ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

| | |
|---|---|
| **Název:** | Knihovna funkcí pro podporu diagnostického protokolu s pomocí UDP |
| **Student:** | Jaromír Mikušík |
| **Vedoucí:** | Ing. Pavel Kubalík, Ph.D. |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Počítačové inženýrství |
| **Katedra:** | Katedra číslicového návrhu |
| **Platnost zadání:** | Do konce letního semestru 2017/18 |

## Pokyny pro vypracování

Prozkoumejte existující diagnostické protokoly používané pro vyčítání dat a nahrávání software do mikrokontroléru.
Vyberte vhodný diagnostický protokol a vytvořte pro něj vhodnou knihovnu funkcí umožňující jeho spolehlivé fungování nad UDP protokolem.
Implementované funkce budou navrženy a implementovány s ohledem na minimální latence a timeouty běžné při implementaci nad TCP protokolem.
Knihovna funkci bude rozdělená na části server a klient.
Server bude implementován v jazyce C++ obecně pro libovolný mikrokontrolér.
Klient bude s ohledem na použití na PC implementován v jazyce Java.
Výsledné řešení otestujte.

## Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 24. ledna 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra číslicového návrhu

Bakalářská práce

# Knihovna funkcí pro podporu diagnostického protokolu s pomocí UDP

***Jaromír Mikušík***

Vedoucí práce: Ing. Pavel Kubalík, Ph.D.

11. května 2017

# Poděkování

Děkuji vedoucímu práce Ing. Pavlu Kubalíkovi, Ph.D. za jeho ochotu a rady během vytváření práce, kolegům Ing. Michalu Sarnovskému a Ing. Martinu Makovičkovi za pomoc při návrhu knihovny a členům rodiny za obrovskou podporu během studia.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 11. května 2017 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Jaromír Mikušík. Všechna práva vyhrazena.
*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.*

### Odkaz na tuto práci

Mikušík, Jaromír. *Knihovna funkcí pro podporu diagnostického protokolu s pomocí UDP*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

# Abstrakt

Cílem práce je vytvořit knihovnu funkcí pro podporu přenosu diagnostického protokolu. Použití této knihovny umožní paralelní diagnostiku procesorů v řídících systémech, kde je potřeba zajistit, aby nedocházelo ke kolizím zpráv v rámci jejich komunikace. Knihovna obsahuje i vlastní komunikační protokol. Použití této knihovny v diagnostické aplikaci na PC a v rámci systému povede ke zrychlení práce servisních techniků a vyvojářů HW a SW. V příloze lze nalézt zdrojové kódy klienta v jazyce Java a serveru jazyce C++.

**Klíčová slova** diagnostika mikrokontrolerů, knihovna, UDP, mikrokontroler, přenos diagnostického protokolu, Java, C++

# Abstract

The goal of this thesis is to create a library to support transferring of diagnostic protocol. Using this library allows parallel diagnostic of the processors within control systems where it is crucial to prevent collisions of data within their communication. The library also contains a special communication protocol. Using this library inside a diagnostic application on PC and within the control system increases the speed and effectivity of the work of service engineers and SW and HW developers. The attachment contains source code of the client written in Java and server written in C++.

**Keywords** microcontroller diagnostic, library, UDP, microcontroller, diagnostic protocol transfer, Java, C++

# Obsah

# Seznam obrázků

# Seznam tabulek

# Úvod

V dnešní době se vše zrychluje, výjimkou nejsou ani procesy vývoje, testování a výroby řídících systémů. Vývoj SW pro takovéto systémy se dá urychlit použitím knihoven, které jsou navrženy tak, aby se daly jednoduše použít znova. Testování HW i SW a případný servis je možné urychlit umožněním paralelního přístupu k jednotlivým částem systému.

V práci se zabývám analýzou, návrhem a implementací knihovny pro podporu přenosu firemního diagnostického protokolu DCP. Knihovna je rozdělena na dvě části: klient integrovatelný do firemní diagnostické aplikace TST a server integrovatelný do aplikací v již existujících řídících systémech.

Vytvoření a použití knihovny pomůže urychlit a zefektivnit pracovní úkony servisních techniků a vývojářů SW i HW ve firmě Poll. Mezi tyto úkony patří zejména testování a diagnostika systémů. Zrychlení těchto úkonů přispěje ke zrychlení procesů vývoje a servisu řídících systémů, které firma dodává.
Práce pokračuje podle následující struktury:
Nejprve se věnuje úvodu do síťových modelů. Následuje rešerše diagnostických a komunikačních protokolů. Dále se zabývá analýzou a návrhem knihovny a jejího protokolu. Po těchto dovu částech následuje popis implementace jednotlivých částí knihovny. Na závěr je funkčnost knihovny ověřena pomocí několika testů.

Kapitola 1

# Cíl práce

Cílem práce je vytvořit knihovnu funkcí pro podporu přenosu diagnostického protokolu pomocí UDP. Použití této knihovny zajistí podporu paralelní diagnostiky a zamezí kolizi dat komunikace mikrokontrolerů v rámci systému. Knihovna je rozdělena na dvě části:

1. Klient
   Napsán v jazyce Java kvůli integraci do firemní diagnostické aplikace TST (Tuning & Service Tool).

2. Server
   Napsán v jazyce C++ s ohledem na integraci do již existujících systémů napsaných v tomtéž jazyce.

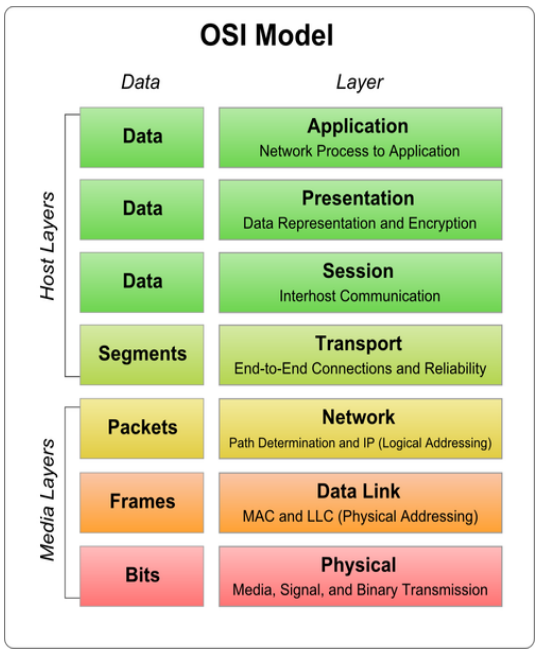
Kapitola 2

# Modely síťové komunikace

*Tato kapitola čerpá primárně ze zdroje [1], s. 41–54.*

## 2.1 OSI model

OSI (Open System Interconnection) model popisuje spojování a přenášení dat mezi systémy, které jsou schopné komunikovat s jinými systémy. Model je rozdělen na sedm abstraktních vrstev seskupených podle podobnosti funkcionality a platí, že nižší vrstva poskytuje rozhraní vrstvám vyšším.




Obrázek 2.1: Vrstvy OSI modelu a reprezentace dat v nich [2]

Popis jednotlivých vrstev OSI modelu:

- Aplikační
  Definuje způsob komunikace aplikací, představuje rozhraní pro uživatele. Jako příklad lze uvést protokol HTTP používaný pro přenos webových stránek.

- Prezentační
  Zajišťuje formátování a prezentaci dat, to zahrnuje například kódování znaků, šifrování a dešifrování, kompresi a dekompresi.

- Relační
  Vytváří logické rozhraní pro aplikace, synchronizuje spojení, stará se o přihlašování a udržování spojení. Jako příklad lze uvést přístup ke sdíleným diskům.

- Transportní
  Přenáší data mezi aplikacemi nebo zařízeními. Řeší spolehlivost, kontrolu a fragmentaci dat. Patří zde TCP a UDP.

- Síťová
  Stará se o adresaci a směrování dat mezi jednotlivými segmenty sítě, dále pak každému zařízení přiřazuje jednoznačnou adresu v rámci sítě. Zde je nejznámější IP.

- Spojová (linková)
  Zajišťuje detekci, příp. korekci chyb přenášených dat, řídí přístup k lince a také tok na ní. Každému zařízení v segmentu sítě pak přiřazuje jednoznačnou adresu. Patří zde např. Ethernet.

- Fyzická
  Umožňuje přenos bitů kanálem. Určuje vlastnosti a způsob sdílení média, dále pak mechanické a elektrické vlastnosti rozhraní. Např. Ethernet 10BaseT nebo také standard bezdrátového rozhraní 802.11.

Tento model se však neuchytil kombinací těchto čtyřech důvodů:

1. Špatné načasování
   Než stihl standard OSI přijít na trh, výrobci už dodávali produkty s TCP/IP protokoly a nechtěli podporovat novou technologii, dokud k tomu nebyli donuceni.

2. Špatné technologie
   Rozdělení na sedm vrstev je spíše politického než technického rázu, což se projevuje např. tím, že relační a prezentační vrstvy jsou téměř prázdné, zatímco spojová a síťová jsou přeplněny. Dále některé funkcionality, jako např. kontrola chyb, se opakují ve více vrstvách, což je nadbytečné a neefektivní.

3. Špatná implementace
   Kvůli vysoké složitosti modelu byly implementace zpočátku pomalé, což udělalo OSI modelu špatné jméno. I když se implementace s časem zlepšovaly, špatná pověst už modelu zůstala.

4. Špatná politika
   OSI model byl považován za výtvor evropských, posléze americké vlády. Myšlenka, že se byrokraté snaží vytvořit standard technického rázu, modelu nijak nepomohla.

## 2.2 TCP/IP model

Model je pojmenován podle dvou hlavních protokolů, na kterých je postaven: TCP a IP. Už od počátku návrhu tohoto modelu se počítalo s tím, že zajistí hladké propojení více sítí. Model měl také zachovávat přenos mezi koncovými zařízeními i v případě selhání přenosové linky. Z hlediska podpory různorodých aplikací bylo potřeba, aby tento model byl flexibilní.


Obrázek 2.2: Vrstvy TCP/IP modelu vzhledem k OSI modelu

Následuje stručný popis vrstev modelu TCP/IP:

- Aplikační
  Obsahuje protokoly používané aplikacemi, tyto aplikace si sami udržují informace, které by v OSI modelu patřily do prezentační a relační vrstvy. Patří zde např. již zmiňovaný HTTP nebo také DNS používáný pro překlad internetových adres.

- Transportní
  Umožňuje udržovat spojení mezi koncovými zařízeními a odpovídá transportní vrstvě v OSI modelu. Patří zde již zmiňované protokoly TCP a UDP.

- Internetová
  Tato vrstva je schopná posílat pakety do jakékoliv sítě, tyto pakety putují nezávisle na sobě a mohou dorazit v přeházeném pořadí, v tom případě se o jejich správné seřazení musí postarat vyšší vrstva. Do této vrstvy patří protokol IP.

- Síťová
  Popisuje, co musí splňovat např. ethernet nebo sériová linka pro přenos IP paketů.

## 2.3 Porovnání TCP a UDP

*Tato sekce čerpá ze zdrojů [3], [4] a [5].*

TCP a UDP patří mezi dva nejpoužívanější protokoly transportní vrstvy. Jsou postaveny na IP a přenáší se pomocí paketů. I když pakety mají rozdílnou strukturu, princip jejich přenosu se nijak zásadně neliší.

Tabulka 2.1: Rozdíl funkcionalit TCP a UDP

| | TCP | UDP |
|---|---|---|
| podpora spojení | ✓ | ✗ |
| detekce chyb | ✓ | ✓ |
| potvrzení přijetí paketu | ✓ | ✗ |
| znovuodeslání paketu | ✓ | ✗ |
| řazení paketů | ✓ | ✗ |

Jak je vidět z tabulky 2.1, TCP zajišťuje spolehlivý přenos dat včetně řazení, potvrzování i znovuodeslání paketů. UDP na druhou stranu pakety neřadí ani nečeká na jejich potvrzení. Oba protokoly kontrolují data na chyby, avšak UDP chybné pakety zahazuje, TCP znovu odesílá. TCP tedy sice zajišťuje spolehlivý přenos, ale platí za to větší časovou odezvou.

TCP se použije tam, kde je žádoucí, aby data nedošla zpřeházená, jako příklad lze uvést přenos webových stránek nebo obecně jakýchkoli souborů. UDP se naopak použije tam, kde se mohou nějaké pakety poztrácet, např. při streamování videa.

Kapitola 3

# Rešerše

Komunikačních protokolů existuje celá řada. Některé se používají pro přenos komunikace na internetu, jiné při automatizaci průmyslových procesů nebo řízení budov.

## 3.1 Standardizované automotive protokoly

Zde jsou popsány tři standardizované protokoly používané pro diagnostiku pozemních vozidel.

### 3.1.1 OBD-II - On-Board Diagnostics II

Standard specifikuje typ konektoru, formát zpráv a také diagnostické protokoly, které mohou být použity. Problémem však je, že už nespecifikuje vnitřní uspořádání modulů. Důsledkem pak je, že téměř každá automobilka používá svůj vlastní protokol, který je navržen na míru pro její typy vozidel. Je třeba také zmínit, že implementace tohoto protokolu jsou neveřejné.

### 3.1.2 UDS - Unified diagnostic services

Protokol používaný zejména v prostředí elektronického řízení vozidel, v tomto prostředí se nachází např. řízení automatické převodovky nebo ovládání ABS[1], zkrátka vše, co jde řídit elektronicky. Primárně je určen pro sběrnici CAN (Controller Area Network), existuje však i varianta UDSonIP popisující přenos po IP.

[1] anti-lock braking system, bezpečnostní systém, který při brždění zaručí, aby se kola stále točila vzhledem k vozovce

### 3.1.3 DoIP - Diagnostic over Internet Protocol

Popisuje strukturu diagnostických zpráv v Ethernetovém paketu při komunikaci diagnostické aplikace s vozidlem. Většinou se používá v kombinaci s UDS. Výměnou sběrnice CAN za Ethernet se dosahuje větší propustnosti dat, použití IP pak zaručuje jednoduchou integraci v různých sítích.[6]

## 3.2 IoT protokoly

Tato sekce popisuje několik otevřených protokolů používaných v IoT (Internet of Things, internet věcí), což je označení pro připojení všech druhů zařízení (*„věcí“*, např. mobilních telefonů, praček, lamp, kávovarů ...) na *internet*.

### 3.2.1 OPC - Open Platform Communications

Standard specifikuje abstraktní vrstvu a její protokoly pro komunikaci mezi zařízeními a aplikacemi. To znamená, že ani aplikace ani zařízení nepotřebují pro komunikaci mezi sebou dodatečné ovladače, jeden o druhém nevědí vůbec nic. Rozšíření tohoto standardu se nazývá OPC–UA (Unified Architecture). Rozdíl mezi obyčenjou a UA variantou je multiplatformnost, UA umožňuje použít OPC na všech platformách. Rozšířené OPC také poskytuje více funkcí jako např. šifrování dat nebo autentizaci uživatelů. Jako implementaci lze uvést open-source projekt FreeOpcUa[7] v jazycích C++ a Python.

### 3.2.2 AMQP - Advanced Message Queueing Protocol

AMQP[8] je otevřený internetový protokol pro přenos zpráv. Podobně jako model OSI je rozdělen do více vrstev. Jednotlivé vrstvy popisují přenos mezi dvěma koncovými body, abstraktní formát zprávy a jejich kódování nebo také bezpoečnost.

## 3.3 Shrnutí

Standardy vyjmenovaných protokolů jsou pro účely této práce přílíš komplexní, proto jsem přistoupil k vytvoření nového protokolu. Z toho také vyplývá, že pro jeho podporu je nutné vytvořit novou knihovnu.

Kapitola 4

# Analýza a návrh

V této kapitole je popsán síťový model knihovny a jakým vrstvám odpovídá v modelech OSI (sekce 2.1) a TCP/IP (sekce 2.2). Dále jsou zde teoreticky popsány základní jednotky CLP (Communication Locking Protocol, protokol vytvořený v rámci této práce), bez kterých by nemohl fungovat. CLP počítá také s použitím TCP/IP, pro identifikaci klienta se použije IP adresa a port, ze kterého odesílá data. Je zde popsáno API (programové rozhraní aplikace) klienta, serveru i některých jejich částí.

## 4.1 Síťový model

<table>
<tr><th>OSI</th><th>Komunikační protokol</th><th>TCP/IP</th></tr>
<tr><td>Aplikační</td><td rowspan="2">DCP</td><td rowspan="3">Aplikační</td></tr>
<tr><td>Prezentační</td></tr>
<tr><td>Relační</td><td>spojení + zámky</td></tr>
<tr><td>Transportní</td><td>UDP</td><td>Transportní</td></tr>
<tr><td>Síťová</td><td>IP</td><td>Internetová</td></tr>
<tr><td>Linková</td><td rowspan="2">Síťové rozhraní</td><td rowspan="2">Síťové rozhraní</td></tr>
<tr><td>Fyzická</td></tr>
</table>

Obrázek 4.1: Síťový model protokolu vedle modelů OSI a TCP/IP

Na obrázku 4.1 můžeme vidět, že model knihovny více odpovídá modelu TCP/IP, avšak implementuje také relační vrstvu OSI modelu pomocí spojení (sekce 4.4.4) a zámků (sekce 4.4.3).

Jako protokol tranposrtní vrstvy je použit UDP místo TCP. Důvod je jednoduchý, vlastnosti, které má TCP navíc oproti UDP (viz sekce 2.3), jsou součástí vyšších vrstev nebo nejsou potřeba vůbec.

Přeposílání paketů, ať už ztracených nebo chybných, zajišťuje ovladač DCP: pokud nedostane odpověď na zprávu do určitého časového limitu, pošle ji znova.

Řazení paketů není potřeba, přepokládá se, že linka spoující diagnostikované zařízení a diagnostickou aplikaci nebude extra vytížená a bude obsahovat malé množství zařízení ovlivňující tok dat v síti (směrovače, rozbočovače).

## 4.2 Struktura dat

Následující části popisují strukturu UDP paketů a dat CLP. Hlavička IP tu popsána není, nicméně je potřeba zmínit, že obsahuje položku obsahující IP adresu odesílatele, ta je důležitá pro identifikaci klienta.

### 4.2.1 UDP paket


Obrázek 4.2: Struktura UDP paketu

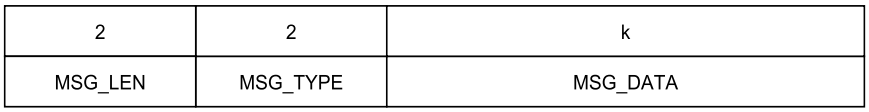
Na obrázku 4.2 lze vidět strukturu paketu z pohledu transportní vrstvy. Hlavička UDP obsahuje pouze čtyři 16bitové položky:

- Zdrojový port
  Číslo portu odesílatele. Ačkoli se podle standardu jedná o nepovinnou položku, v tomhle případě je společně s IP adresou potřeba k identifikaci klienta.
- Cílový port
  Číslo portu příjemce. Server ve výchozím nastavení poslouchá na portu 2021.
- Délka
  Délka celého paketu včetně hlavičky.
- Kontrolní součet
  Obsahuje kontrolní součet používaný ke kontrole dat paketu.

### 4.2.2 Struktura CLP

*Pozn.:U všech obrázků platí, že velikosti jednotlivých položek jsou uvedeny v bajtech.*
Data jsou přenášena ve formátu big-endian, tzn. že se bajt s nejvyšší významností nachází vlevo, významnost ostatních bajtů poté klesá směrem doprava.

Na následujícím obrázku je uvedena struktura zprávy CLP, kterou posílá klient.



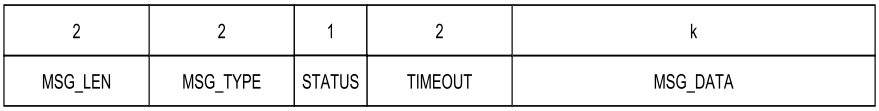

Obrázek 4.3: Struktura zprávy klient→server

Popis jednotlivých položek zprávy z obrázku 4.3:

- MSG_LEN
  Délka celé zprávy v bajtech.
- MSG_TYPE
  Typ protokolu, který se přenáší v MSG_DATA.
- MSG_DATA
  Data přenášeného diagnostického protokolu.

Server zprávu vyhodnotí a může poslat dva typy odpovědi:

1. Odpověď bez chyby. Zpráva byla úspěšně předána ovladači diagnostického protokolu na straně serveru, který jí vyhodnotil a posílá zpátky odpověď. V takovémhle případě má odpověď následující strukturu:




Obrázek 4.4: Struktura zprávy bez chyby server→klient

Oproti zprávě posílané klientem se liší přidáním dvou položek:

- STATUS
  Bajt indikující, zda jde o zprávu s chybou nebo bez. V případě zprávy bez chyby má hodnotu 0.
- TIMEOUT
  Informace o tom, kolik vteřin zbývá do ukončení aktuálního spojení. V současné implementaci se položka nepoužívá a je vždy nastavena na hodnotu 0.

2. Odpověď s chybou podle následující struktury:

| 2 | 2 | 1 | 2 | 4 |
|---|---|---|---|---|
| MSG_LEN | MSG_TYPE | STATUS | TIMEOUT | ERROR_TYPE |

Obrázek 4.5: Struktura chybové zprávy server→klient

V tomhle případě je položka STATUS nastavena na hodnotu 1, podle toho jde určit, že se jedná o zprávu s chybou. Samotná chyba je pak pomocí kódu specifikována v položce ERROR_TYPE a bude odpovídat jednomu z následujících případů:

- MAX_CON_REACHED
  Na serveru bylo dosaženo maximálního počtu aktivních spojení.
- INVALID_MSG_LEN
  Délka přijaté zprávy neodpovídá délce zprávy uvedené v hlavičce.
- MEM
  Serveru došla operační paměť.
- UNSUPPORTED_DIAG_PROTOCOL
  Nepodporovaný typ protokolu.
- DEVICE_NOT_FOUND
  ID procesoru se v systému nenachází nebo nebylo registrováno.
- OPERATION_DENIED
  Operace zamítnuta, klient nemá právo odemknout/zamknout zámek s id daného procesoru, protože ho má nejspíš zamknutý jiný klient.

## 4.3 Klient

Sekce se zabývá rozhraními klienta v jazyce Java. Sekce 4.3.1 popisuje, co musí splňovat ovladač diagnostického protokolu, aby mohl přijímat zprávy. Následuje sekce 4.3.2, která popisuje rozhraní posluchače chyb. Sekce 4.3.3 popisuje API klienta. Kapitola je zakončena sekcí popisující výčtové typy.

### 4.3.1 Třída pro ovladač diagnostického protokolu

Třída `DiagProtocol` určuje, co musí splňovat ovladač diagnostického protokolu, aby mohl být informován o datech příchozích ze serveru.

```
public abstract class DiagProtocol {
  /* Typ protokolu. Nutno zadat při inicializaci. */
  private MsgType type;
  /* Vytvoří instanci ovladače s daným typem. Tento typ se
   * používá k identifikaci jednotlivých ovladačů.*/
  public DiagProtocol(MsgType pType) {
    type = pType;
  }
  public MsgType getType(){
    return type;
  }
  /* Umožní ovladači diagnostického protokolu přijímat
   * data od serveru. */
  public abstract void notifyNewMessage(byte[] message);
}
```

Listing 4.1: Třída DiagProtocol

### 4.3.2 Rozhraní posluchače chyb

Rozhraní `IErrorListener` umožní aplikaci zjišťovat a reagovat na stav komunikace.

```
public interface IErrorListener {
  /* Umožní třídě implementující toto rozhraní reagovat na stav
   * komunikace. */
  public void notifyError( ErrType err );
}
```

Listing 4.2: Rozhraní IErrorListener

*Pozn.: současná implementace třídy* `CLPClient` *(4.3.3) posílá stav komunikace* všem *registrovaným posluchačům. To má za následek nesprávné vyhodnocování*

*stavu komunikace při použití více diagnostických protokolů. V případě jednoho diagnostického protokolu funguje vše v pořádku.*

### 4.3.3 Ovladač protokolu CLP

Třída `CLPClient` definuje rozhraní klienta. Její součásti je také logika pro odesílání dat pomocí UDP. Následuje výčet a popis funkcí, které je možné zavolat z aplikace.

```
/* Vrátí instanci klienta. Při prvním zavolání se provede
 * inicializace a vytvoření objektu, při dalším zavolání
 * se pouze vrátí už dříve vytvořený objekt.*/
public static CLPClient getInstance();
/* Nastaví port serveru. Výchozí hodnota je 2021, netřeba nastavovat. */
public void setPort( int port );
/* Nastaví IP adresu serveru. Je nutné ji nastavit. */
public void setDestIP( InetAddress addr );
/* Vrací true, pokud je aktivní spojení se serverem. Z hlediska UDP to
 * pouze znamená, zda je nastavena IP adresa a port u soketu. */
public boolean isConnected();
/* Pokusí se o vytvoření spojení. Opět z hlediska UDP jde pouze
 * o nastavení cílové IP adresy a portu. */
public boolean connect() throws SocketException;
/* Ukončí spojení tak, že soketu odebere IP adresu a port. */
public boolean disconnect();
/* Provede operaci odpojení a připojení. */
public boolean reconnect() throws SocketException;
/* Přidá ovladač diagnostického protokolu do seznamu.
 * Nutné pokud ovladač vyžaduje, aby mu byla přeposílána data. */
public void registerDiagProtocol( DiagProtocol dp );
/* Odebere ovladač diagnostického protokolu ze seznamu. Po
 * zavolání již ovladači nebudou posílána data, dokud se znova
 * nezaregistruje. */
public void removeDiagProtocol( DiagProtocol dp );
/* Přidá posluchače chyb do seznamu, aby mohl dostávat data. */
public void registerErrorListener( IErrorListener sl );
/* Odebere posluchače chyb ze seznamu. */
public void removeErrorListener( IErrorListener sl );
/* Hlavní metoda pro posílání zpráv s použitím CLP a UDP.
 * K datům se přidá hlavička se všemi náležitostmi. Takto
 * vytvořená zpráva se zařadí do fronty k odeslání. */
public void writeMessage(MsgType type, byte[] message);
```

Listing 4.3: Popis API CLPClient

### 4.3.4 Výčtové typy

`ErrType` popisuje chybové kódy a také jim přiřazuje textové řetězce, které se mohou použít pro výpis do konzole. Jednotlivé chyby odpovídají chybám v uvedených v sekci 4.2.2, konkrétně část popisující chybovou odpověď.

`MsgType` reprezentuje typy zpráv (a také diagnostických protokolů), které lze přenášet. Přidáním dalšího typu lze zajistit jednoduchou podporu přenosu dalšího diagnostického protokolu. Momentálně obsahuje dva typy, `DCP` a `LOOPBACK`. `LOOPBACK` byl použit na začátku vývoje pro odladění většiny chyb a také při testování.

`Status` reprezentuje stav přijaté zprávy, zda je bez chyby nebo s chybou.

## 4.4 Server

Pro server byl vybrán jazyk C++ jelikož je potřeba ho zaintegrovat do již běžících systémů, které jsou rovněž napsány v tomto jazyce.

### 4.4.1 Použité knihovny

Sekce popisuje knihovny použité při implementaci knihovny a také jejich rozhraní.

#### 4.4.1.1 Light Weight TCP/IP stack

Knihovna určená primárně pro mikrokontrolery implementující TCP/IP. Funkce z této knihovny použité při implementaci mají prefix `udp_`. Detailní dokumentaci lze najít na [9].

#### 4.4.1.2 `dco::desc`

Firemní rozhraní pro přenášení zpráv v rámci procesoru, tzv. deskriptory.

```
// Struktura deskriptoru reprezentující zprávu.
typedef struct dco_desc_n {
  t_buffer      data;  // ukazatel na pole s daty
  struct dco_desc_n*  next;  // ukazatel na další deskriptor
  t_uint16      length;  // velikost validních dat
  union {
    t_uint16  all;
    struct {
      t_uint16  newMesg  : 1; // příznak označující začátek zprávy
      // následují další nedůležité příznaky
    }  bits;
  }  flags;
} t_desc;
// Metoda vrátí deskriptor s naalokovaným prostorem pro data o velikosti
// dataLength. Metoda nastaví atribut length na 0.
t_desc* getNewDesc(t_uint16 dataLength);
// Uvolní deskriptor z paměti.
void freeDesc(t_desc* desc);
// Maximální možná velikost dat v deskriptoru. Nastaven na 1024.
int getDescMaxLength();
```

Listing 4.4: Rozhraní `dco::desc`

#### 4.4.1.3 `core::thread`

Firemní rozhraní, pomocí kterého lze v mikrokontroleru jednoduše implementovat vlákna.

```
// Rozhraní reprezentující vlákno.
class IThread{
public:
  // Konstruktor pro vytvoření vlákna.
  IThread(bool breakable = false);
  // Metoda volána z manažera vláken. Obsahuje kód samotného vlákna.
  virtual void loop() = 0;
}
// Pomocí této metody lze registrovat vlákna do manažera.
core::mainTM->registerThread( t_uint16 priority,
                              core::thread::IThread* thr );
```

Listing 4.5: Rozhraní `core::thread`

#### 4.4.1.4 `tim::stm`

Firemní rozhraní pro jednoduché a přesné měření času za pomocí tiků procesoru.

```
// Vrátí momentální počet tiků.
t_uint32 currentTicksLo();
// 64bitová verze předchozí metody.
t_uint64 currentTicks();
// Vrátí rozdíl oldTicks a momentálního počtu tiků.
t_uint32 ticksChangeLo(t_uint32 oldTicks);
// Stejné jako předchozí metoda, liší se typem parametru.
t_uint32 ticksChange(t_uint64 oldTicks);
// Převádí počet tiků na milisekundy.
t_uint32 ticks2milis(t_uint32 ticks);
// Převádí počet tiků na mikrosekundy.
t_uint32 ticks2micros(t_uint32 ticks);
// Převádí milisekundy na tiky.
t_uint32 milis2ticks(t_uint16 milis);
// Převádí mikrosekundy na tiky.
t_uint32 micros2ticks(t_uint16 micros);
```

Listing 4.6: Rozhraní `tim::stm`

#### 4.4.1.5 `dco::ifc::MessageListener`

Firemní rozhraní pro posluchače zpráv, tento posluchač čeká na zprávy ve formě deskriptorů z `dco::desc`, které pak následně zpracovává.

```
class MessageListener{
public:
  // Metoda sloužící pro notifikaci posluchače, když mu přišla nová zpráva.
  virtual void notifyNewMessage(desc::t_desc* messageDesc) = 0;
};
```

Listing 4.7: Rozhraní `dco::ifc::MessageListener`

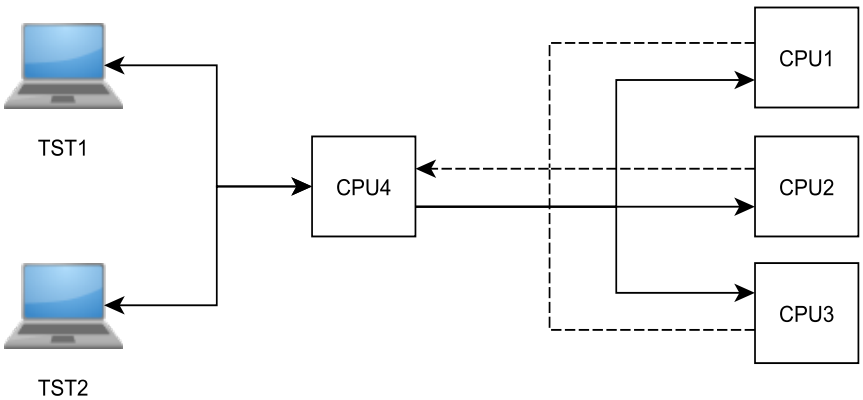
### 4.4.2 Výčtové typy

Na serveru existují tři typy pro reprezentaci typu a statusu zprávy a také chybových kódu.

Typy zpráv, resp. diagnostických protokolů, jsou popsány výčtovým typem `t_msgType`. `t_errType` pak reprezentuje chybové kódy a `t_status` status zprávy. Hodnoty těchto typů odpovídají položkám dat CLP popsaným v sekci 4.2.2.

### 4.4.3 Zámek

Následující obrázek ukazuje část zapojení komunikace procesorů z praxe. S jeho pomocí bude zdůvodněno použití zámků pro zamykání komunikace.




Obrázek 4.6: Část zapojení komunikace procesorů z praxe

Komunikační linka spojující procesor 4 s procesory 1, 2 a 3 nepodporuje identifikaci toho, kdo právě vysílá. Pokud tedy `CPU1` začne posílat data `CPU4`,

tak `CPU4` neví, od koho data přicházejí. Stejně tak `CPU2` a `CPU3` nemají informaci o tom, zda zrovna někdo odesílá data. Pokud tedy více procesorů vysílá data zároveň, dochází ke kolizi na lince a `CPU4` není schopen data přijímat ani interpretovat.

Aby tedy nedocházelo ke kolizím na takovýchto komunikačních linkách, je potřeba zamezit paralelnímu vysílání více procesorů. K tomu slouží komunikační zámky. Jejich funkce vychází z faktu, že procesor odesílá data pouze pokud má odpovědět na nějakou zprávu. Při odeslání zprávy se zamkne příslušný zámek, odesílání dat na danou komunikační linku je pak povoleno pouze pro jednoho klienta.

Zámek potřebuje pro svou funkci tyto informace:

- Seznam identifikátorů procesorů, které se mají zamykat společně. Např. podle obrázku 4.6 by jeden zámek obsahoval id 1, 2 a 3 a další pouze 4.
- Identifikátor vlastníka zámku. Jako vlastník se zde bere identifikátor spojení, které je popsáno v sekci 4.4.4.

Třída `com::smgr::LockManager::Lock` definuje rozhraní zámku. Při vytvoření je zámek odemčený. Při jeho uzamknutí dojde k nastavení časovače na nějakou nastavitelnou dobu. Pokud během této doby dojde k dalšímu uzamčení, časovač se resetuje. Pokud uběhne daný časový limit bez pokusu o zamknutí, dojde k automatickému odemčení.

Při inicializaci serveru je nejprve třeba vytvořit zámky a také jim nsatavit hodnotu časovače pomocí následujících funkcí:

```
// Vytvoří odemknutý zámek. První parametr obsahuje pole
// identifikátorů procesorů, které má tento zámek zamykat.
// Druhý parametr pak udává počet těchto identifikátorů.
Lock( const t_uint8* devId, const t_uint32 cnt );
// Nastaví doby nečinnosti zámku v milisekundách, po které
// se má automaticky odemknout.
void setTimeout( t_uint64 millis );
```

Listing 4.8: Rozhraní zámku

### 4.4.4 Spojení

Při zahájení komunikace klienta se serverem je klientovi přiřazeno spojení. Jedinou věc, kterou klient musí udělat, aby zahájil komunikaci, je poslat UDP paket se zprávou na správný port a pokud vše proběhne v pořádku, klient může bez problémů komunikovat. Důvod selhání navázání spojení může být dosažení limitu aktivních spojení na serveru, nepodoporovaný protokol nebo neznámé id zařízení.

Spojení obsahuje tyto položky:

- Identifikátor spojení
  Jednoznačný identifikátor spojení v rámci serveru. Používá se k zamykání zámků.
- Stav spojení
  V jakém stavu se spojení zrovna nachází:
  - `FREE`
    spojení se neopužívá, je možné připojení klienta
  - `RECEIVING`
    spojení přijímá první zprávu, potřebu tohoto mezistavu popisuje sekce 5.3.2
  - `ACTIVE`
    spojení je aktivní, klient používající tohle spojení má zamknutý alespoň jeden zámek
- Struktura s informacemi o IP adrese a portu klienta
  Zaručí, aby si spojení pamatovalo, komu má poslat data.
- Velikost přijatých dat
  Více informací v následující položce.
- Velikost přijímané zprávy
  Při příjmu nové zprávy si spojení z hlavičky vyčte, jak dlouhá má zpráva být. Dále si pamatuje, kolik dat už bylo přijato. Pokud byla přijata celá zpráva, tak se položka resetuje a je schopná přijmout novou zprávu. Tahle vlastnost se dá použít i k přijmutí zprávy, která dorazila ve více paketech. Spojení je také schopné detekovat chybu, kdy přijatá zpráva je delší, než bylo uvedeno v hlavičce.

Spojení je na serveru definovánu třídou `com::smgr::SessionManager::ComSession`. Tato třída definuje rozhraní spojení, které je využíváno manažerem spojení popsaného v sekci 4.4.7. Při použití knihovny není potřeba se spojeními zabývat.

### 4.4.5 Ovladač diagnostického protokolu

Třída `com::smgr::DiagProtocol` určuje, co musí splňovat ovladač diagnostického protokolu na straně serveru. Důležité jsou zejména následující metody:

```
// Jelikož CLP neví nic o struktuře dat, které přenáší, je potřeba,
// aby ovladač protokolu z těchto dat vrátil id procesoru, kterému data patří.
// K tomu slouží následující dvě metody, které mají stejnou funkci, liší se
// pouze typem parametrů.
t_uint8 getIdFromMsg( const dco::desc::t_desc* msg )const;
t_uint8 getIdFromData( const void* data )const;
// Metoda, pomocí které se ovladači oznámí, že mu přišla nová zpráva.
void notifyNewMessage( dco::desc::t_desc* msg );
```

Listing 4.9: Třída `com::smgr::Diagprotocol`

### 4.4.6 Manažer zámků

Třída `com::smgr::LockManager` implementuje manažera zámků pro jednodušší obsluhu zámků. Obsahuje v sobě vlákno, které iteruje přes jednotlivé zámky a kontroluje, zda nepřekročily dobu nečinnosti a nemělo by dojít k jejich odemknutí. Tato doba se resetuje s každým uspěšným zavoláním metody `lock` nad daným zámkem. Zámky jsou ukládany ve formě spojového seznamu. Z hlediska programátora jsou důležité následující funkce:

```
// Výchozí nastavení maximálního počtu zámků.
// Předefinováním lze změnit při překladu aplikace.
#ifndef CFG_COM_SMGR_LOCK_MGR_MAX_LOCKS
#define CFG_COM_SMGR_LOCK_MGR_MAX_LOCKS 5
#endif
// Bezparametrový konstruktor. Zámky do něj lze přidávat pomocí metody
// registerLock.
LockManager();
// Zaregistruje zámek do manažera. Tuto funkci lze použít pouze při
// inicializaci, dokud neběží vlákna, pro registraci za běhu lze použít
// metodu registerLockWhileRunning.
void registerLock( Lock* l );
// Metoda určená pro přidávání zámků za běhu ostatních vláken.
void registerLockWhileRunning( Lock* l );
```

Listing 4.10: API `com::smgr::LockManager`

Tento manažer je využíván manažerem spojení. Manažer spojení pak používá následující funkce:

```
// Pokusí se zamknout/odemknout zámek s~daným id zařízení předáváním ve druhém
// parametru. První parametr pak odpovídá identifikátoru spojení. Metody jsou
// schopné detekovat nepovolenou operaci nebo neregistrované id zařízení.
t_errType lock( const t_uint32 clientId, const t_uint8 devId );
t_errType unlock( const t_uint32 clientId, const t_uint8 devId );
// Podle identifikátoru zařízení najde a vrátí identifikátor vlastníka.
t_uint32 getClientId( const t_uint8 devId );
// Vrátí true, pokud klient s~daným identifikátorem vlastní nějaký zámek.
bool hasSomethingLocked( const t_uint32 clientId );
```

Listing 4.11: Rozhraní manažera zámků pro manažera spojení

### 4.4.7 Manažer spojení

Třída `com::smgr::SessionManager` zajišťuje logiku příjmu zpráv od klienta pomocí UDP a jejich následné zpracování. Detailnější popis těchto činností lze nalézt v sekci 5.3. Manažer spojení má následující rozhraní:

```
// výchozí nastavení maximálního počtu aktivních spojení
#ifndef CFG_COM_SMGR_MAX_NUMBER_OF_SESSIONS
#define CFG_COM_SMGR_MAX_NUMBER_OF_SESSIONS 3
#endif
// Výchozí nastavení maximálního počtu podporovaných protokolů.
#ifndef CFG_COM_SMGR_MAX_NUMBER_OF_PROTOCOLS
#define CFG_COM_SMGR_MAX_NUMBER_OF_PROTOCOLS 3
#endif
// Nastavení předchozích dvou parametrů lze předefinovat při překladu.

// Konstruktor vytvoří a připraví server. Jako parametry bere manažera
// zámků a číslo portu, na kterém má server poslouchat.
// DEFAULT_PORT je výchozí port a má hodnotu 2021.
SessionManager( LockManager* lockMgr,
                t_uint16 port = DEFAULT_PORT );
// Tuto metodu použije ovladač protokolu, pokud chce odeslat zprávu.
// Důležitými parametry jsou msgType a data. První parametr udává typ
// zprávy, resp. protokolu, druhý obsahuje data ve formě deskriptoru.
// Z hlediska ovladače protokolu nejsou zbylé dva parametry důležité.
short send( t_msgType msgType, dco::desc::t_desc* data,
            struct udp_pcb* pcb = NULL, t_uint32 timeout = 0 );
// Metoda zaregistruje ovladač diagnostického protokolu, pokud počet
// registrovaných nepřesáhl CFG_COM_SMGR_MAX_NUMBER_OF_PROTOCOLS.
void registerProtocol( DiagProtocol* prot );
// Metoda odregistruje ovladač protokolu, ten už pak nebude moci
// přijímat data.
void unregisterProtocol( DiagProtocol* prot );
```

Listing 4.12: API manažera spojení

Kapitola 5

# Implementace

Kapitola popisuje použití rozhraní definovaných v předchozí kapitole pomocí ukázky inicializace klienta i serveru. Dále jsou popsány implementační detaily uvolňování zámků i spojení. Poté následuje sekce popisující průběh komunikace mezi klientem a serverem.

## 5.1 Server

### 5.1.1 Inicializace

Ukázka použití jednotlivých funkcí při inicializaci serveru. Po provedení této metody je server připraven ke komunikaci s klienty.

```
1  void init(){
2    t_uint8 ids[] = {1,2,3}; t_uint8 ids2[] = {4};
3    const t_uint32 LOCK_TIMEOUT = 10000;
4    com::smgr::LockManager::Lock* locks[] = {
5      new com::smgr::LockManager::Lock( ids, sizeof(ids)/sizeof(*ids) ),
6      new com::smgr::LockManager::Lock( ids2, sizeof(ids2)/sizeof(*ids2) ),
7    }
8    com::smgr::LockManager* lckMgr = new com::smgr::LockManager();
9    for (t_uint32 i = 0; i < sizeof(locks)/sizeof(*locks); i++){
10     locks[i]->setTimeout( LOCK_TIMEOUT );
11     lckMgr->registerLock( locks[i] );
12   }
13   com::smgr::SessionManager* sMgr = new com::smgr::SessionManager( lckMgr );
14   com::smgr::DiagProtocol* dcpInst = new dcp::dcp::DCP( sMgr );
15   sMgr->registerProtocol( dcpInst );
16 }
```

Listing 5.13: Inicilializace serveru

Následuje stručné vysvětlení jednotlivých kroků, čísla v seznamu odpovídají číslům řádků kódu 5.13:

2 pole s jednotlivými skupinami identifikátorů procesorů

3 konstanta pro určení velikosti doby neaktivity pro zámek

4 vytvoření jednotlivých zámků, první parametr jsou seznamy identifikatorů, druhý parametr se dopočítá podle velikosti prvního

8 vytvoření prázdného manažera zámků

10 všem zámkům se nastaví doba nečinnosti potřebná pro odemknutí

11 zámky je nutné zaregistrovat do manažera zámků

13 vytvoření manažera spojení

14 vytvoření instance ovladače diagnostického protokolu, zde je použit `DCP`

15 registrace ovladačů diag. protokolů do manažera spojení

### 5.1.2 Uvolňování spojení

Uvolňování spojení probíhá na nekonečné smyčce. Server iteruje přes všechna spojení a zjišťuje jejich stav. Pokud je spojení ve stavu `ACTIVE`, zjistí si jeho identifikátor. Zavoláním metody `hasSomethingLocked` nad manažerem zámků zjistí, zda klient používající dané spojení vlastní nějaký zámek. Pokud ne, jsou nad daným spojením zavolány funkce `disconnect` a `clearMsg`. Tím dojde k uvolnění spojení pro další klienty.

### 5.1.3 Uvolňování zámků

Zámky jsou uvolňovány na nekonečné smyčce manažera zámků. Ten kontroluje, zda zamknutým zámkům nevypršela doba neaktivity a pokud ano, nastaví zámku vlastníka na hodnotu 0. Tento zámek pak přesune ze seznamu zamknutých zámků do seznamu odemknutých.

## 5.2 Klient

### 5.2.1 Inicializace

Následuje ukázka inicializace klienta. Server má IP adresu 10.0.0.151 a poslouchá na portu 2021.

```
1  public void CLPClientInit(){
2    CLPClient client = CLPClient.getInstance();
3    try {
4      client.setDestIP(InetAddress.getByName("10.0.0.151"));
5      client.setPort(2021);
6      client.connect();
7    } catch (Exception e) {
8      e.printStackTrace();
9    }
10   client.registerDiagProtocol( new DCP() );
11   client.registerErrorListener( new ErrorListener() );
12 }
```

Listing 5.14: Inicilializace klienta

Následuje stručné vysvětlení jednotlivých kroků, čísla v seznamu odpovídají číslům řádků kódu 5.14:

2 v aplikaci může existovat pouze jedna instance ovladače CLP, tzv. singleton[10]

4 nastavení IP adresy serveru

5 nastavení portu serveru

6 připojení na server

8 ošetření výjimek

10 vytvoření a registrace ovladače `DCP`

11 registrace posluchače chyb, nepovinné

## 5.3 Průběh komunikace

Sekce popisuje průběh komunikace z hlediska implementace. Tento průběh je rozdělen do čtyř kroků. Jako diagnostický protokol byl použit DCP.

### 5.3.1 Klient posílá zprávu

Klient se rozhodne komunikovat s nějakým procesorem. Přes uživatelské rozhraní spustí přenos DCP zpráv. V tu chvíli ovladač DCP začne generovat příslušnou zprávu `msg`, kterou předá do ovladače CLP zavoláním metody

```
writeMessage( MsgType.DCP, msg );
```

Listing 5.15: Předání zprávy z DCP do CLP(4.3)

V této metodě se zprávě přidá hlavička, tzn. délka a typ zprávy. Zpráva je pak včetně této hlavičky zařazena do fronty k odeslání a posléze odeslána na odesílacím vláknu.

### 5.3.2 Server přijímá zprávu

Server přijme zprávu. Nejprve se podívá, zda její odesílatel nemá na serveru aktivní spojení. Pokud ano, pokračuje se dál. Pokud ne, server zkusí nalézt volné spojení. Když volné spojení nenalezl, odešle zprávu s chybou `MAX_CON_REACHED` (dosaženo maximálního počtu aktivních spojení). Pokud volné spojení nalezl, naplní strukturu s daty o odesílateli (IP adresu a port) a změní stav spojení z `FREE` na `RECEIVING`.

Pokud by se při příjmu první zprávy nastavil stav spojení na `ACTIVE`, mohlo by dojít k jeho uvolnění dřív, než by se stihl zamknout zámek. Proto při příjmu první zprávy nastaví stav spojení na `RECEIVING`. Po přijetí celé první zprávy máme zaručeno, že klient už má zamklý nějaký zámek, tudíž můžeme nastavit stav spojení na `ACTIVE`.

Po úspěšném přidělení spojení dojde ke kontrole přijetí celé zprávy pomocí metody `isMsgComplete`. Pokud byla přijata celá zpráva, následuje pokus o nalezení ovladače DCP. Pokud nebyl ovladač nalezen, server odešle zprávu s chybou `UNSUPPORTED_DIAG_PROTOCOL` (nepodporovaný protokol).

Z ovladače se pak pomocí metody `getIdFromMsg` získá id procesoru, kterému zpráva patří. Následuje pokus o zamčení zámku pomocí metody `lock`. Tato metoda může vrátit jednu ze tří následujících hodnot:

- `NONE`, pokud klient může odemknout/zamknout daný zámek nebo jde o odemknutí zámku, který není zamknutý, což se nepovažuje za chybu.
- `DEVICE_NOT_FOUND`, pokud žádný zámek neobsahuje dané id procesoru. Tato chyba se okamžitě pošle klientovi.

- `OPERATION_DENIED`, pokud chceme odemknou nebo zamknout zámek, který momentálně vlastní někdo jiný. Tato chyba se také okamžitě pošle klientovi.

Pokud tedy operace zamknutí skončí bez chyby, data se předají ovladači DCP pomocí metody `notifyNewMessage`.

### 5.3.3 Server posílá odpověď

Vyhodnocení zprávy z předchozích dvou kroků způsobila odeslání odpovědi. Tato odpověď `rsp` je z DCP předána pomocí zavoláním funkce

```
send( en_msgType_DCP, rsp );
```

Listing 5.16: Předání zprávy z DCP do CLP na serveru(4.12)

Ve funkci se ke zprávě přidá hlavička se všemi náležitostmi, které jsou potřeba (viz 4.2.2). Ze zprávy se vyčte id procesoru, který ji posílá pomocí metody `getIdFromMsg`. Tohle id se použije pro nalezení id spojení pomocí metody `getClientId`. Podle tohoto id se najde přislušné spojení, ze kterého se získá struktura s informacemi o klientovi pomocí funkce `getPcb`. Tato struktura se použije jako parametr funkce použité proodesílání dat po UDP.

### 5.3.4 Klient přijímá odpověď

Ovladač CLP přijme zprávu. Z hlavičky zprávy se vyčtou data, mj. také typ zprávy a status. Pokud se status zprávy rovná `SUCCES`4.3.4, data se přepošlou odpovídajícímu ovladači pomocí metody `notifyNewMsg`. Pokud došla chybová odpověď, ovladači DCP se nic neposílá. Ať už byl status jakýkoli, proběhne notifikace všech posluchačů chyb pomocí metody `notifyError`.

Kapitola 6

# Testování

Kapitola se zabývá testováním implementace knihovny. Popisuje nastavení serveru, testovací protokol a strukturu výpisu testování. Poslední sekce se zabývá jednotlivými testovacími scénáři a jejich vyhodnocení.

## 6.1 Testovací prostředí

### 6.1.1 Testovací protokol

Pro testování byl použit protokol `LOOP_BACK`. Data tohoto protokolu mají následující strukturu:

| 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|
| CPU_ID | 0xA | 0xB | 0xC | 0xD | 0xE |

Obrázek 6.1: Struktura dat protokolu `LOOP_BACK`

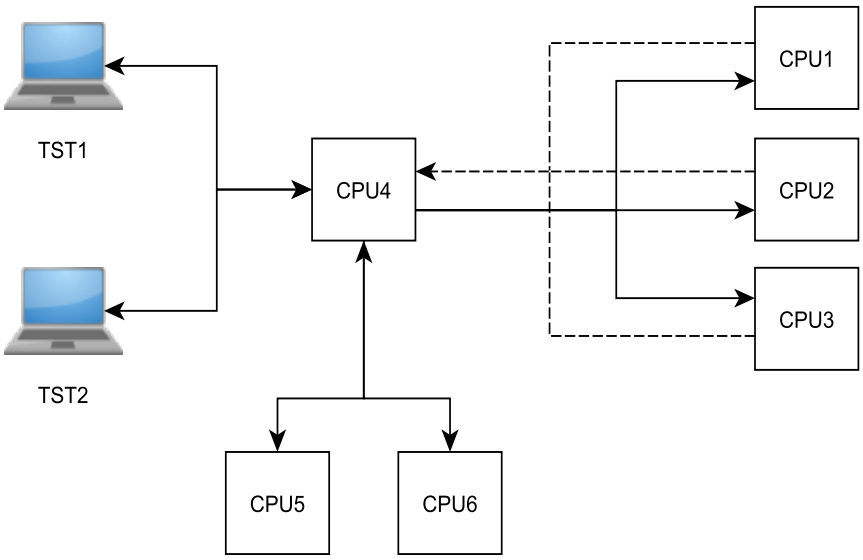
- `CPU_ID` slouží k identifikaci procesoru, kterému patří zpráva
- následujících pět bajtů má konstatní hodnotu

Princip tohoto protokolu je jednoduchý. Všechny zprávy, které přijme, odešle zpět klientovi. Číslený identifikátor tohoto protokolu je roven hodnotě `0xFF`.

### 6.1.2 Nastavení serveru

Testování probíhá na jednom procesoru. Výsledky testování nejsou touto skutečností nijak ovlivněny. V procesoru budou vytvořeny zámky, které budou

simulovat zapojení ostatních procesorů podle obrázku 6.2. Ovladač testovacího protokolu bude simulovat komunikaci s procesory.




Obrázek 6.2: Zapojení procesorů pro testování

Nastavení serveru bude tedy následující:

- maximální počet aktivních spojení omezen na dva
- existují tři zámky s následujícími identifikátory:
  a) 1, 2, 3
  b) 4
  c) 5, 6
- doba neaktivity zámků nastavena na tři vteřiny
- maximální počet podporovaných protokolů nastaven na tři
- maximální počet zámků nastaven na pět
- pouze jeden registrovaný diagnostický protokol `LOOP_BACK`

### 6.1.3 Struktura položek zápisu testování

Průběh testování se zapisuje do souboru. Jeden řádek reprezentuje jednu operaci. Struktura jednoho řádku pro operaci odeslání pak bude následující:

```
S: DATE TIME CLI_ID CPU_ID DATA
```

Listing 6.1: Struktura položky ideslání dat v zápisu testování

Následuje popis jednotlivých položek výpisu:

**S** označuje operaci odeslání dat

**DATE** datum ve formátu den.měsíc.rok, např. 04.05.2017

**TIME** čas ve formátu hodiny:minuty:vteřiny.milisekundy, např. 09:54:13.845

**CLI_ID** id klienta, aplikace bude používat id 1, 2 a 3

**CPU_ID** id procesoru, kterému klient posílá data

**DATA** odeslaná data, budou mít vždy hodnotu `0xa 0xb 0xc 0xd 0xe`

Struktura řádku pro zápis operace příjmu dat je podobná:

```
R: DATE TIME CLI_ID CPU_ID STATUS DATA
```

Listing 6.2: Struktura položky příjmu dat v zápisu testování

Následuje popis jednotlivých položek výpisu. Položky, které zde nejsou vyjmenovány mají stejný význam jako u předchozího případu:

**R** označuje operaci přijmutí dat

**CPU_ID** id procesoru, od kterého přišla data

**STATUS** může se rovnat dvěma hodnotám: `OK` a `ER`

**DATA** přijatá data, budou mít hodnotu `0xa 0xb 0xc 0xd 0xe`, pokud položka status bude `OK`, jinak se nahradí textovým popisem chyby

## 6.2 Testovací scénáře

Pro testování je navrženo celkem pět scénářů, jejichž kombinací lze dostat reálný případ komunikace. Scénáře jsou na sobě nezávislé, tzn. není potřeba testovat jejich kombinace.

### 6.2.1 Operace zamítnuta

Scénář testuje zamykání a odemykání jednotlivých zámků. Klient 1 nejprve pošle data procesoru 1, to způsobí zamknutí procesorů 1, 2 a 3 na serveru. Dále se klient 2 pokusí komunikovat s procesorem 2, ve výpisu by se měla objevit chyba „Operace zamítnuta“. Po pár vteřinách se vypne komunikace klienta 1. Po třech vteřinách by mělo dojít k uvolnění zámku procesoru 1 a tím tedy i procesorů 2 a 3 a následně k zabrání tohoto zámku klientem 2. Ve výpisu by se chyba „Operace zamítnuta“ měla nahradit přijatými daty.

```
S: 06.05.2017 20:28:34.761 1 1 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:34.771 1 1 OK 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 20:28:35.271 1 1 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:35.282 1 1 OK 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 20:28:35.599 2 2 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:35.700 2 2 ER Operace zamitnuta
S: 06.05.2017 20:28:35.782 1 1 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:35.792 1 1 OK 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 20:28:36.200 2 2 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:36.210 2 2 ER Operace zamitnuta
S: 06.05.2017 20:28:36.293 1 1 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:36.304 1 1 OK 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 20:28:36.711 2 2 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:36.721 2 2 ER Operace zamitnuta
S: 06.05.2017 20:28:37.221 2 2 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:37.231 2 2 ER Operace zamitnuta
S: 06.05.2017 20:28:37.731 2 2 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:37.741 2 2 ER Operace zamitnuta
S: 06.05.2017 20:28:38.242 2 2 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:38.352 2 2 ER Operace zamitnuta
S: 06.05.2017 20:28:38.852 2 2 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:38.862 2 2 ER Operace zamitnuta
S: 06.05.2017 20:28:39.363 2 2 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:39.473 2 2 OK 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 20:28:39.976 2 2 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:40.076 2 2 OK 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 20:28:40.576 2 2 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 20:28:40.586 2 2 OK 0xa 0xb 0xc 0xd 0xe
```

Listing 6.3: Část výpisu testovacího scénáře 1

Z výpisu je patrné, že server dokáže rozpoznat naplatnou operaci, resp. dokáže vyhodnotit situaci, kdy klient 2 chce komunikovat s procesorem ve skupině, která je obsazena klientem 1. Test neplatné operace lze vyhodnotit jako úspěšný.

Podle časových značek lze určit, že tři vteřiny poté, co klient 1 přestal posílat data, klient 2 už nedostává chybu a může komunikovat s procesorem 2. Z toho vyplývá, že zámek se automaticky uvolnil. Test uvolňování zámků lze vyhodnotit jako úspěšný.

Výsledky tohoto testu také ověřují zamykání celé skupiny procesorů.

### 6.2.2 Překročení maximálního počtu aktivních spojení

Scénář testuje správné vyhodnocení překročení maximálního počtu aktivních spojení na serveru. Nejprve pomocí klientů s id 1 a 2 posílá data na server, ten jim přiřadí dvě spojení. Následně se pokusí odeslat data pomocí klienta 3, ale jako odpověď by měl dostat „Překročen maximální počet spojení“. Tento scénář také testuje, zda se dva různí klienté mohou připojit ke dvěma různým skupinám procesorů najednou.

```
S: 06.05.2017 19:59:23.507 2 4 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 19:59:23.521 1 1 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:23.527 2 4 OK 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:23.533 1 1 OK 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 19:59:24.028 2 4 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 19:59:24.032 3 5 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 19:59:24.034 1 1 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:24.128 2 4 OK 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:24.133 3 5 ER Prekrocen maximalni pocet spojeni
R: 06.05.2017 19:59:24.144 1 1 OK 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 19:59:24.628 2 4 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 19:59:24.634 3 5 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:24.638 2 4 OK 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 19:59:24.644 1 1 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:24.645 3 5 ER Prekrocen maximalni pocet spojeni
```

Listing 6.4: Část výpisu testovacího scénáře 2

Z výpisu je patrné, že na serveru mají klienti 1 a 2 přiřazená spojení a komunikace běží. Klient 3 dostává chybovou hlášku o překročení maximálního počtu spojení. Z výpisu je také jasné, že komunikace více klientů s více skupinami procesorů funguje. Test tedy lze vyhodnotit jako úspěšný.

### 6.2.3 Nepodporovaný protokol

Scénář testuje správné vyhodnocování nepodporovaného protokolu. Testovací aplikace pomocí klienta 1 odešle pár zpráv, kde položka `MSG_TYPE` bude nastavena na hodnotu `DCP`. Ovladač DCP není registrován, tudíž by mělo dojít k vyhodnocení chyby. Ve výpise testování by se měla objevit chyba „Nepodporovaný protokol“.

```
S: 06.05.2017 19:59:32.634 1 1 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:32.734 1 1 ER Nepodporovany protokol
S: 06.05.2017 19:59:33.234 1 1 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:33.244 1 1 ER Nepodporovany protokol
S: 06.05.2017 19:59:33.744 1 1 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:33.754 1 1 ER Nepodporovany protokol
```

Listing 6.5: Část výpisu testovacího scénáře 3

Z výpisu je patrné, že server nepodporuje/nezná protokol `DCP`. Test tedy lze vyhodnotit jako úspěšný.

### 6.2.4 Procesor nenalezen

Scénář testuje správné vyhodnocování nalezených id. Aplikace odešle pomocí klienta 1 data procesoru, který není na serveru zaregistrován. V odpovědi by se měla ukázat chyba „Zařízení nenalezeno“.

```
S: 06.05.2017 19:59:40.479 1 73 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:40.581 1 73 ER Zarizeni nenalezeno
S: 06.05.2017 19:59:41.082 1 5 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:41.192 1 5 OK 0xa 0xb 0xc 0xd 0xe
S: 06.05.2017 19:59:41.692 1 -51 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:41.702 1 -51 ER Zarizeni nenalezeno
S: 06.05.2017 19:59:42.202 1 84 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:42.312 1 84 ER Zarizeni nenalezeno
S: 06.05.2017 19:59:42.812 1 -11 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:42.823 1 -11 ER Zarizeni nenalezeno
S: 06.05.2017 19:59:43.324 1 18 0xa 0xb 0xc 0xd 0xe
R: 06.05.2017 19:59:43.424 1 18 ER Zarizeni nenalezeno
```

Listing 6.6: Část výpisu testovacího scénáře 4

Z výpisu je patrné, že server dokáže vyhodnotit neexistující id procesoru. Na řádcích 3 a 4 lze vidět, že komunikace s procesorem 5 funguje, zatímco ostatní náhodná čísla server nerozpozná. Test lze vyhodnotit jako úspěšný.

### 6.2.5 Zpráva ve více paketech

Scénář testuje schopnost serveru přijímat zprávy rozdělené do více paketů. Apliakce odešle zprávu procesoru 4 rozdělenou na několik částí. Po odeslání všech částí zprávy by měla aplikace dostat odpověď.

```
S: 09.05.2017 11:15:49.804 1 4 0xe
R: 09.05.2017 11:15:50.305 1 4 OK 0xa 0xb 0xc 0xd 0xe
S: 09.05.2017 11:15:50.805 1 4 0xa
S: 09.05.2017 11:15:51.307 1 4 0xb
S: 09.05.2017 11:15:51.808 1 4 0xc
```

```
S: 09.05.2017 11:15:52.309 1 4 0xd
S: 09.05.2017 11:15:52.810 1 4 0xe
R: 09.05.2017 11:15:53.311 1 4 OK 0xa 0xb 0xc 0xd 0xe
```

Listing 6.7: Část výpisu testovacího scénáře 5

Výpis ukazuje odesílání zprávy po jednom bajtu každou půlvteřinu. Po odeslání celé zprávy server pošle odpověď. Z toho vyplývá, že server dokáže přijímat zprávy rozdělené do více paketů.

# Závěr

Cílem práce bylo vytvořit knihovnu funkcí pro podporu přenosu diagnostického protokolu pomocí UDP. Knihovna měla být rozdělena na dvě části: klient a server. Klient měl být implementován v jazyce Java pro jednoduchou integraci do firemní diagnostické aplikace a server v C++ pro použití v mikrokontrolerech.

Knihovna měla také podporovat paralelní připojení klientů k serveru a také zachovat integritu komunikace mezi mikrokontrolery v rámci systému. Pro vyřešení tohoto problému byl také navržen protokol CLP, na kterém je knihovna postavena.

Vytvořená knihovna splňuje všechny definované požadavky. Je postavena na UDP, umožňuje přenášet více než jeden diagnostický protokol a také paralelní komunikaci více klientů se serverem. Její použití v jiných aplikacích je velmi jednoduché.

Na knihovně i CLP budu v rámci plnění zaměstnaneckých povinností dále pracovat. Jenda z plánovaných úprav je dynamické vytváření zámků pomocí speciálního typu zprávy. Tato funkcionalita se použije v systémech, kde není předem určeno zapojení mikrokontrolerů.

Další možná úprava spočívá v oddělení ovladače UDP od manažera spojení. To by zaručilo jednoduchou výměnu typu komunikace mezi serverem a klientem.

# Literatura


[1] Tanenbaum, A. S.; Wetherhall, D. J.: *Computer Networks [online]*. Prentice Hall, páté vydání, 2011, [cit. 25.4.2017]. Dostupné z: https://montcs.bloomu.edu/Readings/Computer%20Networks%20-%20A%20Tanenbaum%20-%205th%20edition.pdf

[2] TECH-FAQ: The OSI Model – What It Is; Why It Matters; Why It Doesn't Matter. [online]. [cit. 1. 5. 2017]. Dostupné z: http://www.tech-faq.com/osi-model.html

[3] Postel, J.: Transmission Control Protocol [online]. RFC 793, RFC Editor, Fremont, CA, USA, Září 1981, doi:10.17487/RFC0793, [cit. 27. 4. 2017]. Dostupné z: https://www.rfc-editor.org/rfc/rfc793.txt

[4] Postel, J.: User Datagram Protocol [online]. RFC 768 (Internet Standard), Srpen 1980, doi:10.17487/RFC0768. Dostupné z: https://www.rfc-editor.org/rfc/rfc768.txt

[5] Hoffman, C.: What's the Difference Between TCP and UDP? [online]. 2014, [cit. 25.4.2017]. Dostupné z: https://www.howtogeek.com/190014/htg-explains-what-is-the-difference-between-tcp-and-udp/

[6] Softing Automotive: DoIP, Diagnostic Communication over Internet Protocol [online]. [cit. 23.4.2017]. Dostupné z: https://automotive.softing.com/fileadmin/sof-files/pdf/de/ae/poster/DoIP_faltblatt_softing.pdf

[7] FreeOpcUa: Open Source C++ OPC-UA Server and Client Librarys and Tools [online]. [cit. 11.5.2017]. Dostupné z: https://freeopcua.github.io/

[8] ISO/IEC: Information technology — Advanced Message Queuing Protocol (AMQP) v1.0 specification [online]. Technická zpráva, ISO/IEC,

2014, [cit. 2. 5. 2017]. Dostupné z: http://standards.iso.org/ittf/PubliclyAvailableStandards/c064955_ISO_IEC_19464_2014.zip

[9] Free Software Foundation: lwIP, Lightweight TCP/IP stack [online]. [cit. 24.4.2017]. Dostupné z: https://savannah.nongnu.org/projects/lwip/

[10] Techopedia: Singleton [online]. [cit. 4. 5. 2017]. Dostupné z: https://www.techopedia.com/definition/15830/singleton

Příloha **A**

# Seznam použitých zkratek

**AMQP** Advanced Message Queueing Protocol

**API** Application Program Interface

**CLP** Communication Locking Protocol

**CAN** Controller Area Network

**CPU** Central Processing Unit

**DCP** Diagnostic Control Protocol

**DNS** Domain Name System

**DoIP** Diagnostic over Internet Protocol

**HTTP** Hypertext Transfer Protocol

**IoT** Internet of Things

**IP** Internet Protocol

**OBD-II** On-Board Diagnostics II

**OPC** Open Platform Communication

**OSI** Open Systems Interconnection

**TCP** Transmission Control Protocol

**TST** Tuning & Service Tool

**UDP** User Datagram Protocol

**UDS** Unified Diagnostic Services

Příloha **B**

# Obsah přiloženého CD

- `sources`
  - `client` ..... zdrojové kódy implementace klienta
  - `server` ..... zdrojové kódy implementace serveru
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `chapters` ..... zdrojové formy kapitol práce ve formátu LaTeX
- `text` ..... text práce
  - `BP_mikusik_jaromir.pdf` ..... text práce ve formátu PDF