 0) {
        fprintf(stderr, "Nepodarilo se odeslat zpravu\n");
        exit(3);
    }

    result = recv(sockfd, buf, sizeof(buf)-1, 0);
        // přijetí odpovědi
    if (result < 0) {
        fprintf(stderr, "Prijeti zpravy se nezdarilo\n");
        exit(4);
    }
    if (result > 0) {
        buf[result]='\0'; // přidání znaku ukončující string
        printf("Prijata zprava je: %s\n", buf); // tisk odpovědi
    }

    close(sockfd); // zavření soketu

    printf("Klient uspesne skoncil\n");
    exit(0);
}
```

Nakonec JCL úkoly pro překlad na mainframe. K překladu je důležité vědět, že všechny potřebné hlavičky se nachází v datové sadě TCPIP.SEZACMAC. Tento příklad je pro překlad serveru. Skript pro překlad klienta bude úplně shodný, až na cestu k souboru se zdrojovým kódem.

```
//VLARA80A JOB (90300000),'VLARA80',NOTIFY=&SYSUID,
//          MSGCLASS=H,CLASS=A,MSGLEVEL=(1,1),REGION=0M
//MYLIB JCLLIB ORDER=(CBC.SCCNPRC)
//*
//COMPCL   EXEC PROC=CBCCL,
// CPARM='OPTFILE(DD:OPTFILE)',
// INFILE='VLARA80.BP.SOURCES(SRV)',
// OUTFILE='VLARA80.BP.LOAD(SRV),DISP=SHR'
//COMPILE.SYSLIB DD DSN=TCPIP.SEZACMAC,DISP=SHR
//               DD DSN=CEE.SCEEH.H,DISP=SHR
//LKED.SYSLIB DD DSNAME=CEE.SCEELKED,DISP=SHR
//            DD DSNAME=TCPIP.SEZACMTX,DISP=SHR
```

C Obsah přiloženého CD

Na CD jsou přiloženy zdrojové kódy naší aplikace.

- Client/
  - Makefile
  - SConscript
  - client.cpp
  - clientlib.cpp
  - clientlib.h
- Library/
  - Makefile
  - SConscript
  - input.cpp
  - input.h
  - message.cpp
  - message.h
  - messenger.cpp
  - messenger.h
  - outputSocket.h
  - protocol.h
- ServerThread/
  - Makefile
  - SConscript
  - cmdExec.cpp
  - cmdExec.h
  - cmdFuncs.cpp
  - cmdFuncs.h
  - serverThread.cpp
- SConstruct
- Makefile

Zdrojové kódy jsou psány pro GNU/Linux. Lze je přeložit pomocí nástroje GNU/Make a přiložených Makefile souborů. Lze přeložit i pomocí Scons a přiložených SConstruct, SConscript souborů.