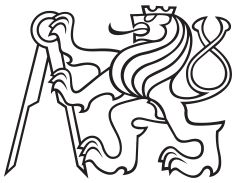
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra měření

# Diagnostika systému White Rabbit

**Filip Tefr**

Vedoucí: doc. Ing. Jaroslav Roztočil, CSc.
Konzultant: Ing. Vojtěch Vigner, Ph.D.
Studijní program: Kybernetika a robotika
Studijní obor: Kybernetika a robotika
Květen 2021

# Poděkování

Chtěl bych poděkovat panu docentovi Jaroslavovi Roztočilovi za odborné vedení práce a cenné rady, které mi pomohly tuto práci zkompletovat. Mé poděkováni patří též panu ing. Vojtěchovi Vignerovi za spolupráci a konzultace při zpracování praktické části práce.


# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 21. května 2021

# Abstrakt

Cílem této práce práce bylo navrhnout a vytvořit automatický diagnostický systém pro White Rabbit zařízení, který uživateli usnadní detekci a opravu chyb. Pro vývoj systému byl použit jazyk C++ spolu s QT frameworkem. Výsledný systém je složen ze dvou částí. Serverová aplikace poskytuje možnost vzdáleného monitorování zařízení pomocí REST API a provádí automatickou kontrolu všech připojených WR zařízení. Chování aplikace je možné dynamicky nastavit v konfiguračním souboru. Druhou částí systému je grafická aplikace, která slouží pro vzdálené monitorování WR zařízení připojených k serveru. Diagnostický systém byl otestován v laboratoři přesného času a frekvence FEL ČVUT. Výsledkem testů je fakt, že systém je plně funkční a je připraven k reálnému nasazení.



**Klíčová slova:** Automatická diagnostika, White Rabbit, EtherBone, WR-LEN, C++, QT framework



**Vedoucí:** doc. Ing. Jaroslav Roztočil, CSc.
Fakulta elektrotechnická
Technická 2
166 27 Praha 6 - Dejvice



# Abstract

This thesis aims to design and develop automatic diagnostic system for White Rabbit devices, which helps user to detect and repair system failures. The system was developed in C++ language with a help of QT framework. Final system consists of two parts. Server application provides a possibility of remote connection to local network and performs automatic detection of failures of White Rabbit devices. Behavior of application is fully configurable in configuration file. The second part of system is graphical application which is used for remote monitoring of White Rabbit devices. Diagnostic system was tested in laboratory of precise time and frequency at FEE CTU. The test results shows that final system is fully functional and is ready to be deployed in real operation.



**Keywords:** Automatic diagnostics, White Rabbit, EtherBone, WR-LEN, C++, QT framework

**Title translation:** White Rabbit system diagnostics

# Obsah

# Obrázky



# Tabulky

# Kapitola 1
# Úvod

V posledních letech můžeme vidět velký vývoj v oblasti distribuovaných systémů, které se používají čím dál častěji než centralizované systémy. Tyto systémy mají mnohem lepší škálovatelnost a budoucí rozšířitelnost.

Společně s nástupem distribuovaných systémů je však třeba vyřešit mnoho problémů, které rozdělení systémů přináší. Mezi ty nejdůležitější patří komunikace mezi jednotlivými částmi systému, její odezva a také problémy s časovou synchronizací, protože každá část systému používá vlastní referenční hodinový signál. Právě tématu časové synchronizace v distribuovaných systémech se budeme věnovat v této práci.

K časové synchronizaci v distribuovaných systémech lze použít mnoho protokolů, mezi ty nejznámější patří například PTP nebo synchronní ethernet (synchronizuje jen frekvenci). Tyto protokoly nabízejí přesnost synchronizace kolem jedné mikrosekundy, která v mnoha případech nemusí být dostatečná. Proto vznikl projekt White Rabbit, jehož součástí jsou vědecké laboratoře a univerzity z celého světa. Cílem projektu je vytvořit systém, který nahradí existující synchronizační systém v urychlovači částic v CERNu. Výstupem projektu je systém, který kombinací různých synchronizačních protokolů nabízí přesnost synchronizace na méně než jednu nanosekundu na vzdálenosti desítky kilometrů. Jelikož specifikace projektu jsou volně dostupné, je možné výsledky použit v jiných oblastech, např v průmyslu.

Protože systém White Rabbit může být často použit v rozsáhlých sítích, je jeho diagnostika a kontrola časově velice náročná. V případě chyby je nutné zkontrolovat mnoho prvků sítě, které se mohou nacházet na velkých územích. Právě pro vzdálené připojení do jednotlivých White Rabbit zařízení lze v mnoha případech použít protokol EtherBone. Cílem této práce je s pomocí tohoto protokolu vytvořit diagnostický systém pro sítě White Rabbit. Součástí systému bude možnost vzdáleného monitorování všech zařízení v sítí, a zejména také systém automatické kontroly. Aplikace bude v pravidelných intervalech zcela autonomně kontrolovat stav White Rabbit zařízení. Pokud bude detekována chyba, systém automaticky upozorní uživatele emailem nebo pomocí SMS.

Výsledný diagnostický systém bude nasazen v laboratoři přesného času a frekvence na FEL ČVUT, kde bude zapojen do již existující sítě s césiovými atomovými hodinami, které se mimo jiné podílejí na tvorbě mezinárodního

atomového času. Úkolem bude otestovat funkčnost diagnostického systému v reálných podmínkách a zjistit, zda nenarušuje běžnou časovou synchronizaci.

Celá práce je rozdělena do tří částí. V kapitole 2 jsou vysvětleny teoretické znalosti, které jsou potřeba k vývoji diagnostického systému. Je zde podrobně popsán projekt White Rabbit a protokol EtherBone. V kapitole 3 je popsán vlastní vývoj diagnostického systému. V poslední části práce (kapitola 4) jsou uvedeny výsledky práce a testování systému v reálných podmínkách.

# Kapitola 2
# Teoretická část

V této kapitole budou vysvětleny teoretické předpoklady, které jsou nutné pro vývoj diagnostického systému. V sekci 2.1 je popsán systém White Rabbit, který bude aplikace kontrolovat. V sekci 2.3 je popsán protokol EtherBone, který slouží pro vzdálenou komunikaci s White Rabbit zařízeními a v aplikaci bude použit pro kontrolu jejich stavu.

## 2.1 Projekt White Rabbit

Projekt White Rabbit [3] je mezinárodní spolupráce laboratoří za účelem vývoje nové technologie pro řízení a časovou synchronizaci měřících přístrojů. Cílem tohoto projektu je vytvoření sítě na bázi Ethernetu pro distribuci přesného času. Cílem původního řešení bylo nahradit systém General Machine Timing v CERNu, který vyžaduje připojení více než 2000 koncových zařízení a přesnost synchronizace méně než jedna nanosekunda a determiničnost paketů. Nicméně systém lze použít i v jiných aplikacích v průmyslu nebo v telekomunikacích.

White Rabbit je založen na již existujících standardech, konkrétně:

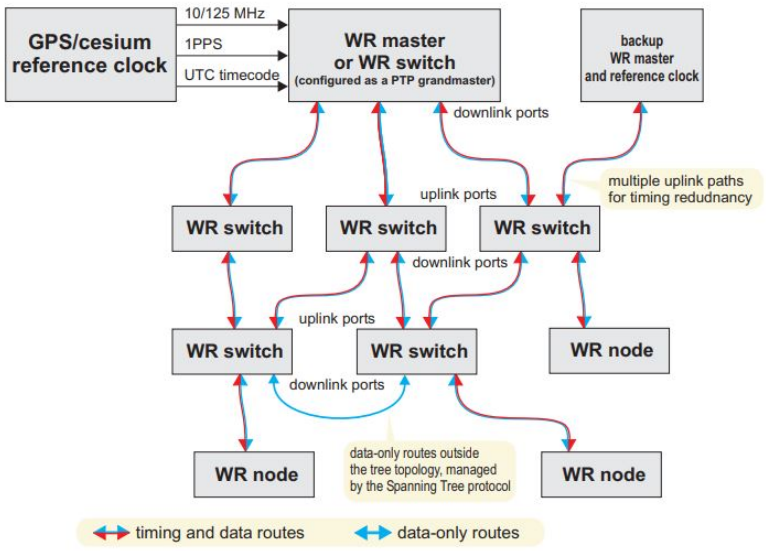
- Ethernet (IEEE 802.3)
- Synchronní Ethernet (SyncE [6])
- PTP (IEEE1588-2008 [4])

Tato kombinace technologií umožňuje plně nahradit původní systém tím, že umožňuje časovou synchronizaci v rozsáhlých distribuovaných systémech s tisíci zařízeními na vzdálenosti desítek kilometrů. Celý systém je vydán pod open-source licencí, takže je možné koupit White Rabbit kontabilní zařízení od mnoha komerčních výrobců.

Síť White Rabbit (WRN) se skládá ze dvou typů zařízení:

- WR switch - rozvětvení sítě, stejná funkce jako u ethernetu
- WR node - koncová zařízení

Tyto prvky mohou být propojeny metalickým nebo optickým vedením, nicméně vývoj se soustředí především na použití optiky, která poskytuje lepší




**Obrázek 2.1:** White Rabbit Network (převzato z [8])

vlastnosti. Ukázku celé sítě WRN můžete vidět na obrázku 2.1. V tomto příkladu jsou pro zdroj přesného času použit signál GPS, v přesnějších aplikacích ze častěji používají atomové hodiny. Tento signály je přiveden do WR master, který je nakonfigurován jako PTP grandmaster. Tento node slouží jako master WRN a poskytuje synchronizaci ostatním zařízením v sítí. Také vidíme, že je tento node pro případ poruchy zdvojen.

V WRN se můžou vyskytovat dva typy paketů:

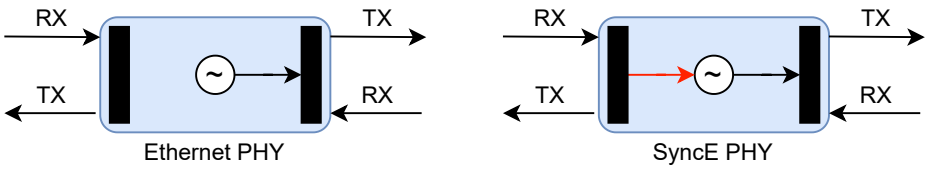
- Časování - přenos přesného času a frekvence
- Data - přenos ostatních dat

Z toho plyne, že celá síť umožňuje jedním vedením vést informaci o časové synchronizaci a také standardní ethernetové rámce, které mohou sloužit k přenosu dalších informací.

Přesnější popis protokolů a jejich propojení bude uveden v dalších kapitolách.

### 2.1.1 Synchronní Ethernet

Synchronní Ethernet (SyncE [6]) je rozšíření klasického Ethernetu (IEEE 802.3), které dovoluje mimo standardních paketů distribuovat v síti také synchronizační signál se stejnou frekvencí (WRN používá frekvenci 125 MHz). Protokol je založen na Master-Slave architektuře, kde Master frekvenci generuje, ostatní zařízení ho pak přijímají. Při synchronizaci však dochází k posunu fáze signálu, jelikož není žádným způsobem kompenzováno zpoždění vznikající přenosem signálu. SyncE žádným způsobem nezasahuje do přenosu standardních paketů, synchronizační signál je zakódován v ethernetovém rámci, a proto není datová propustnost sítě nijak omezena.




**Obrázek 2.2:** Rozdíl mezi ethernetem a SyncE

Každé zařízení připojené do sítě ethernet obsahuje vlastní oscilátor, který používá ke generování rámce. Zařízení rámec přijme, dekóduje ho, a odvysílá dalším zařízením. Pro časování je použit interní oscilátor. Neexistuje tedy žádná vazba mezi příjmem signálu a interním oscilátorem. Ten se používá jen pro generování signálu.

Jak již bylo řečeno, master zařízení přidává do ethernetového rámce informaci o frekvenční synchronizaci. Slave tento rámec přijme a upraví frekvenci svého interního oscilátoru tak, aby byla stejná jako u mastera. Tato skutečnost je znázorněna na obrázku 2.2. Celá procedura frekvenční synchronizace probíhá následovně:

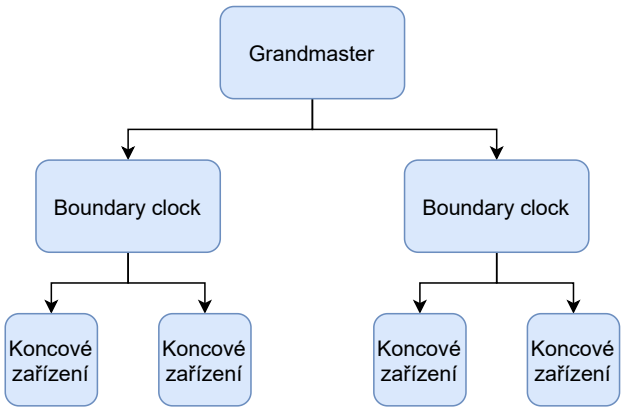
1. Master použije svůj interní oscilátor a zákóduje jeho frekvenci do ethernetovského rámce
2. Master posílá paket
3. Slave přijímá paket
4. Slave dekóduje frekvenční signál, který je následně vyčištěn fázovým závěsem
5. Slave upraví frekvenci svého interního oscilátoru dle frekvence mastera

Mezi hlavní výhodu tohoto mechanismu patří fakt, že běžný provoz v rámci sítě není synchronním ethernethem nijak omezen, pakety se mohou pohybovat maximální možnou rychlostí. Nevýhodou je, že tento protokol žádným způsobem nekompenzuje zpoždění, které vzniká při přenosu signálu. Synchronizovanou frekvenci tento způsob neovlivní, nicméně mezi oscilátory jednotlivých prvků vzniká fázové zpoždění. Pro synchronizaci fáze se musí ve WRN použít jiný mechanismus, který bude popsán v dalších kapitolách.

### 2.1.2 Precision Time Protocol

PTP (IEEE1588 [4]) je mezinárodní standard sloužící pro časovou synchronizaci v počítačových sítích. Pomocí toho protokolu lze zjistit časový offset zařízení oproti referenci. V lokálních sítích je možné dosáhnout přesnosti na méně než jednu mikrosekundu. Na rozdíl od jiných časových protokolů je podpora PTP většinou podporována v hardwaru zařízení, což umožňuje přesnější synchronizaci.

PTP pracuje s hierarchií Master-Slave. Na vrcholu stromu se nacházejí tzv. grandmaster clock, které jsou zdrojem přesného času a distribuují tento čas




**Obrázek 2.3:** Zapojení zařízení v PTP síti

dalším částem sítě. V těchto částech se mohou nacházet tzv. boundary clocks, které slouží jako most mezi grandmasterem a dalšími prvky v síti. Boundary clocks jsou lokálním zdrojem přesného času. Ukázku možné konfigurace sítě můžete vidět na obrázku 2.3.

V základní verzi protokolu (IEEE-2002) master vysílá v pravidelných intervalech zprávy s časovou značkou, které klienti přijímají. Pomocí několika dalších zpráv je vypočítán offset časové stupnice mezi masterem a slavevem.

Časová synchronizace se skládá ze čtyř částí:

1. Master odešle zprávu *Sync* v čase $t_1$
2. Slave přijímá zprávu *Sync* v čase $t_2$
3. Slave posílá v čase $t_3$ zprávu *Delay_Req*
4. Master přijme zprávu v čase $t_4$, tento čas odešle slaveovi ve zprávě *Delay_Resp*

S využitím těchto časů lze vypočítat offset časových stupnic jako

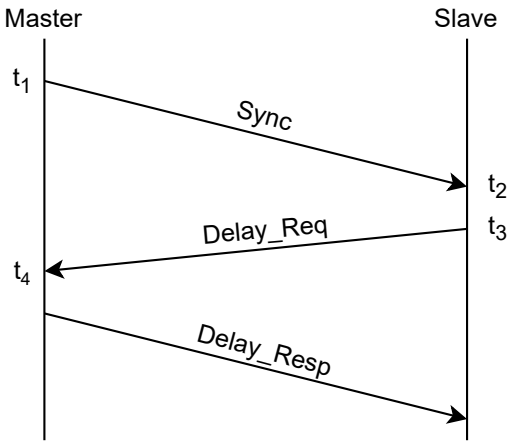
$$o = t_2 - t_1 + \delta \tag{2.1}$$

kde $\delta$ je zpoždění cesty získané vzorcem

$$\delta = \frac{(t_4 - t_1) - (t_3 - t_2)}{2} \tag{2.2}$$

Grafickou ukázku celé transakce zpráv můžete vidět na obrázku 2.4.

Tento způsob synchronizace předpokládá hvězdicové zapojení sítě (obrázek 2.3). V praxi, zejména v průmyslu, se často používají i jiné topologie (např. lineární). Nevýhoda toho mechanismu (Boundary Clock) je, že očekává konstantní zpoždění trasy a nebere v úvahu variabilní zpoždění v jednotlivých síťových prvcích (např. switch). Proto byl standard v nové revizi rozšířen (IEEE-2008) o tzv. Transparent Clock. Toto zařízení měří čas zpracování jednotlivých zpráv a přidává ho k původní časové značce. Může tak kompenzovat variabilní zpoždění vznikající zpracováním paketu. Nicméně tento koncept




**Obrázek 2.4:** Systém zpráv PTP protokolu

narušuje princip standardního průchodu paketu prvkem, jelikož zařízení do něj zasahuje a modifikuje jeho strukturu. Tento problém je řešen v novějších revizích normy.

### 2.1.3 White Rabbit extension to PTP

Standardní PTP má několik problémů, které White Rabbit musí řešit, aby se dosáhlo požadovaných vlastností. PTP má limitovanou přesnost časových značek a také nebere v úvahu asymetrii zpoždění fyzické vrstvy. Některé zdroje asymetrie se dají vyřešit vhodným návrhem hardwaru, jiné mají ale původ ve vlastnostech použitého fyzického média (různá rychlost světla v optickém vlákně, různá délka tras na DPS). Tyto nesymetrie musí být odstraněny důmyslnou kalibrací.

Pro vyřešení těchto problémů byl vyvinut protokol WRPTP (White Rabbit extension for PTP [8]). Tento protokol rozšiřuje standardní PTP. Před samotnou synchronizaci je vložena fáze WR Link Setup, která má za úkol identifikaci WR Node (Master, Slave), syntonizaci, výpočet zpoždění a jejich distribuci v rámci sítě. Mechanismus výpočtu všech zpoždění probíhá na rozdíl od porovnání časových značek měřením rozdílu fáze oscilátoru, jehož frekvenci mají díky synchronnímu ethernetu shodnou.

Celá fáze WR Link Setup je složena z následujících fází:

1. **WR Port A** (PTP Master) v pravidelných intervalech posílá zprávu WR Announce message
2. **WR Port B** přijme tuto zprávu, použije algoritmus Best Master Clock (BMC) k určení pozice zařízení v WRN
3. **WR Port B** se přepíná do stavu WR Slave a spouští fázi WR Link Setup tím, že Masterovi pošle zprávu *M_SLAVE_PRESENT*
4. **WR Port A** se přepíná do stavu WR Master, posílá zprávu *M_LOCK* ke spuštění syntonizace

5. WR Slave pošle zprávu *M_LOCKED*, jakmile je syntonizace dokončena
6. WR Master pošle zprávu *M_CALIBRATE*, jejíž cílem je změřit fixní zpoždění Slave-Master $\Delta_{SM}$
7. WR Master pošle zprávu *M_CALIBRATED*, jakmile je zpoždění vypočítáno
8. WR Slave pošle zprávu *M_CALIBRATE*, jejíž cílem je změřit fixní zpoždění Master-Slave $\Delta_{MS}$
9. WR Slave pošle zprávu *M_CALIBRATED*, jakmile je zpoždění vypočítáno
10. WR Master pošle zprávu *M_WR_MODE_ON*, fáze WR Link Setup je dokončena
11. V pravidelných intervalech je používán PTP protokol k výpočtu zpoždění a časového offsetu

Celý mechanismus výměny zpráv můžete vidět na obrázku 2.5, kde je možné vidět graficky obvyklou posloupnost posílání zpráv během WR Link Setup mezi Masterem a Slavem. Jsou také vidět stavy stavového automatu WR FSM, který je blíže popsán v kapitole 2.1.3.

### Model zpoždění WR Link

K dosažení sub-nanosekundové přesnosti White Rabbit je nutné znát přesné zpoždění Master-Slave a Slave-Master. PTP předpokládá, že jsou tato zpoždění stejná a nerozděluje je. Nicméně tato abstrakce je pro White Rabbit nedostatečná, proto musel být model zpoždění změněn. Zpoždění Master-Slave může být vypočítáno dle vzorce

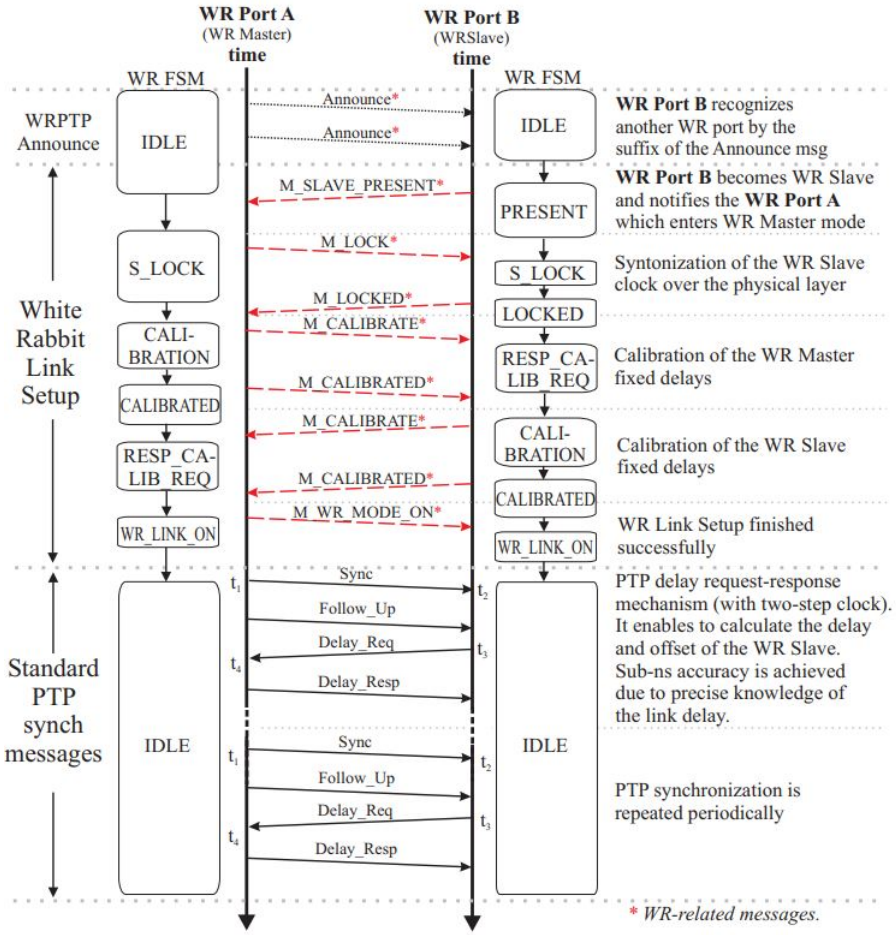
$$delay_{ms} = \Delta_{tx_m} + \delta_{ms} + \Delta_{rx_m} \tag{2.3}$$

kde $\Delta_{tx_m}$ resp. $\Delta_{rx_m}$ jsou pevná zpoždění vysílacího obvodu mastera resp. přijímacího obvodu slaveva. Zpoždění $\delta_{ms}$ je proměnné zpoždění přenosu. Zpoždění $delay_{sm}$ může být vypočítáno podobným způsobem. Pevná zpoždění ($\Delta_{tx_m}$, $\Delta_{rx_m}$, $\Delta_{tx_s}$, $\Delta_{rx_s}$) jsou požadována za konstantní a jsou měřena během kalibrační fáze. Proměnlivá zpoždění $\delta_{ms}$ a $\delta_{sm}$ se musí měřit průběžně a závisí také na použitém médiu. Pro největší přesnost se používá mnohavidové optické vlákno, kde je dána asymetrie zpoždění pouze rozdílnou rychlostí světla způsobenou různými vlnovými délkami, nejčastěji 1550 a 1310 nm.

Poměr mezi zpožděními $\delta_{ms}$ a $\delta_{sm}$ lze vyjádřit koeficientem relativního zpoždění:

$$\alpha = \frac{\delta_{ms}}{\delta_{sm}} - 1 = \frac{n_{1550}}{n_{1310}} - 1 \tag{2.4}$$

Koeficient $\alpha$ může být vypočítán pomocí indexů lomu nebo změřen. S využitím koeficientu $\alpha$, round-trip zpožděním ($delay_{mm}$) a fixních zpoždění $\Delta_{tx_m}$, $\Delta_{rx_m}$,




**Obrázek 2.5:** WR Link Setup (převzato z [8])

$\Delta_{tx_s}$, $\Delta_{rx_s}$ zpoždění $delay_{ms}$ lze vypočítat dle vzorců

$$\begin{aligned} \Delta &= \Delta_{tx_m} + \Delta_{rx_m} + \Delta_{tx_s} + \Delta_{rx_s} \\ delay_{mm} &= \Delta + \delta_{ms} + \delta_{sm} \\ delay_{ms} &= \frac{1+\alpha}{2+\alpha}(delay_{mm} - \Delta) \end{aligned} \tag{2.5}$$

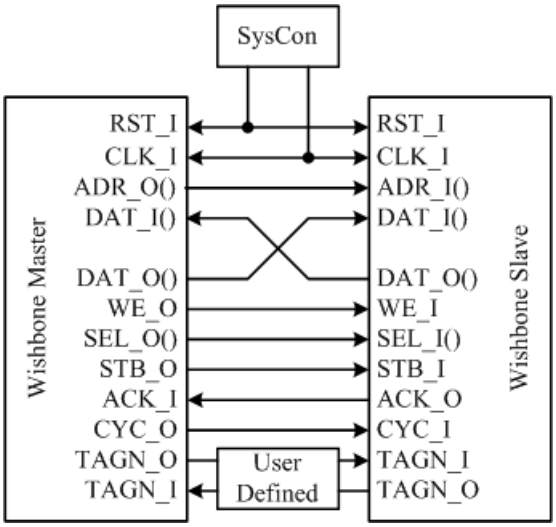
S využitím PTP protokolu a zpožděním $delay_{ms}$ je pak vypočítán konečný offset master-slave

$$offset_{ms} = t_2 - t_1 + delay_{ms} \tag{2.6}$$

obdobně jako ve vzorci 2.1, kde zpoždění $\delta$ je nahrazeno asymetrickým zpožděním získaným dle 2.5.

### Stavový automat WR FSM

Celá fáze *WR Link Setup* je řízena stavovým automatem White Rabbit (WR FSM), který řídí posloupnost fází při synchronizaci pro mastera i slaveva. Běžná posloupnost stavů je zobrazena na obrázku 2.5.




**Obrázek 2.6:** Sběrnice Wishbone (převzato z [5])

## 2.2 Sběrnice Wishbone

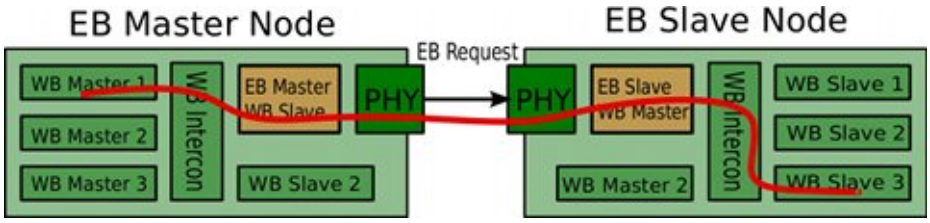
Wishbone [5] je open-source sběrnice používaná pro fyzické propojení jednotlivých nezávislých částí (IP cores) SoC (System on Chip) integrovaných obvodů. V rámci White Rabbit je tato sběrnice používána pro propojení jednotlivých částí zařízení, které jsou většinou implementovány pomocí FPGA.

Wishbone je tzv. logická sběrnice, standard definuje pouze používané signály, časování a způsob komunikace. Fyzická vrstva a způsob jejího elektrického zapojení není standardem definována, jelikož sběrnice je navržena pro použití v FPGA a tedy způsob propojení závisí na použité technologii.

Sběrnice je navržena pro modulární použití, podporuje různé způsoby komunikace (čtení, zápis, blokový přenos, read-modify-write). Bitovou šířku sběrnice je možné také uživatelsky zvolit, lze také nastavit pořadí bitů (big endian, little endian).

Sběrnice Wishbone je založena na topologii Master-slave, masterů může být ke sběrnici připojeno více. Provoz sběrnice řídí arbiter, jehož chování je definováno uživatelem. Wishbone je paralelní sběrnice. Pro propojení je použito několik signálů, jejichž seznam je zobrazen na obrázku 2.6. Mezi hlavní signály sběrnice patří:

- *RST_I* - Resetovací signál z arbitru
- *CLK_I* - Hodinový signál z arbitru
- *ADR_O*, *ADR_I* - Adresa
- *DAT_O*, *DAT_I* - Data
- *WE_O*, *WI_I* - Write enable
- *STB_O*, *STB_I* - Chip select




**Obrázek 2.7:** Architektura EtherBone (převzato z [1])

- *ACK_O*, *ACK_I* - Acknowledge
- *CYC_O*, *CYC_I* - Bus cycle

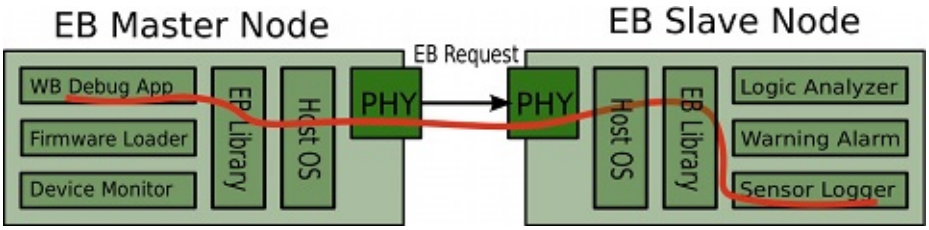
Signály *RST_I* a *CLK_I* jsou generovány arbitrem sběrnice, slouží k řízení provozu sběrnice. K adresaci slouží signály *RST_x*, pro přenos dat *DAT_x*, pro řízení přenosu slouží signály *STB_x* (alternativa chip select), *ACK_x* (potvrzování) a *CYC_x* (řízení cyklu sběrnice).

Vzdálené připojení k této sběrnici bude hlavní způsob jak diagnostikovat stav systémů White Rabbit. Bližší způsob diagnostiky bude popsán v dalších kapitolách, zejména v kapitole 2.3.

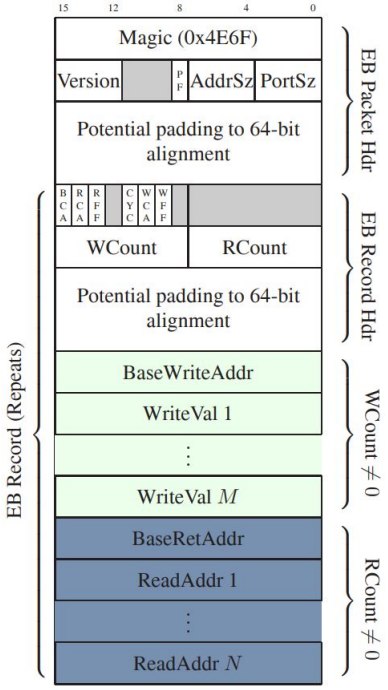
## 2.3 EtherBone

Pro vzdálenou diagnostiku White Rabbit zařízení se potřebujeme připojit externě ke sběrnici Wishbone (kapitola 2.2). Pro tento účel byl vyvinut protokol EtherBone [7], který slouží ke vzdálenému připojení ke sběrnici Wishbone pomocí protokolů TCP/IP. Mezi hlavní přednosti tohoto protokolu patří deterministická latence a také jednoduchá hardwarová implementace. Jelikož pro TCP/IP protokoly lze použít ethernetovou fyzickou vrstvu (stejnou jako WR), je pro diagnostiku White Rabbit zařízení možné využít již vybudovanou síť,není nutné budovat další spojení.

Architekturu sítě EtherBone můžeme vidět na obrázku 2.7. V tomto případě WB Master 1 chce zapsat informace do registru vzdáleného zařízení WR Slave 3. Požadavek na sběrnici Wishbone je přeložen periférií WR Interconnect na EtherBone paket. Ten je odvysílán pomocí fyzické vrstvy do vzdáleného zařízení, kde je přeložen zpět na Wishbone požadavek a odeslán konečnému zařízení. EB Master či EB Slave zařízení může být také simulováno softwarově. Tuto situaci můžete vidět na obrázku 2.8, kde je simulováno master i slave zařízení. Tento způsob spojení bude využíván pro vzdálenou diagnostiku, kde EB Master bude nahrazen softwarovou knihovnou na PC, která bude přistupovat k fyzickému EB Slave zařízení (v našem případě White Rabbit). Detailní použití knihovny EtherBone Core, používanou pro simulaci EB Master, bude popsáno v kapitole 2.3.




**Obrázek 2.8:** Emulace EtherBone (převzato z [1])




**Obrázek 2.9:** EtherBone packet (převzato z [7])

## Formát paketu

EtherBone může být použit ve dvou variantách. Jedna varianta je postavena na protokolu TCP, druhá na UDP. Varianta s UDP je více používána, navíc ji podporuje i použité White Rabbit zařízení. Z těchto důvodů bude dále popsána jen tato varianta.

Základní stavbu EtherBone UDP paketu můžete vidět na obrázku 2.9. Paket obsahuje hlavičku se základními informacemi a jednu nebo více operací čtení nebo zápisu.

Hlavička začíná konstantou 0x4E6F, poté následuje informace o verzi protokolu, délky adresy a šířce sběrnice (jelikož tyto parametry lze pro Wishbone sběrnici uživatelsky nastavit). Po hlavičce následují samotné příkazy.

Hlavička příkazu obsahuje několik bitových flagů, které slouží k dalším nastavením přenosu (např. endianing), poté jsou uvedeny informace o počtu operací čtení a zápisu. Po hlavičce následují samotné operace, nejdříve čtení, pak zápis.

# Kapitola 3
# Praktická část

Cílem této práce je navrhnout a vyvinout diagnostický systém pro White Rabbit, který bude zapojen k již existujícímu systému v laboratoři přesného času a frekvence FEL ČVUT. Zdrojem přesného času v laboratoři jsou přesné cesiové atomové hodiny, které se podílení i na tvorbě mezinárodní časové stupnice UTC. K distribuci přesné časové reference těchto hodin do ostatních systémů je použit White Rabbit. Diagnostický systém bude sloužit zejména k automatické diagnostice, detekci chyb a jejich oznámení uživateli, který je může ihned vyřešit. Součástí systému bude také vzdálené monitorování stavu všech zařízení White Rabbit.

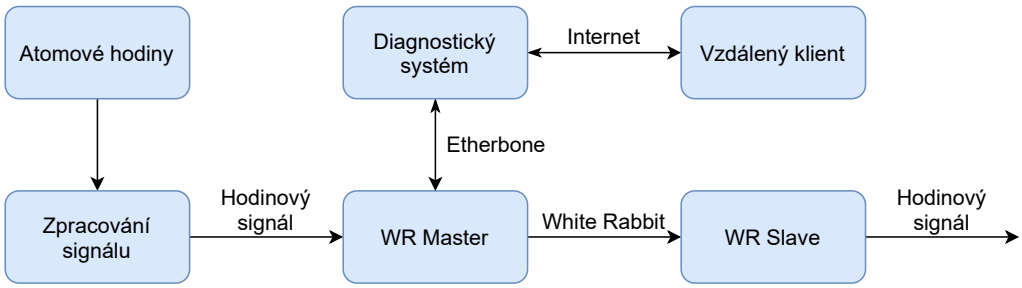
## 3.1 Analýza problému

Na obrázku 3.1 můžeme vidět schéma aktuálního systému v laboratoři. Systém se skládá z atomových hodin a obvodu pro zpracování signálu, na jehož výstupu je přesný hodinový signál. Tento systém byl pro testovací účely doplněn o dvě WR zařízení, která budou sloužit pro následné testování celého řetězce. Hodinový signál z atomových hodin je přijímán WR masterem, který ho následně pomocí protokolu White Rabbit předává WR slaveovi. Ten generuje stejný hodinový signál jako na začátku řetězce. K tomuto systému bude připojen diagnostický systém, který pomocí protokolu EtherBone bude komunikovat se všemi WR zařízeními v síti. Zároveň bude možné se na systém napojit vzdáleně přes internet a monitorovat zařízení bez nutnosti fyzického přístupu do lokální sítě.

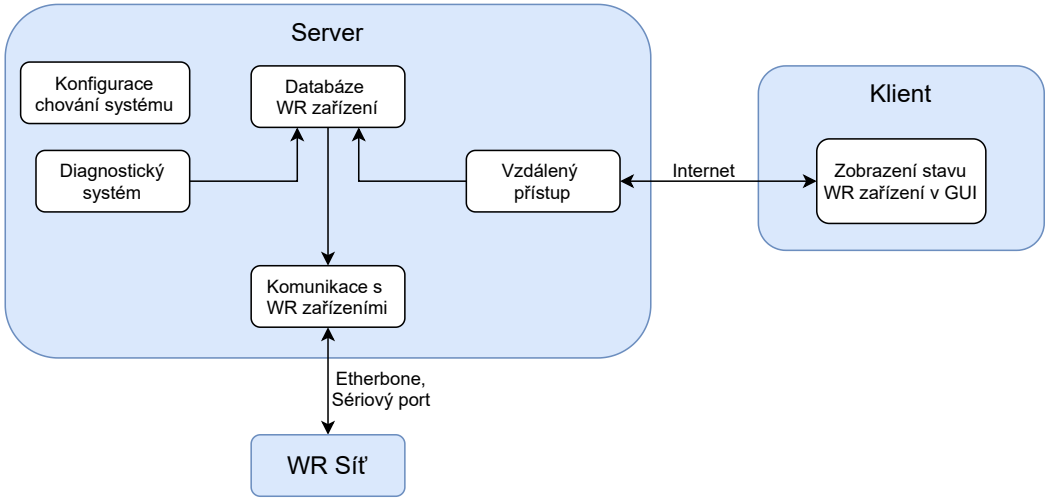
## 3.2 Návrh diagnostického systému

Na začátku vývoje bylo nutné stanovit hlavní požadavky, které musí diagnostický systém splňovat. Mezi ty nejdůležitější patří automatická diagnostika a vyhodnocení stavu WR zařízení bez jakéhokoliv zásahu uživatele a také možnost vzdáleného přístupu do sítě White Rabbit. Na základě těchto požadavků byly stanoveny následující funkce systému:

- Podpora komunikace s WR zařízeními přes protokol Etherbone




**Obrázek 3.1:** Blokové schéma WR systému




**Obrázek 3.2:** Blokové schéma diagnostického systému

- V případě výpadku sítě bude též možné použít sériový port
- Vytvoření API pro vzdálenou komunikaci přes internet
- Automatická diagnostika všech připojených WR zařízení
- Konfigurace chování systému přes konfigurační soubor dle přání uživatele
- Zobrazení aktuálního stavu vzdálených WR zařízení v grafické aplikaci

Na základě těchto požadavků byly vytvořeny základní komponenty systému, jejichž propojení můžete vidět na blokovém diagramu, který je zobrazený na obrázku 3.2. Veškerá funkcionalita systému byla rozdělena do dvou samostatných částí. Serverová aplikace má za úkol komunikaci s WR zařízeními a jejich automatickou diagnostiku. Druhou částí systému je klientská aplikace, která bude sloužit ke vzdálenému připojení uživatele na server. Jedná se o grafickou aplikaci, kde uživatel uvidí aktuální stav všech WR zařízení připojených k serveru, s nimiž může také vzdáleně komunikovat. Obě části systému budou podrobně popsány v následujících kapitolách.

## 3.3 Vývojové prostředí a QT framework

K vývoji diagnostického systému byl vybrán programovací jazyk C++ zejména kvůli dostupnosti knihoven a dlouhodobým zkušenostem, např knihovna pro podporu protokolu EtherBone (EtherBone Core [1]) je dostupná jen v jazyce C. Pro jazyk C++ je v projektu použit nejnovější standard c++17, takže je nutné, aby vývojář použil kompilátor, který tento standard podporoval.

Pro vývoj grafického klienta a velké části serverové aplikace byl použit framework QT [13]. QT je multiplatformní (Windows, Linux, maxOS, android, iOS, ..) framework, který slouží k ulehčení vývoje aplikací. Hlavním důvodem jeho použití je zjednodušení práce programátora a přenositelnost kódu na jiné platformy. QT obsahuje mnoho modulů, které lze k vývoji použít. Mezi ty nejpoužívanější patří moduly pro vytváření grafického rozhraní nebo pro síťovou komunikaci. QT také obsahuje několik nástrojů pro podporu programování. Mezi ty nejdůležitější patří zejména *QT Designer*, který slouží k návrhu grafického rozhraní a následnému automatickému generování zdrojového kódu; dále také nástroj *qmake*, který slouží k automatické generaci make souborů z tzv. project souborů (.pro). Pro vývoj samotné aplikace lze také použít IDE *QT Creator*, který má vestavěnou podporu QT frameworku.

Na konec je nutné zmínit, že k vývoji diagnostického systému byl použit operační systém linux, konkrétně Ubuntu 20.04, kde byly všechny části systému vyzkoušeny. Pro překlad zdrojového kódu byl použit kompilátor *g++*. Nicméně grafickou aplikaci lze spustit i na operačním systému Windows, serverovou část aplikace z drobnými úpravami také.

### 3.3.1 Příklad zdrojového kódu

Příklad jednoduchého zdrojového kódu s využitím frameworku QT můžete vidět ve výpisu zdrojového kódu 3.1. Kód slouží k vytvoření jednoduché grafické aplikace s jedním tlačítkem s textem *Hello world !*.

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[]) {
    // Create main application
    QApplication app(argc, argv);
    // Create button a display it
    QPushButton button ("Hello␣world␣!");
    button.show();
    // Run main event loop
    return app.exec();
}
```

**Zdrojový kód 3.1:** Příklad zdrojového kódu

Pro kompilaci aplikace je třeba vytvořit jednoduchý *.pro* soubor 3.2, který obsahuje základní informace o aplikaci (použité QT moduly nebo zdrojové

kódy aplikace). Pomocí tohoto souboru spolu s nástrojem *qmake* je pak vygenerován Makefile, který je použit k následné kompilaci.

```
TARGET     = qt_example # Application name
TEMPLATE   = app
CONFIG    += c++17 # Use standard c++17
QT        += core gui widgets # Used QT modeles

SOURCES = qt_example.cpp # Source codes of aplication
```

**Zdrojový kód 3.2:** Příklad .pro souboru

K následné kompilaci a spuštění aplikace je nutné zadat následující příkazy:

```
qmake
make
./qt_example
```

**Zdrojový kód 3.3:** Kompilace a spuštění aplikace

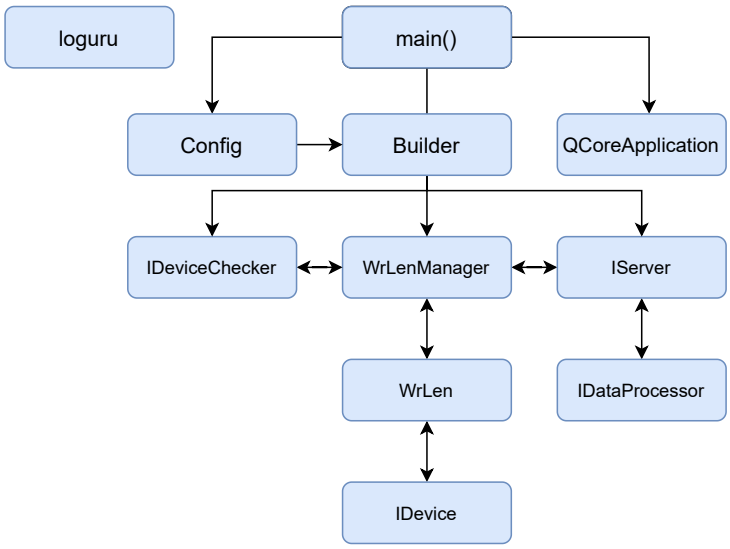
## 3.4 Popis serverové části systému

Jak již bylo zmíněno v předchozích kapitolách, hlavními úkoly serverové části systému jsou automatická diagnostika WR zařízení a umožnění vzdáleného přístupu uživatelům do WR sítě.

V kapitole 3.2 byl proveden návrh takového systému. Blokové schéma a možné propojení jednotlivých komponentů je zobrazeno na obrázku 3.2. S využitím tohoto diagramu bylo navrženo propojení rozhraní a tříd, které budou funkcionalitu těchto bloků implementovat. Návrh a propojení těchto tříd můžete vidět na obrázku 3.3. Po spuštění aplikace je ve funkci *main()* vytvořena třída *Config*, která je zodpovědná za načtení konfigurace ze souboru a její následné propagaci do dalších částí programu. Dále je vytvořena instance třídy *QCoreApplication*, která zpracovává události z operačního systému (např. síťovou komunikaci) a také obsahuje EventLoop, kterou využívají další komponenty frameworku QT. Poslední vytvořenou třídou je *Builder*, který vytvoří a propojí hlavní komponenty diagnostického systému. Podrobnější popis těchto komponentů a jejich propojení bude blíže popsáno v dalších kapitolách. K logování aktuálního stavu aplikace je použita knihovna loguru [2]. Knihovna obsahuje řadu maker, které usnadňují logování stavu aplikace, jako jsou například automatické vkládání jména a řádku souboru, kde chyba nastala. Samozřejmostí je také ukládání stavu aplikace do souborů pro pozdější analýzu.

### 3.4.1 Rozhraní IDevice

Základním úkolem serverové aplikace je komunikace s WR zařízeními. Tento úkol obstarává rozhraní *IDevice*. Aplikace podporuje dva způsoby komunikace




**Obrázek 3.3:** Propojení tříd v serverové aplikaci

- vzdáleně přes ethernet pomocí protokolu EtherBone a lokálně pomocí sériového portu.

### EtherboneDevice

Hlavním způsobem vzdálené komunikace s WR zařízeními je protokol EtherBone [7]. S využitím tohoto protokolu je možné se vzdáleně připojit na interní sběrnici wishbone, ke které je připojena také UART periférie. Přesný popis tohoto způsobu připojení je popsán v kapitole 2.3.

K implementaci protokolu je použita knihovna *EtherBone Core [1]* napsaná v jazyce C. Příklad použití této knihovny můžete vidět ve zdrojovém kódu 3.4.

```
// Device address in format udp4/ip/port
const char* devAddress = "udp4/192.168.0.1/60368"

// Open remote socket
eb_socket_open(EB_ABI_CODE, nullptr, EB_ADDRX | EB_DATAX, &
    socket);
// Open comunication witj device
eb_device_open(socket, devAddress, EB_ADDRX | EB_DATAX, 3, &
    device);
// Find address of UART periphery
eb_sdb_find_by_identity(device, VENDOR_ID_CERN, WR_UART_ID, &
    sdbDevice, &nDevices);
uint32_t uartBase = sdbDevice.sdb_component.addr_first;

// Write one byte to device
char c_write = 'a';
```

```
eb_device_write(device, uartBase + VUART_TX_REG, EB_BIG_ENDIAN
    | EB_DATAX, c, 0, nullptr);

// Read data from device
eb_data_t dataRaw = 0;
eb_device_read(device, uartBase + VUART_RX_REG, EB_BIG_ENDIAN |
     EB_DATAX, &dataRaw, 0, nullptr);
// Number of characters in TX buffer
int cnt = (dataRaw & VUART_RX_CNT_MSK) >> 9;
// One char
char c_read = (dataRaw & VUART_RX_DAT_MKS);

// Close connection
eb_device_close(device);
eb_socket_close(socket);
```

**Zdrojový kód 3.4:** Příklad použití knihovny EtherBone Core

Na začátku kódu je vytvořen socket, přes který se bude komunikovat s WR zařízeními, následně je navázáno samotné spojení se zařízeními. Při spojení je třeba správně nastavit vlastnosti wishbone sběrnice (šířka sběrnice, endianing, apod.), protože jednotlivé implementace sběrnice se mohou lišit. Po navázání spojení je třeba zjistit adresu UART komponenty. K tomuto účelu slouží funkce *eb_sdb_find_by_identity()*. Po zjištění této adresy je již možné použít metody *eb_device_read()* a *eb_device_write()* ke čtení resp. k zápisu dat do UART registrů.

Přesnou implementaci *EtherboneDevice* můžete vidět ve zdrojových kódech aplikace.

#### SerialDevice

Druhý způsob komunikace s WR zařízeními je použití sériového portu. Tuto možnost lze použít jen na kratší vzdálenosti a na rozdíl od protokolu EtherBone je nutné použít další spojení.

Pro komunikaci po sériovém portu byl použit framework QT konkrétně třída *QSerialPort*, která obsahuje rozhraní pro komunikaci po sériovém portu nezávisle na operačním systému. V konstruktoru třídy se nastaví vlastnosti spojení (baudrate, počet bitů, stopbitů, apod.). Poté lze použít metody *read()* a *write()* ke čtení resp. k zápisu dat. Přesný popis těchto metod lze nalézt v dokumentaci frameworku QT [12].

### 3.4.2 Třída WrLen

Třída *WrLen* slouží ke zastřešení komunikace s WR zařízeními. Obsahuje několik metod k navázání spojení a několik předdefinovaných příkazů. Pro vlastní komunikaci třída využívá rozhraní *IDevice*, které je popsáno v kapitole 3.4.1. Díky tomuto přístupu je možné samotnou implementaci komunikace

řešit nezávisle na třídě *WrLen*. Výhodou je také volba různého způsobu připojení (EtherBone, sériový port).

Ukázku použití třídy *WrLen* můžete vidět ve zdrojovém kódu 3.5.

```
#include "WhiteRabbit/WrLen/WrLen.h"
#include "Recources/DeviceList.h"

const auto name = "master"; // Device name
const auto devDesc = DeviceList::Device("master", "192.168.0.1"
    , "/dev/ttyUSB0"); // Create device descriptor
auto ethDev = std::make_unique<WR::EtherboneDevice>(devDesc);
    // Create new EtherboneDevice

WR::WrLen wrLen(std::move(ethDev)); // Create WrLen instance
    with etherbone comunication

auto response = wrLen.GetVerCmd(); // Read VerCmd informations
    from device
std::string vendor = response.fru.vendor; // Read device vendor
     from response
```

**Zdrojový kód 3.5:** Příklad použití třídy WrLen

Na začátku je získán popis zařízení z databáze, která se vytváří v konfiguračním souboru aplikace. Bližší popis konfiguračního souboru je uveden v kapitole 3.4.6. Poté je tento deskriptor použit k vytvoření instance třídy *EtherboneDevice* (viz kapitola 3.4.1), která je předána konstruktoru třídy *WrLen*. Následně je možné využít tuto třídu k získání informací o zařízení. K tomuto účelu slouží několik metod *GetXXXCmd()*, kde XXX je jméno případu. Pro obecnou komunikaci lze použít metodu *GetCmd()*. Každá tato metoda vrací strukturu se stejnojmenným názvem, která obsahuje zformátované informace příkazu. Ukončení spojení je provedeno v destruktoru třídy, takže je spojení automaticky ukončeno, jakmile třída *WrLen* zmizí z aktuální kontextu.

### 3.4.3 Databáze zařízení

Pro správu všech připojených WR zařízení byla vytvořena třída *WrLenManager*. Třída obsahuje databázi zařízení obsahující základní informace nutné k vytvoření spojení. Konfigurace této databáze se provádí v konfiguračním souboru, který je popsán v kapitole 3.4.6.

Pro navázání spojení je třeba znát jen jméno WR zařízení. Parametry spojení jsou uloženy v databázi. Výběr fyzické vrstvy (kapitola 3.4.1) je proveden dle aktuálních možností. Prioritu má protokol EtherBone, pokud není tato možnost dostupná a je k dispozici sériový port, je použit tento způsob. Informace o vytvořených spojeních jsou cachovány, takže není nutné spojení vytvářet stále dokola.

Použití třídy je relativně jednoduché, příklad můžete vidět ve zdrojovém kódu 3.6.

```
#include "WhiteRabbit/WrLen/WrLenManager.h"

auto deviceList = config.deviceList; // Get DeviceList from
    config file
WR::WrLenManager manager(deviceList); // Create instance of
    WrLenManager
auto& wrLen = manager["master"]; // Create connection with
    device with name "master"

auto response = wrLen.GetVerCmd(); // Read VerCmd informations
    from device
```

**Zdrojový kód 3.6:** Příklad použití třídy WrLenManager

Na začátku kódu je získána databáze zařízení z konfiguračního souboru, která je následně předána konstruktoru třídy *WrLenManager*. Poté je možné použít přetížený *operator[]()* k automatickému navázání spojení. Metoda vrací referenci třídy *WrLen* (kapitola 4.1), kterou lze následně použít pro čtení informací ze zařízení.

### 3.4.4 Vzdálený přístup

Jedním ze dvou hlavních úkolů aplikace je umožnění vzdáleného přístupu do White Rabbit sítě a komunikace s WR zařízeními přes internet. K tomuto účelu slouží rozhraní *IServer*, které by mělo poskytovat veřejné API sloužící k tomuto účelu. Během vývoje bylo vyzkoušeno více implementací, které používaly různé způsoby komunikace.

Prvním testovaným způsobem bylo navržení vlastního formátu binárních zpráv, jejichž výměna probíhala pomocí protokolů TCP/IP. Výhodou byla jejich velikost, hlavní nevýhodou jejich složitá implementace, budoucí rozšiřitelnost, a také možné vytvoření vlastních klientů, jejichž vývojáři by museli podrobně studovat formát zpráv.

Kvůli značným nevýhodám výše uvedené realizace serveru byl zvolen zcela jiný přístup - REST API webový server. REST je protokol založený na webovém protokolu HTTP, který slouží pro výměnu informací mezí klientem a serverem v distribuovaných systémech. API je založeno na práci se zdroji (tzv. resource), které jsou reprezentovány unikátním identifikátorem URI (Uniform Resource Identifier). Tyto zdroje mohou reprezentovat různá data nebo stavy aplikace.

Pro čtení nebo manipulace se zdroji slouží HTTP požadavky. Standard definuje čtyři hlavní metody:

- **GET** - Získání zdroje, používá se např. pro získání stránky z webového serveru
- **POST** - Odeslání zdroje, používá se např. pro odeslání dat z formuláře

- **DELETE** - Mazání zdroje
- **PUT** - Aktualizace zdroje

Pro popis těchto zdrojů se používají různé notace, mezi ty nejznámější patří *XML* a *JSON*.

Implementace REST API v aplikaci je provedena ve třídě *RestServer*. Třída využívá modul QtHTTPServer [14], který je součásti QT frameworku. Server poskytuje několik zdrojů, pomocí kterých je možné získat seznam připojených WR zařízení, se kterými může klient následně komunikovat. Pro přenos jednotlivých příkazů se používá jazyk JSON, jehož struktura ke podobná příkazům definovaných v kapitole 4.1. Tento formát je lidsky čitelný, čehož lze například využít při diagnostice komunikace. Výhodou tohoto přístupu je použití ověřeného postupu. Podpora REST API je vestavěná do mnoha knihoven a programů, takže je komunikace se serverem jednoduchá.

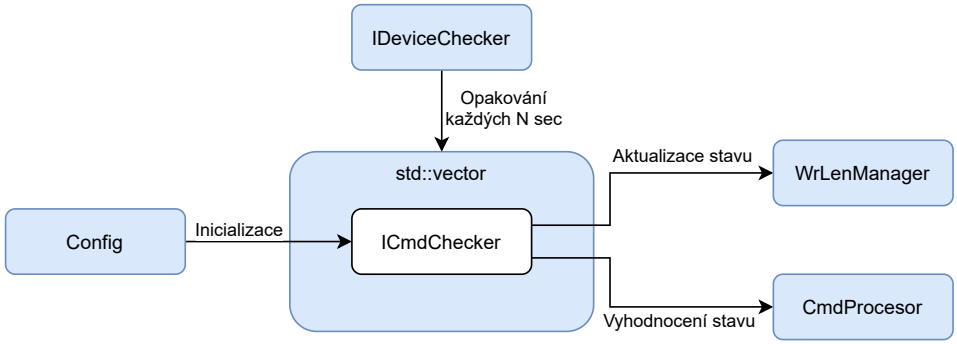
Příklad komunikace se serverem můžete vidět ve zdrojovém kódu 3.7. Je zde vidět získání zdroje **/devices/master/ver**, který reprezentuje základní informace o zařízení se jménem *master*. Z těchto informací můžeme např získat výrobce WR zařízení, jeho název nebo použitý firmware. Pro otestování funkcionality serveru byl použit program *curl*, který mimo jiné podporuje protokol HTTP.

```
curl 127.0.0.1:10000/devices/master/ver

{
    "FRU": {
        "device": "WRLEN",
        "fid": "2018-07-12 13:27:43.617560",
        "partnum": "7SWRLENv?.0-S09",
        "serial": "7SWRLENv?.0-S09_360",
        "vendor": "7S"
    },
    "GW": {
        "date": "18/02/05 11:54",
        "id": "0x751e41c0",
        "version": "3.13.35"
    },
    "WRCore": {
        "build": "7344bd0",
        "info": "Built for 128 kB RAM, stack is 2048 bytes",
        "time": "Feb 5 2018, 11:48:25"
    }
}
```

**Zdrojový kód 3.7:** Výstup příkazu VerCmd

Přesný popis komunikace se serverem a dokumentaci všech dostupných zdrojů naleznete v příloze v sekci B.1.




**Obrázek 3.4:** Blokové schéma diagnostické části aplikace

### 3.4.5 Automatická diagnostika

Druhým a asi nejdůležitějším úkolem diagnostického systému je automatická kontrola stavu všech WR zařízení, vyhodnocení případných chyb a jejich oznámení uživateli, který na ně následně může příslušným způsobem reagovat.

Diagnostický systém byl navržen tak, aby byl lehce konfigurovatelný dle přání uživatele. Pravidla pro vyhodnocování stavů WR zařízení nejsou v programu pevně definována, ale vytvářejí se dynamicky v konfiguračním souboru. Po spuštění aplikace si systém pravidla přečte a při diagnostice zařízení je aplikuje.

Pro implementaci funkcionality diagnostického systému je použito několik tříd. Hlavní třídou je rozhraní *IDeviceChecker* resp. jeho implementace *WrLenDeviceChecker*. Tato třída v definovaných intervalech aktualizuje stav WR zařízení s využitím třídy *WrLenManager* (viz kapitola 3.4.3) a poté jejich stav vyhodnocuje. Pokud je detekována nějaká chyba, může třída zavolat jakýkoliv uživatelsky definovaný skript, který může sloužit např. k odeslání emailu s informacemi o chybě. Pro aktualizaci, uchování a samotné vyhodnocení informací slouží rozhraní *ICmdChecker* resp. jeho implementace *WrLenCmdChecker*. Poslední důležitou třídou používanou pro diagnostiku je třída *CmdProcessor*, která má za úkol dynamické vytváření pravidel popsaných v konfiguračním souboru a jejich následnou aplikaci. Vztah mezi výše popsanými částmi systému můžete vidět na obrázku 3.4. Přesný popis tvorby pravidel bude popsán v kapitole 3.4.5.

#### Tvorba pravidel

Jak již bylo řečeno, o zpracování a tvorbu pravidel z konfiguračního souboru se stará třída *CmdProcessor*. Diagnostický systém funguje na principu tvorby jednoduchých pravidel, pomocí nichž se porovnávají jednotlivé parametry zařízení s očekávanými hodnotami. Tuto funkcionalitu zastřešuje rozhraní *IRule*, implementace tohoto rozhraní reprezentují různé druhy pravidel. K dispozici máme následující čtyři druhy pravidel:

- **equal** - Rovnost
- **nequal** - Nerovnost

- **less** - Menší než
- **more** - Větší než

Pro definici parametru, který chceme kontrolovat, slouží pomocné rozhraní *INodeDescriptor*. V programu existují následující dvě implementace:

- **SimpleNodeDescriptor** - Slouží k popisu jednoduchých skalárních parametrů
- **VectorNodeDescriptor** - Slouží k popisu parametrů nacházejících se v poli

Tyto deskriptory slouží k popisu parametrů zařízení pomocí ID (u pole je potřeba znát i index), které se deskriptoru předají v konstruktoru zařízení. Poté lze v kódu použít metodu *Get()*, která vrací referenci na daný parametr. Tuto metodu používá již výše zmíněné rozhraní *IRule*, které slouží pro vyhodnocování samotných pravidel.

Bližší popis diagnostického systému lze najít v dokumentaci, popis uživatelské konfigurace lze najít v kapitole 3.4.6.

### 3.4.6 Konfigurace systému

Již v předchozích kapitolách bylo zmíněno, že většinu chování diagnostického systému je možné modifikovat dle přání uživatele. K tomuto účelu slouží konfigurační soubor *config.yaml*, který je napsán v jazyce *YAML*. Tento jazyk je často ke konfiguraci aplikací používán, jeho konstrukce je podobná javascriptu (použitý v REST serveru v kapitole 3.4.4), ale dovoluje použití komentářů, které zpřehledňují strukturu souboru. V programu je tato funkcionalita implementována ve třídě *Config*, která slouží ke zpracování konfiguračního souboru a propagaci nastavení do ostatních částí programu.

Konfigurační soubor je rozdělen do několika částí, které slouží k modifikaci funkce diagnostického systému. Mezi ty nejdůležitější části patří:

- **programConfig** - Základní konfigurace programu
- **devices** - Databáze zařízení
- **rules** - Tvorba pravidel pro automatický diagnostický systém

Tyto částí budou podrobněji popsány v následujících kapitolách. Podrobnější výpis z dokumentace, kde je popsána konfigurace programu, je uvedena v příloze v sekci B.2.

#### Základní konfigurace programu - sekce *programConfig*

V této části konfiguračního souboru se nastavuje základní chování programu. Je zde možné například modifikovat TCP/IP Port, na kterém běží REST server (kapitola 3.4.4) nebo periodu automatické kontroly WR zařízení (kapitola 3.4.5). Přehled všech parametrů této části můžete vidět v tabulce 3.1.

| Parametr | Typ | Defaultní hodnota | Význam |
|---|---|---|---|
| httpPort | Int | 10000 | Číslo portu, na kterém běží REST API server |
| checkerPeriodMs | Int | 10000 | Perioda v ms, ve kterou program automaticky kontroluje stav nařízení |
| wrLenReadDelayMs | Int | 2000 | - |
| errorCmd | String | "" | Příkaz, který se vykoná po detekování chyby |
| logDir | String | Logs | Složka, do které se logují informace o programu a stavu zařízení |

**Tabulka 3.1:** Přehled základní konfigurace programu

| Parametr | Typ | Defaultní hodnota | Význam |
|---|---|---|---|
| ip | String | - | IP adresa WR zařízení |
| serialPort | String | - | Jméno sériového portu, ke kterému je WR zařízení připojeno |
| name | String | - | Jméno WR zařízení, používá se pro identifikaci |
| log | Bool | False | Logovat informace o zařízení |

**Tabulka 3.2:** Seznam parametrů pro databázi zařízení

### Databáze zařízení - sekce *devices*

V této části konfiguračního souboru se definuje seznam WR zařízení, která jsou připojena k diagnostickému systému, a se kterými má aplikace pracovat. Tento seznam využívá třída *WrLenManager* (kapitola 3.4.3), která pomocí těchto dat inicializuje spojení se zařízeními. Ke každému zařízení je třeba uvézt jeho jméno, IP adresu a název sériového portu (pokud je připojen). Všechny tyto parametry včetně jejich popisu jsou zobrazeny v tabulce 3.2.

### Tvorba pravidel - sekce *rules*

V poslední části diagnostického systému se nastavují pravidla, podle kterých se vyhodnocuje stav připojených WR zařízení. Každý parametr lze vyhodnocovat pomocí následujících pravidel:

- **equal** - Rovnost
- **nequal** - Nerovnost
- **less** - Menší než
- **more** - Větší než

Pravidla je možné nastavit pro každé zařízení zvlášť, jako identifikátor slouží jméno zařízení. Lze také použít identifikátor **global**, jehož pravidla jsou použita pro všechna vyjmenovaná zařízení. Pokud není zařízení v seznamu, není jeho stav vyhodnocován ani globálními pravidly. Chceme-li pro WR zařízení použít jen globální pravidla, je nutné vložit prázdný node bez pravidel (viz příklad). Pokud je ve jméně parametru přípona [id], jedná se o tzv. vektorový parametr, který se používá pro indexování pole parametrů. Indexace se provádí pomocí hranatých závorek, počítá se od nuly. K pravidlu lze také přidat parametr *text*, který se používá ke generování dynamického popisu chyby. K popisu chyby se použije tento parametr s doplněním informací o aktuální a očekávané hodnotě parametru.

Jednotlivé parametry jsou rozděleny do několika skupin dle příkazu, ve kterém se nacházejí. Aktuálně je možné vybírat z následujících skupin:

- **verCmd** - Základní informace o zařízení (výrobce, jméno, apod.)
- **statCmd** - Aktuální stav zařízení
- **macCmd** - Aktuální stav ethernet portů

Ukázku konfigurace pravidla pro jeden parametr můžete vidět ve zdrojovém kódu 3.8. Pravidlo je vytvořeno pro zařízení **master**. Podmínkou je, aby parametr **FRU_serial** ze skupiny **verCmd** byl roven **12345**.

```
# Node with rules for DeviceChecker
rules:
    # Device name
    master:
        # Parameter group
        verCmd:
          # Parameter name
          FRU_serial:
            text: "FRU_serial␣has␣wrong␣value"
            equal: "12345"
```

**Zdrojový kód 3.8:** Příklad konfigurace pravidel

Podrobný popis všech možných parametrů, které je možné sledovat pomocí pravidel, je uveden v úryvku z dokumentace v příloze B.2.

### Ukázka konfiguračního souboru

Ukázku konfiguračního souboru serverové části aplikace můžete vidět ve zdrojovém kódu 3.9. Jsou zde zobrazeny všechny výše popsané sekce souboru *programConfig*, *devices*, *rules*. Předpokládá se, že k systému budou připojena dvě WR zařízení **master** a **slave** s IP adresami 10.20.30.12 resp. 10.20.30.11.

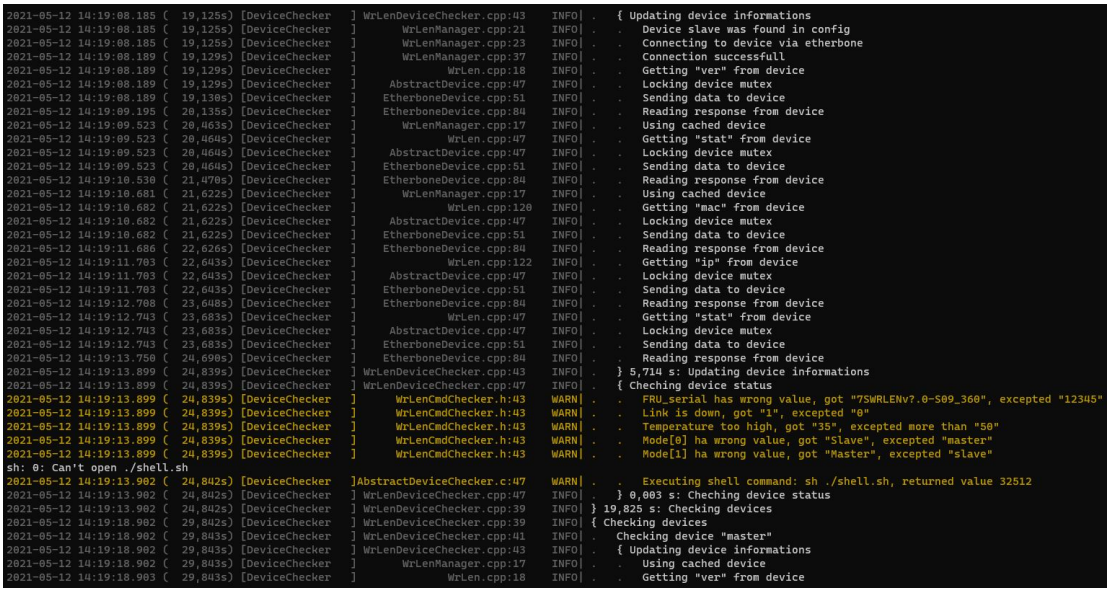
V sekci pravidel pro automatickou kontrolu stavu zařízení vidíme použití příkazu **global**, který definuje globální pravidla pro všechna zařízení. Dále jsou zde příklady pravidel pro všechna připojená zařízení.

```
% YAML 1.0
---
programConfig:
  # Port of REST API server
  httpPort: 10000
  # Period in ms, when devices status is checked
  checkerPeriodMs: 5000
  # TODO
  wrLenReadDelayMs: 2000
  # Command which is called by DeviceChecker when error occurs
  errorCmd: "sh␣./shell.sh"
  # Directory, where logs are stored
  logDir: "Logs"

# Array of connected WR-LEN devices
devices:
    -
      ip: 10.20.30.12
      serialPort: /dev/ttyUSB1
      name: master
      log: true

    -
      ip: 10.20.30.11
      serialPort: /dev/ttyUSB0
      name: slave
      log: true

# Node with rules for DeviceChecker
rules:
  global:
    verCmd:
      FRU_serial:
        text: "FRU_serial␣has␣wrong␣value"
        equal: "12345"
    statCmd:
      Temp:
        text: "Temperature␣too␣high"
        more: 50
  master: ~
  slave:
    statCmd:
      Link[0]:
        text: "Link␣is␣down"
        equal: false
    macCmd:
```




**Obrázek 3.5:** Ukázka výstupu serverové části aplikace

```
    Mode[0]:
      text: "Mode[0] has wrong value"
      equal: "master"
    Mode[1]:
      text: "Mode[1] has wrong value"
      equal: "slave"
```

**Zdrojový kód 3.9:** Příklad konfiguračního souboru

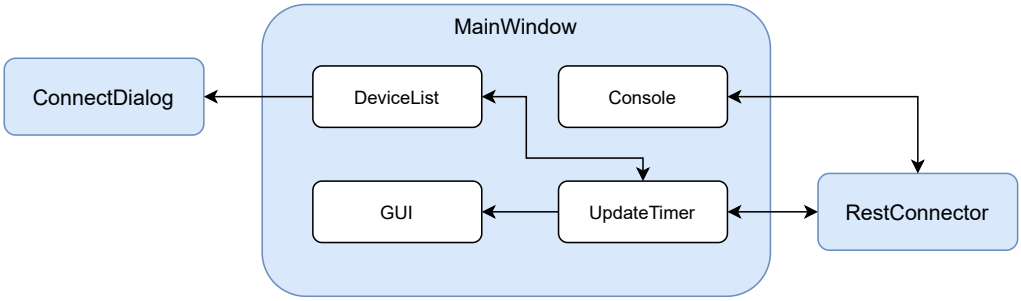
### 3.4.7 Logování

Z důvodu diagnostiky aktuálního i minulého stavu celého systému jsou všechny stavy aplikace vypisovány do standardního výstupu a následně ukládány do souboru. Ke zjednodušení logování se používá knihovna *loguru* [2]. Tato knihovna obsahuje několik maker, pomocí kterých lze rozlišit důležitost výstupu (INFO, WARNING, ERROR), nebo také automatické vložení jména souboru a čísla řádku místa výskytu chyby. Ukázku výstupu aplikace můžete vidět na obrázku 3.5, kde probíhá automatická kontrola stavu zařízení.

Všechny logy aplikace se ukládají do složky definované v konfiguračním souboru. Nacházejí se zde všechny výstupy aplikace a také minulé stavy všech připojených WR zařízení. Názvy souborů se generují dynamicky dle času spuštění aplikace, stavy WR zařízení se ukládají do složek dle jména zařízení.

## 3.5 Popis klientské části systému

Důležitou částí diagnostického systému je možnost vzdáleného připojení klienta do WR sítě a následné monitorování stavu všech připojených zařízení.




**Obrázek 3.6:** Blokový diagram klientské části aplikace

Serverová část aplikace obsahuje REST API server, který je důkladně popsán v kapitole 3.4.4. Klientská aplikace může využít tohoto API k zobrazení stavu WR zařízení uživateli. Pro ověření konceptu klient-server byla navrhnuta a vytvořena GUI aplikace vytvořená pomocí frameworku QT [13], která obsahuje přehledné grafické rozhraní pro zobrazení všech parametrů uživateli. Aplikace se připojí ke vzdálenému diagnostickému systému a pomocí již zmíněného REST API průběžně aktualizuje a zobrazuje většinu parametrů, pomocí kterých lze vzdáleně monitorovat stav WR sítě.

Klientská aplikace musí obsahovat následující funkce:

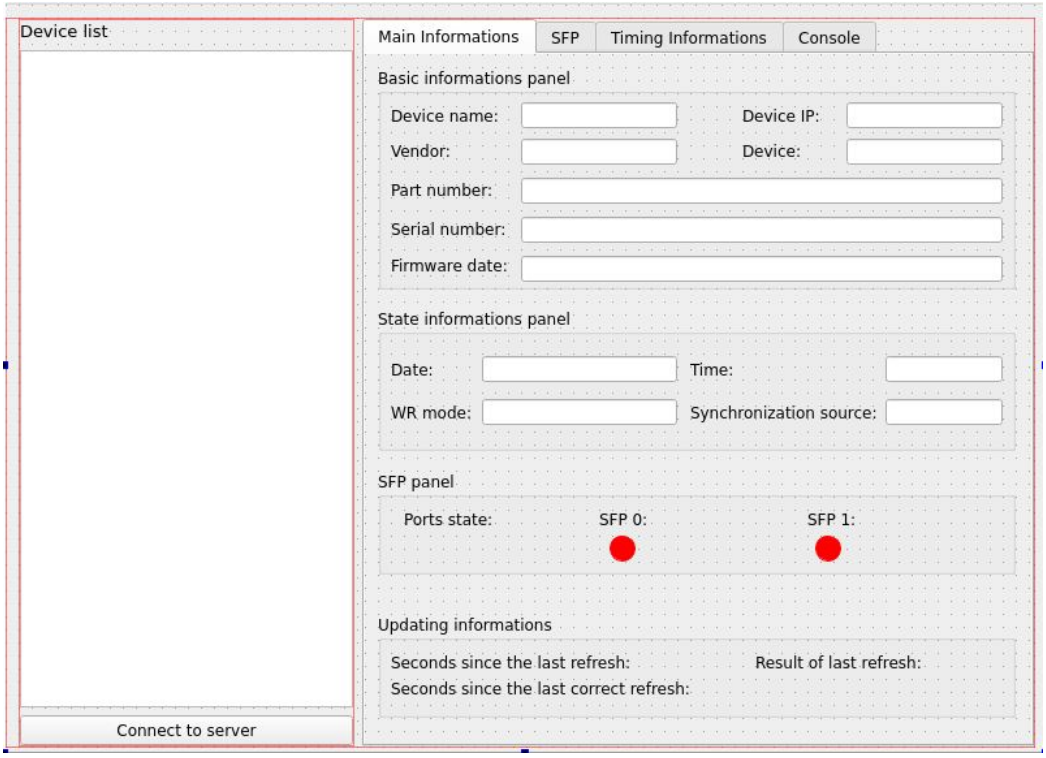
- Připojení a komunikace se vzdáleným serverem pomocí REST API
- Zobrazení stavů WR zařízení v grafické aplikaci, které jsou průběžně aktualizovány
- Podpora přímé komunikace s WR zařízením pomocí vzdálené emulace konzole

S využitím těchto požadavků a blokového diagramu celého systému na obrázku 3.2 byl navržen systém tříd a jejich propojení, které implementují dané funkcionality. Blokové schéma tohoto návrhu můžete vidět na obrázku 3.6. Hlavní částí klientské aplikace je okno *MainWindow*, které slouží k zobrazení všech potřebných informací. Tato třída spojuje všechny potřebné komponenty, které jsou potřebné pro funkci aplikace. Mezi nejdůležitější komponentu patří třída *RestConnector*, která obsahuje metody potřebné pro komunikaci s REST serverem. Všechny metody jsou asynchronní, takže nezpomalují chod aplikace.

Aplikace se skládá z několika fází, které se periodicky opakují. Životní cyklus aplikace probíhá následovně:

1. Pro připojení k serveru slouží dialog *ConnectDialog*, uživatel zadá potřebné údaje a potvrdí tlačítkem OK.
2. Je přečten seznam připojených zařízení, uložen do třídy *DeviceList* a zobrazen v pravé části aplikace.
3. Následně je spuštěn časovač, který periodicky čte stav vybraného zařízení. Informace o komunikaci jsou zobrazovány ve spodní části aplikace.

V následujících kapitolách bude podrobněji popsán vývoj a jednotlivé části klientské aplikace.




**Obrázek 3.7:** Grafické rozhraní klienta

### 3.5.1 Grafické rozhraní aplikace

Hlavní součástí aplikace je grafické rozhraní, které bude sloužit pro vizualizaci stavu WR zařízení. Pro vývoj GUI byl použít framework QT, který je popsán v kapitole 3.3. Součástí frameworku je nástroj QT Designer, který programátorovi zjednodušuje vývoj grafického rozhraní. Jedná se o program, který slouží pro automatické generování zdrojového kódu na základě návrhu GUI. Vygenerovaný kód je pak možné vložit do aplikace a implementovat následné chování. Tímto způsobem je možné oddělit grafický návrh a logiku aplikace. Každou část vyvíjet nezávisle.

Na základě požadavků na aplikaci byl vytvořen jednoduchý layout, který je zobrazen na obrázku 3.7.

V levé dolní části aplikace se nachází tlačítko pro připojení k serveru. Po stisku se zobrazí dialog, kam se zadá IP adresa serveru a potvrdí se stiskem tlačítka OK. Po úspěšném připojení, se přečte seznam připojených WR zařízení, který se zobrazí v listu vlevo. Hlavní část okna tvoří panel se stavem vybraného zařízení. Nachází se zde několik záložek, které obsahují jednotlivé parametry zařízení. Součástí aplikace je také emulátor konzole, který slouží pro přímou komunikaci s WR zařízením. Poslední částí aplikace je status bar, který se nachází ve spodní části aplikace. Jsou zde zobrazeny informace o poslední aktualizaci parametrů a také čas, který uplynul od poslední úspěšné aktualizace. Tímto způsobem lze zjistit aktuálnost parametrů.

### 3.5.2 Komunikace se serverem

Pro komunikaci s REST API serverem (kapitola 3.4.4) slouží třída *RestConnector*. Ta interně používá třídu *QNetworkRequest*, která se používá k odesílání HTTP požadavků. Veškeré metody jsou asynchronní, po vykonání se zavolá uživatelsky definovaná funkce, takže metody neblokují interakci aplikace s uživatelem.

Použití třídy můžete vidět ve zdrojovém kódu 3.10. Pro připojení k serveru je použita metoda *SetDomain()*. Poté je možné použít metodu *GetDeviceList()* pro získání seznamu zařízení, která jsou připojena ke vzdálené diagnostické aplikaci. Pro čtení samotných parametrů slouží metody *GetXXX()*. Všechny metody jsou asynchronní. Po jejich dokončení je zavolána funkce, která je předána jako jeden z parametrů, v tomto případě se jedná o anonymní lambda funkci.

```
#include <stdio>
#include "wrlen/RestConnector.h"

// Init variables
WR::RestConnector restConnector;
DeviceList::csptr deviceList;
// Connect to server
restConnector.SetDomain("http://127.0.0.1:10000");
// Get device list from remote server
restConnector.GetDeviceList([deviceList](const DeviceList::
    csptr list){
    deviceList = list;
});

.
.
.

// Get VerCmd
restConnector.GetVersion((*deviceList)["master"], [](const WR::
    VerCmd& response){
    // Print vendor of WR device
    std::cout << response.fru.vendor << std::endl;
});
```

**Zdrojový kód 3.10:** Příklad použití třídy *RestConnector*

## 3.6 Zdrojové kódy

Veškeré zdrojové kódy a dokumentace jsou dostupné v git repositáři FEL ČVUT [11]. Součástí repositáře jsou již zmíněné zdrojové kódy aplikace spolu

s jejich kompletní dokumentací. Nacházejí se zde také uživatelské návody pro kompilaci, nastavení aplikace a také zdrojové kódy této práce pro LaTeX.

Repositář obsahuje následující složky:

- **daemon** - Zdrojové kódy serverové aplikace (kapitola 3.4.4)
- **doc** - Dokumentace ke zdrojovým kódům
- **external** - Použité externí knihovny
- **gui** - Zdrojové kódy klientské aplikace (kapitola 3.5)
- **scripts** - Podpůrné skripty
- **thesis** - Zdrojové kódy práce pro LaTeX

# Kapitola 4
# Výsledky a testování

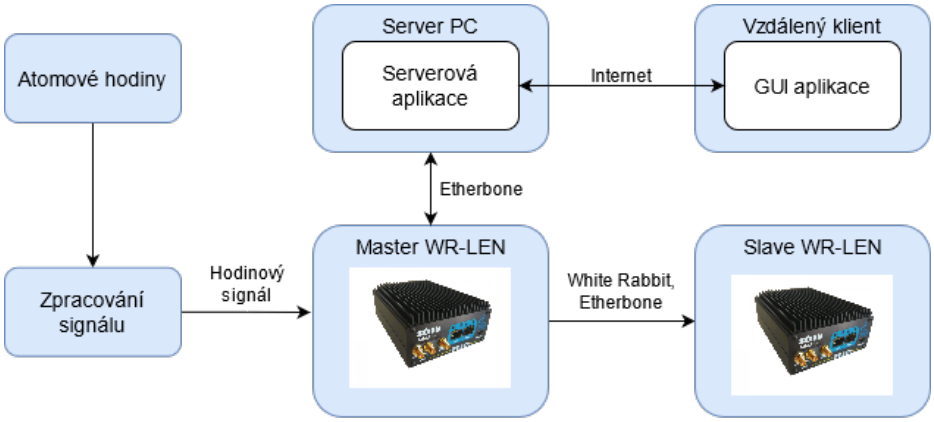
V kapitole 3 byl popsán celý postup návrhu a vývoje diagnostického systému White Rabbit, v sekci 3.4 vývoj serverové části systému, v sekci 3.5 vývoj vzdáleného klienta. Po úspěšném dokončení vývoje bylo třeba provést důkladné otestování obou částí systému tak, aby bylo ověřena jeho funkčnost a mohl být nasazen do reálného provozu.

Serverová část aplikace byla testována v laboratoři přesného času a frekvence FEL ČVUT. Pro otestování komunikace s WR zařízeními byly použity moduly WR-LEN vyrobené firmou Seven Solutions. Popis těchto zařízení je uveden v kapitole 4.1. Zařízení byla zapojena v konfiguraci Master-Slave, jak je vidět na obrázku 4.1. Serverová část diagnostického systému byla spuštěna na notebooku s operačním systémem Ubuntu 20.04, na který se vzdáleně připojoval notebook s klientskou aplikací.

## 4.1 WR-LEN

K otestování funkce diagnostického systému byly použity moduly WR-LEN (obrázek 4.2). White Rabbit Lite Embedded Node (WR-LEN) [10] je zařízení vyrobené firmou Seven Solutions podporující technologii White Rabbit. Výrobce udává, že zařízení je schopné poskytnout přesnost časové synchronizace na vzdálenost až 80 km s použitím optického vlákna s přesností menší než jedna nanosekunda. Pro připojení do sítě zařízení disponuje dvěma SFP porty, z nichž jeden je nakonfigurován jako master, druhý jako slave. Dále zařízení obsahuje vstup pro 1 PPS signál a 10 MHz signál, které jsou také synchronizovány White Rabbit sítí. Pro komunikaci se zařízením lze použít protokol EtherBone (kapitola 2.3), případně lze použít integrovaný UART.

Pro komunikaci s WR-LENy výrobce poskytuje grafickou a konzolovou aplikaci napsané v jazyce python, jejichž zdrojové kódy jsou volně dostupné [9]. Tato aplikace slouží pro emulaci sériového portu pomocí protokolu EtherBone. Aplikace sloužila jako inspirace pro přesný postup navázání spojení a vzdálenou komunikaci. Manuál k této aplikaci a seznam všech možných příkazů lze nalézt na webových stránkách výrobce [10].




**Obrázek 4.1:** Zapojení WR testovací sítě


**Obrázek 4.2:** White Rabbit Lite Embedded Node (převzato z [10])

### 4.1.1 Omezení aplikace

Z důvodu použití těchto zařízení jim byl zdrojový kód aplikace přizpůsoben. Týká se to zejména komunikace protokolem EtherBone, pomocí kterého se aplikace připojuje k interní UART periferii WR-LENu. Nelze zaručit, že jiná WR zařízení budou mít stejné adresy registrů UART periferie. Ve zdrojovém kódu se také posílají do zařízení příkazy, které mohou mít specifickou strukturu, a nemusejí být stejné pro jiná WR zařízení. Pokud by uživatel chtěl použít aplikaci pro jiná zařízení, musí si být vědom těchto omezení a zdrojový kód aplikace případně modifikovat.

## 4.2 Testování serverové části aplikace

V serverové části aplikace bylo třeba otestovat dvě funkce aplikace - REST API server, který slouží ke vzdálené komunikaci, a automatický diagnostický systém. Systém byl nakonfigurován tak, aby využíval dvě připojená zařízení (viz obrázek 4.1). Perioda pro automatickou diagnostiku byla nastavena na 5000 ms. Téměř totožný konfigurační soubor můžete vidět ve zdrojovém kódu 3.9.

### 4.2.1 REST API server

REST API server byl testován ve dvou fázích. V první fázi byl vypnut automatický diagnostický systém, aby nebyl blokován přístup k WR-LENům. V druhé fázi testování byl automatický diagnostický systém zapnut a bylo sledováno, zda nedochází k přílišnému blokování a zpoždění komunikace.

Pro testování vzdálené komunikace byl použit nástroj *curl*, který je součástí většiny linuxových distribucí. Tento program slouží pro komunikaci v různých počítačových sítích, podporuje také protokol HTTP, který používá REST server. Výstup odezvy serveru pro zdroj */devices/slave/stat*, který slouží k získání stavu zařízení se jménem slave, můžete vidět ve zdrojovém kódu 4.1. Odezva serveru je relativně rychlá, průměrná odezva systému byla do 1,5 sekundy, kde je většina času spotřebována samotnou komunikací s WR-LENy, odezva HTTP komunikace se liší dle délky komunikace.

Po zapnutí automatického diagnostického systému nastává problém se sdílením zdrojů, jelikož není možné komunikovat s jedním WR-LENem zároveň z diagnostického systému a REST serveru. Ve zdrojovém kódu je přístup k zařízením chráněn mutexy, takže se nemůže stát, že by jedna komunikace přerušovala druhou. Nicméně může nastat případ, že diagnostický systém bude během automatické kontroly blokovat přístup k WR-LENům REST serveru, takže se odezva serveru může zpozdit. Pro zachování přijatelné odezvy serveru je tedy nutné, aby perioda automatické kontroly nebyla příliš krátká. Při testech bylo zjištěno, že perioda **5000 ms** je vhodným kompromisem mezi odezvou REST serveru a aktuálnosti dat z automatické kontroly WR zařízení.

```
curl x.x.x.x:10000/devices/slave/stat

{
    "Temp": 35,
    "TempFPGA": 43,
    "WRMode": "WRC_SLAVE_WR0",
    "wr": [
        {
            "CableRttDelay": 610711,
            "ClockOffset": -22510,
            "Link": true,
            "Locked": true,
```

```
            "MasterPhyDelay": {
                "RX": 46407,
                "TX": 167843
            },
            "MasterSlaveDelay": 534045,
            "PhaseSetpoint": 4443,
            "PhaseTracking": true,
            "RX": 24512825,
            "RoundTripTime": 1053611,
            "ServoState": "'SYNC_NSEC'",
            "SlavePhyDelay": {
                "RX": 46407,
                "TX": 46407
            },
            "SynchronizationSource": "wr0",
            "TX": 7005938,
            "TotalLinkAsymmetry": -14479,
            "UpdateCounter": 6030703,
            "_ad": "65000",
            "_aux": "0",
            "_hd": "56772",
            "_md": "32361",
            "time": {
                "Nanoseconds": 299286160,
                "Seconds": 6989433
            }
        },
        {
            "Link": false,
            "Locked": true,
            "RX": 0,
            "TX": 17439671
        }
    ]
}
```


**Zdrojový kód 4.1:** Testování vzdálené komunikace

### 4.2.2 Diagnostický systém

Druhou části serverové části je automatický diagnostický systém. Ten v pravidelných intervalech čte stav všech připojených WR zařízení a kontroluje správnost jednotlivých parametrů dle pravidel definovaných v konfiguračním souboru. Pro účely testování byla zvolena jednoduchá pravidla, jejichž správnost je možné jednoduše ověřit (např. sériové číslo nebo teplota zařízení).

Výstup jedné periody testování můžete vidět ve zdrojovém kódu 4.2, kde bylo testováno zařízení **master**. Při kontrole se ze zařízení nejdříve přečtou

všechny důležité parametry, které jsou uloženy. Zde si můžeme všimnout, že byl diagnostický systém přerušen požadavkem z REST serveru, který v tuto chvíli dostal přednost. Požadavek serveru byl obsloužen a následně pokračovalo vykonávání automatické kontroly.

Při ověřování správnosti parametrů dle definovaných pravidel se našly dvě chyby:

- Parametr **FRU_serial** měl špatnou hodnotu, byla očekávána hodnota **12345**
- Teplota WR-LENu byla příliš nízká, očekávaná hodnota byla větší než **50**

Po vykonání kontroly byl vykonán uživatelský skript **shell.sh**, který v tomto případě uložil chyby do souboru. Skript také může sloužit např. k oznámení chyby emailem nebo SMS. Z logu můžeme také vidět návratovou hodnotu skriptu, **0** v tomto případě znamená úspěšné vykonání.

```
Checking device "master"
 .   { Updating device informations
 .   .   Using cached device
 .   .   Getting "ver" from device
 .   .   Locking device mutex
 .   .   Sending data to device
 .   .   Reading response from device
 .   .   Using cached device
 .   .   Getting "stat" from device
 .   .   Locking device mutex
 .   .   Sending data to device
 .   .   Reading response from device
 .   .   Using cached device
 .   .   Getting "mac" from device
 .   .   Locking device mutex
 .   .   Sending data to device
 .   .   Reading response from device
 .   .   Getting "ip" from device
 .   .   Locking device mutex
 .   .   Sending data to device
 .   .   { GET http://x.x.x.x:10000/devices/slave/stat HTTP 1.1
 .   .   .   Using cached device
 .   .   .   Getting "stat" from device
 .   .   .   Locking device mutex
 .   .   .   Sending data to device
 .   .   .   Reading response from device
 .   .   .   Getting "stat" from device
 .   .   .   Locking device mutex
 .   .   .   Sending data to device
 .   .   .   Reading response from device
```

```
.   .   .   Request was served succesfully
.   .   } 1,156 s: GET http://x.x.x.x:10000/devices/slave/stat
    HTTP 1.1
.   .   Reading response from device
.   } 5,016 s: Updating device informations
.   { Cheching device status
.   .   FRU_serial has wrong value, got "7SWRLENv1.1-S03_723",
    excepted "12345"
.   .   Temperature too high, got "33", excepted more than "50
    "
.   .   Executing shell command: sh ./shell.sh, returned value
    0
.   } 0,005 s: Cheching device status
```

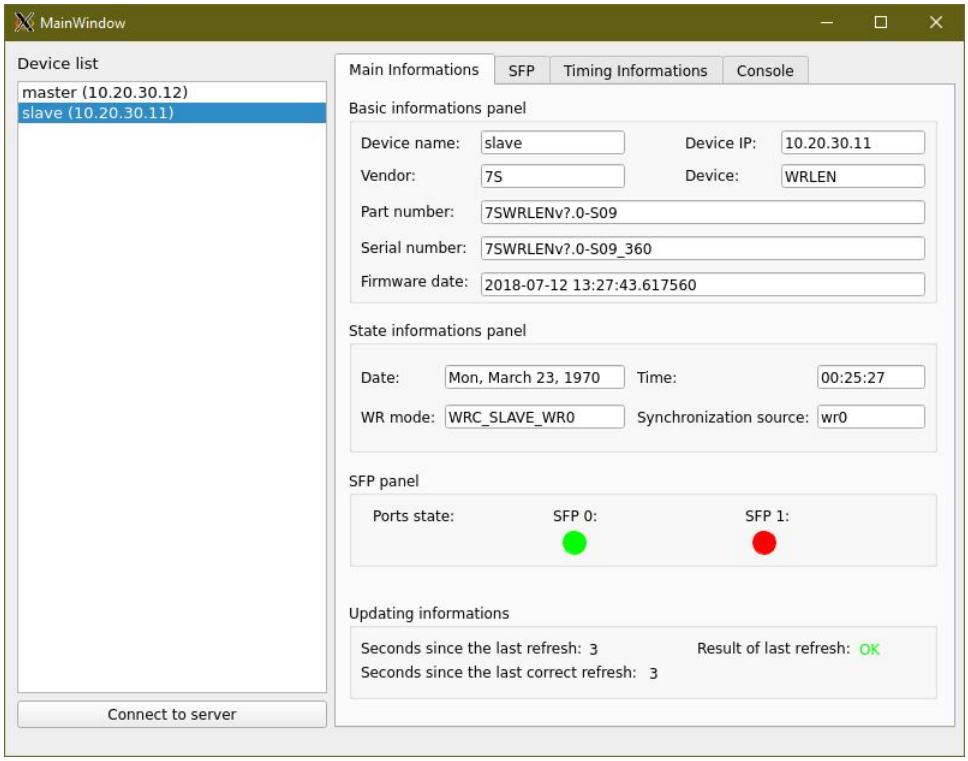
**Zdrojový kód 4.2:** Testování automatický diagnostický systému

## 4.3 Klientská aplikace

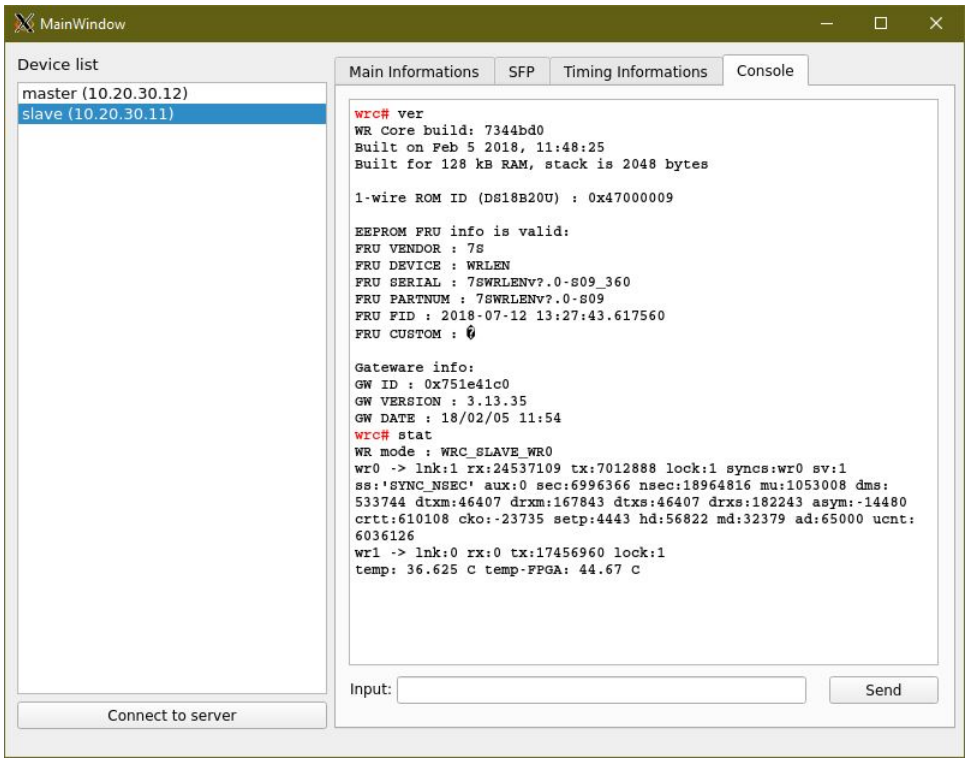
Druhou částí systému, která byl vytvořena a je jí také nutné otestovat, je klientská grafická aplikace, která slouží pro vzdálený přístup do White Rabbit sítě a vizualizaci stavů všech připojených WR zařízení, která jsou připojena k serveru. Aplikace pro přístup k serveru používá REST API server, jehož funkčnost byla otestována v předchozí kapitole 4.2.1. Při testování grafické aplikace je třeba se soustředit zejména na odezvu na uživatelské vstupy a správnost zobrazených dat.

Po zapnutí aplikace je třeba se připojit k serveru stiskem tlačítka *Connect to server*. Po připojení se v levé části obrazovky zobrazí seznam připojených zařízení, vpravo jsou zobrazeny všechny důležité parametry. Obrazovku s hlavními parametry můžete vidět na obrázku 4.3, kde jsou zobrazeny základní informace o vybraném zařízení (jméno, výrobce), dále aktuální datum a čas, a také stav SFP portů. V dolní části obrazovky je zobrazen výsledek poslední aktualizace parametrů a čas od poslední úspěšné aktualizace. Parametry WR zařízení se automaticky obnovují po 30 sekundách, aby nebyl příliš často narušen automatický diagnostický systém na serveru.

Na obrázku 4.4 můžete vidět jednu z dalších záložek aplikace, která umožňuje přímou komunikaci s WR zařízením - slouží jako emulace sériového portu. Vložený příkaz se odešle na server, který získá odpověď od zařízení a odešle ji zpět klientovi. Tímto způsobem může uživatel získat parametry, které nejsou přístupné v grafickém rozhraní nebo změnit nastavení vzdáleného zařízení.




**Obrázek 4.3:** Ukázka grafického rozhraní aplikace




**Obrázek 4.4:** Ukázka přímé komunikace

# Kapitola 5

# Závěr

Cílem této práce práce bylo navrhnout a vytvořit automatický diagnostický systém pro White Rabbit zařízení, který uživateli usnadní detekci a opravu chyb. Tento cíl se podařilo splnit. Výsledný systém je složen ze dvou částí. Serverová aplikace poskytuje možnost vzdáleného monitorování zařízení pomocí REST API a provádí automatickou kontrolu všech připojených WR zařízení. Chování aplikace je možné dynamicky nastavit v konfiguračním souboru. Mezi nastavitelné parametry patří např. port, na kterém běží REST server nebo perioda automatické kontroly. Pro účel autonomní diagnostiky byl vytvořen systém pravidel, který uživateli umožňuje specifikovat, jaké parametry zařízení se budou kontrolovat a jejich očekávané hodnoty.

Druhou částí systému je grafická aplikace, která slouží pro vzdálené monitorování WR zařízení připojených k serveru. Aplikace komunikuje pomocí REST API se serverem a periodicky čte vlastnosti vybraného zařízení, které se zobrazují přehledně v grafickém rozhraní. Součástí aplikace je také emulátor konzole, která slouží pro přímou komunikaci se zařízeními a může sloužit např. pro změnu nastavení.

Celý diagnostický systém byl otestován v reálném provozu v laboratoři, kde se pro implementaci White Rabbit používají moduly WR-LEN vyrobené firmou Seven Solutions. Při testech bylo ověřeno, že jednotlivé části diagnostického systému jsou plně funkční. Diagnostický systém žádným způsobem neovlivňuje časovou synchronizaci a v přijatelném čase poskytuje informace pro vzdálené monitorování sítě. Klientská aplikace je responzivní, vzdálené připojení k serveru je plně funkční.

Testy potvrdily, že výsledný diagnostický systém je plně funkční a je schopen nasazení do reálného provozu.

# Příloha A

## Literatura


[1] EtherBone Core. https://ohwr.org/project/etherbone-core. Open Hardware Repository, Citováno: 28.4.2021.

[2] Loguru: a lightweight and flexible C++ logging library. https://github.com/emilk/loguru. Github Repository, Citováno: 28.4.2021.

[3] White Rabbit. https://ohwr.org/project/white-rabbit/wikis/home. Citováno: 28.4.2021.

[4] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems - Redline. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002) - Redline*, pages 1–300, 2008.

[5] Cicuttin A. Introduction to the WISHBONE Bus Interface. http://indico.ictp.it/event/a11204/session/35/contribution/22/material/0/0.pdf, 2012. Citováno: 28.4.2021.

[6] Ferrant J.-L., Gilson M., Jobert S., Mayer M., Ouellette M., Montini L., Rodrigues S., Ruffini S. Synchronous ethernet: a method to transport synchronization. *IEEE Communications Magazine*, 46(9):126–134, 2008.

[7] Kreider M., Terpstra W., Lewis J., Serrano J., Wlostowski T. Etherbone - a network layer for the wishbone soc bus. In *Contributions to the Proceedings of ICALEPCS 2011*, page 1423, France, 2011.

[8] Lipiński M., Włostowski T., Serrano J.,Alvarez P. White rabbit: a ptp application for robust sub-nanosecond synchronization. In *2011 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pages 25–30, 2011.

[9] Seven Solutions. py7slib. https://github.com/HERA-Team/py7slib. Github Repository, Citováno: 16.5.2021.

[10] Seven Solutions. WR-LEN. https://sevensols.com/home/timing-products/wr-len/. Citováno: 16.5.2021.

[11] Tefr F. WR-LEN monitoring. https://gitlab.fel.cvut.cz/tefrfili/wr-len-monitoring. Gitlab Repository FEE CTU, Citováno: 21.5.2021.

[12] The Qt Company. Qt documentation. `https://doc.qt.io`. Citováno: 4.5.2021.

[13] The Qt Company. Qt framework. `https://www.qt.io`. Citováno: 2.5.2021.

[14] The Qt Company. QtHTTPServer. `https://github.com/qt-labs/qthttpserver`. Github Repository, Citováno: 5.5.2021.

# Příloha B

## Dokumentace

Na následujících stránkách jsou zobrazeny úryvky z dokumentace k diagnostickému systému, které popisují hlavní chování a ovládání programu.

# REST API

Pro externí komunikaci se diagnostickým serverem lze použít jednoduché REST API. Jedná se o HTTP requesty, které vracejí JSON soubor s odpovědí. Pro otestování spojení lze použít nástroj `curl`. K otestování komunikace byl použit lokální server na portu 10000. Server poskytuje následující zdroje:

## /test

**Popis**: Otestování spojení
**HTTP metoda**: GET
**Příklad použití**: curl 127.0.0.1:10000/test
**Odpověď**: Text

```
Test of server
```

## /devices

**Popis**: Seznam připojených zařízení z konfiguračního souboru
**HTTP metoda**: GET
**Příklad použití**: curl 127.0.0.1:10000/devices
**Odpověď**: Zformátovaný seznam zařízení

```
{
    "devices": [
        {
            "ip": "10.20.30.12",
            "name": "master",
            "serialPort": "/dev/ttyUSB1"
        },
        {
            "ip": "10.20.30.11",
            "name": "slave",
            "serialPort": "/dev/ttyUSB0"
        }
    ]
}
```

## /devices/<arg>/ver

**Parametr <arg>**: Jméno zařízení z konfiguračního souboru
**Popis**: Zavolání příkazu *ver* na zařízení <*arg*>, základní informace o WR-LENu
**HTTP metoda**: GET
**Příklad použití**: curl 127.0.0.1:10000/devices/slave/ver
**Odpověď**: Zformátovaná odpověď příkazu

```
{
    "FRU": {
        "device": "WRLEN",
        "fid": "2018-07-12 13:27:43.617560",
        "partnum": "7SWRLENv1.0-S09",
        "serial": "7SWRLENv1.0-S09_360",
        "vendor": "7S"
    },
    "GW": {
        "date": "18/02/05 11:54",
        "id": "0x751e41c0",
        "version": "3.13.35"
    },
    "WRCore": {
        "build": "7344bd0",
        "info": "Built for 128 kB RAM, stack is 2048 bytes",
        "time": "Feb  5 2018, 11:48:25"
    }
}
```


## /devices/<arg>/stat

**Parametr <arg>**: Jméno zařízení z konfiguračního souboru
**Popis**: Zavolání příkazu *stat* na zařízení <*arg*>, aktuální stav WR-LENu
**HTTP metoda**: GET
**Příklad použití**: curl 127.0.0.1:10000/devices/slave/stat
**Odpověď**: Zformátovaná odpověď příkazu


```
{
    "Temp": 36,
    "TempFPGA": 44,
    "WRMode": "WRC_SLAVE_WR0",
    "wr": [
        {
            "CableRttDelay": 610299,
            "ClockOffset": -15349,
            "Link": true,
            "Locked": true,
            "MasterPhyDelay": {
                "RX": 46407,
                "TX": 167843
            },
            "MasterSlaveDelay": 533839,
            "PhaseSetpoint": 4456,
            "PhaseTracking": true,
            "RX": 12382711,
            "RoundTripTime": 1053199,
            "ServoState": "'SYNC_NSEC'",
            "SlavePhyDelay": {
                "RX": 46407,
```

```
                "TX": 46407
            },
            "SynchronizationSource": "wr0",
            "TX": 3530479,
            "TotalLinkAsymmetry": -14479,
            "UpdateCounter": 2985958,
            "_ad": "65000",
            "_aux": "0",
            "_hd": "56782",
            "_md": "32167",
            "time": {
                "Nanoseconds": 252631600,
                "Seconds": 3521245
            }
        },
        {
            "Link": false,
            "Locked": true,
            "RX": 0,
            "TX": 8785818
        }
    ]
}
```


## /devices/<arg>/eth

**Parametr <arg>**: Jméno zařízení z konfiguračního souboru
**Popis**: Zavolání příkazu *eth* na zařízení *<arg>*, seznam aktuálně připojených zařízení k jednotlivým portům
**HTTP metoda**: GET
**Příklad použití**: curl 127.0.0.1:10000/devices/slave/eth
**Odpověď**: Zformátovaná odpověď příkazu


```
{
    "entries": [
        [
            {
                "MAC": "8:0:30:b4:6b:b5",
                "age": "59"
            },
            {
                "MAC": "2c:53:4a:0:0:cc",
                "age": "60"
            }
        ],
        [
        ]
    ]
}
```

## /devices/<arg>/mac

**Parametr <arg>**: Jméno zařízení z konfiguračního souboru
**Popis**: Zavolání příkazu *mac* na zařízení *<arg>*, IP a MAC adresy jednotlivých portů WR-LENu + aktuální mód operace portu (master nebo slave)
**HTTP metoda**: GET
**Příklad použití**: curl 127.0.0.1:10000/devices/slave/mac
**Odpověď**: Zformátovaná odpověď příkazu

```
{
    "entries": [
        {
            "ip": "10.20.30.11",
            "mac": "08:00:30:b1:51:b9",
            "mode": "Slave"
        },
        {
            "ip": "10.20.30.11",
            "mac": "08:00:30:b1:51:ba",
            "mode": "Master"
        }
    ]
}
```

## /devices/<arg>

**Parametr <arg>**: Jméno zařízení z konfiguračního souboru
**Popis**: Poslání obecného příkazu na zařízení, spolu s tímto příkazem je nutné poslat JSON s parametrem `cmd`, který specifikuje volaný příkaz **HTTP metoda**: POST
**Příklad použití**: curl 127.0.0.1:10000/devices/slave -d '{"cmd":"ver"}'
**Odpověď**: Nezformátovaná odpověď příkazu

```
{
    "response": "WR Core build: 7344bd0\r\nBuilt on Feb  5 2018,
11:48:25\r\nBuilt for 128 kB RAM, stack is 2048 bytes\r\n\r\n1-wire ROM ID
(DS18B20U) : 0x47000009 \r\n\r\nEEPROM FRU info is valid: \r\n FRU VENDOR
: 7S \r\n FRU DEVICE  : WRLEN \r\n FRU SERIAL  : 7SWRLENv?.0-S09_360 \r\n
FRU PARTNUM : 7SWRLENv?.0-S09 \r\n FRU FID     : 2018-07-12 13:27:43.617560
\r\n FRU CUSTOM  : ? \r\n\r\nGateware info: \r\n GW ID      :
0x751e41c0\r\n GW VERSION : 3.13.35\r\n GW DATE    : 18/02/05 11:54"
}
```

## B.2 Konfigurace serveru



# Konfigurace programu

Veškerá konfigurace serveru se provádí v konfikuračním souboru `config.yaml` ve formátu YAML. Soubor obsahuje jednoduchou stromovou strukturu, pomocí které lze jednuduše přispůsobit chování programu potřebám uživatele.

Soubor je rozdělen do několika částí. Každá znich ovládá jinou část programu. Popis jednotlivých části je uveden níže.

## programConfig

V této části se ovládá základní chování programu. Je zde možné nastavit následující parametry:

| Název parametru | Typ | Defaultní hodnota | Význam |
|---|---|---|---|
| httpPort | Int | 10000 | Číslo portu, na kterém běří REST API server |
| checkerPeriodMs | Int | 10000 | Perioda v ms, v kterou program automaticky kontroluje stav nařízení |
| wrLenReadDelayMs | Unsigned Int | 2000 | - |
| errorCmd | String | "" | Příkaz, který se vykoná po detekování chyby |
| logDir | String | Logs | Složka, do které se logují informace o programu a stavu zařízení |

Ukázku konfigurace můžete vidět níže.

```
programConfig:
  # Port of REST API server
  httpPort: 10000
  # Period in ms, when devices status is checked
  checkerPeriodMs: 10000
  # TODO
  wrLenReadDelayMs: 2000
  # Command which is called by DeviceChecker when error occurs
  errorCmd: "sh ./shell.sh"
```

## devices

V této části souboru se nastavuje seznam zařízení, se kterými aplikace komunikuje. Jedná se o pole zařízení, u každého z nich je možné nastavit následující parametry:

| Název parametru | Typ | Povinost | Význam | Defaultní hodnota |
|---|---|---|---|---|

| Název parametru | Typ | Povinost | Význam | Defaultní hodnota |
|---|---|---|---|---|
| ip | String | Povinný | IP adresa WR-LENu | - |
| serialPort | String | Nepovinný | Jméno sériového portu, ke kterému je WR-LENu připojen | - |
| name | String | Povinný | Jméno WR-LENu, používá se pro identifikaci | - |
| log | Bool | Nepovinný | Logovat informace o zařízení | False |

Ukázku konfigurace můžete vidět níže.

```
# Array of connected WR-LEN devices
devices:
    -
      ip: 10.20.30.12
      serialPort: /dev/ttyUSB1
      name: master
      log: True

    -
      ip: 10.20.30.11
      serialPort: /dev/ttyUSB0
      name: slave
      log: True
```

## rules

V této části konfigurace se nastavují pravidla, podle kterých se vyhodnocuje stav připojených WR-LENů. Každý parametr lze vyhodnocovat pomocí následujících pravidel:

| Název pravidla | Význam |
|---|---|
| equal | Rovnost |
| nequal | Nerovnost |
| less | Menší než |
| more | Větší než |

Pravidla je možné nastavit pro každé zařízení zvlášť, jako identifikátor slouží jméno zařízení. Lze také použít identifikátor **global**, jehož pravidla jsou použity pro všechna vyjmenovaná zařízení. **Pokud není zařízení v seznamu, není jeho stav vyhodnocován ani globálními pravidly. Pokud chceme pro WR-LEN použít jen globální pravidla, je nutné vložit prázdný node bez pravidel (viz příklad).** Pokud je ve jméně parametru přípona [id], jedná se o tzv. vektorový parametr, který se používá pro indexování pole parametrů. Indexace se provádí pomocí hranatých závorek od nuly. Ke pravidlu lze také přidat parametr `text`, který se používá ke generování popisu chyby. K popisu chyby se použije tento parametr s doplnění o aktuální a očekávanou hodnotu parametru.

## Skupiny parametrů

Jednotlivé parametry jsou rozděleny do několika skupin dle příkazu, ve kterém se nacházejí. Aktuálně je možné vybírat z následujích skupin:

**verCmd**

| Název parametru | Typ | Význam | Očekávané hodnoty |
|---|---|---|---|
| WrCore_build | String | Jméno WR Core build | - |
| WrCore_buildTime | String | Čás WR Core build | - |
| WrCore_buildInfo | String | Informace WR Core build | - |
| FRU_vendor | String | Jméno výrobce | 7S |
| FRU_device | String | Jméno zařízení | WRLEN |
| FRU_serial | String | Sériové číslo | - |
| FRU_partnum | String | Výrobní číslo | - |
| FRU_fid | String | - | - |
| Gateware_id | String | - | - |
| Gateware_version | String | - | - |
| Gateware_date | String | - | - |

**statCmd**

Bližší informace na https://afi-project.jinr.ru/attachments/download/884/wrpc-v3.0.pdf, sekce *Appendix B WRPC GUI elements*. Vektorové parametry jsou označeny příponou [id], kde id je číslo portu číslované od 0.

| Název parametru | Typ | Význam | Očekávané hodnoty |
|---|---|---|---|
| WRMode | String | Operační mód WR PTP CORE | WR Slave, WR Master |
| Temp | Float | Teplota | - |
| TempFPGA | Float | Teplota FPGA | - |
| Link[id] | Bool | Status spojení | True, False |
| Rx[id] | Int | Číslovač packetů RX | - |
| Tx[id] | Int | Číslovač packetů TX | - |
| Locked[id] | Bool | Soft PLL lock | True, False |
| SynchronizationSource[id] | String | **WR Slave only**, Jméno portu, ze kterého s eWR daemon synchronizuje | wr0, wr1 |

| Název parametru | Typ | Význam | Očekávané hodnoty |
|---|---|---|---|
| PhaseTracking[id] | Bool | **WR Slave only**, Phase tracking zapnuto? | True, False |
| ServoState[id] | String | **WR Slave only**, Aktuální stav WR stavového automatu | Uninitialized, SYNC_SEC, SYNC_NSEC, SYNC_PHASE, **TRACK_PHASE** |
| Time_sec[id] | Int | **WR Slave only**, Unixový čas v sekundách (od 1.1.1970) | - |
| Time_nsec[id] | Int | **WR Slave only**, Čas [ns] | - |
| RoundTripTime[id] | Int | **WR Slave only**, Round-trip spoždění (delay__MM) [ps] | - |
| MasterSlaveDelay[id] | Int | **WR Slave only**, Odhadnuté spoždění master -> slave (delay_MS)[ps] | - |
| MasterPhyDelay_rx[id] | Int | **WR Slave only**, Spoždění vysílání/příjmu hardware mastera RX [ps] | - |
| MasterPhyDelay_tx[id] | Int | **WR Slave only**, Spoždění vysílání/příjmu hardware mastera TX [ps] | - |
| SlavePhyDelay_rx[id] | Int | **WR Slave only**, Spoždění vysílání/příjmu hardware slave RX [ps] [ps] | - |
| SlavePhyDelay_tx[id] | Int | **WR Slave only**, Spoždění vysílání/příjmu hardware slave TX [ps] [ps] | - |
| totalLinkAsymmetry[id] | Int | **WR Slave only**, Asimetrie WR spojení (delay_MM - 2.delay_MS) [ps] | - |
| CableRttDelay[id] | Int | **WR Slave only**, Round-trip spoždění optického vlákna [ps] | - |
| ClockOffset[id] | Int | **WR Slave only**, Ofset hodin Slave-Master (offset_MS)[ps] | - |
| PhaseSetpoint[0] | Int | **WR Slave only**, Aktuální fázový posun [ps] | - |
| UpdateCounter[0] | Int | **WR Slave only**, Časováč je zvýšen pokaždé, když je stav WR servo aktualizován | - |

**macCmd**

| Název parametru | Typ | Význam | Očekávané hodnoty |
|---|---|---|---|
| Mac[id] | String | MAC adresa portu | - |
| Ip[id] | String | IP adresa portu | **in training** pokud není adresa přizařena |
| Mode[id] | String | Mód operace portu | Master, Slave |

Příklad konfigurace pravidel

```
# Node with rules for DeviceChecker
rules:
  global:
    verCmd:
      FRU_serial:
        text: "FRU_serial has wrong value"
        equal: "12345"
    statCmd:
      Temp:
        text: "Temperature too high"
        more: 50
  # Empty node, just master just with global rules
  master: ~
  slave:
    statCmd:
      Link[0]:
        text: "Link is down"
        equal: false
    macCmd:
      Mode[0]:
        text: "Mode[0] ha wrong value"
        equal: "master"
      Mode[1]:
        text: "Mode[1] ha wrong value"
        equal: "slave"
```

## errorCmd

Pokud parametry zařízení nevyhovují pravidlům, je zavolán příkaz `errorCmd` nastavený v konfikuračním souboru. Příkaz je volán se dvěma parametry. Prvním parametrem je jméno zařízení, druhým parametrem jsou zformátované textové poposy chyb odělené znakem pro konec řádku `\n`. Příklad volaného příkazu můžete vidět níže

```
sh ./shell.sh "master" "[-1] FRU_serial has wrong value, got 7SWRLENv?.0-
S09_360, excepted 12345\n[-1] Link is down, got 1, excepted 0"
```

## Logování

Všechny informace o programu a zařízení (pokud je povoleno) jsou logovány do souborů. Výstup programu je logován do složky `<logDir>`. Jméno souboru je vytvořeno dynamicky dle času spuštění. Stav WR-LENů je logován do složek `<logDir>/<devName>`. Každá relace je ukládána do samostatného souboru, jehož jméno je také vytvářeno dle času spuštění relace.

# ZADÁNÍ DIPLOMOVÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Tefr** Jméno: **Filip** Osobní číslo: **465865**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Kybernetika a robotika**

Studijní obor: **Kybernetika a robotika**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Diagnostika systému White Rabbit**

Název diplomové práce anglicky:

**White Rabbit system diagnostics**

Pokyny pro vypracování:

1. Navrhněte HW a SW řešení (SW aplikaci) diagnostiky systému White Rabbit (WR) s moduly WR-LEN a WR Switch firmy Seven Solutions.
2. SW aplikace bude nepřetržitě monitorovat stav WR systému a v případě výpadku některého zařízení ohlásí tuto chybu pomocí SMS nebo e-mailovou zprávou na zvolenou adresu.
3. Aplikaci vyzkoušejte na reálném WR systému v Laboratoři přesného času a frekvence FEL a otestujte, zda aplikace nenarušuje primární funkci WR systému, tj. distribuci přesného času a frekvence.

Seznam doporučené literatury:

[1] Hedekvist, P. O. - Ebenhag, S. C.: Time and Frequency Transfer in Optical Fibers. In Recent Progress in Optical Fiber Research. InTech, Rijeka 2012, p. 371-386.
[2] Open Hardware Repository, White Rabbit, https://www.ohwr.org/projects/white-rabbit/wiki
[3] Dierikx, E. – Xie, Y.: White Rabbit Good Practice Guide. Dutch Metrology Institute, May 2019.
[4] Technická dokumentace Seven Solutions pro zařízení WR-LEN a WR Switch.

Jméno a pracoviště vedoucí(ho) diplomové práce:

**doc. Ing. Jaroslav Roztočil, CSc., katedra měření FEL**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **23.01.2021** Termín odevzdání diplomové práce: **21.05.2021**

Platnost zadání diplomové práce:
**do konce letního semestru 2021/2022**

| doc. Ing. Jaroslav Roztočil, CSc.<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |
|---|---|---|

## III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání

Podpis studenta