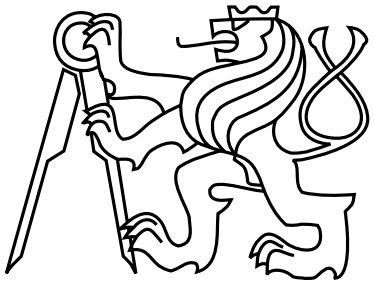
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Software pro distribuované řízení a vyčítání dat ze sítě částicových pixelových detektorů Timepix3

Bc. Jakub Begera

Vedoucí práce: Ing. Štěpán Polanský

Studijní program: Otevřená informatika

Obor: Softwarové inženýrství

8. ledna 2019

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Begera** Jméno: **Jakub** Osobní číslo: **420021**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Software pro distribuované řízení a vyčítání dat ze sítě částicových pixelových detektorů Timepix3

Název diplomové práce anglicky:

Software for distributed control and data readout for network of Timepix3 particle pixel detectors

Pokyny pro vypracování:

The objective of this diploma thesis is to design and develop software for distributed acquisition control and data readout from the network of Timepix3 particle pixel detectors. The software will provide settings of acquisition parameters for the detectors (e.g., acquisition time, detector mode, threshold, etc.) as well as data readout and data persistence. An adequate data structure and suitable storage type should be chosen. The system will be able to operate in a distributed mode for the possibility of horizontal scalability and manageability of a higher number of detectors. Another motivation for horizontal scalability is the fact that the Timepix3 detector is theoretically able to generate data flow up to 5.12 Gbps, which cannot be handled from multiple detectors by a single node. The software will also implement the Katherine communication protocol [3] ? the Timepix3 Ethernet Embedded Readout.ATLAS-TPX, a network of 32 Timepix detectors installed within ATLAS experiment at LHC at CERN, is controlled from a single central server [1], i.e., without the possibility of horizontal scalability. The Second Long Shutdown of LHC is planned in between the years 2019 and 2020, in which the modernization of ATLAS-TPX network is proposed with the usage of Timepix3 detectors and software which will be designed and developed within this diploma thesis.

Seznam doporučené literatury:

[1] Manek, P., Begera, J., Bergmann, B., Burian, P., Janecek, J., Polansky, S., ? Suk, M. (2017). Software system for data acquisition and analysis operating the ATLAS-TPX network. In 2017 International Conference on Applied Electronics (AE). IEEE. https://doi.org/10.23919/ae.2017.8053593
[2] Poikela, T., Plosila, J., Westerlund, T., Campbell, M., Gaspari, M. D., Llopart, X., ? Kruth, A. (2014). Timepix3: a 65K channel hybrid pixel readout chip with simultaneous ToA/ToT and sparse readout. Journal of Instrumentation, 9(5), C05013?C05013. https://doi.org/10.1088/1748-0221/9/05/c05013
[3] Burian, P., Broulím, P., Jára, M., Georgiev, V., & Bergmann, B. (2017). Katherine: Ethernet Embedded Readout Interface for Timepix3. Journal of Instrumentation, 12(11), C11001?C11001. https://doi.org/10.1088/1748-0221/12/11/c11001.
[4] Z. Vykydal et al., The Medipix2?-based network for measurement of spectral characteristics and composition of radiation in ATLAS detector, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, Volume 607, Issue 1, 1 August 2009, Pages 35-?37, ISSN 0168-?9002, h ttp://dx.doi.org/10.1016/j.nima.2009.03.104.
[5] D. Turecek et al., Remote control of ATLAS?-MPX Network and Data Visualization, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, Volume 633, Supplement 1, May 2011, Pages S45-?S47, ISSN 0168?-9002, http://dx.doi.org/10.1016/j.nima.2010.06.117.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Štěpán Polanský, katedra dozimetrie a aplikace ionizujícího záření FJFI

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

Datum zadání diplomové práce: **17.01.2018** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2019**

Ing. Štěpán Polanský podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. 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

Poděkování

Na tomto místě bych rád poděkoval svým kolegům z ÚTEF ČVUT v Praze za poskytnutí know-how a za podporu při realizaci této práce, zejména Ing. Petrovi Burianovi, Ph.D. za rady a trpělivost při implementaci komunikačního protokolu vyčítacího rozhraní *Katherine*. Také bych rád poděkoval vedoucímu této práce Ing. Štěpánu Polanskému za předané zkušenosti, ochotu a čas, který mi věnoval. Dále bych rád poděkoval M. Sc. Saúl Calderón Ramírez za vedení této práce během mého studijního pobytu na *Instituto Tecnológico de Costa Rica* a za jeho cenné podněty. Mé poděkování také patří mé rodině a přátelům, kteří mě během studia podporovali. Bez nich by tato práce nikdy nemohla vzniknout.

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 dne 8. 1. 2019 .....

Abstract

The aim of this master thesis is to design and implement Pixnet software – modular distributed system for control and data readout from network of particle pixel detectors. System consists of two main components – handler and master. Whereas handler is used for operation of subset of detectors and master is used for controlling of handlers and for providing API for controlling of whole system as well. Primary consumer of this API is a *single-page* web application but it can be consumed by 3rd party systems as well. Scalability of system is realized by adding handlers instances and distributing detectors across them.

The presented system has a modular architecture – instance of detector is given by his communication and data module, which has to implement defined interfaces. Thus, the system allows to operate heterogenous networks of detectors in homogenous way. Furthermore, the presented system includes a module implementation supporting *Timepix3* detector, operated by *Katherine* readout interface.

For development and testing purposes, an emulator of the *Katherine* interface has beed developed. Emulator implements subsets of commands which are needed to basic configuration of detector and data acquisition setup.

Key words: Timepix3, ATLAS-TPX, Katherine, distributed control, ionizing radiation, pixel detector, Kotlin, detectors network, modular architecture, data acquisition software

Abstrakt

Tato diplomová práce ze zabývá návrhem a implementací software Pixnet – modulárním distribuovaným systémem pro řízení a vyčítání dat ze sítě částicových pixelových detektorů. Systém se skládá ze dvou hlavních komponent, kterými jsou handler a master. Handler slouží pro operování podmnožiny jemu přiřazených detektorů a master k řízení handlerů a poskytování API pro řízení celého systému. Primárním konzumentem tohoto API je webová *single-page* aplikace, ale mohou jím být i systémy třetích stran. Škálování systému je realizováno přidáváním instancí handlerů a distribucí detektorů mezi jejich instancemi.

Navržený systém je modulární – instance detektoru je reprezentována jeho komunikačním a datovým modulem, které musí implementovat definované rozhraní. Systém tedy umožňuje operování heterogenní sítě detektorů homogenním způsobem. V rámci této práce byl implementován komunikační a datový modul, které umožňují podporu detektoru *Timepix3*, komunikujícího prostřednictvím vyčítacího rozhraní *Katherine*.

Pro účely vývoje a testování byl v rámci této práce rovněž vyvinut emulátor vyčítacího rozhraní *Katherine*, který implementuje všechny příkazy, které jsou důležité pro konfiguraci a spuštění akvizice dat.

Klíčová slova: Timepix3, ATLAS-TPX, Katherine, distribuované řízení, ionizující záření, pixelový detektor, Kotlin, detektorová síť, modulární architektura, software pro akvizici dat

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Kapitola 1

Úvod

Tato diplomová práce se zabývá návrhem a implementací softwarového systému **Pixnet** - software pro distribuované řízení sítě částicových pixelových detektorů. Software byl navržen především pro řízení detektorů *Timepix3*, ale díky modulární architektuře (po implementaci potřebných modulů) je možné ho použít na řízení libovolného detektoru.

Systém se skládá z několika částí - backendové aplikace **handleru** (software pro řízení a vyčítání dat z jemu přiřazených detektorů - viz kapitola 4), backendové aplikace **mastera** (software pro řízení sítě handlerů - viz 6.1) a frontendové webové *single-page* aplikace **mastera** (software poskytující webové uživatelské rozhraní pro řízení mastera pomocí jeho API - viz 6.2).

V rámci této práce byly implementovány moduly, které systému Pixnet umožňují řízení a vyčítání dat z detektoru *Timepix3*, připojeným pomocí vyčítacího rozhraní *Katherine*, viz kapitola 5.

1.1 Motivace

Podobný systém pro řízení sítě částicových pixelových detektorů již existuje [21, 3] a byl vyvinut pro účely řízení a vyčítání dat z experimentu *ATLAS-TPX* - sítě *Timepix* detektorů, která byla instalována v rámci experimentu *ATLAS* na *LHC* v *CERN* a měřila data během *LHC* `run 2`[1].

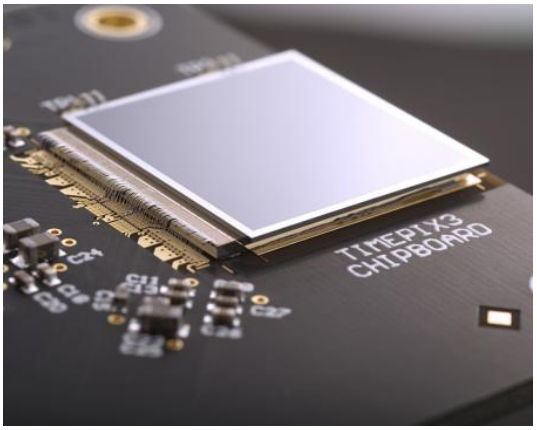
V letech 2019 až 2020 je plánován LS2[2], v rámci kterého proběhne modernizace detektorové sítě *ATLAS-TPX* za použití detektorů *Timepix3*. Z pohledu softwarové architektury modernizace přináší nová omezení, které existující systém není schopný zajistit. Především se jedná o vysoký datový tok (detektor *Timepix3* je teoreticky schopný generovat až $5,12$ Gb/s), který by současné nedistribuované řešení nebylo schopné zpracovat. Díky modulární architektuře je další výhodou nově navrženého systému možnost homogenního způsobu řízení nehomogenní detektorové sítě, tj. takové sítě, která se skládá z více druhů detektorů.

[1] Označení druhého období provozu částicového urychlovače *LHC*, který probíhal v letech 2014 až 2018.
[2] Z angl. *Long Shutdown 2*, technologická odstávka částicového urychlovače *LHC*.

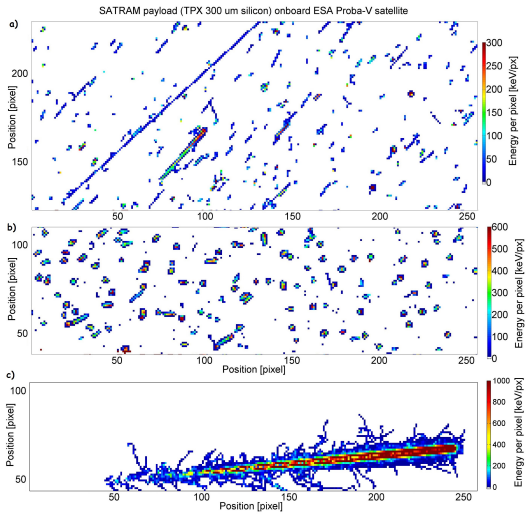
1.2 Timepix3 detektor

Hybridní částicový pixelový detektor Timepix3 [24] je nástupcem detektoru Timepix [20] a je vyvíjen v rámci Medipix[3] kolaborace v CERN[4], mezi jejíž členy patří od roku 1999 i ÚTEF[5] ČVUT v Praze.

Detektor se skládá z matice 256×256 čtvercových nezávislých pixelů, každý o hraně 55 μm. Jednotlivé pixely se skládají z citlivého polovodičového senzoru (nejčastěji *Si*, nebo *CdTe*) a vyčítací CMOS elektroniky (čítače, komparátory apod.). Princip funkce detektoru lze přirovnat digitálnímu fotoaparátu. Podobně jako u digitálního fotoaparátu, začátek a konec akvizice dat je řízen závěrkou (tzv. *shutter signál*). Po dobu otevření závěrky pak citlivý polovodičový objem detektoru zaznamenává interakce s nabitými částicemi a dále je zpracovává dle nastaveného módu. V kapitole 2 budou popsány jednotlivé módy, včetně *Time-Over-Threshold* módu, kde hodnota čítače pixelu na konci akvizice odpovídá deponované energii interagovaných částic s daným pixelem (mezi energií a TOT je nelineární závislost, která je dána fyzikálními vlastnostmi každého pixelu a je předmětem energetické kalibrace detektoru [12]).



(a) Fotografie Timepix3 detektoru, osazeného na Timepix3 CERN chipboard [15].



(b) Ukázka dat z detektoru Timepix, naměřených zařízením *SATRAM* (z angl. *Space Application of Timepix based Radiation Monitor*), které od roku 2013 obíhá Zemi na heliosynchronní kruhové polární dráze ve výšce 820 km, jako součást *ESA Proba-V* satelitu [23].

Obrázek 1.1: Detektor Timepix3 a vizualizace naměřených dat.

Timepix3 detektor přináší oproti svému předchůdci několik výhod. Je schopný operovat i

[3] <http://medipix.web.cern.ch/>

[4] Evropská organizace pro jaderný výzkum (z francouzského *Conseil Européen pour la Recherche Nucléaire*), se sídlem v Ženevě.

[5] Ústav technické a experimentální fyziky.

v kontinuálním módu, ve kterém je každý pixel detektoru schopný detekovanou událost ihned zpracovat nezávisle na ostatních pixelech. Tím se téměř odstraňuje mrtvá doba detektoru, zvyšuje detekční účinnost, ale i se zvyšuje datový tok z detektoru, jehož maximální teoretická hodnota je až $5,12$ Gb/s.

1.3 Struktura práce

Práce je rozdělená do následujících kapitol:

- **Kapitola** 2 – Úvod do problematiky hybridních částicových pixelových detektorů: V této kapitole bude čtenář seznámen se základními fyzikálními principy detekce ionizujícího záření v polovodiči, jsou zde představeny hybridních částicové pixelové detektory rodiny *Medipix*, včetně jejich módu, aspektů kalibrace a dostupných vyčítacích rozhraní.
- **Kapitola** 3 – Návrh architektury: Tato kapitola se zabývá návrhem softwarové a hardwarové architektury systému Pixnet, jsou zde popsány jeho komponenty a jejich vzájemné interakce. Rovněž zde je diskutována škálovatelnost systému a je zde uveden seznam použitých technologií, potřebných pro implementaci systému.
- **Kapitola** 4 – Handler: V této kapitole bude čtenář seznámen s handlerem – komponentou systému Pixnet pro řízení jemu přiřazených detektorů.
- **Kapitola** 5 – Vyčítací rozhraní Katherine a jeho implementace: Tato kapitola je věnována vyčítacímu rozhraní *Katherine* (zařízení pro řízení a vyčítání dat z detektoru *Timepix3* s embedovaným počítačem, komunikující po ethernetu). Je zde popsán návrh a implementace modulů, umožňující jeho integraci do systému Pixnet. Také je představen emulátor, který emuluje *Katherine* na síťové vrstvě.
- **Kapitola** 6 – Master: V této kapitole bude představen master – centrální element systému Pixnet, který řídí handlery, přiřazuje jim detektory a umožňuje centralizované řízení detektorů.
- **Kapitola** 7 – Testování: V této kapitole bude čtenář seznámen s metodami testování pro zajištění kvality jednotlivých softwarových komponent a s provedenými experimenty.
- **Kapitola** 8 – Závěr: V závěru práce bude čtenář seznámen z dosaženými výsledky a plány na budoucí vývoj systému.

Kapitola 2

Úvod do problematiky hybridních částicových pixelových detektorů

Ionizující záření je lidskými smysly nedetekovatelné, avšak jeho studium nám pomáhá pochopit podstatu hmoty, její vlastnosti a vzájemné interakce. To lidstvu umožnilo rozvinout mnohé aplikace, jako je například protonová terapie [22], defektoskopie nebo zkoumání pravosti uměleckých děl [14]. První pokusy o detekci ionizujícího záření sahají do 19. století. Například na počátku 20. století se pomocí mlžné komory prvně podařilo zachytit trajektorii nabitých částic. Rozvoj polovodičové technologie dal vzniknout novým detekčním technologiím, jako například částicovým pixelovým detektorům.

Existuje celá řada částicových pixelových detektorů, ale v této kapitole budou popsány pouze hybridní pixelové detektory, pro které je typické, že se skládají ze dvou nezávisle vyrobených částí – senzoru a vyčítacího čipu. To oproti monolitickým detektorům, kde vyčítací elektronika je součástí senzoru, přináší řadu výhod, jako například snížení výrobních nákladů nebo možnost kombinace vyčítacího čipu se senzory různých materiálů (*Si*, *GaAs*, *CdTe* apod.) a tlouštěk (vetšinou $300\mu m$, nebo $500\mu m$).

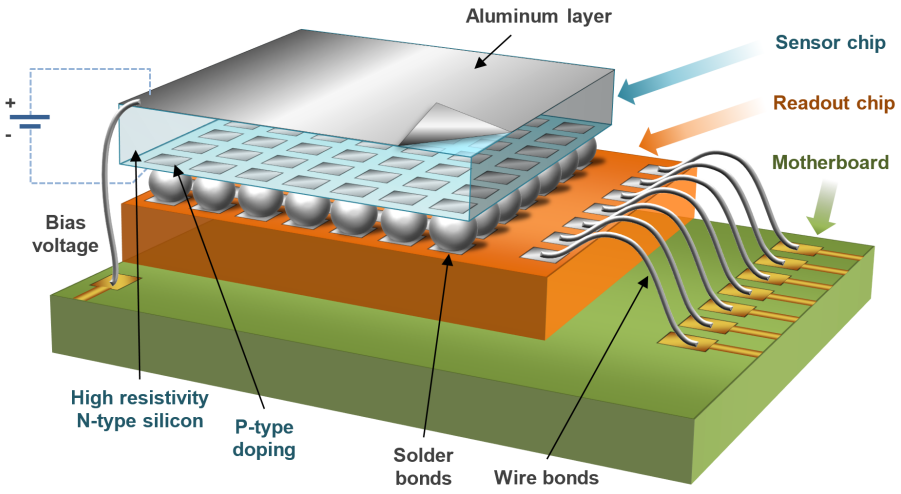
Na tomto místě je třeba zmínit, že existuje více druhů těchto detektorů (*AGH Fermilab, Pilatus, Philips Chromaix* apod.)[2], v této práci budou použity pouze detektory z rodiny detektorů Medipix.

2.1 Hardwarová architektura detektorů rodiny Medipix

Většina hybridních částicových pixelových detektorů rodiny Medipix obsahuje matici čtvercových 256×256 pixelů. Každý z nich má hranu o délce 55 μm, takže detektor obsahující 65536 pixelů má aktivní plochu $1,96\ cm^2$.

Na obrázku 2.1 je znázorněna struktura detektoru Timepix3. Vrchní část detektoru tvoří polovodičový senzor, který je nejčastěji vyroben z křemíku, ale výjimkou není ani *GaAs* nebo *CdTe*. Jednotlivé pixely senzoru jsou spojeny s integrovaným `ASIC`[1] vyčítacím čipem pomocí technologie zvané *Bump-Bonding*. Vyčítací čip je pak propojen se základní deskou pomocí *wire-bond*, z které je ještě přivedeno měřící napětí na senzor detektoru (tzv. *bias*).

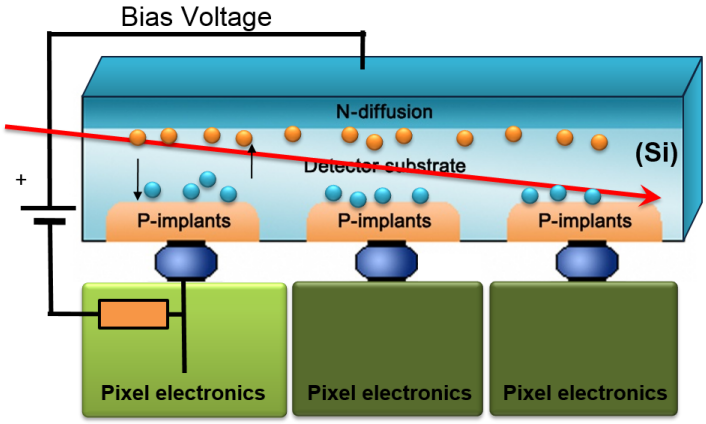
[1] z angl. Application Specific Integrated Circuit.



Obrázek 2.1: Struktura hybridního polovodičového pixelového detektoru Timepix3, skládajícího se z vyčítacího čipu (*Readout chip*) a polovodičového senzoru (*Sensor chip*) [23].

2.2 Princip detekce

Princip detekce ionizujícího záření v polovodiči je znázorněn na obrázku 2.2.



Obrázek 2.2: Princip detekce ionizujícího záření detektorem Timepix3, kde červená šipka znázorňuje interagující částici, elektrony jsou znázorněny žlutě, díry modře [23].

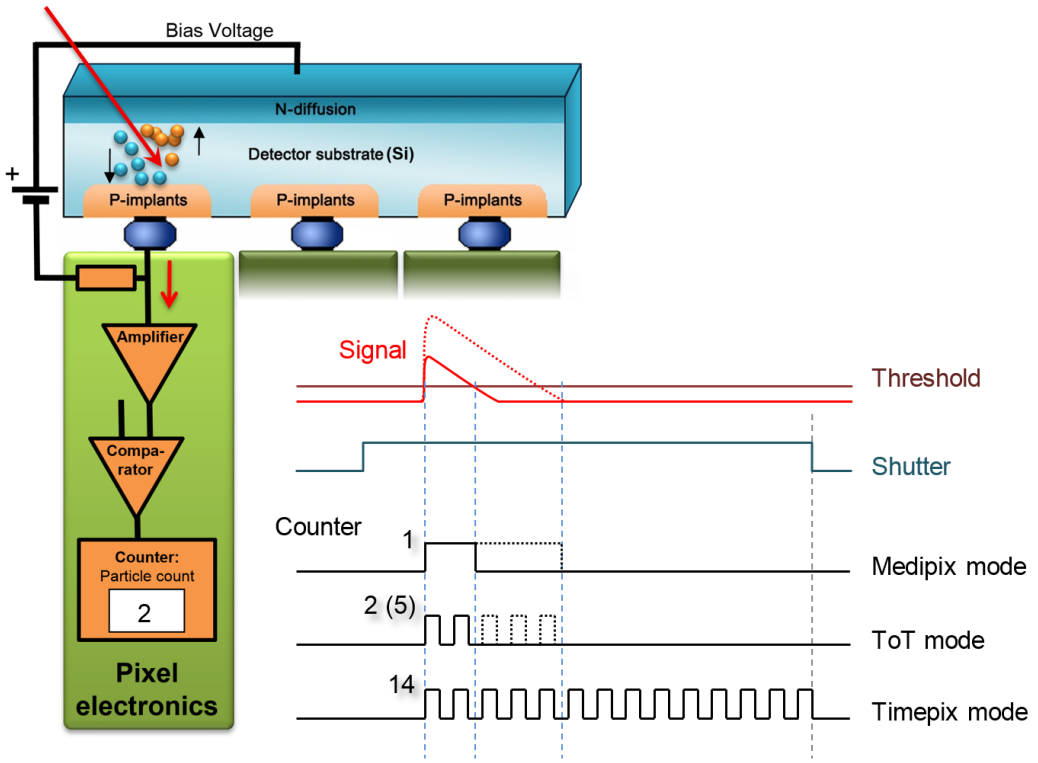
Jako náhradní schéma jednoho pixelu si lze představit diodu zapojenou v závěrném směru, kterou bez přítomnosti ionizujícího záření protéká minimální proud. Vnikne-li do senzoru částice ionizující záření a dojde-li k její interakci se senzorem, resp. část její energie je deponována do polovodičového objemu senzoru, dojde v senzoru ke vzniku elektron-děrových párů a díky lavinovému efektu i k následnému otevření PN přechodu (viz obr. 2.2).

Vzniklý proudový impulz je měřícím odporem převeden na napětí, které je komparátorem porovnáno s prahovým napětím (tzv. *threshold*). Výsledek je dále CMOS[2] obvodem zpracován dle použitého měřícího módu, jak bude ukázáno v kapitole 2.3.

Na rozdíl od CCD[3] technologie, CMOS readout *Timepix*/*Medipix* detektorů negeneruje temný proud[4], díky odstínění signálu od šumu pomocí komparačního napětí. To znamená, že doba jedné akvizice je teoreticky neomezena, protože detektor je schopný detekovat jen ty částice, jejichž deponovaná energie (resp. amplituda vzniklého napěťového pulzu) je větší, než *threshold*.

2.3 Operační módy detektoru

V této podkapitole bude vysvětlena většina operačních módů, ve kterých detektory rodiny *Medipix* jsou schopny pracovat.



Obrázek 2.3: Zpracování signálu pixelem detektoru dle nastaveného módu (*Medipix*, *TOT* a *ToA*) [23].

[2] Z angl. *Complementary Metal–Oxide–Semiconductor*, výrobní technologie logických integrovaných obvodů.

[3] Z angl. *Charge-coupled device*.

[4] Termín charakterizující vyčítací šum u CCD snímačů. Obvykle je udáván v elektronech za sekundu při konstantní teplotě a ve tmě.

Interagovaná částice vyvolá napěťový impulz, jehož tvar je dán deponovanou energií. Pro účely analýzy se ale používá pouze binární informace o překročení prahového napětí v čase. Výsledek této analýzy je po jejím dokončení uložen ve 14-bitovém registru pixelu.

Na obr. 2.3 je znázorněn příklad zpracování analýzy signálu následujícími módy:

Medipix mód (Counting mód) V tomto módu je čítač inkrementován v každém cyklu měřící frekvence, pokud měřící napětí překročilo prahové napětí pixelu. Na konci akvizice pak hodnota čítače odpovídá počtu zaznamenaných částic.

Time-Over-Threshold (TOT) Pracuje-li pixel v tomto módu, pak je jeho čítač inkrementovaný v každém cyklu měřící frekvence, pokud měřící napětí je vyšší, než prahové napětí pixelu. Hodnota uložená v čítači odpovídá deponované energii interagovaných částic. Mezi energií a `TOT` je nelineární závislost a její zkoumání je předmětem energetické kalibrace detektoru, jak bude ukázáno v kapitole 2.5.2. Tento mód má široké spektrum aplikací, například [28] nebo [22].

Time-of-Arrival (ToA) Tímto módem disponují pouze detektory *Timepix* a *Timepix3*, avšak nesdílí stejný princip. Zatímco *Timepix* detektor začne inkrementovat čítač v každém cyklu měřící frekvence po první náběžné hraně z komparátoru, *Timepix3* na náběžnou hranu uloží do 14-bitového registru aktuální časové razítko z hodin detektoru. V obou případech `ToA` udává čas první interakce částice v dané akvizici. Na obr. 2.3 jako *Timepix mode*.

2.4 Vyčítání naměřených dat

Jednotlivé detektory rodiny *Medipix* mají různou hardwarovou podporu pro vyčítání naměřených dat. Detektory vždy podporují alespoň jeden z těchto módů:

Frame-Based Pracuje-li detektor v tomto módu, pak jsou všechny registry čítačů pixelů vyčítány najednou, po dokončení aktuálního snímku. Vždy je třeba vyčíst všechny pixely bez ohledu na naměřenou hodnotu.

Data-Driven Tento mód, také označovaný jako *Event-Driven* nebo též kontinuální, byl prvně použit v detektoru *Timepix3*. Pracuje-li detektor v tomto módu, pak v průběhu akvizice dat (resp. když *shutter* signál je nastaven na úroveň `HIGH`) každý pixel po zpracování události notifikuje readout interface o tom, že nová data jsou připravena k vyčtení a readout interface je pak bez prodlení vyčte a dále zpracuje.

Na obrázku 2.4 je vidět hlavní motivace pro zavedení podpory *Data-Driven* módu u detektoru *Timepix3*. Ukázalo se, že *Data-Driven* mód je efektivnější při takových měření, kde okupance snímků je menší než zhruba 50 %. Po překročení této meze je efektivnější použití *Frame-Based* módů, protože není třeba přenášet souřadnice zasažených pixelů. Podle [24] může být vyčítací čas definován následovně:

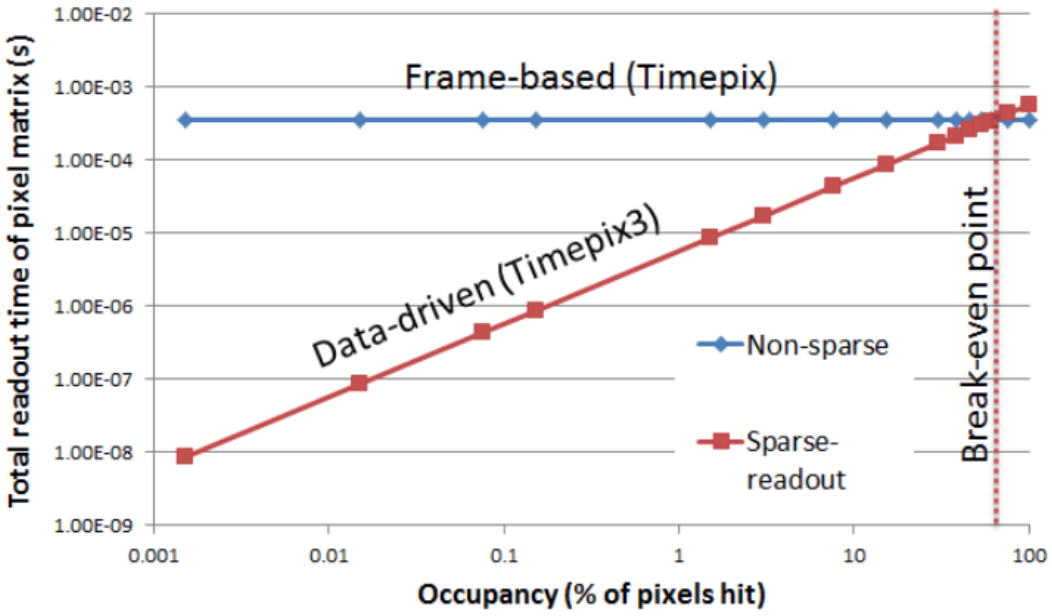
$$T_{readout} = N_{pixels} * bits_{pixel}/BW, \tag{2.1}$$

kde:

N_{pixels} je počet pixelů které je potřeba vyčíst (pro *Frame-Based* mód jsou to všechny pixely detektoru (256×256) a pro *Data-Driven* je to počet zasažených pixelů),

$bits_{pixel}$ je počet bitů na pixel ($28b$ v *Frame-Based* módu a $28b + 16b$ v *Data-Driven* módu kvůli nutnosti přenášení adresy pixelu) a

BW je počet bytů za vteřinu, které je možné vyčíst z detektoru (*bandwidth*).



Obrázek 2.4: Doba vyčítání detektoru za použití *Frame-based* (non-sparse) a *Data-Driven* (sparse) módu [24].

2.5 Kalibrace

Každý detektor má své specifické vlastnosti, které jsou dány nejenom výrobním procesem, ale i závislostí na opotřebení a únavě materiálu v čase, okolní teplotě nebo na nastavených měřících parametrech (například *bias*). Hlavní motivací pro kalibraci detektorů je minimalizace systematické chyby měření. Z pohledu aplikace získaných kalibračních dat je možné kalibrační metody rozdělit do dvou kategorií:

(i) Použití v průběhu akvizice dat – jedná se o parametry, které jsou použity pro nastavení akvizice dat v detektoru a mají přímý vliv na naměřené hodnoty, které danou metodou není možné dodatečně kalibrovat. Do této kategorie spadá například *threshold equalizace* (viz 2.5.1).

(ii) Transformace naměřených dat – v tomto případě jsou kalibrační data aplikovaná dodatečně na naměřená data. Tento přístup má výhodu v možnosti dodatečné kalibrace

již naměřených dat. Do této kategorie spadá například *Energetická kalibrace* (viz 2.5.2) nebo *Time-Walk korekce* (viz 2.5.3).

2.5.1 Threshold equalizace

V podkapitole 2.2 a 2.3 již bylo vysvětleno použití prahového napětí (*threshold*) v průběhu akvizice dat detektorem. Každý pixel detektoru má ale rozdílné fyzikální vlastnosti dané výrobním procesem, s čímž souvisí i citlivost (resp. oddělení užitečného signálu od šumu) jednotlivých pixelů. Kromě globální hodnoty thresholdu je možné pro každý pixel upravit citlivost pomocí lokální 4*b* hodnoty thresholdu (viz obr. 2.8).

Vlastní proces equalizace probíhá tak, že se pro všechny pixely detektoru provede měření se všemi kombinacemi thresholdu, přičemž je třeba minimalizovat interakce detektoru s částicemi. V běžné praxi stačí detektor dostatečně odstínit a provést několik sad měření. Výstupem tohoto procesu je pak globální threshold a jeho 4-bitové korekce pro jednotlivé pixely.

Během tohoto procesu vznikne rovněž maskovací matice detektoru, která obsahuje šumějící, nebo jinak poškozené pixely. To jsou například takové pixely, které bez přítomnosti interagujících částic hlásí překročení thresholdu.

2.5.2 Energetická kalibrace

V kapitole 2.3 byl již popsán *Time-Over-Threshold* mód, ve kterém je detektor schopný měřit deponovanou energii interagovaných částic, která je udávána v `TOT`. Jak již bylo ukázáno, vztah mezi energií v keV a `TOT` je nelineární a závisí na fyzikálních vlastnostech daného pixelu, což je předmětem energetické kalibrace, která bude v této podkapitole popsána.

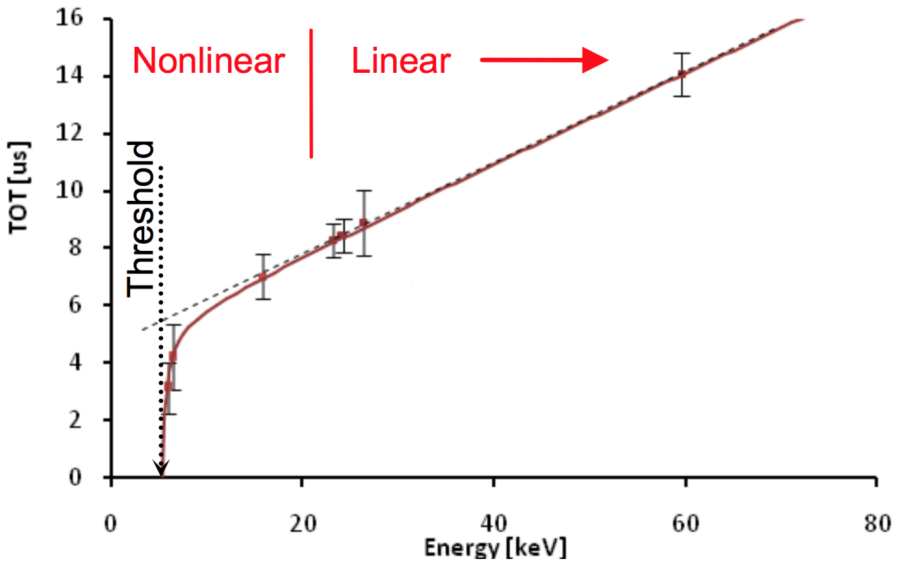
Tato metoda [12] spočívá v provedení několika sad měření se zdroji ionizujících záření, jejichž energie jsou předem známy, a v jejich analytickém zpracování a vytvoření kalibrační funkce 2.2 pro každý pixel detektoru. V předchozí práci [3] byly tyto metody podrobně popsány a byl vytvořen software, který uživateli umožňuje vytvoření energetické kalibrace detektoru z naměřených dat.

$$f_{calib}(x) = ax + b - \frac{c}{x - t} \tag{2.2}$$

Pro získání kalibračních dat se jako efektivní řešení v praxi ukázalo měření rentgenových fluorescencí[5] [11]. Pro zajištění dobré kvality kalibrace je třeba naměřit takový počet událostí, aby ze snímků získaná spektra byla dobře rozeznatelná. Z naměřených dat jsou vyfiltrovány pouze tzv. *Single-hit* události[6], aby se minimalizovaly negativní vlastnosti efektu sdílení náboje (*Charge-sharing*). Tento efekt je dán společnou elektrodou senzoru a může způsobit, že vzniklé elektrony jsou zpracovány více pixely najednou a část deponované energie nemusí

[5] Děj, ke kterému dochází při ozařování materiálu (nejčastěji se pro kalibraci využívá *Cu*, *Fe*, In apod.) rentgenovým zářením, při kterém jsou z něj vyráženy excitované elektrony. Při vyražení elektronu na nižší energetické úrovni, elektron z vyšší energetické úrovně obsadí jeho místo a přebytečnou energii emituje formou vyzářeného fotonu – fluorescenčního záření, jehož charakteristické monoenergetické spektrum je pro většinu prvků dobře známé.

[6] Události, ve kterých částice interagovala pouze s jedním pixelem detektoru



Obrázek 2.5: Kalibrační funkce, udávající závislost mezi energií v keV a TOT [12], vzniklá proložením získaných kalibračních bodů funkcí 2.2 a sestávající se ze dvou částí – (i) nelineární části pro oblast nižších energií (hyperbola) a (ii) lineární části pro vyšší energie (přímka).

být ASIC čipem zpracována, protože vzniklý signál může být nižší než threshold daného pixelu.

Z jednotlivých měření jsou pro každý pixel detektoru vytvořena spektra TOT hodnot. Na obrázku 2.6 je znázorněn příklad takového spektra, získaného z fluorescence mědi. Požadovaný kalibrační bod se získá ze střední hodnoty TOT a tabulkové hodnoty energie fluorescenčního záření mědi. Střední hodnota je získána proložením spektra funkcí 2.3 – jedná se o součet Gaussovy funkce a Gaussovy chybové funkce (kvůli levé asymetrii vzniklé *Charge-sharing* efektem)

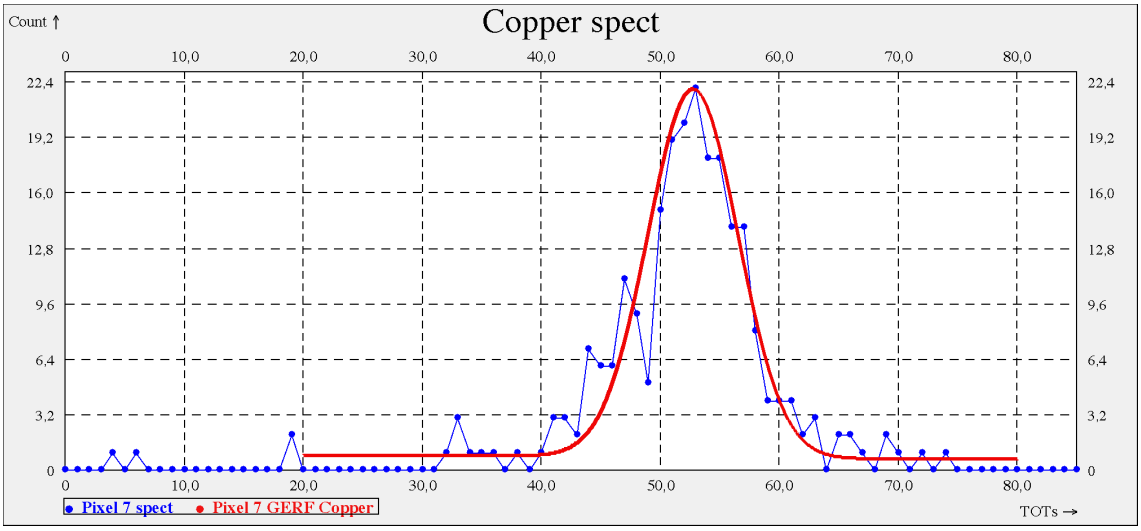
$$f_{GERF}(x) = \underbrace{Ae^{-\frac{(x-\mu)^2}{2\sigma^2}}}_{\text{Gaussova funkce}} + \underbrace{\frac{avg_{right} - avg_{left}}{\sigma\sqrt{2\pi}} \int_{-\infty}^{t} e^{-\frac{(t-\mu)^2}{2\sigma^2}} + avg_{left}}_{\text{Gaussova chybová funkce}}, \tag{2.3}$$

kde:

- A je amplituda,
- μ je střední hodnota hledané energie,
- σ je rozptyl střední hodnoty energie μ, který je možné vypočítat ze vzorce 2.4, kde FWHM[7] udává šířku gausiánu v polovině jeho výšky a
- avg_{right}, avg_{left} je průměrná hodnota spektra na pravém (resp. levém) úpatí gausiánu.

$$\sigma = \frac{2\sqrt{2ln_2}}{FWHM} \tag{2.4}$$

[7] z angl. Full Width at Half Maximum



Obrázek 2.6: Příklad energetického spektra jednoho pixelu Timepix detektoru s proloženou funkcí 2.3 [3].

2.5.3 Time-Walk korekce

Time-Walk efekt je nežádoucí jev, který vzniká při interakci ionizujícího záření o různé energii. Velikost deponované energie má vliv na amplitudu a sklon náběžné hrany napěťového pulzu na výstupu zesilovače pixelu. Při interakci ionizující částice s více pixely je pak díky tomuto jevu interagovanými pixely zaznamenána jiná hodnota ToA, i přes to že událost byla způsobena stejnou částicí a hodnoty ToA by měly být stejné. Viz obrázek 2.7, kde jsou pro interakci stejné částice čtyřmi sousedními pixely zaznamenány různé hodnoty ToA.

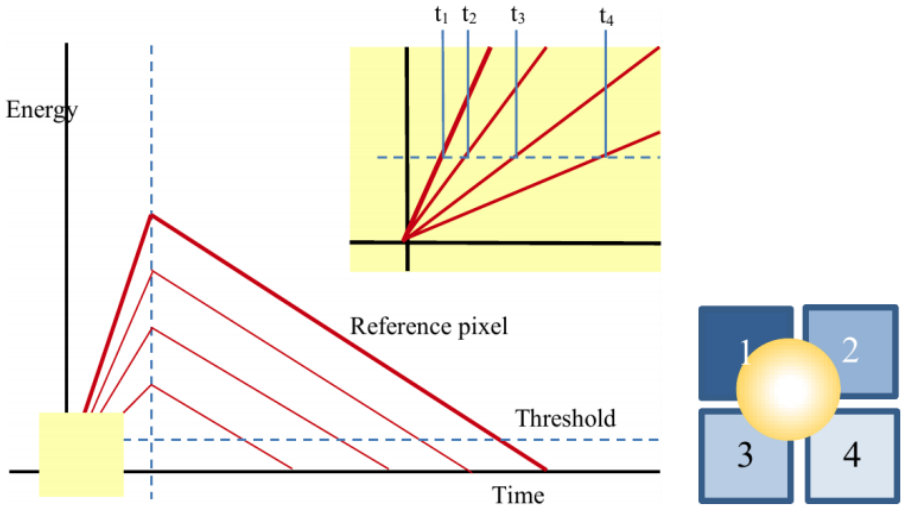
S použitím detektoru *Timepix3* je možné tuto energeticky závislou chybu eliminovat za použití kalibrační metody [30], protože tento detektor umožňuje měřit v ToA a TOT módu současně. Tato kalibrační metoda spočívá v analytickém zpracování dat získaných z měření se zdrojem α částic (v [30] použito $^{241}\mathtt{Am}$), které generuje clustery o více pixelech. Nejprve je však třeba jej energeticky zkalibrovat 2.5.2.

O vybraných clusterech o velikosti 3 a 4 pixely pro $^{241}\mathtt{Am}$ víme, že součet jejich energií je $59.5keV$. Z clusteru je vybrán pixel s energií $30keV$, který je použit jako referenční (na 2.5.3 jako t_1), zbylá energie je náhodně rozdělena mezi ostatní pixely (na 2.5.3 jako t_{2-4}). Jednotlivé rozdíly $t_i + t_1$ jsou analytickými metodami, popsanými v [30], zpracovány a výsledkem tohoto procesu jsou konstanty c, d pro každý pixel detektoru, kterými lze vypočítat jeho *Time-Walk offset*

$$\Delta T = \frac{c}{(E - E_0)^d} \ , \tag{2.5}$$

kde:

ΔT je *Time-Walk* korekce pixelu $[ns]$ (výsledná hodnota ToA je pak rovna $ToA - \Delta T$),



Obrázek 2.7: *Time-walk* efekt: příklad interakce jedné částice se čtyřmi pixely detektoru, kde v každém pixelu byla deponována jiná energie, což na výstupů zesilovačů pixelů způsobilo jiné hodnoty napětí. Kvůli rozdílné charakteristice náběžné hrany pulzů byl threshold překročen v různých časech (t_{1-4}) [30].

E je energie pixelu [keV],
E_0 je threshold pixelu [keV] a
c, d jsou konstanty.

2.5.4 ToA offset korekce

ToA offset se o jev, ke kterému dochází u *Timepix3* detektorů z důvodu chyby hardware, který způsobuje špatnou synchronizaci časového razítka napříč sloupci detektoru [5]. Tato chyba vzniká jen při zapnutí detektoru a poté je ToA offset již stabilní, takže je možné vytvořit korekci.

Odstranění tohoto jevu spočívá ve vytvoření kompenzační tabulky pomocí středních hodnot interních testovacích pulzů [24]. Pro získání správných hodnot ToA z naměřených dat stačí jen odečíst příslušnou hodnotu z korekční tabulky.

2.6 Přehled detektorů rodiny Medipix

V této podkapitole budou stručně představeny jednotlivé detektory, které byly vyvinuty v rámci Medipix kolaborace [13].

Medipix1 V roce 1998 byl vyvinut detektor *Medipix1* [26], také zvaný *Photon-Counting-Chip*, který byl prvním velkoplošným detektorem používajícím CMOS technologii. S maticí 64×64 pixelů, každým o hraně 170 μm, má celkovou aktivní plochu $1,1\ cm^2$. I

přes malý počet pixelů a jejich velkou rozteč, tento detektor prokázal dobré rozlišovací vlastnosti, především pak v rentgenovém zobrazování.

Detektor je vyrobený pomocí 1 μm technologie a je schopný operovat pouze v Medipix módu (viz 2.3) a podporuje pouze vyčítání po snímcích (viz 2.4).

Medipix2 V roce 2001 byl vyvinut detektor *Medipix2* [19], jako náhrada za svého předchůdce *Medipix1*. Díky 250 nm technologii bylo možné zvýšit počet pixelů a zároveň snížit jejich rozteč – detektor má 256×256 pixelů, každý o hraně 55 μm. Navíc má dva thresholdy s diskriminací na 4 pixely.

Stejně jako jeho předchůdce je schopný operovat pouze v Medipix módu (viz 2.3) a podporuje pouze vyčítání po snímcích (viz 2.4).

Timepix V roce 2006 byl vyvinut detektor *Timepix* [20], který vychází z detektoru *Medipix2*. Detektory mají stejné rozměry, ale architektura pixelů se změnila. Nově každý pixel detektoru umožňuje nezávisle měřit čas interakce částice, její energii, nebo je schopný počítat jednotlivé interakce.

Detektor je schopný operovat v `TOT`, `ToA`, nebo v Medipix módu (viz 2.3) a podporuje pouze vyčítání po snímcích (viz 2.4).

Medipix3 V roce 2011 byl vyvinut detektor *Medipix3* [1], jako nástupce *Medipix2* detektoru. Rozměry detektoru zůstaly zachovány (256×256 pixelů, každý o hraně 55 μm), ale díky 130 nm technologii byla funkce jednotlivých pixelů vylepšena. Detektor nově umožňuje zvýšení energetického rozlišení díky potlačení *Charge-Sharing* efektu (viz 2.5.2) pomocí integrace náboje do clusteru 4×4 pixelů.

Každý pixel má dva 14-bitové čítače. Jednotlivé pixely můžou být naprogramovány tak, aby vždy měřily za pomoci jednoho čítače, zatímco je hodnota z druhého čítače vyčítána, což umožňuje měření bez mrtvé doby.

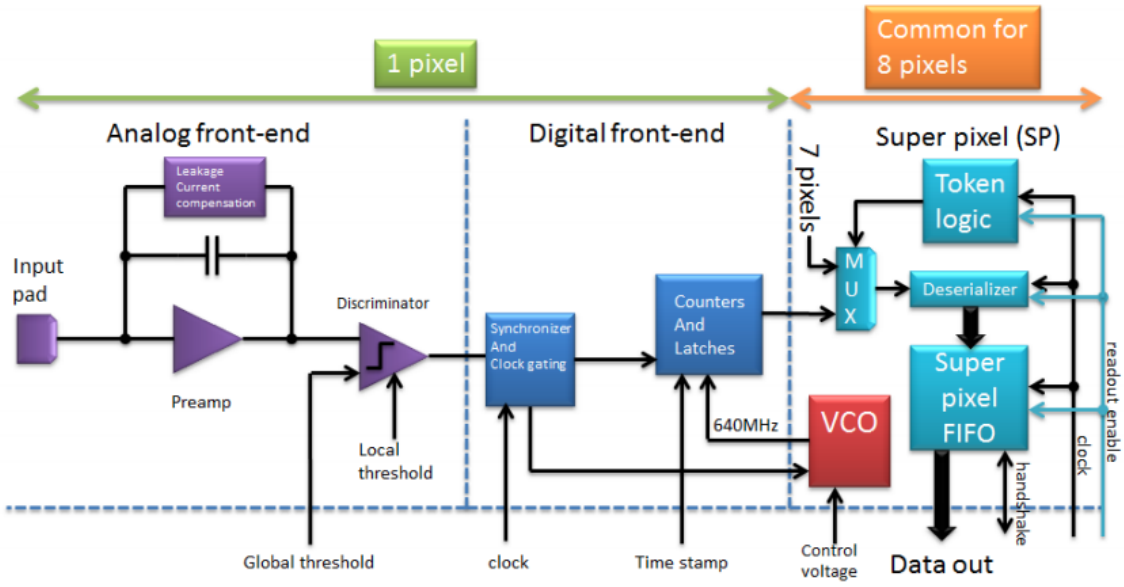
Také je možné zapojit tento readout čip (s pixelem o hraně 55 μm) na senzor s pixely o hraně 110 μm, takže výsledný super-pixel má k dispozici 8 čítačů a pomocí různých úrovní thresholdů jej lze použít jako 8-kanálový spektrometr.

Stejně jako jeho předchůdce je schopný operovat pouze v Medipix módu (viz 2.3) a podporuje pouze vyčítání po snímcích (viz 2.4).

Timepix3 V roce 2014 byl vyvinut detektor *Timepix3* [24], jako nástupce *Timepix* detektoru. Rozměry byly zachovány, ale byla použita 130 nm výrobní technologie, jako u *Medipix3*. Detektor nově umožňuje měřit v `ToA` a `TOT` současně a jeho časové rozlišení bylo vylepšeno více než na šestinásobek.

Detektor nově umožňuje kromě vyčítání po snímcích i *Data-Driven* mód, kde jsou data v rámci akvizice vyčítána z detektoru kontinuálně (viz 2.4). Takto navržená architektura umožňuje vyčítat události až do $40 M_{hits} * cm^{-2} * s^{-1}$ bez globální mrtvé doby detektoru (pouze s lokální mrtvou dobou – pro jednotlivé pixely, ze kterých jsou vyčítána naměřená data).

Na obrázku 2.8 je znázorněno blokové schéma jednoho pixelu *Timepix3* detektoru, který se skládá z analogové a digitální části (pro popis funkcionality viz 2.2). Na



Obrázek 2.8: Schéma pixelu detektoru *Timepix3* se společnou elektronikou pro 8 pixelů (tzv.*Super-pixel*) [24].

obrázku vpravo je pak společná elektronika, sdílená vždy 8 sousedními pixely (tzv. *Super-pixel*).

2.7 Vyčítací rozhraní

V této podkapitole bude popsáno několik vyčítacích rozhraní, které byly vyvinuty (v rámci *Medipix kolaborace*[8]) ÚTEF ČVUT v Praze a jeho spin-off společností ADVACAM s.r.o. Pro komunikaci s *Timepix3* detektorem bude v rámci této práce použito nově vyvinuté vyčítací rozhraní **Katherine** [5] 2.7.4.

2.7.1 FITPix

FITPix [17] vyčítací rozhraní bylo vyvinuto v roce 2010 a skládá ze z programovatelného hradlového pole (FPGA)[9], USB 2.0 rozhraní, DAC[10] a ADC[11] převodníků a obvodů pro generování měřícího napětí (*bias*).

Zařízení je s detektorem propojeno pomocí *LVDS*[12] a je schopné jeho plnohodnotného řízení, vč. nastavování hodnot (měřící frekvence, threshold, bias apod.), řízení akvizice a vyčítání naměřených dat rychlostí až 90 snímků za sekundu. Zařízení rovněž umožňuje připojení externího trigger signálů pro synchronizované měření s více detektory současně.

[8] <https://medipix.web.cern.ch>
[9] Z angl. Field Programmable Gate Array
[10] Převodník digitálního signálu na analogový.
[11] Převodník analogového signálu na digitální.
[12] Z angl. Low-voltage differential signaling

FITPix je možné použít pouze jako nízkoúrovňové vyčítací rozhraní a veškerá business logika musí být implementována v připojeném počítači (např. deserializace a derandomizace naměřených dat apod.). Jako řídící software je použit *Pixelman*[29]. Zařízení je kompatibilní se všemi detektory uvedenými v 2.6, kromě *Medipix1*.

2.7.2 AdvaDAQ

AdvaDAQ [30] je nástupcem vyčítacího rozhraní FITPix a kompatibilní se všemi detektory uvedenými v 2.6 (kromě *Medipix1*). Principem funkce vychází ze svého předchůdce, ale díky použitému rozhraní USB 3.0 je schopné dosahovat maximálního datového toku $2,9\ Gb/s$. Deserializace naměřených dat byla implementována do firmware *FPGA*. Pro řízení tohoto rozhraní, vizualizaci a analýzu naměřených dat byl vyvinut software Pixet.

2.7.3 ATLAS Pix

ATLAS Pix [21, 3] je zařízení vyvinuté v rámci experimentu *ATLAS-TPX*, síti hybridních částicových detektorů *Timepix*, instalované v rámci experimentu *ATLAS* na *LHC* v *CERN*, operující od roku 2014.

Zařízení vzniklo modifikací vyčítacího rozhraní *FITPix* a disponuje `FPGA`, `LVDS` zesilovačem a počítačem *Raspberry Pi*, ve kterém je implementován software, poskytující API pro své vzdálené řízení přes `TCP` protokol. Komunikace mezi `FPGA` a počítačem je realizována pomocí `SPI`[13] protokolu.

2.7.4 Katherine

Katherine [5] je pokročilé vyčítací rozhraní dedikované pro řízení jednoho *Timepix3* detektoru, které v sobě obsahuje embedovaný počítač, čímž se vymezuje od ostatních vyčítacích rozhraní bez procesoru, kde business logika musí být implementována v připojeném počítači, což má negativní dopad na vytížení jeho procesoru.

Hlavním benefitem zařízení je možnost jeho použití při experimentech, kde je nutná větší vzdálenost mezi detektorem a vyčítacím rozhraním (u *Katherine až* 100 m), především kvůli nedostatečné radiační a elektromagnetické odolnosti použité elektroniky. To umožňuje aplikace v blízkosti jaderných reaktorů nebo na částicových urychlovačích (například měření luminosity v rámci experimentu ATLAS na LHC v CERN [27]).

Zařízení je kompatibilní s *Timepix3* detektory, které jsou osazeny na CERN chipboardu, nebo na kompatibilní `PCB`[14] s `VHDCI`[15] 68-pinovým konektorem. S detektorem může být propojeno napřímo (viz obr. 2.9 vlevo), prodlužovacím `VHDCI` kabelem o maximální délce až 10 m, nebo speciálním extenderem na bázi ethernetu pro vzdálenosti až 120 m. S rostoucí vzdáleností ale klesá maximální datový tok (viz tabulka 2.1).

Jeho maximální výstupní datový tok je daný použitým gigabitovým ethernetem, které odpovídá ekvivalentu počtu událostí $16 M_{hits} * cm^{-2} * s^{-1}$ v *Data-Driven* módu (viz 2.4). *Timepix3* detektor je schopný detekovat až $80 M_{hits} * cm^{-2} * s^{-1}$, takže při vyšší zátěži

[13] Z angl. Serial Peripheral Interface.
[14] Z angl. Printed Circuit Board.
[15] Z angl. Very-High-Density-Cable-Interconnect.

Obrázek 2.9: Vyčítací rozhraní *Katherine* s přípojeným detektorem *Timepix3* vlevo a popisem vstupních a výstupních portů vpravo[5].

Vzdálenost	**Typ propojení**	**Počet kanálů – datový tok**	**Hit Rate**
3 m	VHDCI	2×640 Mb/s	16 M_{hits}/s
10 m	VHDCI	4×160 Mb/s	10 M_{hits}/s
20 m	Ethernet	2×640 Mb/s	16 M_{hits}/s
100 m	Ethernet	4×80 Mb/s	5 M_{hits}/s

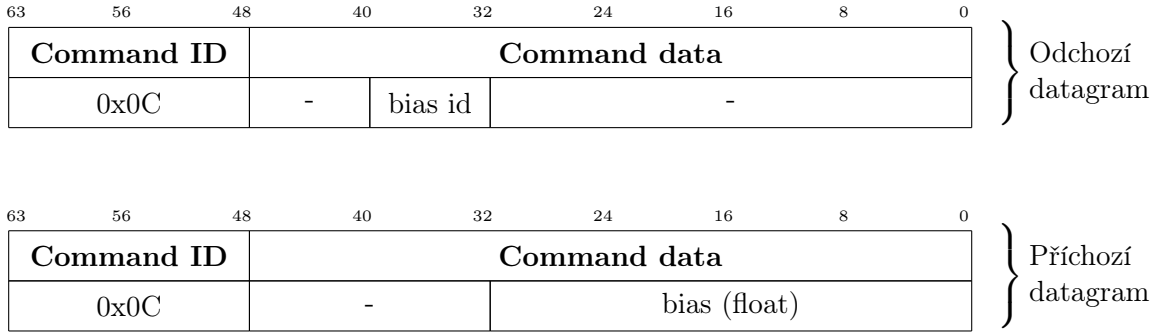
Tabulka 2.1: Katherine: závislost maximálního datového toku na typu propojení mezi detektorem a vyčítacím rozhraním [5].

není zařízení schopné přenést všechny zaznamenané události. Nicméně, *Katherine* disponuje vestavěnou `DDR3` pamětí o velikosti 1 GB, která funguje jako buffer – když je aktuální počet zpracovávaných událostí vyšší, než je maximální datový tok spojení s nadřazeným počítačem, tak se data hromadí v bufferu, aby později při nižší intenzitě zpracovávaných událostí mohla být přenesena.

Pro úplnost výčtu vstupních a výstupných portů je třeba doplnit, že *Katherine* také disponuje `GPIO`[16] konektorem (viz obr. 2.9 vpravo), zahrnujícím 4 obecné signály pro integraci s dalšími zařízeními a další rozšíření, jako například pro trigger synchronizaci (pro možnost zapojení například do detektorového teleskopu [6]). Zařízení dále disponuje `LEMO` konektorem (viz 2.9), poskytujícím zdroj vysokého napětí (± 300 V) pro *bias* (měřící napětí) detektoru.

Vyčítací rozhraní má implementovanou podporu pro `ToA` korekci (viz 2.5.4), které je automaticky vykonána v rámci startovací sekvence, kde jsou poškozené sloupce identifikovány pomocí středních hodnot testovacích pulzů a je sestavena kompenzační tabulka. V průběhu akvizice dat jsou `ToA` hodnoty automaticky opraveny odečtením příslušných hodnot z výše zmíněné tabulky.

[16] Z angl. General Purpose Input/Output.



Obrázek 2.10: Katherine: příklad requestu a response řídícího datagramu pro vyčtení biasu.

2.7.4.1 Komunikace s nadřazeným PC a řízení akvizice dat

Vyčítací rozhraní *Katherine* je dle dané konfigurace schopné operovat ve dvou módech:

SFTP klient (autonomní mód) V tomto módu zařízení operuje z pohledu řízení a akvizice dat zcela nezávisle. Po spuštění zařízení, resp. dokončení startovací sekvence, jsou z dodané konfigurace automaticky nastaveny měřící a akviziční parametry detektoru a je spuštěna akvizice dat. Získaná data jsou pak nepřetržitě posílána pomocí *SFTP*[17] na server, kde jsou ukládána do definovaného adresáře v `ASCII` souborech. Výhoda tohoto módu spočívá v jednoduchosti obsluhy a hodí se především pro takové aplikace, kde není očekáván vysoký datový tok a kde není potřeba měnit nastavení detektoru, protože každá jeho změna vyžaduje restart zařízení.

Manuální mód Pracuje-li zařízení v tomto módu, pak k jeho funkci je třeba řízení z nadřazeného počítače pomocí proprietárního `UDP`[18] protokolu. Protokol využívá dvou `UDP` portů – **řídícího** a **datového**.

Řídící port je vyhrazen pro přenos řídících a konfiguračních paketů. Komunikace je implementována pomocí 36 příkazů, kde každý z nich se skládá z 64-bitového datagramu pro request a 64-bitového datagramu pro response, takže se jedná o synchronní potvrzovanou komunikaci[19]. Pro příklad viz obrázek 2.10, kde je zobrazen request datagramu pro vyčtení biasu a jeho response.

Datový port je určen pro jednosměrnou komunikaci z *Katherine* do nadřazeného počítače a slouží k přenosu naměřených dat. Každý datagram se skládá z 3-bitové hlavičky 43-bitových dat.

V rámci této práce bude použit druhý zmíněný – manuální mód, protože je méně náročný na šířku pásma a umožňuje vzdálené řízení detektoru. V dalších kapitolách (viz 5) bude komunikační rozhraní vyčítacího rozhraní *Katherine* popsáno podrobněji.

[17] Z angl. Secure File Transfer Protocol (protokol pro zabezpečený přenos souborů mezi počítači pomocí `TCP` protokolu).
[18] Z angl. User Datagram Protocol.
[19] Potvrzování je realizováno na aplikační úrovni `ISO/OSI` modelu.

Kapitola 3

Návrh architektury

V této kapitole bude čtenář seznámen s návrhem a koncepcí softwarového systému **Pixnet** – software pro distribuované řízení sítě částicových pixelových detektorů, který byl navržen a implementován v rámci této práce. V této kapitole bude popsána motivace pro vznik tohoto systému a budou představeny jednotlivé komponenty systému a jejich vzájemná komunikace. Pro detailnější popis návrhu a implementace komponent viz kapitoly 4, 5 a 6.

3.1 Motivace

Hlavní motivací pro vznik tohoto systému je fakt, že moderní částicové pixelové detektory jsou schopné generovat velký datový tok, například *Timepix3* má teoretické maximum 5,12 Gb/s (viz 2.6), takže nedistribuovaný systém, který by operoval na jedné instanci, by nebyl schopný zpracovat datový tok, který je síť o více detektorech schopna generovat.

Zde je možné namítnout, že každý systém je možné škálovat vertikálně[1]. Zatímco cenu škálování horizontálně škálovatelného systému zle popsat lineární závislostí výpočetního výkonu na ceně, u vertikálně škálovatelného systému tato závislost roste exponenciálně. Jelikož vertikální škálování takového systému je vysoce neefektivní, nebude dále uvažováno a tato práce se bude věnovat jenom návrhu a implementaci horizontálně škálovatelného řešení.

Další motivací pro vytvoření tohoto systému je možnost řízení heterogenní sítě detektorů homogenním způsobem. Heterogenní sítí detektorů rozumíme takovou síť, ve které jsou detektory různých typů (například *Timepix*, *Timepix3* apod, viz 2.6), komunikující různými komunikačními protokoly prostřednictvím různých vyčítacích rozhraních (například *Katherine*, *ATLAS Pix* apod, viz 2.7). V další části textu bude detailně popsána navržená a implementovaná modulová architektura, která výše zmíněné umožňuje.

Pro potřeby experimentu *ATLAS-TPX* byl již vyvinut software [21, 3] pro řízení sítě detektorů *Timepix* 2.6, prostřednictvím vyčítacího rozhraní *ATLAS Pix* 2.7.3. Software však nevyhovuje požadavkům diskutovaných výše:

[1] Škálováním v kontextu počítačových systémů rozumíme změnu vlastností daného systému za účelem zvýšení, nebo snížení jeho výpočetního výkonu (ev. jiného sledovaného parametru). Zatímco u vertikálního škálování měníme vlastnosti jednoho uzlu systému (například přidáváním procesorů, pamětí, kapacity úložiště apod.), u horizontálního škálování přidáváme jednotlivé uzly – samostatné jednotky (např. počítače). Pro úplnost je třeba doplnit, že vertikální škálování má své omezení z hlediska použitého hardware, u horizontálního škálování žádná taková omezení nejsou.

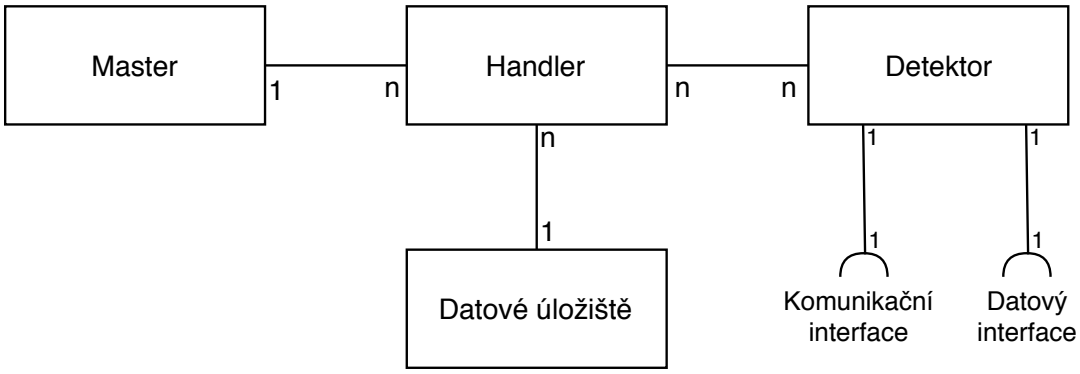
(i) **Škálovatelnost** – systém je navržen bez možnosti horizontálního škálování. Všechny detektory sítě jsou řízeny z jednoho uzlu a všechna vygenerovaná data jsou jím zpracovávána. Možnost použití pouze jednoho uzlu představuje nejslabší článek systému, který nemůže být použit pro řízení a vyčítání dat z větší sítě detektorů.

(ii) **Modularita** – systém implementuje pouze komunikační protokol vyčítacího rozhraní *ATLAS Pix* 2.7.3. Přidání podpory nového vyčítacího rozhraní představuje významnou modifikaci architektury systému a pro nasazení nové verze je nutná odstávka celého systému.

Pro potřeby modernizace sítě *ATLAS-TPX* (za použití detektorů *Timepix3*) bylo rozhodnuto o vývoji software, který bude navržen a implementován tak, aby požadavky na škálovatelnost a modularitu byly zohledněny.

3.2 Softwarová architektura

V této podkapitole budou popsány jednotlivé komponenty softwarové architektury systému Pixnet a bude vysvětlena jejich funkce. Na obrázku 3.1 jsou zobrazeny základní komponenty systému, včetně jejich kardinalit. Detektory sítě jsou řízeny handlery, které komunikují s detektory pomocí dodaného komunikačního protokolu (na obr. 3.2 jako *komunikační interface*) a vyčítají z nich data, která jsou pak ukládána do datového úložiště (pomocí dodané implementace datového interface). Jednotlivé handlery jsou řízeny centrálním uzlem systému – masterem. Master je zodpovědný za konfiguraci systému, držení jeho stavu a přiřazování detektorů masterům.



Obrázek 3.1: Pixnet: softwarová architektura.

V následujících podkapitolách budou detailněji popsány jednotlivé komponenty z obrázku 3.2.

3.2.1 Detektor

Detektor je logická jednotka systému, která je reprezentována implementací komunikačního a datového interface, konfigurací a svým stavem. Fyzické propojení s detektorem je realizováno implementací komunikačního interface.

Komunikační interface obsahuje například metody pro navazování spojení s detektorem, metody pro nahrávání konfigurace a metody pro získání podporovaných příkazů detektoru a jejich vykonání. Každý příkaz má svoje ID, název a model vstupních a výstupních hodnot. Modelem hodnot rozumíme množinu parametrů různých datových typů (`boolean`, `integer`, `float` apod.), které mohou nabývat hodnot omezených zadaným intervalem, nebo jedné z předdefinovaných diskrétních hodnot.

Datový interface obsahuje kromě inicializačních metod (předání konfigurace detektoru apod.) také metodu pro předání reference na asynchronní frontu naměřených dat, do které jsou data vkládána implementací komunikačního interface. Datový interface může mít několik implementací – například data mohou být ukládána do datového úložiště handleru a následně asynchronně nahrána do centrálního datového úložiště, nebo mohou být rovnou synchronně nahrávána do datového úložiště.

3.2.2 Datové úložiště

Klíčový požadavek na datové úložiště je škálovatelnost, jelikož pro sítě o vetším počtu detektorů by ukládání dat do jednoho uzlu znamenalo omezení maximálního datového toku, dané kapacitou daného uzlu.

Jednou z možností implementace může být využití distribuovaného souborového systému, například *Hadoop Distributed File System* [25], který se v praxi[2] používá pro distribuované ukládání dat s možností jejich redundance na více uzlech (pro potřeby zálohování).

Další možností může být použití některé z `NoSQL` distribuovaných databází, například *MongoDB* (viz 3.5).

3.2.3 Handler

Handler je komponenta, kterou je řízena podmnožina detektorů detektorové sítě. K jednomu handleru je tedy možné připojit n detektorů, kde n je omezeno sumou datového toku přes všechny připojené detektory v závislosti na maximálním možném datovém toku handleru.

Handler komunikuje s detektorem pomocí dodané implementace jeho komunikačního interface a data detektorem vygenerovaná jsou pomocí implementace datového interface uložena do datového úložiště, nebo zpracována jiným způsobem.

Handler je zodpovědný za zavedení dodaných implementací komunikačního a datového interface do systému. Například po přiřazení detektoru handleru, handler získá seznam podporovaných příkazů detektoru a poskytne je masteru. To systému umožňuje řízení heterogenní sítě detektorů homogenním způsobem, jak již bylo zmíněno v 3.1.

Aby bylo možné handlery, potažmo detektory, řídit centralizovaně, handler poskytuje `API`[3]. Pomocí `API` jsou jednotlivé handlery připojovány do systému, handlerům jsou přiřazovány a odebírány detektory, inicializuje se konfigurace detektorů a je řízena akvizice dat.

[2] například v roce 2010 internetová společnost *Yahoo!* používala `HDFS` pro persistenci 25 PB podnikových dat [25].

[3] Z angl. *Application programming interface* (aplikační programové rozhraní).

3.2.4 Master

Master je centrální prvek systému, jehož prostřednictvím jsou řízeny handlery. Master poskytuje `API` pro své řízení pomocí pomocí frontendové aplikace. Tato aplikace je webový tenký klient, který uživateli poskytuje uživatelské rozhraní pro řízení systému.

Když uživatel přidává nový detektor do systému, tak nejprve skrze uživatelské rozhraní frontend aplikace zadá parametry detektoru, včetně implementace komunikačního a datového rozhraní, jak je znázorněno na obr. 3.2. Poté je detektor nahrán do mastera pomocí jeho `API`, kde je následně perzistentně uložen do jeho databáze. Při přiřazování detektoru handleru je konfigurace detektoru včetně implementace komunikačního a datového interface nahrána do zvoleného handleru, kde je dále zpracována. Zpracování konfigurace probíhá v následujících krocích:

1. Vytvoření instance komunikačního interface a ověření jeho validity.
2. Vytvoření instance datového interface a ověření jeho validity.
3. Syntaktická analýza (tzv. *parsing*) konfigurace detektoru a její nahrání do instancí komunikačního a datového interface.
4. Vytvoření asynchronní fronty měřených dat a její předání instancím komunikačního a datového interface.

Po dokončení inicializační sekvence je uživatel notifikován a v případě úspěšného dokončení je detektor připraven vykonávat příchozí příkazy a měřit data.

Jako datové úložiště pro konfigurace detektorů, informace o jejich stavu a o stavu jednotlivých handlerů bude navržena relační `SQL`[4] databáze, ke které bude přistupovat pouze backend mastera.

3.3 Hardwarová architektura

V předchozí podkapitole byla popsána softwarová architektura. Fyzická instalace sítě přináší další omezení, především z pohledu síťové infrastruktury, která budou popsána v této podkapitole.

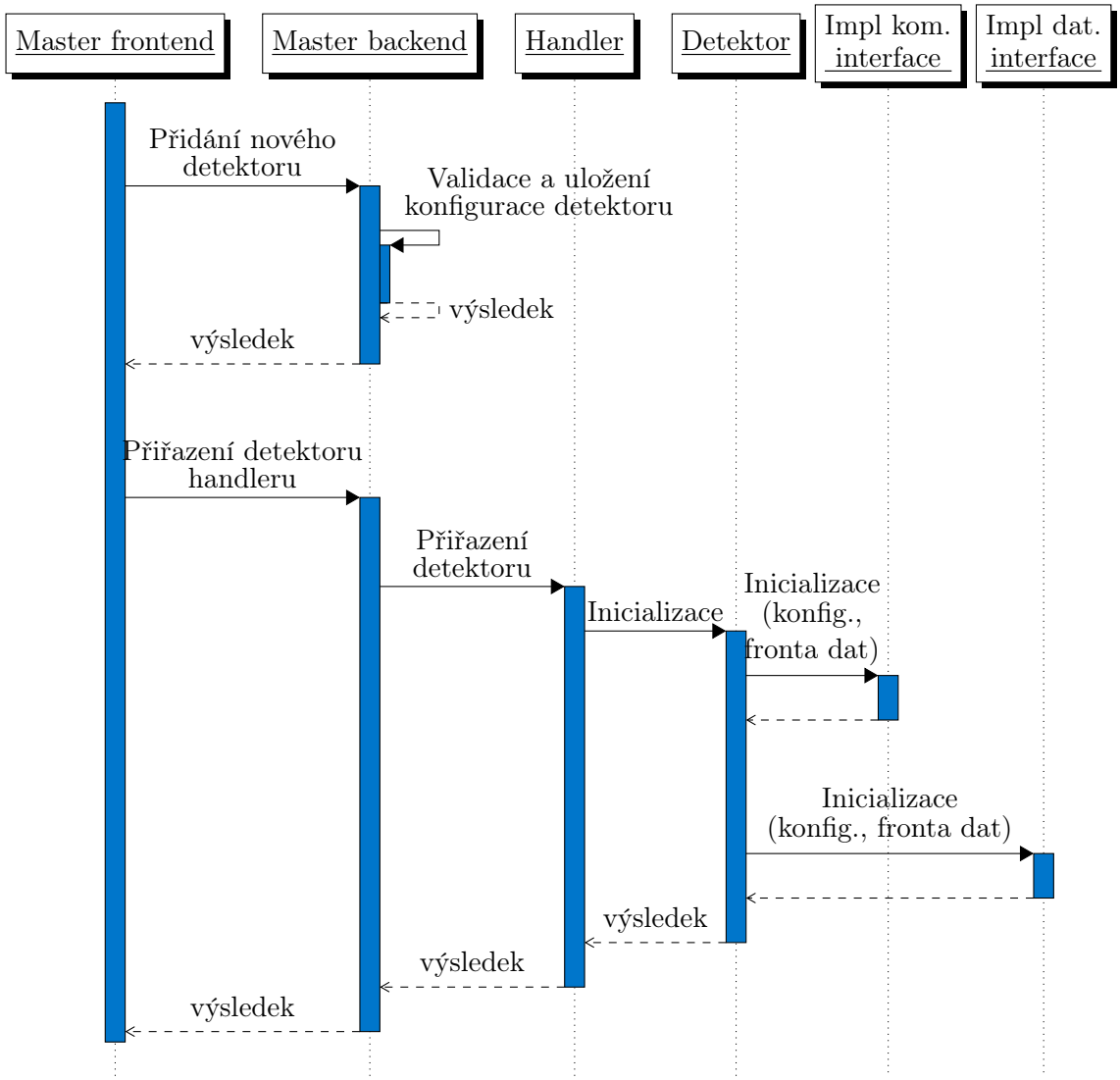
Na obrázku 3.3 je příklad sítě sestávající se ze dvou podsítí (tzv. *subnetwork*). Podsítí rozumíme takovou podmnožinu detektorů a handlerů, ve které libovolný detektor může být přiřazen libovolnému handleru. Pro úplnost je třeba doplnit, že nutnou podmínkou je, aby všechny handlery měly spojení s masterem a s distribuovaným úložištěm naměřených dat.

Master se skládá ze tří komponent:

Backend server implementující business logiku pro komunikaci s handlery (jako například přiřazování detektorů, řízení akvizice dat apod.). Server zároveň poskytuje *API* pro možnou integraci s webovým frontend serverem se systémy třetích stran[5], kterými muže být plnohodnotně řízen.

[4] Z angl. *Structured Query Language* (strukturovaný dotazovací jazyk).

[5] Například pro experiment *ATLAS-TPX* v CERN je plánována integrace s `DCS` [4] (*Detector Control System*), který umožňuje centrální řízení detektorových systému, včetně integrace do bezpečnostního systému `DSS` (*Detector Safety System*), datového akvizičního systému `DAQ` (*Data Acquisition System*) apod.

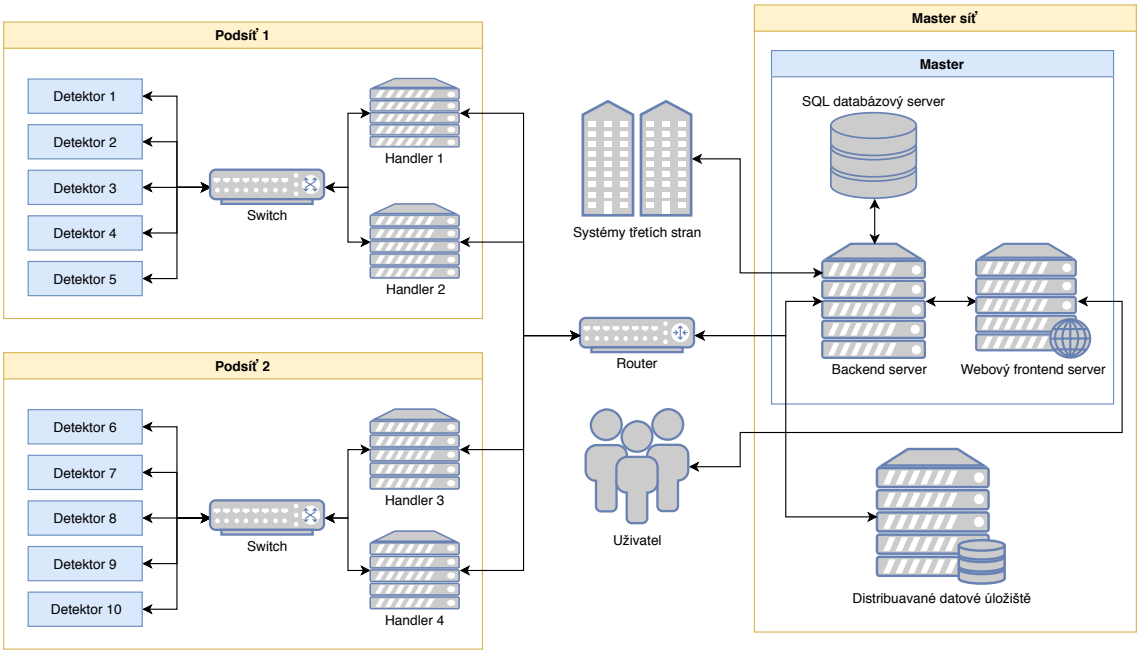


Obrázek 3.2: Sekvenční diagram znázorňující příklad přidání detektoru do systému a jeho přiřazení handleru.

Databázový server poskytující relační `SQL` databázi pro persistenci konfigurace systému, včetně implementace potřebných rozhraní, stavových informací sítě apod.

Frontend server poskytující webovou aplikaci s uživatelským rozhraním pro řízení mastera pomocí jeho `API`, resp. pro řízení celé sítě.

Na příkladu zmíněném výše je síť využívající jedno centrální datové úložiště. Realizace této komponenty je závislá na poskytnuté implementaci datových rozhraní jednotlivých detektorů. Může být implementovaná pomocí centralizovaného, nebo distribuovaného systému. Umístění úložiště také není omezeno – může být umístěno v síti mastera, v jednotlivých podsítích, v síti mastera i v jednotlivých podsítích (buffer pro malou šířku pásma spojení podsíť



Obrázek 3.3: Pixnet: hardwarová architektura s příkladem realizace sítě o dvou podsítích, kde v každé jsou dva handlery a pět detektorů, mastera (s *SQL* databází pro persistenci konfigurace a frontend server poskytující webovou aplikaci) a centrálního datového úložiště naměřených dat.

– master síť), nebo třeba v internetu (cloudové úložiště, jako například *Firebase Firestore*, *Amazon 3S*, *Microsoft Azure* apod.).

3.4 Škálovatelnost

Škálování systému Pixnet teoreticky není omezeno. Kritickým bodem každého datového akvizičního systému je vyčítání a ukládání naměřených dat. Tento problém je řešen horizontálním škálováním, spočívajícím v přidáváním handlerů, které zajišťují komunikaci s detektory a ukládání naměřených dat.

Možnost škálování datového úložiště je závislá na instanci hardwarové architektury popsané výše. V [9] bylo ukázáno, že `NoSQL` databázové systémy (jako například *MongoDB* nebo *Apache Cassandra*) jsou dobře horizontálně škálovatelné a jsou schopny zajistit dostatečnou rychlost zápisu vstupních dat. Takže například pro instanci sítě s vhodně zvoleným distribuovaným datovým úložištěm, umístěným v síti mastera (viz obr. 3.3), budou požadavky na persistenci naměřených dat splněny.

3.5 Použité technologie

V této podkapitole jsou stručně shrnuty technologie použité při implementaci systému.

Java je staticky typovaný, objektově orientovaný programovací jazyk a byl při zahájení implementace zvolen jako primární programovací jazyk pro vývoj tohoto systému. Byl zvolen především pro svou jednoduchost, výkonost a přenositelnost.

Java je interpretovaný programovací jazyk – při kompilaci kód není kompilovaný do strojového kódu zvolené platformy, ale do tzv. *bytecode*, který je platformně nezávislý. *Bytecode* pak může být spuštěn na libovolném počítači či zařízení, které disponuje interpretem Javy – `JVM` (*Java Virtual Machine*). Java není čistě interpretovaný programovací jazyk, protože v pozdějších verzích byla přidána podpora pro `JIT` (tzv. *Just-in-time*) kompilátoru, který umožňuje dynamickou kompilaci *bytecode* do strojového kódu dané platformy, díky čemuž může Java ve srovnání s neinterpretovanými programovacími jazyky (`C++` apod.) dosahovat srovnatelných výsledků.

Java také obsahuje nástroj pro generační správu paměti (tzv. *Garbage Collector*), který se stará o automatické uvolňování paměti mazáním objektů, na které neukazuje žádná reference. Tento nástroj je pro dlouhodobě běžící serverové systémy velice užitečný.

Kotlin je moderní staticky typovaný, objektově orientovaný programovací jazyk, který běží nad `JVM`. Hlavní výhoda Kotlinu je jeho interoperabilita – kód může být zkompilovaný do Java *bytecode*, do JavaScriptu, nebo i do nativního kódu s tím, že může používat závislosti dané platformy (například pro kompilaci do Java *bytecode* může používat Java knihovny, takže část kompilovaného kódu může být napsána v Javě a část v Kotlinu).

Oproti Javě poskytuje mnoho výhod, jako například *null-safety* (každá proměnná má kromě svého datového typu i příznak, jestli může být `null`, což eliminuje spoustu chyb, se kterými se můžeme setkat například v Javě), podpora funkcionálního stylu programování (oproti Javě může být v proměnné uložená funkce), *extension functions* (podobně jako v `C#`, je možné přidat funkci do dané třídy bez nutnosti její modifikace, nebo vytvořené poděděné třídy) a další vylepšení.

Především díky výhodám popsaným výše bylo v pozdější fázi vývoje rozhodnuto o použití Kotlinu, coby primárního programovacího jazyku pro handler a backend mastera. Také bylo experimentováno s použitím Kotlinu pro frontend mastera, pomocí vytvoření společné *code-base* pro backend a frontend mastera (sestávajícího se z business logiky a modelu), ale od tohoto přístupu bylo upuštěno z důvodů omezené podpory kompilátoru Kotlin kódu do JavaScriptu a z důvodu omezené komunitní podpory.

JavaScript je interpretovaný, objektově orientovaný programovací jazyk, který je zpravidla používán pro tvorbu webových aplikací, ale má i jiná využití. V kontextu webových aplikací, JavaScript kód je interpretován na klientské straně v prohlížeči a pomocí jeho *API* manipuluje s načtenou HTML stránkou, resp. její vnitřní interpretací webovým prohlížečem – tzv. `DOM` (*Document Object Model*).

V rámci této práce je použit pro implementaci frontendové části mastera – webové aplikace poskytující uživatelské rozhraní pro obsluhu systému Pixnet.

Gradle Je automatizovaný build[6] systém, který vychází z *Apache Ant* a *Apache Maven* a používá `DSL`[7] (založeném na *Groovy* a *Kotlin*). Umožňuje automatizované stahování

[6] Sestavování počítačových programů.

[7] Z angl. *domain-specific language* (doménově specifický jazyk)

a správu závislostí a knihoven z online repositářů (kromě *Gradle* i *Maven* a *Ivy* pro kompatibilitu s dalšími build systémy).

Spring je aplikační framework pro vývoj J2EE[8] aplikací a v rámci systému Pixnet bude použit v rámci implementace handlera a mastera. Spring framework je sada modulů, které nabízejí nástroje jako například webový controller (pro poskytování REST API), *Dependency Injection*[9], JPA (*Java Persistent API* pro práci s databází), podporu testování apod.

Novější alternativou je Spring Boot, který je nadstavbou nad původním Spring frameworkem. Spring Boot odstraňuje nutnost komplikovaného definování konfigurace a také přináší možnost zapouzdření do samostatně spustitelné aplikace s embedovaným webovým serverem (*Tomcat*, nebo *Jetty*), takže již není třeba nasazovat WAR[10] na webový server, protože Spring Boot aplikace je zkompilovatelná do spustitelné JAR[11] aplikace.

ReactJS je JavaScriptová knihovna pro vývoj uživatelského rozhraní webových aplikací, vyvíjená společností Facebook. React je používán k vývoji tzv. *single-page* aplikací, tj. takových webových aplikací, které k interakci s uživatelem používají jen jednu stránku, jejíž obsah dynamicky přepisují (na rozdíl od stahování nové stránky z webového serveru).

React je založen na principu zapouzdřených komponent, kde každá komponenta má svoje vlastnosti a svůj stav. Komponenty jsou znovupoužitelné a zanořitelné do jiných komponent. Navíc komponenty nepracují přímo s DOM prohlížeče, ale přistupují pouze k jeho virtuální reprezentaci, což umožňuje optimalizovat operace s DOM. Například když nějaká komponenta změní svůj stav (ev. tranzitivně i stav jiných komponent), tak se na stránce přegenerují jen takové komponenty, jejichž stav byl změněn.

React bude použit pro implementaci frontendové webové aplikace mastera. Vedle Reactu existuje ještě React Native, který je určen pro hybridní vývoj mobilních aplikací, tím se ale v této práci nebudeme zabývat.

PostgreSQL je nejpoužívanější open source objektově-relační databázový systém. Oproti klasickým relačním databázím (například *MySQL*) má objektově orientovaný databázový model, který je přímo podporován databázovým schématem a dotazovacím jazykem.

Tento databázový systém bude použit pro persistenci konfigurace sítě a bude k němu přistupovat backend mastera.

MongoDB je NoSQL dokumentový databázový systém. Oproti tradičním relačním databázovým systémům jsou data ukládána do BSON[12] dokumentů. Mezi hlavní výhody *MongoDB* patří:

[8] Označení pro Java *Enterprise Edition* – platformy pro vývoj podnikových systémů, odvozené od JavaSE (Standard Edition)

[9] Technika pro vkládání závislostí mezi komponentami počítačového programu, aniž by v době kompilace měly komponenty na sebe referenci.

[10] Z angl. *Web Application Archive* (archív se zkompilovanou webovou J2EE aplikací).

[11] Z angl. *Java Archive* (archív se zkompilovanou Java aplikací).

[12] Binární forma JSON (z ang. JavaScript Object Notation)

Indexace: Na každé pole objektů v dokumentu lze vytvořit index a tím zrychlit vyhledávání v datech.

Replikace: *MongoDB* ukládá data do tzv. *replica set*, která obsahuje jednu, nebo více replik dat. Pro případ více replik, jedna vždy funguje jako primární a ostatní jako sekundární, do kterých jsou replikována data z primární repliky. Hlavní výhoda spočívá ve vyšší dostupnosti dat (data lze číst z více replik současně) a spolehlivosti (když jedna replika selže, je nahrazena replikou jinou).

Horizontální škálování: *MongoDB* má podporu pro horizontální škálování pomocí tzv. *shardingu* [16]. *Sharded cluster* je pak množina uzlů s jednotlivými *shardy*, kde každý obsahuje podmnožinu *shardovaných* dat.

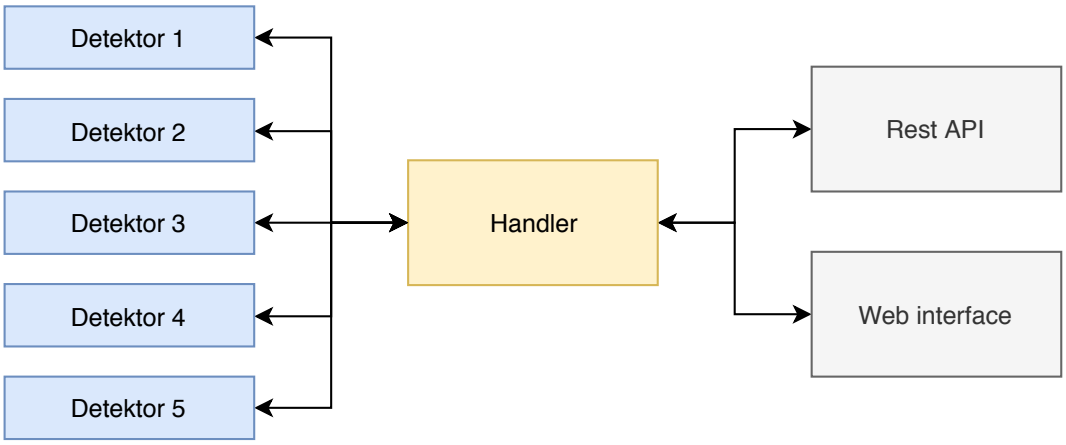
Kapitola 4

Handler

V této kapitole bude čtenář podrobně seznámen s handlerem – komponentou pro distribuované řízení detektorů. Jak již bylo zmíněno v kapitole 3, handler je komponenta zajišťující komunikaci a vyčítání dat z detektorů (viz obr. 4.1) skrze poskytnuté implementace komunikačního a datového interface.

Handler implementuje *Spring framework* (viz 3.5), aby mohl poskytovat `REST API` pro management detektorů a pro poskytování stavových informací o své instanci. Handler dále poskytuje jednoduché webové rozhraní s přehledem připojených detektorů.

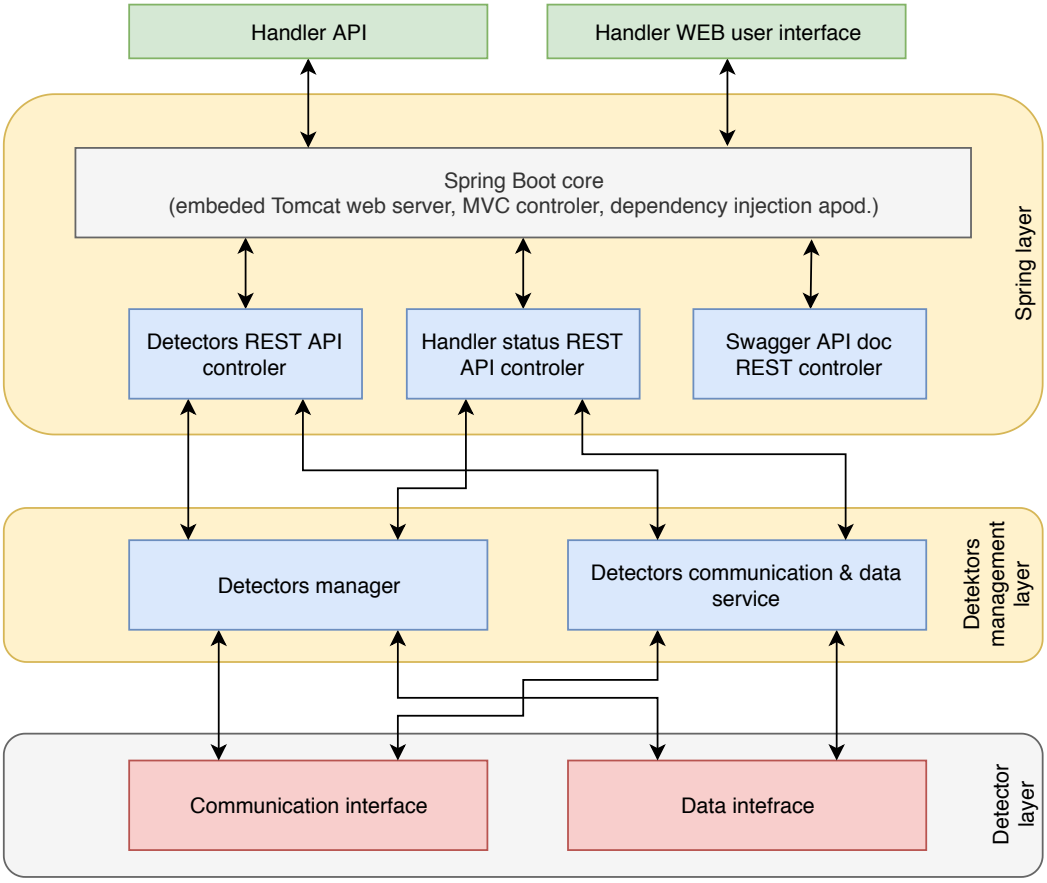
Na obrázku 4.1 je znázorněn příklad instance handleru z pohledu vstupu (pět přiřazených detektorů) a pohledu výstupu (poskytované `REST API` a webové rozhraní.)



Obrázek 4.1: Pixnet – handler: příklad instance handleru s pěti připojenými detektory, poskytujícího REST API pro své řízení a webové uživatelské rozhraní s přehledem připojených detektorů.

4.1 Vrstvy softwarové architektury

Tato podkapitola je věnována softwarové architektuře handleru. Jsou zde představeny její jednotlivé vrstvy, tj. od vrstvy pro komunikaci s detektorem, management detektorů, Spring implementaci, až po výstupní vrstvu poskytující `REST API` a webové rozhraní – viz obr. 4.2. Vybrané vrstvy budou popsány detailněji v dalších podkapitolách.



Obrázek 4.2: Pixnet – handler: softwarová architektura s vrstvami pro (i) rozhraní detektoru (*Detector layer*), (ii) management detektorů (*Detectors management layer*),(iii) Spring vrstvu (*Spring layer*) a (iv) výstupní vrstvou pro `REST API` a webové uživatelské rozhraní.

4.1.1 Detektorová vrstva

Detektorová vrstva se skládá ze dvou částí – externě poskytnuté implementace komunikačního a datového interface. Tyto tzv. moduly jsou zavedené až za běhu programu pomocí vyšších vrstev systému.

4.1.1.1 Zavádění modulů

Každý modul, obsahující implementaci komunikačního nebo datového interface, musí pro své zavedení do systému splňovat následující kritéria:

- Musí být zkompilován do Java archívu (`*.jar`) – současná verze implementace podporuje pouze moduly vyvinuté v jazyce Java (resp. také v jazyce Kotlin, zkompilovaného do Java *bytecode*), v dalších verzích je plánováno přidání podpory pro `C++` a `Python`.
- Při vývoji modulu musí být použit datový model, který je součástí poskytovaných knihoven.
- Modul musí obsahovat implementaci jednoho z již zmíněních rozhraní. Navíc, celý název implementující třídy (vč. tzv. `package`) musí být uveden jako atribut s názvem `PluginImplClass` v manifestu `jar` archívu, viz zdrojový kód 4.1.

```
1 Manifest-Version: 1.0
2 PluginImplClass:
  ↪ cz.ctu.ieap.pixnet.handler.detector_communication_katherine.CommImpl
```

Zdrojový kód 4.1: Příklad obsahu souboru `MANIFEST.MF`, obsaženého v `jar` archívu modulu.

4.1.1.2 Komunikační rozhraní

Zdrojový kód 4.2 obsahuje interface, který komunikační modul detektoru musí implementovat.

Prvních pět metod (tj. řádek 3 až 7) má čistě informativní charakter a jejich implementace slouží pro operátora systému.

Na řádcích 8 a 9 je setter a getter pro konfiguraci detektoru. Konfigurace je detektoru předána jako `String`[1] (setter) a musí do něj být serializovatelná (getter). Systém ale s konfigurací detektoru neumí pracovat (kromě komunikačního modulu detektoru, resp. implementace komunikačního interface), tudíž její syntax není vynucována a její podoba je čistě v kompetenci poskytovatele komunikačního modulu. Je avšak doporučeno, aby zvolený formát byl strojově i lidsky čitelný, z důvodu jeho snazší editace[2]. Takový formát může být například `JSON`[3], nebo `YAML`[4], který je použit pro serializace konfigurace komunikačního modulu Katherine (viz 5.2).

Pro získání seznamu podporovaných *value commands* (tj. příkazů pro operace s jednotlivými hodnotami detektoru) slouží metoda `getSupportedValueCommands()`, viz řádek 10. `AbstractValueCommand` má v modelu poskytované knihovny dvě implementace:

[1] Datový typ obsahující textový řetězec

[2] V dalších fázích implementace systému je plánováno přidání podpory editace konfigurace v rámci webového uživatelského rozhraní mastera.

[3] Z angl. *JavaScript Object Notation* (JavaScriptový objektový zápis).

[4] <http://yaml.org/>

```kotlin
interface DetectorComm {

  fun getDetectorType(): DetectorType
  fun getReadoutName(): String
  fun getSensorsCount(): Int
  fun getDetectorWidth(): Int
  fun getDetectorHeight(): Int
  fun setDetectorConfig(config: String)
  fun getDetectorConfig(): String?
  fun getSupportedValueCommands(): List<AbstractValueCommand>
  fun getSupportedExecutionCommands(): List<AbstractExecutionCommand>
  fun getAcceptedFilesKeys(): List<String>
  fun getDataFrameQueue(): BlockingQueue<AbstractDataFrame>

  fun isConnected(): Boolean
  fun connect(): Boolean
  fun disconnect(): Boolean

  fun executeSetValueCommand(commandID: Int, payload: ValuePayload)
  fun executeGetValueCommand(commandID: Int): ValuePayload
  fun executeExecutionCommand(commandID: Int, input: Map<String,
  ↪  ValuePayload>): Map<String,  ValuePayload>
  fun uploadFile(fileKey: String, file: ByteArray)
  fun setCallback(callback: Callback)

  interface Callback {
    val classLoader: ClassLoader?
  }

}
```

Zdrojový kód 4.2: Komunikační interface detektoru, napsaný v jazyce Kotlin (viz 3.5)).

(i) `ValueCommand` obsahuje atributy daného příkazu, tj.:

- **id** – celočíselný unikátní identifikátor daného příkazu,
- **name** – název příkazu, resp. manipulované hodnoty detektoru,
- **valueUnit** – jednotka veličiny manipulované hodnoty detektoru (může nabývat hodnot z *Enum* třídy `ValueUnit` poskytovaného modelu, např. `ValueUnit.VOLT` apod.),
- **accessType** – modifikátor přístupu manipulované hodnoty, který může nabývat těchto hodnot:

  SETTER pro takové hodnoty, které je možné pouze nastavovat,

GETTER pro takové hodnoty, které je možné pouze číst a
SETTER_AND_GETTER pro hodnoty, které je možné nastavovat i číst.

- **valueModel** – datový model hodnoty daného příkazu, který definuje datový typ veličiny (podporovány jsou `Boolean`, `String`, `Integer`, `Long`, `Float` a `Double`) a omezení rozsahu hodnot. Model může být diskrétní (tzn. hodnota může nabývat jen nějaké z předem definovaných hodnot), nebo spojitý (hodnota může nabývat jakékoliv hodnoty ze zadaného intervalu).

Viz zdrojový kód 4.3 pro příklad definice *ValueCommand* pro *bias* (prahové napětí detektoru).

(ii) `ValueCommandGroup` je třída pro seskupování příkazů podobného významu (např. `DAC` hodnoty detektoru) a obsahuje název skupiny příkazů a seznam jednotlivých `ValueCommands`.

```
1 valueCommands.add(ValueCommand(
2     42, // id
3     "Bias", // name
4     ValueUnit.VOLT, // valueUnit
5     SETTER_AND_GETTER, // accessType
6     FloatValueModel(-300f, 300f) // valueModel
7 ))
```

Zdrojový kód 4.3: Příklad definice *ValueCommand* detektoru pro příkaz s názvem *"Bias"*, id 42, jednotkou Volt, modifikátorem přístupu *Setter & Getter* a reálným modelem hodnot, omezeným intervalem $< -300, 300 >$.

Pro vykonání `ValueCommand` je třeba implementovat metody `executeSetValueCommand()` a `executeGetValueCommand()`, viz řádky 19 a 20 komunikačního interface (zdrojový kód 4.2). První metoda slouží pro nastavení hodnot a akceptuje ID příkazu a `valuePayload`, který obsahuje hodnotu jednoho ze šesti podporovaných datových typů, a je možné jej vykonat pouze pro příkazy, které mají `accessType`, umožňující zápis hodnot. Druhá metoda slouží pro čtení hodnot, jako vstupní parametr má ID příkazu a její návratový typ je `ValuePayload`, obsahující hodnotu čteného parametru.

Dalším typem příkazu je získání seznamu podporovaných `ExecutionCommands` (viz 11. řádek komunikačního interface 4.2). Od `ValueCommands` se liší tím, že vstup a výstup není omezen stejnou veličinou (resp. jejím modelem hodnot) ani jejich množstvím. Tento přístup tedy umožňuje definovat příkazy, které mají libovolný počet vstupních hodnot a libovolný počet výstupních hodnot. Obdobně jako `AbstractValueCommand`, i `AbstractExecutionCommands` má v poskytované knihovně dvě implementace:

(i) `ExecutionCommand` je definován obdobně jako `ValueCommand` – má své ID, jméno a také obsahuje seznamy rozšířených modelů hodnot – jeden vstupní a jeden výstupní. Rozšířený model hodnot obsahuje již zmíněný `valueModel`, dále pak jméno hodnoty, její ID (textový řetězec) a `valueUnit`.

Zdrojový kód 4.4 obsahuje příklad s `ExecutionCommand` pro nastavení akvizičního módu detektoru. Z příkladu je patrné, že příkaz akceptuje právě dvě vstupní hodnoty (akviziční mód a *FastVCO* přepínač) a že žádné hodnoty nevrací.

(ii) `ExecutionCommandGroup` je obdobou třídy `ValueCommandGroup` pro `ExecutionCommand`. Obsahuje název skupiny příkazů a jejich seznam.

```
executionCommands.add(ExecutionCommand(
  KatherineExecutionCommands.SET_ACQ_MODE.internalID, // ID (Int)
  "Set acquisition mode", // name
  // model vstupných hodnot
  arrayOf(
    ValueModelVerbose(
      "Acquisition mode", // název hodnoty
      ValueId.acq_mode.name, // ID hodnoty (String)
          // celočíselný diskrétní model hodnot
          IntValueModel(mapOf(
        Pair("ToA & ToT", 0),
        Pair("ToA", 1),
        Pair("Event & iToT", 2)
      )),
      ValueUnit.DIMENSIONLESS // jednotka hodnoty
    ),
    ValueModelVerbose(
      "Fast vco enabled",
      ValueId.fast_vco_en.name,
      BooleanValueModel(mapOf(
        Pair("Enable", true),
        Pair("Disable", false)
      )),
      ValueUnit.DIMENSIONLESS
        )
  ),
  null // model výstupních hodnot
))
```

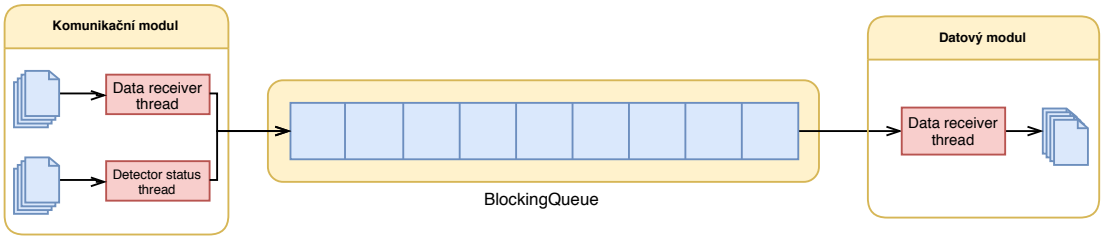
Zdrojový kód 4.4: Příklad definice *ExecutionCommand* pro nastavování akvizičního módu detektoru s vyčítacím rozhraním *Katherine* (viz 2.7.4). Z příkladu je patrné, že vstupní model je tvořen dvěma hodnotami a výstupní model je prázdný.

Pro vykonání `ExecutionCommand` slouží metoda `executeExecutionCommand()`, viz řádek 21 komunikačního interface (zdrojový kód 4.2). Metoda akceptuje ID příkazu a mapu (tzn. seznam párů klíč – hodnota) jednotlivých hodnot. Klíčem v mapě je ID parametru a hodnota je již výše zmíněný `ValuePayload`, obsahující hodnotu parametru. Výstupem volání metody je pak mapa výstupních parametrů příkazu.

Pro připojení detektoru, odpojení detektoru a zjišťování stavu připojení slouží metody `connect()`, `disconnect()` a `isConnected()` (viz řádky 15 až 17 zdrojového kódu 4.2). Aby bylo možné provést jakoukoliv operaci interagující s detektorem (tj. vykonávání příkazů a nahrávání souborů) je nutné, aby byl detektor připojen, resp. metoda `isConnected()` musí vracet `true`.

Do detektorů rodiny *Medipix* je třeba před jejich použitím nahrát konfiguraci jednotlivých pixelů detektoru. Konfigurace je pole bytů, kde nastavení jednoho pixelu je serializováno do jednoho bytu. Z tohoto důvodu je třeba přidat podporu pro nahrávání velkých binárních objektů (tzv. `LOB`[5]). Pro nahrávání souborů do detektoru slouží příkaz `uploadFile()` (viz řádek 22 zdrojového kódu 4.2), který akceptuje ID souboru a jeho binární reprezentaci. Pro poskytnutí seznamu podporovaných souborů, resp. jejich ID, je třeba implementovat metodu `getAcceptedFilesKeys()` (viz řádek 12 zdrojového kódu 4.2).

V kapitole 3.2.1 již byl popsán tok měřených dat v handleru, resp. od komunikačního k datovému modulu. Přenos dat je realizován asynchronní blokující frontou (viz obr. 4.3), tj. frontou ke které může asynchronně přistupovat více vláken současně a zároveň čtení z fronty je implementováno jako blokující operace (tzn. že zablokuje čtecí vlákno, než bude ve frontě nějaký element k vyčtení). Jelikož komunikační modul je zodpovědný za vytvoření instance fronty, tak její implementace je zcela v kompetenci poskytovatele komunikačního modulu. Jedinou podmínkou je, aby fronta implementovala interface `BlockingQueue`[6]. V další části textu (viz 5) bude čtenář seznámen s implementací komunikačního modulu s implementací fronty pomocí `LinkedBlockingQueue`[7].



Obrázek 4.3: Asynchronní blokující fronta naměřených dat s příkladem produkujících vláken v komunikačním modulu a přijímacím vláknu v datovém modulu.

Poslední metodou komunikačního interface je metoda `setCallback()` (viz 23. řádek komunikačního interface (zdrojový kód 4.2)). Tato metoda je handlerem zavolána bezprostředně po inicializaci modulu a modul si může uložit referenci na předaný `callback` (instanci interface `Callback` z komunikačního interface). `Callback` slouží například k získání Java `ClassLoader`, kterým byl modul načten (používá se například pro parsing konfigurace).

[5] Z angl. Large Object.
[6] Z *Java Collections Framework*.
[7] Implementace `BlockingQueue` pomocí spojového seznamu. V této frontě jsou elementy řazení pomocí FIFO (first-in-first-out). Fronty založené na spojové struktuře umožňují (ve srovnání s frontami založenými na dynamickém poly) vyšší datový tok, zejména pro vícevláknové aplikace.

4.1.1.3 Datové rozhraní

Zdrojový kód 4.5 obsahuje interface, které musí být implementováno datovým modulem detektoru.

```
1  interface DataPersistence {
2
3    fun setDataFrameQueue(queue: BlockingQueue<AbstractDataFrame>)
4    fun setDetectorConfig(config: String)
5
6    fun start()
7    fun stop()
8
9    fun setCallback(callback: Callback)
10
11   interface Callback {
12     val classLoader: ClassLoader?
13   }
14
15 }
```

Zdrojový kód 4.5: Datový interface detektoru, napsané v jazyce Kotlin (viz 3.5)).

Metoda `setDataFrameQueue()` (viz řádek 3) je zavolána po inicializaci modulu a jejím prostřednictvím handler předá datovému modulu frontu měřených dat, která byla vytvořena komunikačním modulem (viz předchozí komponenta).

Na 4. řádku je metoda pro předání konfigurace, která probíhá stejně, jako u komunikačního interface. Za tímto účelem je v datovém interface také `Callback` interface a metoda pro předání jeho instance, vytvořené handlerem. `Callback` obsahuje metodu pro předání `ClassLoader`, kterým byl modul zaveden.

Na řádcích 6 a 7 jsou metody `start()` a `stop()`, kterými se spouští a zastavuje vlákno zpracovávající frontu měřených dat.

4.1.2 Vrstva managementu detektorů

Vrstva pro management detektorů slouží jako úložiště detektorů a jako služba pro jejich řízení. Tato vrstva se skládá ze dvou komponent, které budou popsány v této podkapitole.

`DetectorManager` je komponenta, sloužící pro uložení detektorů přiřazených handleru do jeho paměti. Její instance je poskytována pomocí Java Bean Spring frameworkem jako *singleton*, tzn. že v programu existuje pouze jedna instance této komponenty. Komponenta poskytuje úložiště detektorů, včetně *CRUD*[8] operací nad jeho obsahem. Díky Spring frameworku, resp. jeho `Dependency-Injection` modulu lze tuto komponentu jednoduše použít z jiné komponenty systému.

[8] Z angl. *Create, read, update and delete* (vytváření, čtení, editaci a mazání).

Detektor je reprezentován instancí třídy `Detector`, které má následující parametry:

- detectorID – unikátní textový identifikátor detektoru v systému,
- `name` – vlastní pojmenování detektoru (např. s označením fyzického umístění detektoru apod., nemusí být unikátní),
- `detectorComm` – instance komunikačního interface detektoru (viz 4.1.1.2),
- `dataPersistence` – instance datového interface detektoru (viz 4.1.1.3),
- `pluginsClassLoader` – Java `ClassLoader`, kterým byly načteny výše zmíněné moduly a
- `status` – instance třídy `DetectorStatus`, ve které jsou uloženy stavové informace o detektoru (např. stav připojení, měření apod.).

`DetectorService` je komponenta poskytující operace nad detektorem, vč. navazování spojení, nahrávání souborů, vykonávání *ValueCommands* a *ExecutionCommands* apod. Komponenta je implementována jako `Service`[9].

4.1.3 Spring vrstva

Tato vrstva se skládá z několika komponent, Spring Boot jádra a dalších knihoven Spring frameworku (viz obr. 4.2). V této podkapitole bude čtenář seznámen s implementací jednotlivých komponent, tj. *Detectors REST API controller* (viz 4.1.3.1) pro správu a řízení detektorů masterem (ev. jinými systému), *Handler status REST API controller* (viz 4.1.3.2) pro zjišťování stavu handleru masterem, *Swagger API doc REST controller* (viz 4.1.3.3) pro webovou dokumentaci REST API poskytovaného handlerem a jednoduchého webového rozhraní handleru (viz 4.1.3.4).

4.1.3.1 Detectors REST API controller

Tato komponenta je `RestController`, poskytující endpointy[10] pro *CRUD* operace nad detektory a jejich řízení. Komponenta je závislá na `DetectorManager` a `DetectorService` (viz 4.1.1). Všechny endpointy mají prefix URL cesty `/api/detector/`, takže například URL endpointu pro přidání detektoru může být <`http://localhost:8082/api/detector/add`>. Seznam API endpointů této komponenty je v tabulce 4.1.

Pro přenášená data je použit formát JSON. Zdrojový kód 4.6 znázorňuje ukázku API volání s příkladem s nastavením akvizičního módu detektoru (definice tohoto příkladu je uvedena ve zdrojovém kódu 4.4) pomocí `executeExecutionCommand` endpointu.

Dokumentace k jednotlivým endpointům je poskytování jinou komponentou systému, viz kapitola 4.1.3.3.

[9] Dle [8] *Service* je množina operací, která není součástí modelu a zároveň není nositelem stavu.
[10] Označení pro API metodu.

HTTP Metoda	Endpoint	Popis
GET	`/getAll`	Vrátí seznam všech detektorů
GET	`/getById`	Vrátí detektor podle zadaného ID
POST	`/add`	Přidá nový detektor
DELETE	`/remove`	Odstraní detektor podle zadaného ID
POST	`/connect`	Provede připojení zvoleného detektoru
POST	`/disconnect`	Provede odpojení zvoleného detektoru
POST	`/executeValueCommand`	Vykoná `ValueCommand`
POST	`/executeExecutionCommand`	Vykoná `ExecutionCommand`
POST	`/uploadFile`	Nahraje soubor do zvoleného detektoru

Tabulka 4.1: Endpointy komponenty *Detectors REST API controller*.

```
1  > POST /api/detector/executeExecutionCommand HTTP/1.1
2  > Host: localhost:8082
3  > Content-Type: application/json
4  > Content-Length: 133
5  {
6    "detectorID": "katherine_emulator",
7    "commandID": 200,
8    "input": {
9      "acq_mode": {"intValue": 0},
10     "fast_vco_en": {"booleanValue": false}
11   }
12 }
13
14 < HTTP/1.1 200
15 < Content-Type: application/json;charset=UTF-8
16 {
17   "success": true
18 }
```

Zdrojový kód 4.6: Příklad volání API komponenty pro správu detektorů. V příkladu na řádcích 1 až 12 je *request* pro vykonání `ExecutionCommand` pro nastavení akvizičního módu detektoru a na řádcích 14 až 18 je *response* serveru.

4.1.3.2 Handler status REST API controller

Tato komponenta je určena pro poskytování stavu handlerům dalším částem systému (např. master), ev. systémům třetích stran. Komponenta poskytuje jediný endpoint – `/status` (HTTP metoda `GET`)). Ve zdrojovém kódu 4.7 je příklad volání tohoto endpointu, resp. JSON těla odpovědi. Z příkladu je patrné, že handler je online, má verzi software `0.0.0.1` a spravuje dva detektory.

```
{
  "data": {
    "version": "0.0.0.1",
    "status": "ONLINE",
    "detectors": [
      {
        "id": "emulator_katherine",
        "name": "katherine emulator",
        "status": {
          "connection": "ONLINE",
          "measurement": "UNKNOWN"
        }
      },
      {
        "id": "katherine",
        "name": "katherine",
        "status": {
          "connection": "UNKNOWN",
          "measurement": "UNKNOWN"
        }
      }
    ]
  },
  "success": true
}
```

Zdrojový kód 4.7: Příklad volání API komponenty pro poskytování stavu handleru, resp. těla odpovědi endpointu `/status`.

4.1.3.3 Swagger API doc REST controller

Ke každému API je třeba poskytovat dokumentaci pro umožnění jeho implementace. Navíc, každá změna API by se měla současně projevit i v dokumentaci. Manuální udržování API dokumentace je pracné a náchylné na možné odchylky dokumentace od aktuálního stavu API. Proto bylo rozhodnuto použít automatizovaný nástroj na generování dokumentace.

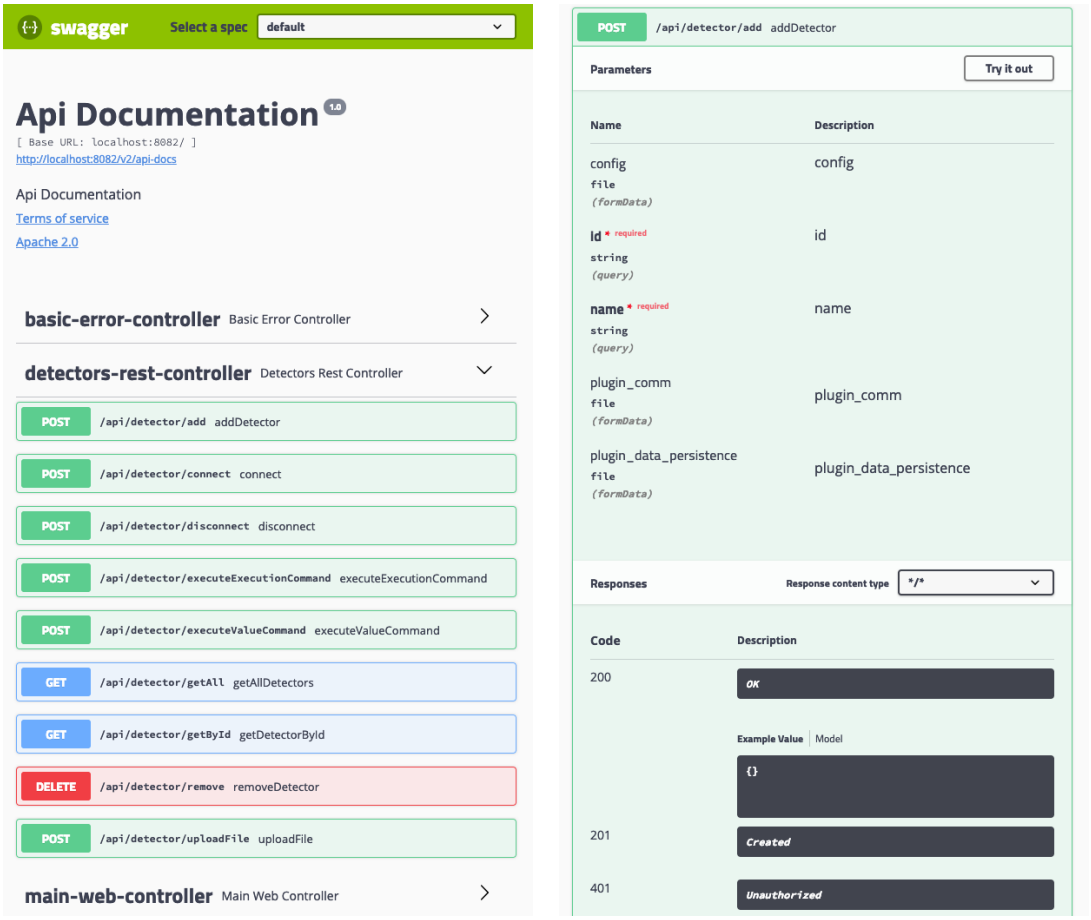
Za tímto účelem byla implementována knihovna *Springfox* [18] – nástroj pro generování strojově i lidsky čitelné dokumentace. *Springfox* je založen na *OpenAPI*[11] specifikaci a *Swagger*[12] frameworku. *Springfox* po spuštění aplikace automaticky analyzuje její komponenty a jednotlivé její třídy pomocí Java reflexe. Na základě analýzy potom vytvoří sémantický model jednotlivých endpointů.

Výstupem je pak online webová dokumentace, dostupná pomocí `swagger-ui.html` end-

[11] *OpenAPI Specification* (dříve *Swagger Specification*) je formát pro popis REST API.

[12] Nadstavba nad *OpenAPI* umožňující návrh, vytváření (resp. generování serverového i klientského kódu), dokumentaci a testování REST API.

pointu (tzn. url dokumentace je například <http://localhost:8082/swagger-ui.html>, dle konfigurace – viz 4.2) – viz obrázek 4.4. Webové rozhraní poskytuje nejen přehled všech endpointů včetně jejich parametrů, ale i definici datového modelu a nástroj pro manuální volání jednotlivých endpointů. Endpoint pro stažení JSON OpenAPI specifikace je `/v2/api-docs`.



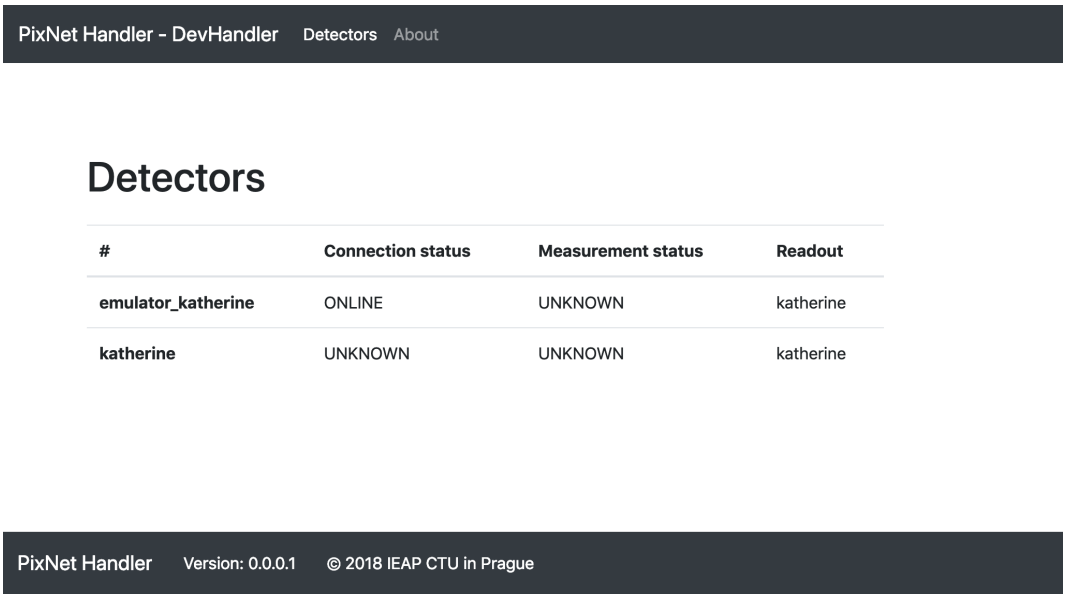
(a) Přehled endpointů. (b) Detail endpointu.

Obrázek 4.4: Springfox online dokumentace handleru.

4.1.3.4 Webové rozhraní handleru

Handler poskytuje jednoduché webové rozhraní, umožňující uživateli přehled přiřazených detektorů, viz obrázek 4.5. Rozhraní je implementováno pomocí *Spring Web MVC* frameworku a *Thymeleaf* (Java template engine pro vytváření HTML stránek na straně serveru). Jedná se tedy o statickou stránku a pro její obnovení je potřeba její opětovné načtení (pro vygenerování nové HTML stránky na straně serveru).

Webové rozhraní je dostupné z endpointu `/detectors`. Pro jednoduchost bylo na tento endpoint přidáno přesměrování z výchozí URL adresy (např. <http://localhost:8082>).



Obrázek 4.5: Screenshot webového rozhraní handleru.

4.2 Konfigurace a nasazení

Pro spuštění handleru je třeba mít nainstalované `JRE 8`[13] nebo vyšší. Zkompilovanou Java aplikaci je třeba spustit s argumentem `handlerConfig` obsahující cestu ke konfiguračnímu souboru (pro příklad viz zdrojový kód 4.8), obsahujícího port na kterém bude aplikace naslouchat a název handleru.

Aplikaci je tedy možné spustit například takto:

```
java -jar handler.jar handlerConfig=config.yaml
```

```
1 portToListen: 8082
2 handlerName: Handler1
```

Zdrojový kód 4.8: `YAML` konfigurační soubor handleru.

[13] *Java SE Runtime Environment, dostupné z* <https://www.oracle.com/technetwork/java/javase/downloads>.

Kapitola 5

Vyčítací rozhraní Katherine a jeho implementace

V této kapitole bude čtenář blíže seznámen s komunikačním protokolem vyčítacího rozhraní *Katherine* (viz 2.7.4) a bude zde také představena implementace komunikačního (viz 4.2) a datového (viz 4.5) interface, vyvinutých za účelem podpory *Katherine* handlerem (viz 4).

Pro účely vývoje a testování byl vyvinut emulátor *Katherine*, který emuluje zařízení na síťové vrstvě, včetně podpory všech příkazů komunikačního protokolu relevantních k akvizici dat. Popis jeho návrhu a implementace bude také zahrnut v této kapitole.

Pro úplnost je třeba dodat, že komunikační a datový modul sdílejí stejnou podmnožinu datového modelu a část business-logiky, vázající se k manipulaci s měřenými daty. Tato společná *code-base* je implementována v rámci separátního modulu, na kterém jsou závislé oba výše zmíněné moduly. Toto řešení umožňuje použití stejné vnitřní reprezentace dat, bez nutnosti jejich serializace a deserializace, což má pozitivní vliv na výkonost systému. Nutnou podmínkou ale je, aby oba moduly byly do systému zavedeny pomocí stejného `ClassLoader` objektu (z důvodu kompatibility přenášených model objektů). Tento mechanizmus je ale už implementován v handleru.

5.1 Komunikační protokol

Jak již bylo vysvětleno v kapitole 2.7.4.1, *Katherine* podporuje dva provozní módy – autonomní a manuální. Tato práce se zabývá pouze použitím manuálního módu, který umožňuje plnohodnotné řízení detektoru a umožňuje (ve srovnání s autonomním módem) efektivněji přenášet naměřená data, díky čemuž je možné maximalizovat tok výstupních dat detektoru.

Komunikace v rámci manuálního módu je založena na proprietárním komunikačním protokolu, který bude popsán v této podkapitole. Klientská část tohoto protokolu je implementována v komunikačním modulu handleru (viz 5.2) a jeho serverová část v *Katherine* emulátoru (viz 5.4).

Protokol je založen na posílání `UDP` datagramů pomocí dvou socketů[1] – jednoho pro

[1] Označení pro koncový bod obousměrné komunikace mezi počítačovými programy v rámci počítačové sítě.

řídící příkazy a druhého pro měřená data. Využití `UDP` se nabízí především z důvodu implementovaného nepotvrzovaného spojení (na rozdíl od `TCP`), takže v případě špatného spojení nedochází k zahlcení a je možné dosáhnout většího datového toku. V dalších fázích vývoje je plánován přechod na `TCP` protokol pro řídící socket, protože objem jeho prostřednictvím přenášených dat je zanedbatelný a potvrzované doručení řídících příkazů by nemuselo být ošetřováno aplikační vrstvou.

5.1.1 Řídící příkazy

Každý řídící příkaz je vždy inicializován klientem – klient pošle 8-bytový datagram na předem definovaný komunikační port *Katherine*, jehož struktura je znázorněna na obr. 5.1a a server (resp. *Katherine*) vždy pošle odpověď ve formě datagramu (viz 5.1b), nebo jiných dat dle specifikace příkazu, na port klienta, ze kterého spojení bylo inicializováno[2]. Response některých příkazů je jen potvrzení (tzv. *Acknowledge*) ve formě ID příkazu a daty $0x00000000$ (dále jen ACQ). Bytové pořadí pro všechny datagramy je *BigEndian*[3].

(a) Request. (b) Response.

Obrázek 5.1: Struktura řídícího datagramu.

Následuje výčet všech důležitých řídících příkazů, s jejich ID, názvem a stručným popisem.

0x01 – Set acq. time (LSB) – nastaví nejnižších 32 bitů akvizičního času v desítkách nanosekund.
Request: `data[31..0]` LSB akvizičního času.
Response: ACQ.

0x02 – Set bias – nastaví *bias* napětí detektoru.
Request: `data[39..32]` ID biasu, `data[31..0]` float[4] hodnota biasu.
Response: ACQ.

0x03 – Acquisition Start – spuštění akvizice dat.
Request: `data[0]` akviziční mód (0 = sekvenční, 1 = data-driven).
Response: ACQ.

0x04 – Internal DAC Settings – nastavení hodnoty DAC (digitálně analogového převodníku) detektoru. Přehled jednotlivých dat je uveden v příloze A.1.

[2]Doplnění této funkcionality bylo vyžádáno až v průběhu implementace komunikačního modulu a během psaní této práce je ještě ve fázi testování. V současné produkční verzi firmware *Katherine* tedy posílá response datagram na staticky konfigurovaný port, což pro operování více *Katherine* o stejné statické konfiguraci z jednoho handleru znamená konflikt portů.

[3]Název pro označení endianity (bytového pořadí), kde na paměťové místo s nejnižší adresou je uložen nejvíce významný byte (MSB).

[4]IEEE 754 standart pro reprezentaci čísel s plovoucí desetinou čárkou.

Request: `data[15..0]` hodnota DAC, `data[16..31]` ID DAC pro čtení.
Response: ACQ.

0x06 – Acquisition Stop – zastaví probíhající akvizici dat (měření).
Request: prázdný.
Response: ACQ.

0x07 – HW Command Start – spustí interní hardwarový příkaz vyčítacího rozhraní. Přehled HW příkazů je uveden v příloze A.2.
Request: `data[7..0]` ID HW příkazu.
Response: ACQ.

0x08 – Sensor Register Setting – nastavení registru připojeného *Timepix3* detektoru ve vyčítacím rozhraní. Přehled registrů je uveden v příloze A.3. Pro propsání nastavení registru do detektoru je třeba ještě vykonat interní HW příkaz s ID **0**.
Request: `data[39..32]` ID registru, `data[31..0]` hodnota registru.
Response: ACQ.

0x09 – Acquisition Mode Setting – nastavení akvizičního módu detektoru.
Request: `data[1..0]` akviziční mód (00 = *ToA & ToT*, 01 = *pouze ToA*, 10 = *Event & iTOT*), `data[7]` povolení *FastToA*[5] (1 = povoleno).
Response: ACQ.

0x0A – Acquisition Time Setting – MSB – nastaví nejvyšších 32 bitů akvizičního času v desítkách nanosekund.
Request: `data[31..0]` MSB akvizičního času.
Response: ACQ.

0x0B – Echo Chip ID – vrátí ID připojeného detektoru.
Request: nic.
Response: `data[31..0]` ID detektoru.

0x0C – Get Bias Voltage – změří a vrátí bias napětí detektoru.
Request: `data[39..32]` ID biasu.
Response: `data[31..0]` float hodnotu biasu detektoru.

0x0F – Internal DAC Scan – naměří a vrátí hodnotu DAC.
Request: `data[7..0]` ID DAC (viz A.1).
Response: `data[31..0]` float hodnota napětí analogového výstupu převodníku.

0x12 – Set All Pixel Config – nastaví konfiguraci jednotlivých pixelů detektoru. V příloze A.4 je příklad převodu z formátu `BMC`[6] do formátu vyžadovaného *Katherine* rozhraním.
Request: nejprve je poslán příkaz z prázdnými daty. Poté *Katherine* očekává 65536 bytů s konfigurací pixelů.
Response: ACQ.

[5] LSB (bit s nejnižší hodnotou) ToA je 25 ns, s aktivovaným FastToA je časové rozlišení zlepšeno na 1.5625 ns

[6] Z angl. *Binary Matrix Configuration*, standardizovaný formát pro konfiguraci pixelů detektorů rodiny Medipix.

0x13 – Number of Frames Setting – nastaví počet snímků, které se naměří, je-li akvizice dat spuštěna v *frame-based* módu (viz 2.3). *Request*: `data[31..0]` počet snímků. *Response*: ACQ.

0x14 – Get All DAC Scan – vrátí sekvenci 22 naměřených hodnot DAC, seřazených dle čtecího ID DAC (viz A.1).
Request: nic.
Response: 22 response datagramů, kde `data[31..0]` je float hodnota napětí analogového výstupu převodníku.

0x15 – Get HW/Readout Temperature – naměří a vrátí teplotu vyčítacího rozhraní.
Request: nic.
Response: `data[31..0]` teplota (v °C, float).

0x17 – Get Readout Status – vrátí informace o své HW a FW konfiguraci.
Request: nic.
Response: `data[7..0]` typ hardware ($0x1$ pro *Katherine* ethernet readout), `data[15..8]` verze HW revize, `data[31..16]` sériové číslo HW, `data[47..32]` verze FW.

0x18 – Get Communication Status – vrátí stav komunikace mezi vyčítacím rozhraním a detektorem.
Request: nic.
Response: `data[7..0]` maska komunikačních kanálů, `data[15..8]` aktuální maximální datový tok mezi vyčítacím rozhraním a detektorem (float, po vynásobení této hodnoty 5 je získána výsledná hodnota v Mb/s), `data[16]` flag připojení detektoru (1 = přípojen).

0x19 – Get Sensor Temperature – naměří a vrátí teplotu senzoru připojeného detektoru.
Request: nic.
Response: `data[31..0]` teplota (v °C, float).

0x20 – Digital Test – provede test digitální části detektoru a vrátí výsledek. Je-li výsledek roven 64, pak test proběhl v pořádku.
Request: nic.
Response: `data[7..0]` výsledek testu.

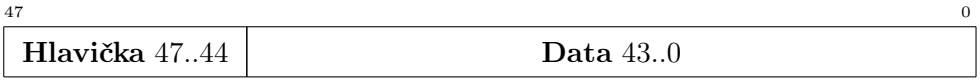
0x28 – ToA Calibration Setup – zapnutí/vypnutí *ToA offset korekce* (viz 2.5.4).
Request: `data[0]` zapnutí *ToA offset korekce* (1 = zapnuto).
Response: ACQ.

5.1.2 Struktura měřených dat

Přenos měřených dat je realizován proudem datagramů ze serveru (vyčítací rozhraní) do klienta (např. handler). Data jsou posílána na předem definovaný port, uložený ve statické konfiguraci *Katherine*[7]. Každý datagram má délku 6 bytů a jeho struktura je znázorněna na

[7] V době psaní této práce probíhal vývoj podpory pro dynamické nastavování klientského portu (přidáním nového příkazu do řídící části komunikačního protokolu).

obr. 5.2. Přehled typů datagramů, vč. jejich významu a obsažených dat, je uveden v tabulce 5.1.



Obrázek 5.2: Struktura datagramu s měřenými daty.

Hlavička	Název	Data
0x7	Nový frame vytvořen	0x0
0x4	Měřená data	Dle konfigurace
0x5	ToA timestamp offset (pouze pro data-driven mód)	ToA offset
0xC	Aktuální snímek vytvořen	Počet poslaných událostí
0x8	Frame start timestamp LSB	32 b LSB
0x9	Frame start timestamp MSB	16 b MSB
0xA	Frame end timestamp LSB	32 b LSB
0xB	Frame end timestamp MSB	16 b MSB
0xD	Počet ztracených pixelů	44 b
0xE	Notifikace o přerušeném měření	0x0

Tabulka 5.1: Přehled typů datagramů pro měřená data.

Z pohledu pořadí toku jednotlivých datagramů rozlišujeme dva případy, dle nastaveného vyčítacího módu:

Frame-based – vyčítání po snímcích (viz 2.4)

1. Klient pošle *Acquisition Start* (0x03) příkaz.
2. Vyčítací rozhraní zahájí akvizici snímku a pošle datagram o vytvoření nového frame (hlavička 0x7).
3. Akvizice dat probíhá po dobu zvoleného akvizičního času. Jsou-li detekovány nějaké události, pak pro každou událost je poslán datagram s měřenými daty (hlavička 0x4). V případě zaznamenání příkazu pro přerušení měření, dojde k poslání datagramu s hlavičkou 0xE a měření je ukončeno.
4. Po ukončení akvizice aktuálního snímku (uzavření *shutteru*) readout pošle timestamp začátku a konce akvizice dat (datagramy s hlavičkami 0x8, 0x9, 0xA a 0xB).
5. Je poslán datagram s informací o počtu ztracených pixelů (hlavička 0xD) a ukončení aktuálního snímku (hlavička 0xC).
6. Je-li počet aktuálně pořízených snímků nižší, než celkový počet snímků (nastavený řídícím příkazem 0x13), pak měření pokračuje bodem 2, v opačném případě měření končí.

Data-driven – režim kontinuální vyčítání dat (viz 2.4)

1. Klient pošle *Acquisition Start* (0x03) příkaz.
2. Vyčítací rozhraní zahájí akvizici dat (resp. jednoho snímku) a pošle datagram o vytvoření nového frame (hlavička 0x7).
3. Akvizice dat probíhá po dobu zvoleného akvizičního času. Jsou-li detekovány nějaké události, pak pro každou událost je poslán datagram s měřenými daty (hlavička 0x4). V případě zaznamenání příkazu pro přerušení měření, dojde k poslání datagramu s hlavičkou 0xE a měření je ukončeno.
4. Po ukončení akvizice aktuálního snímku (uzavření *shutteru*) readout pošle timestamp začátku a konce akvizice dat (datagramy s hlavičkami 0x8, 0x9, 0xA a 0xB).
5. Je poslán datagram s informací o počtu ztracených pixelů (hlavička 0xD) a ukončení aktuálního snímku (hlavička 0xC).
6. Měření je ukončeno.

Tabulka 5.2 znázorňuje formát datagramu s měřenými daty, dle použitého nastavení vyčítacího rozhraní.

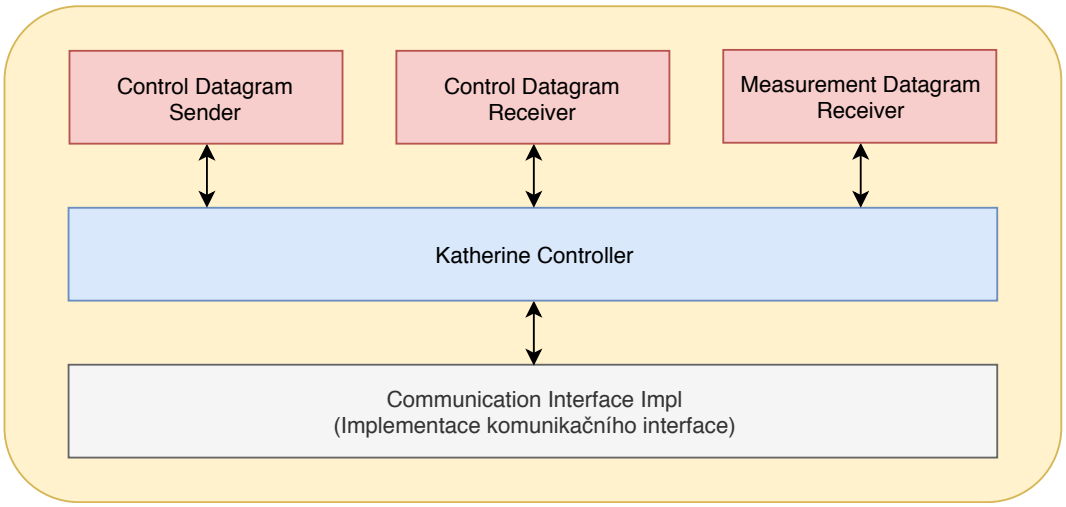
Fast VCO	Mode	Measure Data Frame					
		47..44	*43..36*	*35..28*	*27..14*	*13..4*	*3..0*
Enabled	ToA & ToT	0x4	Coordinate Y	Coordinate X	ToA	ToT	FastToA
	Only ToA	0x4	Coordinate Y	Coordinate X	ToA	---	FastToA
	Event Count and iToT	0x4	Coordinate Y	Coordinate X	iToT	Event Counter	---
Disabled	ToA & ToT	0x4	Coordinate Y	Coordinate X	ToA	ToT	Hit Counter
	Only ToA	0x4	Coordinate Y	Coordinate X	ToA	---	Hit Counter
	Event Count and iToT	0x4	Coordinate Y	Coordinate X	iToT	Event Counter	Hit Counter

Tabulka 5.2: Struktura měřených dat [7].

5.2 Komunikační modul

Pro účely řízení komunikačního rozhraní *Katherine* a vyčítání měřených dat byl implementován komunikační modul, který bude popsán v této podkapitole. Aby bylo možné integrovat modul do systému, musí obsahovat implementaci komunikačního interface (viz 4.1.1.2), které musí být v manifestu zkompilovaného `jar` archívu explicitně uvedeno (viz 4.1.1.1).

Na obrázku 5.3 je přehled vrstev softwarové architektury komunikačního modulu, které budou dále popsány v následujících podkapitolách.



Obrázek 5.3: Softwarová architektura komunikačního modulu, implementujícího komunikační interface.

5.2.1 Implementace komunikačního interface

Tato vrstva slouží jako tzv. fasáda[8] Katherine kontroléru. Implementace komunikačního interface implementuje všechny jeho metody, vč. poskytování přehledu a podpory vykonávání *ValueCommands* a *ExecutionCommands* (pro jejich definici viz soubor `detector_model.json` na přiloženém CD, viz příloha C), navazování spojení s detektorem, nahrávání konfigurace apod.

Po zavedení modulu je vytvořena fronta měřených dat, která je přejatá handlerem.

Připojením detektoru vzniká instance Katherine kontroléru dle předané konfigurace. Při odpojení detektoru je kontrolér notifikován, aby mohl uvolnit svoje prostředky (např. uzavřít síťové spojení) a poté je odstraněn.

5.2.2 Katherine kontrolér

Tato vrstva udržuje spojení s detektorem a implementuje všechny potřebné řídící příkazy detektoru.

Odesílání řídících příkazů je realizováno pomocí komponenty `Control Datagram Sender` (viz obr. 5.3), který implementuje asynchronní frontu datagramů. Fronta je asynchronně zpracovávána nezávislým vláknem, které datagramy z fronty odesílá na řídící port *Katherine*.

O příjem dat z řídícího socketu se stará komponenta `Control Datagram Receiver`, která v separátním vlákně přijímá příchozí data. Každý příchozí datagram (8-bytový vektor) je zpracován a přidán do fronty příchozích řídících dat. Komponenta také implementuje blokující metodu pro přijetí dat, která dle zadaného ID příkazu a maximální doby čekání (tzv. *timeout*) hledá požadovanou odpověď ve frontě dat.

[8] Návrhový vzor fasáda (*Facade Pattern*) – použití pro poskytování zjednodušeného interface pro komplexní části kódu (analogie s pojmem fasáda, známém z architektury).

Příklad použití obou komponent zmíněných výše je ve zdrojovém kódu 5.1. Příklad demonstruje vyčtení biasu z detektoru. Nejprve je pomocí `Control Datagram Sender` odeslán řídící příkaz s ID 0x0C (řádek 3 až 7) a následně pomocí `Control Datagram Receiver`, resp. pomocí jeho blokující metody popsané výše, je zachycena odpověď, ze které je vyčtena požadovaná hodnota napětí. Jestli se odpověď nepodaří přijmout ve zvoleném timeoutu, je vygenerována výjimka, která musí být vyššími vrstvami programu ošetřena.

```
1  @Throws(DetectorException::class)
2  fun getBias(): Float {
3      packetSender.send(
4          PacketBuilder.builder()
5              .commandID(KatherineValueCommands.BIAS.getterCommandId)
6              .build()
7      )
8
9      val packet = packetReceiver.pollPacketFromQueue(
10         KatherineValueCommands.BIAS.getterCommandId,
11         RESPONSE_TIMEOUT
12     )
13     return packet.floatValue
14 }
```

Zdrojový kód 5.1: Příklad implementace řídícího příkazu *Katherine* pro vyčtení biasu pomocí komponent `Control Datagram Sender` a `Control Datagram Receiver`.

Poslední komponentou je `Measurement Datagram Receiver`, který ve svém separátním vlákně přijímá příchozí datagramy na socketu pro detektorem měřená data, jejichž struktura už byla popsána v 5.1.2. Každá příchozí událost je zpracována (identifikace hlavičky apod.) a vložena do fronty měřených dat, která je zpracovávána datovým modulem.

5.3 Datový modul

Datový modul implementuje `DataPersistence` interface (viz 4.1.1.3). Po jeho zavedení do systému je mu handlerem předána fronta dat, kterou po spuštění začne zpracovávat.

Fronta dat obsahuje objekty, které jsou potomky objektu `AbstractDataFrame` (který je součástí modelu poskytovaných knihoven). V rámci modulu se společnou *code-base Katherine* modulů (viz úvod kapitoly 5) byly implementovány tyto třídy, které jsou potomky třídy `AbstractDataFrame`:

`DataFrame` – třída obsahující jednu událost, přenesenou datovým socketem z *Katherine* rozhraní. Objekt obsahuje dekódovanou hlavičku a vlastní data, dle typu datagramu – viz tabulka 5.1.

`ControlFrame` – obsahuje informaci generovanou komunikačním modulem, např. o začátku nebo ukončení akvizice apod.

AcqConfigurationFrame – instance tohoto objektu je poslána komunikačním modulem na začátku akvizice dat a obsahuje informace o stavu a konfiguraci detektoru (např. bias, vyčítací mód, teploty apod.).

Výstupem datového modulu jsou soubory s jednotlivými měřeními ve formátu `ASCII`, které jsou ukládány do zvoleného adresáře dle konfigurace. Příklad obsahu takového souboru je uveden v příloze A.5. Všechny časové údaje jsou v `UTC` čase. Formát pojmenovávání souborů je `název_detektoru_yyyy_MM_dd-HH_mm_ss.txt`.

Rovněž byla implementována varianta datového modulu, který ukládá pořízená data do *MongoDB* databáze (viz 3.5).

5.4 Katherine emulátor

Pro účely vývoje a testování byl vyvinut emulátor, které emuluje funkci *Katherine* vyčítacího rozhraním s připojeným *Timepix3* (viz 2.6) detektorem na síťové vrstvě. Byla implementována podpora pro všechny řídící příkazy uvedené v 5.1.1, které jsou relevantní k akvizici dat (tj. např. akviziční čas, akviziční mód, vyčítací mód, bias, počet snímků apod.) a také příkazy pro měření teplot, vyčítání ID senzoru, či provádění testu digitální částí detektoru. Emulátor rovněž podporuje všechny datagramy pro přenos měřených dat (viz 5.1.2).

Emulátor byl vyvinut v jazyce `Java`, resp. `Kotlin` (viz 3.5).

5.4.1 Softwarová architektura

Na obrázku 5.4 jsou znázorněny jednotlivé komponenty emulátoru. Základním prvkem je jeho jádro (na obr. 5.4 jako *emulator core*), které je vytvořeno po spuštění emulátoru. Společně s jádrem je vytvořen také *Control Datagram Receiver*, který přijímá příchozí datagramy na komunikačním socketu detektoru. Příchozí řídící datagram je dále zpracován dle jeho hlavičky a klientovi je odeslána příslušná odpověď.

Z pohledu zpracování je možné příkazy rozdělit do tří kategorií:

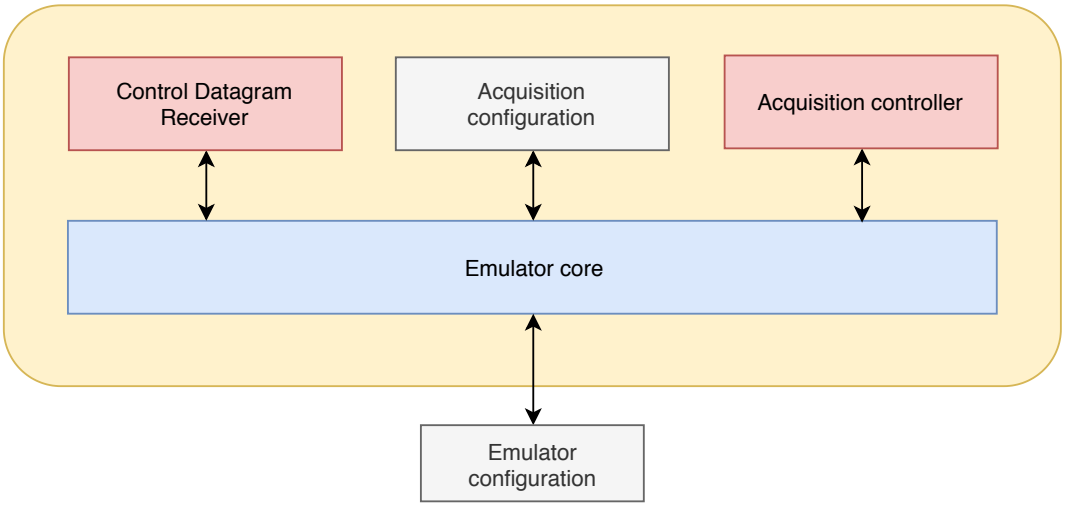
Nastavení akvizičních parametrů – do této skupiny spadají všechny příkazy, které nastavují, nebo vracejí konfigurační parametry detektoru, jako je akviziční čas, bias, akviziční a vyčítací mód apod.

Řízení akvizice – do této kategorie patří příkazy *Acquisition Start* (0x03) a *Acquisition Stop* (0x06). Po spuštění akvizice je vytvořena instance `Acquisition Controller`, který dle konfigurace začne generovat události a posílat je datovým socketem klientovi.

Příkaz *Acquisition Stop* pak přeruší probíhající akvizici.

Ostatní – do této kategorie spadají ostatní příkazy, které nebyly zahrnuty v předchozích kategoriích (např. pro měření teploty, čtení *chip id*, verze FW apod.).

Některé příkazy nebyly implementovány, protože emulace jejich funkce není triviální a je nad rámec požadované funkcionality (jsou to např. příkazy pro nastavování hodnot DAC, registrů apod.). Odpovědí na tyto příkazy bude vždy ACQ (viz 5.1.1).



Obrázek 5.4: Softwarová architektura *Katherine* emulátoru.

5.4.2 Konfigurace a nasazení

Pro spuštění emulátoru vyčítacího rozhraní *Katherine* je třeba mít nainstalované `JRE 8`[9], nebo vyšší. Zkompilovanou Java aplikaci je třeba spustit s argumentem cesty ke konfiguračnímu souboru (pro příklad viz zdrojový kód 5.2). Konfigurační soubor obsahuje číslo portu, na které bude emulátor naslouchat pro řídící příkazy (`portCommands`), číslo portu a adresu pro odesílání měřených dat (*portCommands* a `hots`) a parametr udávající průměrný počet událostí, které budou v rámci jedné akvizice vygenerovány.

Aplikaci je tedy možné spustit například takto:

```
java -jar emulator.jar config.yaml
```

```
1 portCommands: 1555
2 portData: 1556
3 host: 127.0.0.1
4 avgNumberOfEventsPerAcq: 10
```

Zdrojový kód 5.2: `YAML` konfigurační soubor emulátoru vyčítacího rozhraní *Katherine*.

[9] *Java SE Runtime Environment, dostupné z* <https://www.oracle.com/technetwork/java/javase/downloads>.

Kapitola 6

Master

V této kapitole bude čtenář podrobněji seznámen s masterem – centrálním řídícím prvkem systému Pixnet. Hlavním úkolem masteru je řízení handlerů (viz 4), persistence konfigurace a poskytování REST API pro své řízení. Primárním konzumentem API je webová frontend aplikace, poskytující uživatelské rozhraní pro řízení systémů jeho operátorem, pomocí poskytovaného API může být ale řízení systému integrováno i do systémů třetích stran[1].

Tato komponenta se skládá ze dvou nezávislých částí – backendového serveru (6.1) frontendového webového klienta (6.2).

6.1 Backend

Backendová aplikace mastera, podobně jako handlera, byla implementována pomocí Spring Boot frameworku (viz 3.5) a obsahuje embedovaný webový server `Tomcat`. Aplikace komunikuje s handlery, jejichž prostřednictvím řídí jim přiřazené detektory.

6.1.1 Softwarová architektura

Na obrázku 6.1 je znázorněna softwarová architektura backendové aplikace mastera, rozdělená do několika vrstev, které budou v následujících podkapitolách blíže vysvětleny.

6.1.1.1 Vrstva pro persistenci konfigurace a stavu systému

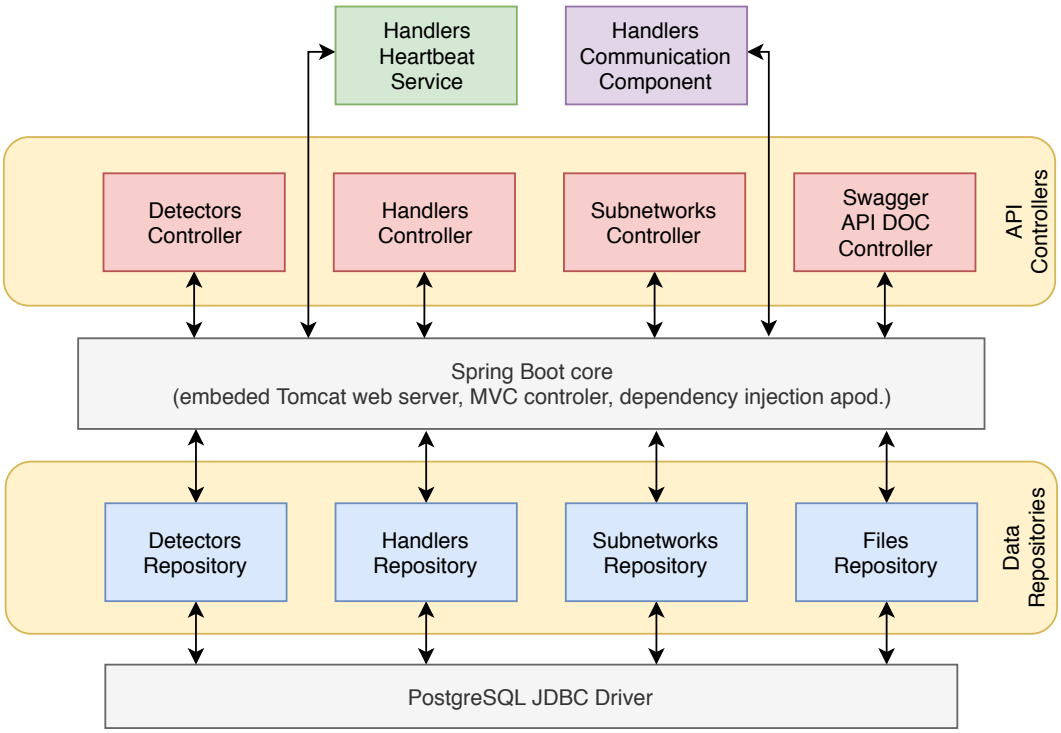
Tato vrstva je zodpovědná za komunikaci s *PostgreSQL* databází (viz 3.5) pomocí *PostgreSQL* `JDBC`[2] ovladače. Pro mapování jednotlivých entit (resp. Java objektů, tzv. `ORM`[3]) byl implementován framework *Hibernate*.

Vrstva poskytuje repozitáře (viz obr. 6.1) pro dotazy nad uloženými daty a pro jejich modifikaci. Rovněž je zodpovědná za automatické vytváření a aktualizaci databázového schématu, připojené databáze. Návrh schématu bude popsán v 6.1.2.

[1] Např. DCS v CERN, jak již bylo popsáno v kapitole 3.3

[2] Z angl. *Java Database Connectivity* – API pro programovací jazyk Java, definující přístup klienta k databázi.

[3] Z angl. *Object-relational mapping* – objektově-relační mapování.



Obrázek 6.1: Pixnet – master: softwarová architektura s vrstvami pro (i) persistenci konfiguraci s stavu systému (*Data Repositories*), (ii) poskytování API (*API controllers*),(iii) komponentu pro komunikaci s handlery (*Handlers Communication Component*) a (iv) službu pro pravidelnou aktualizaci stavu handlerů (*Handler Heartbeat Service*).

V databázi jsou společně se záznamem detektoru uloženy také jeho moduly (resp. implementace komunikačního a datového interface).

6.1.1.2 Komponenta pro komunikaci s handlery

Komponenta poskytuje metody pro komunikaci s handlery pomocí jejich API, popsaném v 4.1. Umožňuje přiřazování detektorů jednotlivým handlerům, navazování spojení s detektory, zjišťování jejich stavu a funkcionality (vč. seznamů podporovaných příkazů), vykonávání jednotlivých příkazů, nahrávání souborů apod.

6.1.1.3 Vrstva s API kontroléry

Tato vrstva se sestává z REST API kontrolérů pro poskytování API metod pro řízení mastera. Těmito kontroléry jsou:

`Subnetworks Controller` – poskytuje CRUD[4] operace nad entitou podsítí (viz 3.3). Pro přehled endpointů viz tabulka 6.1.

[4] Z angl. *Create Read Update and Delete*, operace pro vytváření, čtení, aktualizaci a odstranění.

HTTP Metoda	Endpoint	Popis
POST	`/add`	Přidá novou podsíť
GET	`/getAll`	Vrátí seznam všech podsítí
DELETE	`/deleteAll`	Odstraní všechny podsítě
DELETE	`/deleteById`	Odstraní podsíť podle zadaného ID

Tabulka 6.1: Endpointy komponenty *Subnetworks Controller* (všechny endpointy mají prefix `/api/subnetwork`).

`Handlers Controller` – poskytuje CRUD operace nad entitou handlerů. Pro přehled endpointů viz tabulka 6.2.

HTTP Metoda	Endpoint	Popis
POST	`/add`	Přidá nový handler
GET	`/getAll`	Vrátí seznam všech handlerů
DELETE	`/deleteAll`	Odstraní všechny handlery
DELETE	(pouze prefix)	Odstraní handler podle zadaného ID

Tabulka 6.2: Endpointy komponenty *Handlers Controller* (všechny endpointy mají prefix `/api/handler`).

`Detectors Controller` – kromě poskytování CRUD operací nad entitou detektorů také poskytuje metody pro přiřazování detektorů handlerům, navazování spojení s detektory, vykonávání jejich příkazů (viz *ValueCommands* a *ExecutionCommands*, 4.1.1.2). Pro přehled endpointů viz tabulka 6.3.

`Swagger API DOC Controller` – kontrolér pro poskytování webové API dokumentace, dostupné z endpointu `/apiDoc`. Tato komponenta byla implementována, podobně jako u handleru (viz 4.1.3.3), za pomocí *Springfox* [18] knihovny. Kromě přehledu všech endpointů webové rozhraní také poskytuje definici datového modelu API (ve formátu JSON) a umožňuje manuální volání jednotlivých endpointů. Endpoint pro stažení OpenAPI specifikace ve formátu JSON je `/v2/api-docs`.

6.1.1.4 Handlers Heartbeat Service

Tato služba slouží k periodickému zjišťování stavu jednotlivých handlerů (a tranzitivně i detektorů) systému. V pravidelných intervalech (defaultně nastaveno na 10 s) je navázáno spojení se všemi handlery pomocí jejich `/status` endpointu (viz 4.1.3.2) a jsou aktualizována stavová data v databázi.

HTTP Metoda	Endpoint	Popis
POST	`/add`	Přidá nový detektor
GET	`/getAll`	Vrátí seznam všech detektorů
GET	`/getById`	Vrátí jeden detektor
GET	`/getDetailById`	Vrátí detail jednoho detektoru
DELETE	(pouze prefix)	Odstraní detektor podle zadaného ID
POST	`/bindToHandler`	Přiřadí detektor handleru
POST	`/unbindFromHandler`	Odebere detektor handleru
POST	`/connect`	Naváže spojení mezi handlerem a detektorem
POST	`/disconnect`	Ukončí spojení mezi handlerem a detektorem
POST	`/executeValueCommand`	Vykoná `ValueCommand` detektoru
POST	`/executeExecutionCommand`	Vykoná `ExecutionCommand` detektoru
POST	`/uploadFile`	Nahraje soubor do detektoru

Tabulka 6.3: Endpointy komponenty *Detectors Controller* (všechny endpointy mají prefix `/api/detector`).

6.1.2 Návrh databáze

Na obrázku 6.2 je znázorněno schéma relačního modelu *PostgreSQL* databáze, kde jsou znázorněny jednotlivé entity, vč. jejich atributů, a relace mezi nimi pomocí cizích klíčů.

Ve schématu jsou dále definována různá integritní omezení, která vynucují strukturu dat a pravidla mezi relacemi entit – např. že handler musí mít přiřazenu právě jednu podsíť, že detektor může být přiřazen handleru ze stejné podsítě apod.

6.1.3 Konfigurace a nasazení

Pro spuštění mastera je třeba mít nainstalované `JRE 8`[5] nebo vyšší. Zkompilovanou Java aplikaci je třeba spustit s argumentem `masterConfig` obsahující cestu ke konfiguračnímu souboru (pro příklad viz zdrojový kód 6.1), obsahujícího port na kterém bude aplikace naslouchat příchozí spojení.

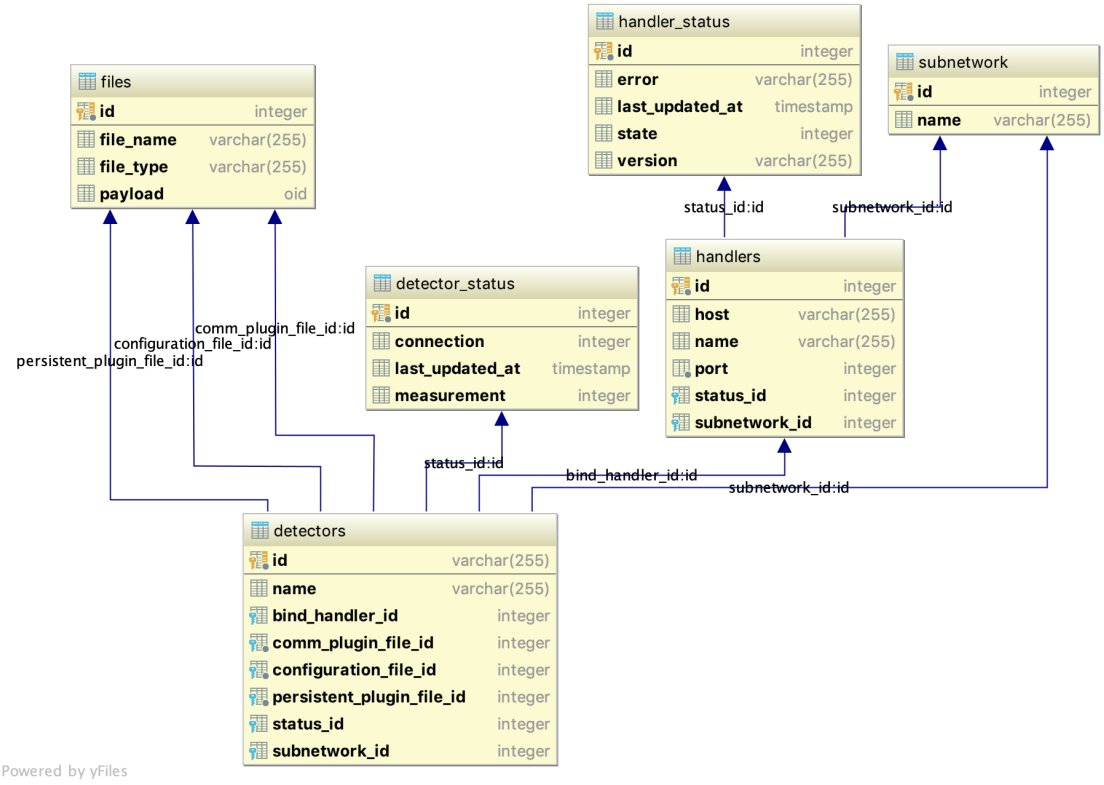
Aplikaci je tedy možné spustit například takto:

```
java -jar master.jar masterConfig=config.yaml
```

```
1 # Connection parameters
2 portToListen: 8080
```

Zdrojový kód 6.1: `YAML` konfigurační soubor mastera.

[5] *Java SE Runtime Environment, dostupné z* <https://www.oracle.com/technetwork/java/javase/downloads>.



Obrázek 6.2: Pixnet – master: databázové schéma.

6.2 Frontend

Pomocí frameworku *ReactJS* (viz 3.5) byl implementován webový klient, který implementuje API, poskytované backendovou aplikací mastera (viz 6.1.1.3).

Jedná se o tzt. *single-page* aplikaci, tj. aplikaci která načte jednu stránku, kterou pak dynamicky aktualizuje dle interakcí uživatele (ev. dalších událostí) bez nutnost stahování nové stránky ze serveru. Jak již bylo vysvětleno v kapitole 3.5, každá *ReactJS* aplikace je založena na komponentách. Každá komponenta má svoje vlastnosti a svůj stav. Komponenty lze opětovně používat a je možné je libovolně zapouzdřovat. Při aktualizaci stavu některé z komponent, *ReactJS* framework změnu detekuje a provede překreslení afektovaných částí uživatelského rozhraní.

Pro navigaci po stránce aplikace používá knihovnu *react-router*[6], která mění obsah stránky renderováním těchto komponent:

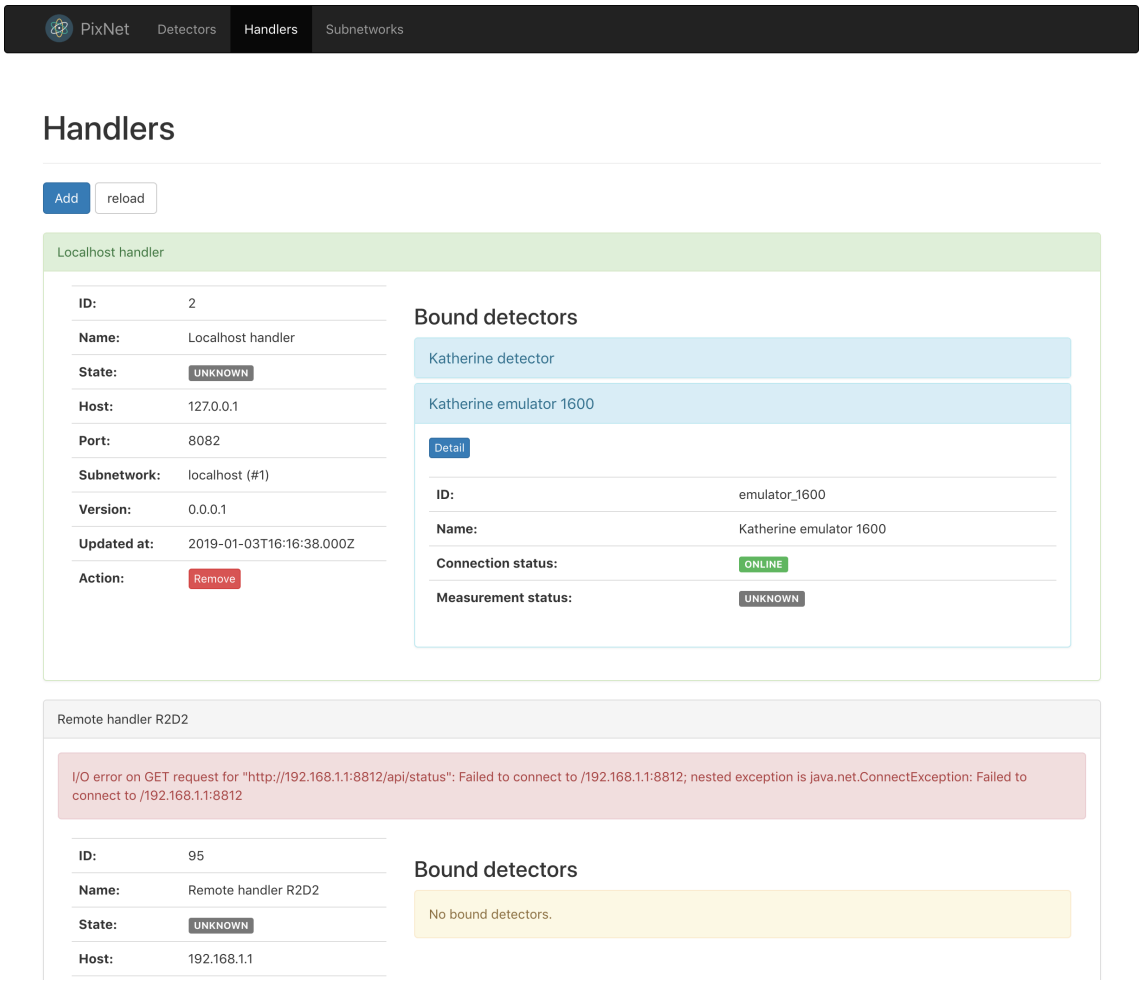
Subnetworks – komponenta vizualizující seznam podsítí a umožňuje přidávání nových podsítí a odebírání existujících.

[6] <https://reacttraining.com/react-router/web>

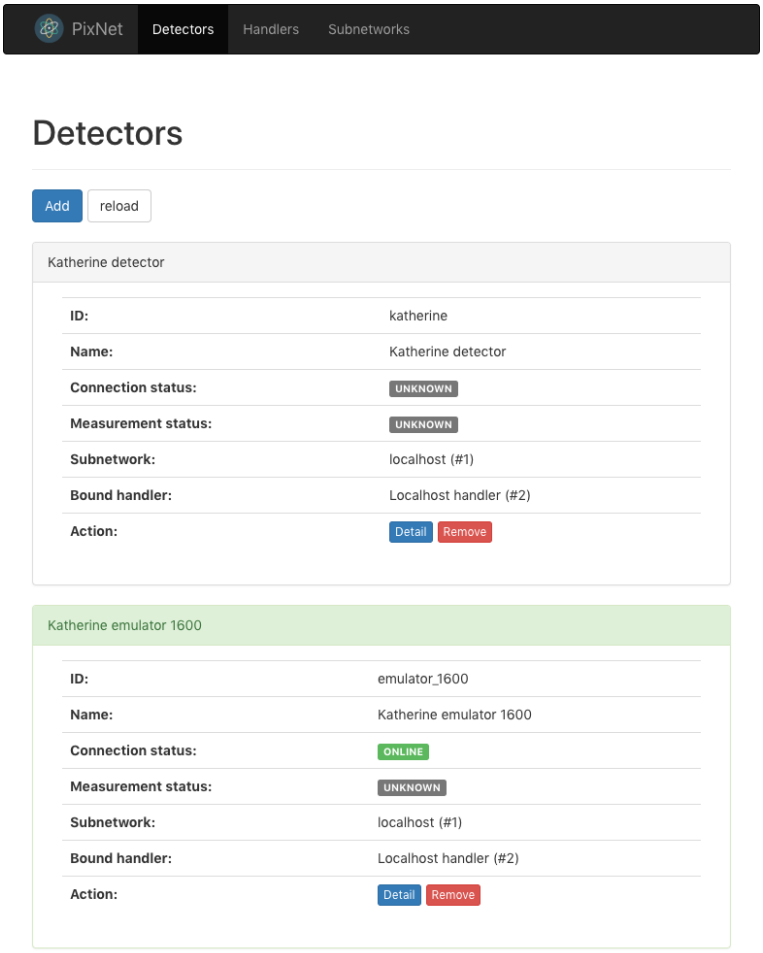
Handlers – tato komponenta zobrazuje seznam přidaných handlerů, včetně podseznamu jim přiřazených detektorů, stavu apod. Komponenta rovněž umožňuje přidání nových hadlerů a odstranění existujících. Pro screenshot viz obr. 6.3.

Detectors – komponenta zobrazující seznam všech detektorů (viz 6.4). Dále umožňuje přidávání nových detektorů a odebírání existujících.

Detector detail – komponenta s detailem detektoru (viz obr. 6.5). Komponenta uživateli nabízí všechny operace nad detektorem, nabízené API backendové aplikace mastera (tj. např. přiřazování detektoru dostupnému handleru, navazování spojení s detektorem, vykonávání jeho příkazů apod.).



Obrázek 6.3: Master – frontend: screenshot seznamu handlerů, dostupného z endpointu `/handlers`.



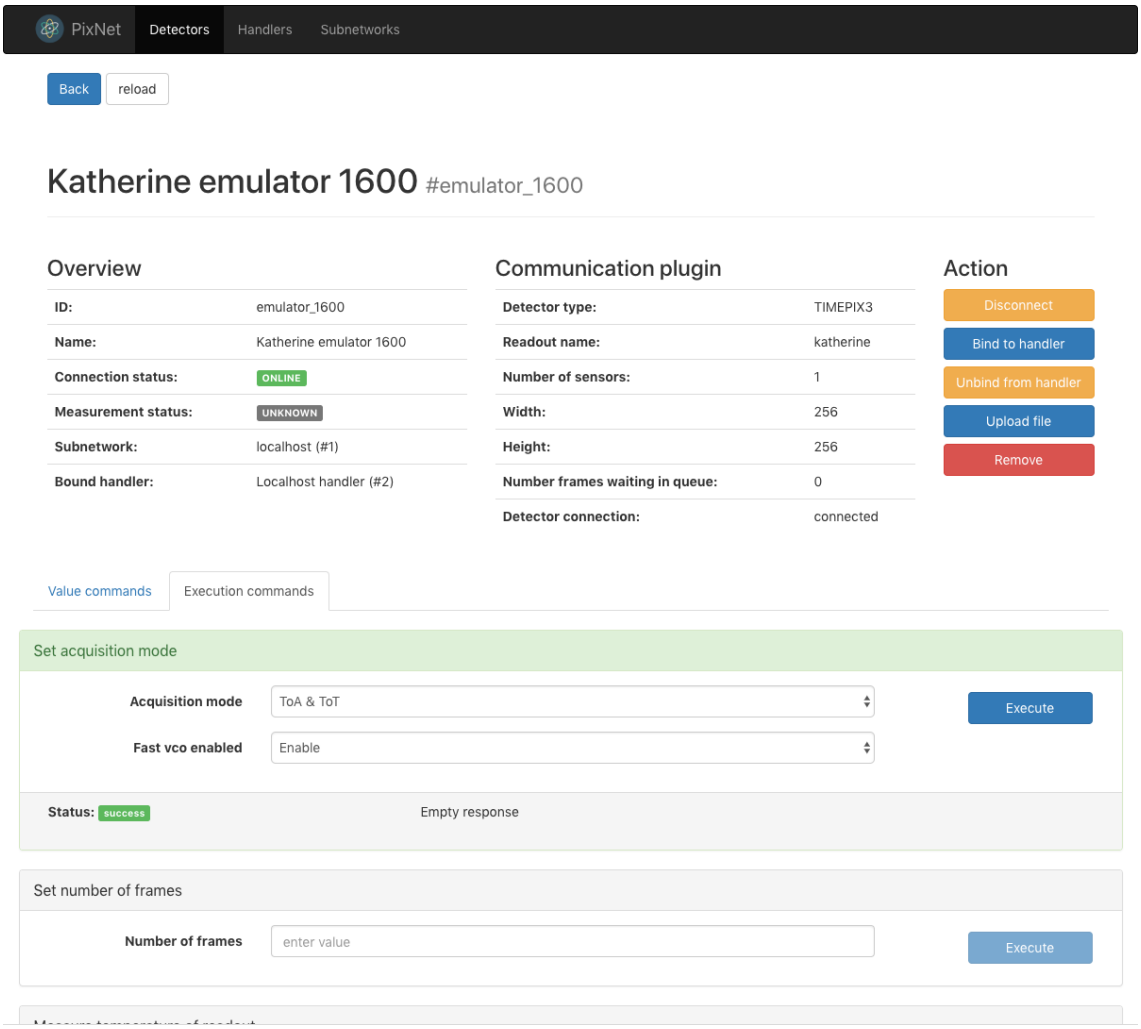
Obrázek 6.4: Master – frontend: screenshot seznamu detektorů, dostupného z endpointu `/detectors`.

6.2.1 Nasazení

Nasazení aplikace je jednoduché – stačí jenom její build (k dispozici na přiloženém CD) nahrát na webový server. Build obsahuje `index.html` společně s vlastní aplikací v `bundle.js` a dalšími soubory (obrázky, CSS styly apod.).

Aplikace byla vyvinuta pomocí systému na správu závislostí `Yarn`[7]. Pro vývojové účely je možné aplikaci spustit příkazem `yarn start`, nebo příkazem `yarn build` je možné udělat build aplikace, nasaditelný na webový server.

[7] <https://yarnpkg.com>



Obrázek 6.5: Master – frontend: screenshot detailu detektoru, dostupného z endpointu `detectors/{ID detektoru}`.

Kapitola 7

Testování

V této kapitole bude čtenář seznámen s metodami testování pro zajištění kvality jednotlivých softwarových komponent a s provedenými experimenty.

7.1 Metody testování

7.1.1 Jednotkové testy

Pro části jednotlivých softwarových komponent byly použity jednotkové testy [10] (tzv. *Unit testy*). Jednotkové testy jsou základním typem testu, který ověřuje funkčnost samostatně testovatelných částí systému - tzv. jednotek. Výhodou těchto testů je jejich automatizovatelnost. Byl použit *JUnit* framework - framework pro jednotkové testy pro platformu Java.

7.1.2 Integrační testy

Integrační testy byly použity pro ověření správné komunikace jednotlivých komponent systému, zejména pak handlera (viz 4) a backendové aplikace mastera (viz 6.1), vyčítacího rozhraní *Katherine* a komunikačního modulu apod.

V rámci těchto testů byla ověřována především správnost implementace komunikačního protokolu vyčítacího rozhraní *Katherine* (resp. její serverové implementace v emulátoru i implementace klienta v komunikačním modulu) a REST API handlera a mastera.

7.1.3 Systémové testy

Systémové testy se provádějí pro ověření funkčnosti systému jako celku. Tyto testy se provádějí až v pozdějších fázích vývoje a testují funkčnost systému z pohledu uživatele. Za tímto účelem bylo definováno několik testovacích scénářů (jako např. přidání detektoru do systému, přiřazení detektoru handleru, spuštění akvizice apod.), pomocí kterých byly tyto testy realizovány.

7.2 Experimenty

Pro ověření funkčnosti a dostatečné propustnosti systému bylo provedeno několik experimentů, které budou v této podkapitole popsány. Pro tyto experimenty byla použita dvě různá prostředí:

lokální – jeden počítač, na kterém byl spuštěn emulátor, handler a master současně. Použitý počítač má CPU 3 GHz Intel Core i7, 16 GB 1600 MHz DDR3 RAM a SSD disk o kapacitě 500 GB (MacBook Pro).

CERN openstack – cloudová infrastruktura v CERN, kterou CERN poskytuje partnerským univerzitám a dalším institucím. Tato infrastruktura je založená na *OpenStack IaaS*[1] platformě, která umožňuje řízení výpočetního výkonu, úložiště a síťových prostředcích v datacentrech skrze webovou aplikaci nebo *OpenStack API*.

Byl vytvořen cluster s těmito virtuálními počítači:

- `pixnet-master.cern.ch` – na tomto uzlu byla nasazena backendová aplikace mastera společně s webovým serverem, poskytujícím webový frontend mastera. Instance má 2 VCPU, 4 GB RAM a 20 GB diskové kapacity.
- `pixnet-handler-1.cern.ch` – uzel pro nasazení backendové aplikace handlera. Instance má 4 VCPU, 8 GB RAM a 40 GB diskové kapacity.
- `pixnet-emulator-katherine1.cern.ch` – uzel pro nasazení emulátoru vyčítacího rozhraní *Katherine*. Instance má 1 VCPU, 2 GB RAM a 10 GB diskové kapacity.
- `pixnet-emulator-katherine2.cern.ch` – uzel pro nasazení emulátoru vyčítacího rozhraní *Katherine*. Instance má 1 VCPU, 2 GB RAM a 10 GB diskové kapacity.

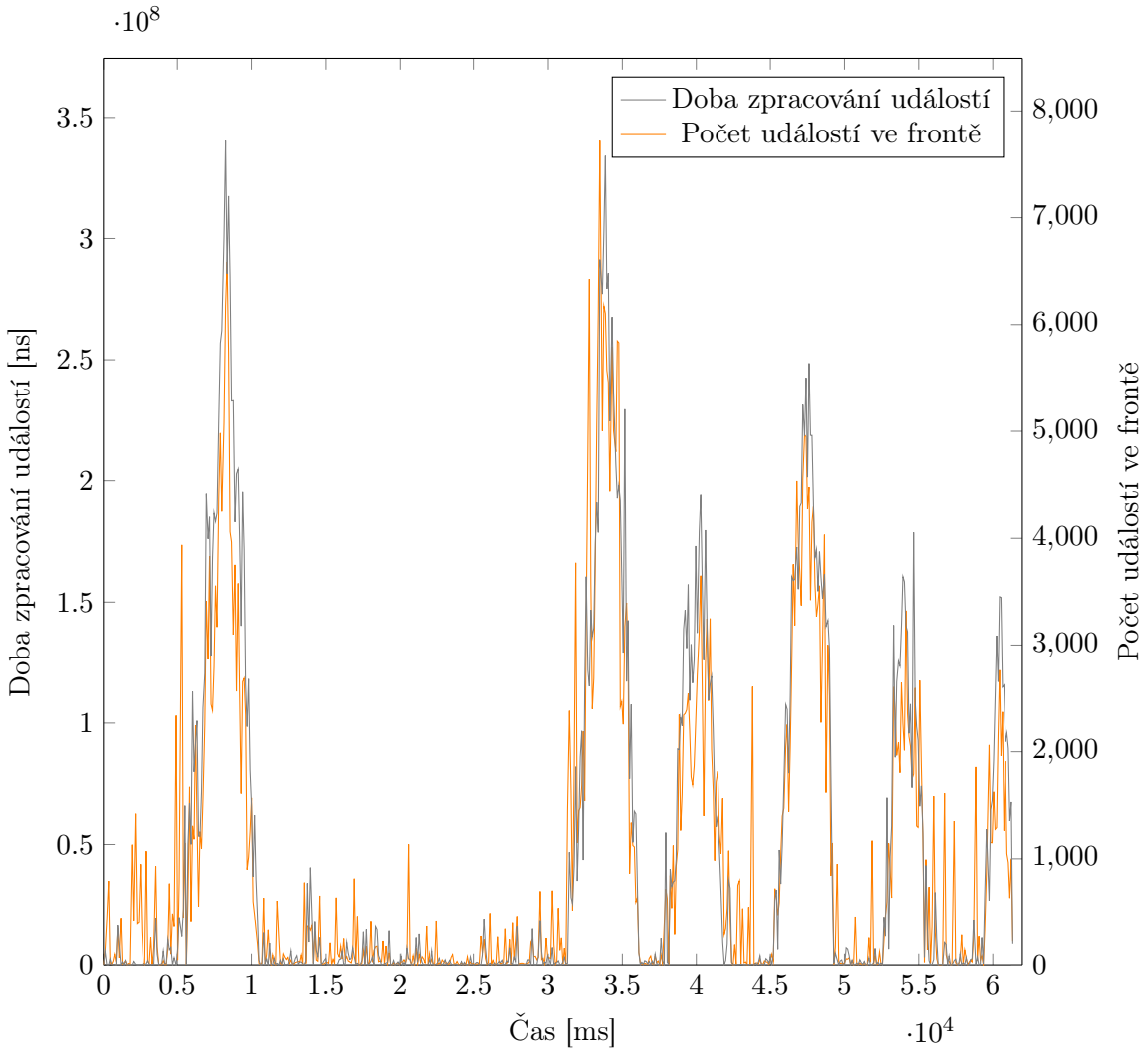
7.2.1 Experiment první: zpracovávání fronty naměřených dat

Kritickou částí každého datového akvizičního systému je proces zpracování dat, od přijetí až po jejich uložení. Jelikož v rámci systému Pixnet data procházejí pouze přes handler (resp. od komunikačního do datového modulu daného detektoru, zavedených v handleru) do datovým modulem definovaného úložiště, budou v rámci tohoto experimentu zkoumány jen metadata jednotlivých událostí (od jejich vzniku až po uložení).

Experiment byl proveden v obou prostředích, v každém s jedním emulátorem vyčítacího rozhraní *Katherine*, jedním handlerem a masterem. Byly sledovány dva parametry - čas zpracování události (tj. doba od jeho přijetí komunikačním modulem až po jeho uložení datovým modulem) a aktuální počet událostí ve frontě. Pro všechna měření byl emulátor nastaven tak, aby generoval fixní počet událostí za sekundu. Jako vyčítací mód emulátoru byl použit *Data-Driven* (viz 2.4), akviziční mód byl *ToA & TOT* a délka akvizice byla vždy 60 s. Na obr. 7.1 je znázorněn průběh jednoho měření, při kterém emulátor generoval 50000 událostí za vteřinu, který zaznamenává vývoj doby zpracovávání událostí a velikosti fronty v čase.

Měření byla provedena pro různé počty generovaných událostí za vteřinu (1, 10, 100, 1000, 10000, 25000, 50000, 75000 a 100000), ze kterých byla stanovena průměrná doba zpracovávání

[1] Z angl. *Infrastructure as a Service*.



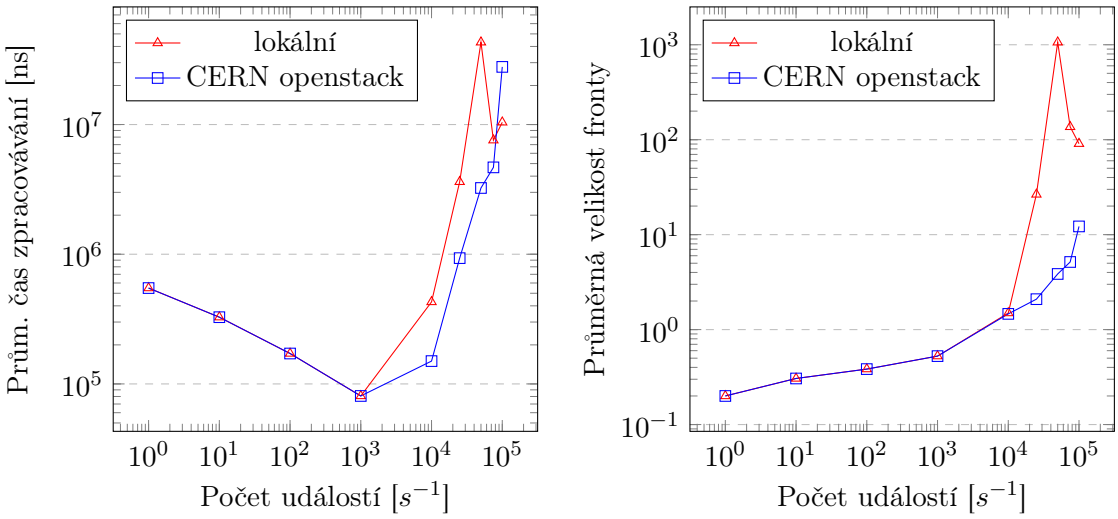
Obrázek 7.1: Vývoj doby zpracovávání událostí a počtu událostí ve frontě v rámci 60 s dlouhé akvizice, při konstantním generovaném množství událostí (50000 s^{-1}).

události v závislosti na počtu příchozích dat (viz obr. 7.2a) a průměrná velikost fronty v závislosti na počtu příchozích dat (viz obr. 7.2b). Jednotlivá měření byla vícekrát zopakována a jejich dílčí hodnoty byly zprůměrovány. Maximální testovaný počet generovaných událostí za vteřinu byl 100000, kvůli omezení emulátoru.

Z výsledku experimentu je patrné, že obě sledované hodnoty rostou s počtem událostí. Oproti očekávaným hodnotám (resp. očekávané rostoucí závislosti) lze z obr. 7.2 pozorovat dva rozdíly.

Tím prvním je vývoj průměrné doby zpracování události od jedné do 1000 událostí za sekundu. Tento efekt je pravděpodobně způsoben nastavenou velikostí *bufferu* a režie přenosu dat.

Druhým rozdílem je různý trend hodnot pro *CERN openstack* a *lokání* prostředí od 50000 událostí za vteřinu. Nejvíce pravděpodobnou příčinou tohoto jevu je rozdílná konfigurace



(a) Závislost průměrné doby zpracování události na počtu událostí za sekundu.

(b) Závislost průměrné velikosti fronty s událostmi na počtu událostí za sekundu.

Obrázek 7.2: Závislost průměrné doby zpracování událostí a jejich průměrného počtu ve frontě na počtu událostí za sekundu.

obou prostředí. Zatímco v rámci *CERN openstack* každá komponenta systémů je spuštěna na jiném počítači, v *lokální* prostředí všechny komponenty sdílí stejný počítač a vzájemně se ovlivňují.

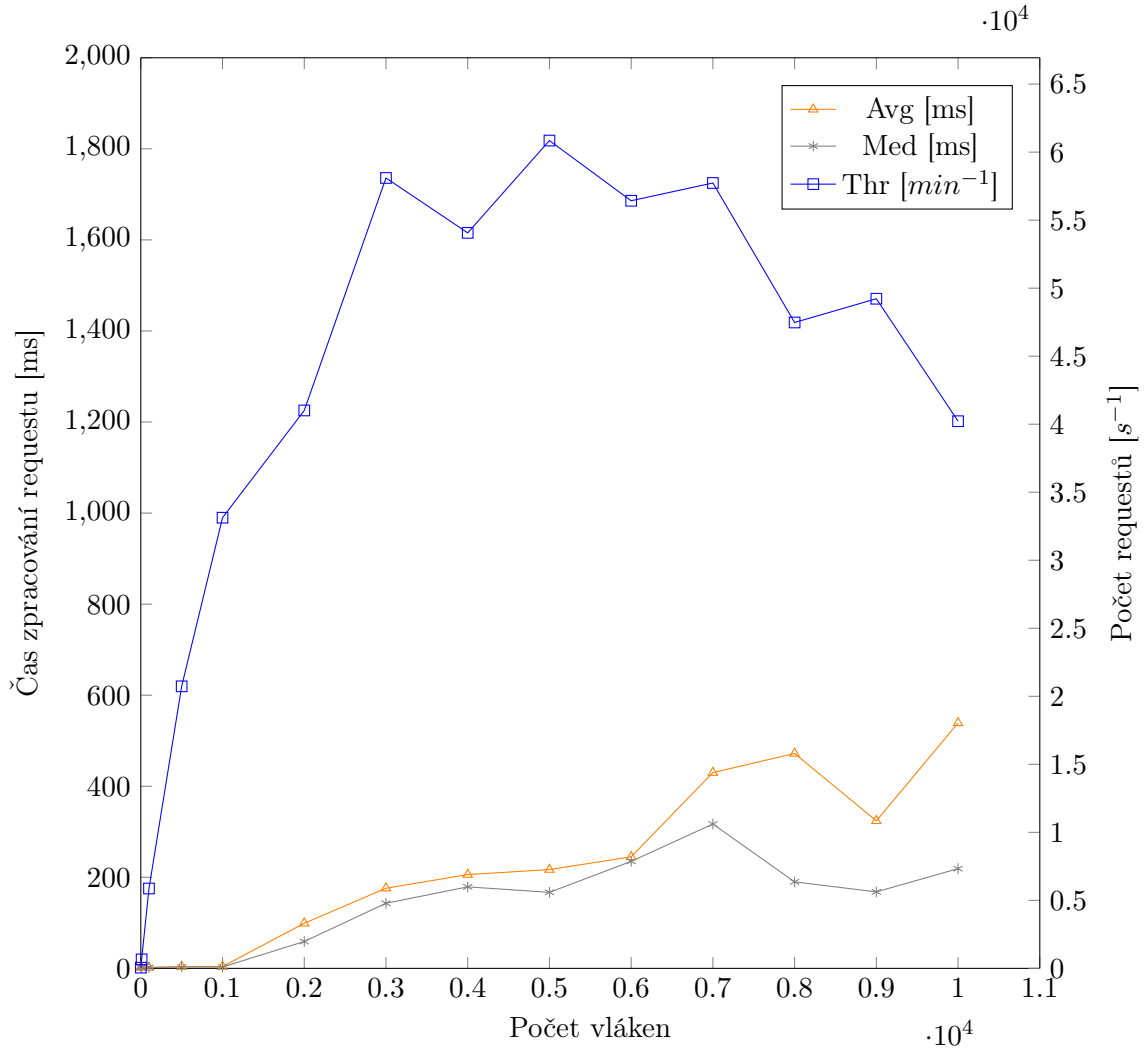
7.2.2 Experiment druhý: zátěžový test API mastera

V rámci tohoto experimentu byl proveden zátěžový test API mastera, který byl proveden v rámci *lokálního* prostředí. Pomocí software *Apache JMeter* bylo provedeno několik měření, ve kterých byl sledován čas zpracovávání requestu (resp. jeho průměr a medián) a průměrný počet zpracovaných událostí za minutu v závislosti na počtu simultánně přistupujících uživatelů. Experiment byl proveden pro 1, 10, 100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000 a 10000 uživatelů, resp. použitých vláken. Každý experiment byl několikrát zopakován a výsledky byly zprůměrovány.

Výsledek experimentu je zobrazen v obrázku 7.3. Z grafu je patrné, že do zhruba 3000 simultánně přistupujících uživatelů počet zpracovaných uživatelů za minutu stoupá a při větším počtu uživatelů začíná klesat.

Dále je patrné, že od zhruba 6000 simultánně přistupujících uživatelů průměrná doba zpracovávání requestu roste oproti jejího mediánu rychleji. To je dáno tím, že některé requesty nestihnou být serverem obslouženy v rámci daného *timeout* a tím je negativně ovlivněna doba jejich zpracování.

Je třeba poznamenat, že měření bylo zatíženou chybou, protože měřící software byl spuštěn na stejném počítači, jako systém Pixnet. Měřící software vytváří velké množství vláken, simulující přistupující uživatele a Tomcat webový server mastera vytváří nová vlákna také, aby velké množství klientů mohl obsloužit.



Obrázek 7.3: Závislost počtu zpracovávaných requestů za minutu (v legendě jako *Thr*, pravá osa) na počtu vláken a závislost času zpracovávání requestu (znázorněno jako *Avg* (průměr) a *Med* (medián), levá osa) na počtu vláken.

Kapitola 8

Závěr

Cílem této práce bylo navrhnout a implementovat distribuovaný systém, který bude schopný řídit a vyčítat data ze sítě hybridních částicových pixelových detektorů *Timepix3* za pomocí vyčítacího rozhraní *Katherine*.

Výsledkem této práce je modulární distribuovaný systém, který uživateli umožňuje operovat teoreticky neomezené množství detektorů různých typů. Implementovaný systém se skládá ze dvou hlavních komponent – handlera a mastera.

Handler slouží pro komunikaci s detektory a vyčítání dat z jemu přiřazených detektorů do uživatelem definovaného datového úložiště. Díky modulární architektuře je možné handleru přiřadit detektory různých typů a jimi naměřená data zpracovávat rozdílnými způsoby (např. odesílat do různých datových úložišť apod.). Komponenta také poskytuje jednoduché webové rozhraní a také *API* pro své řízení jinými komponentami systému, ev. možnost integrace do systémů třetích stran. Počet detektorů, které je možné jednomu handleru přiřadit, je dán sumou jejich datového toku, komplexitou zpracovávání naměřených dat a hardwarovou konfigurací počítače, na kterém je instance handleru nasazena.

Master je centrálním elementem systému, jehož hlavním úkolem je řízení všech handlerů sítě pomocí jejich *API*. Komponenta rovněž implementuje perzistentní úložiště konfigurace a stavu systému. Master dále poskytuje *API* pro své řízení. Primárním konzumentem *API* je webový frontendový klient, který byl implementován jako *single-page* aplikace, která uživateli poskytuje uživatelské rozhraní pro plnohodnotné řízení celého systému. Pomocí *API* mastera může být systém rovněž integrován do systémů třetích stran, jako např. pro účely experimentu *ATLAS-TPX* se systémem `DCS` (CERN systémem pro řízení a akvizici dat detektorových systémů).

Výše zmíněná modulární architektura spočívá v možnosti operování různých typů detektorů. Při přidávání detektoru do systému je uživatel nucen poskytnout také jeho komunikační a datový modul. Komunikační modul musí obsahovat implementaci komunikačního interface (popsaného v 4.1.1.2) a datový modul musí zase obsahovat implementaci datového interface (viz 4.1.1.3).

V rámci této práce byl rovněž vyvinut komunikační a datový modul, jehož prostřednictvím je možné řídit a vyčítat naměřená data z vyčítacího rozhraní *Katherine*, s připojeným *Timepix3* detektorem. Implementovaný datový modul pak umožňuje ukládání dat ve formě souborů do uživatelem definovaného úložiště (např. do distribuovaného souborového systému,

jako je např. HDFS (*Hadoop Distributed File System*)), nebo do *MongoDB* distribuovatelné dokumentové databáze.

Pro účely vývoje a testování byl rovněž vyvinut **emulátor** vyčítacího rozhraní *Katherine*, který emuluje zařízení na síťové vrstvě. Emulátor implementuje všechny příkazy, které jsou důležité pro konfiguraci a spuštění akvizice dat (jako např. nastavení biasu, akvizičního času, vyčítacího a měřícího módu apod.). Emulátor je schopný generovat náhodná data (resp. události vzniklé interakcí s ionizujícím zářením) všech možných kombinací, daných akviziční konfigurací.

8.1 Budoucí vývoj

8.1.1 Podpora více datových modulů jedním detektorem

V kapitole 3.2.1 bylo popsáno, že instance detektoru v handleru je tvořena jeho komunikačním a datovým modulem, které jsou při inicializaci detektoru propojeny asynchronní frontou s naměřenými daty. Tato modifikace systému spočívá v možnosti přidání dalších konzumentů fronty – vícero datových modulů. Tento přístup umožňuje jednoduché přidání dalších komponent pro zpracování dat, např. komponentu pro analýzu naměřených dat v reálném čase, simultánní nahrávání dat do více datových úložišť současně apod.

8.1.2 Podpora pro analýzu a vizualizaci dat v reálném čase

Možnost zobrazení naměřených dat je pro operátora systému důležitá pro zajištění hladkého chodu experimentu a ověření konfigurace systému.

Je plánováno přidání funkcionality která poskytovateli komunikačního modulu umožní definovat důležité hodnoty. Tyto hodnoty (např. okupance snímků, teploty apod.) budou v reálném čase sledovány masterem a v pravidelných intervalech vyčítány, ukládány a budou poskytovány pomocí jeho *API*. Webová aplikace pak bude nabízet vizualizaci vývoje těchto sledovaných hodnot v závislosti na čase.

Rovněž je plánováno přidání podpory pro vizualizaci dat z datového úložiště, ev. výsledky *real-time* analýzy dat.

8.1.3 Zabezpečení

Před nasazením do produkce je plánováno přidání zabezpečení endpointů poskytovaným *API* handlera (pravděpodobně pomocí *Base64*, protože *API* handlera má jediného konzumenta – mastera). *API* mastera bude zabezpečeno pomocí průmyslového standardu *OAuth2*. Důvodem použití *OAuth2* je plánovaný přístup více klientů, kterými jsou uživatelé webové aplikace a ev. systémy třetích stran. Autentizace uživatelů bude rovněž implementována v rámci webové aplikace.

8.1.4 Podpora vyčítacího rozhraní ATLAS Pix

V kapitole 2.7.3 bylo zmíněno vyčítací rozhraní *ATLAS Pix*, pomocí kterého je v současné době realizována detektorová síť *ATLAS-TPX*. Pro zajištění zpětné kompatibility s existujícím systému (resp. možnost operování i detektorů s *ATLAS Pix* rozhraním), bude implementován komunikační modul pro řízení a vyčítání dat z těchto zařízení.

8.1.5 Podpora modifikovaného vyčítacího rozhraní Katherine pro řízení detektorů Timepix2

V současné době je dokončován vývoj nového detektoru *Timepix2*, pro který vzniká modifikace vyčítacího rozhraní *Katherine*. Z tohoto důvodu je nutné vytvořit také modifikovaná komunikační modul.

Literatura

[1] BALLABRIGA, R. et al. Medipix3: A 64k pixel detector readout chip working in single photon counting mode with improved spectrometric performance. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*. 2011, 633, s. S15 – S18. ISSN 0168-9002. doi: https://doi.org/10.1016/j.nima.2010.06.108. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S0168900210012982>. 11th International Workshop on Radiation Imaging Detectors (IWORID).

[2] BALLABRIGA, R. et al. Review of hybrid pixel detector readout ASICs for spectroscopic X-ray imaging. *Journal of Instrumentation*. 2016, 11, 01, s. P01007. Dostupné z: <http://stacks.iop.org/1748-0221/11/i=01/a=P01007>.

[3] BEGERA, J. Calibration and control software for network of particle pixel detectors within the Atlas experiment at the LHC at CERN. Bachelor's thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic, 2016. Dostupné z: <http://hdl.handle.net/10467/64719>.

[4] BOTERENBROOD, H. et al. Design and implementation of the ATLAS detector control system. *IEEE Transactions on Nuclear Science*. June 2004, 51, 3, s. 495–501. ISSN 0018-9499. doi: 10.1109/TNS.2004.828523.

[5] BURIAN, P. et al. Katherine: Ethernet Embedded Readout Interface for Timepix3. *Journal of Instrumentation*. 2017, 12, 11, s. C11001. Dostupné z: <http://stacks.iop.org/1748-0221/12/i=11/a=C11001>.

[6] BURIAN, P. et al. Particle telescope with Timepix3 pixel detectors. *Journal of Instrumentation*. 2018, 13, 01, s. C01002. Dostupné z: <http://stacks.iop.org/1748-0221/13/i=01/a=C01002>.

[7] BURIAN, P. *Documentation of Katherine readout interface* [online]. 2018. [cit. 27. 12. 2018].

[8] EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. New Jersey, USA : Pearson Education (US), 2003. ISBN 978-0321125217.

[9] GANDINI, A. et al. Performance evaluation of NoSQL databases. In *European Workshop on Performance Engineering*, s. 16–29. Springer, 2014.

[10] GRAHAM, D. et al. *Foundations of Software Testing: ISTQB Certification.* Connecticut, US : Intl Thomson Business Pr, 2008. ISBN 9781844803552, 9781844809899.

[11] JAKUBEK, J. Energy-sensitive X-ray radiography and charge sharing effect in pixelated detector. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment.* 2009, 607, 1, s. 192 – 195. ISSN 0168-9002. doi: http://dx.doi.org/10.1016/j.nima.2009.03.148. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S0168900209006408>. Radiation Imaging Detectors 2008Proceedings of the 10th International Workshop on Radiation Imaging Detectors.

[12] JAKUBEK, J. Precise energy calibration of pixel detector working in time-over-threshold mode. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment.* 2011, 633, Supplement 1, s. S262 – S266. ISSN 0168-9002. doi: http://dx.doi.org/10.1016/j.nima.2010.06.183. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S0168900210013732>. 11th International Workshop on Radiation Imaging Detectors (IWORID).

[13] Kolektiv autorů Medipix kolaborace. *Web Medipix* [online]. 2018. [cit. 1. 10. 2018]. Dostupné z: <https://medipix.web.cern.ch>.

[14] Kolektiv autorů Medipix kolaborace. *Particle detectors meet canvas* [online]. 2018. [cit. 3. 1. 2019]. Dostupné z: <http://cds.cern.ch/record/2311083>.

[15] Kolektiv autorů Medipix kolaborace. *The Medipix Chips and Collaborations: from medical imaging to space dosimetry* [online]. 2018. [cit. 3. 1. 2019]. Dostupné z: <https://kt.cern/success-stories/medipix-chips-and-collaborations-medical-imaging-space-dosimetry>.

[16] Kolektiv autorů MongoDB. *MongoDB at Scale* [online]. 2018. [cit. 16. 12. 2018]. Dostupné z: <https://www.mongodb.com/mongodb-scale>.

[17] KRAUS, V. et al. FITPix — fast interface for Timepix pixel detectors. *Journal of Instrumentation.* 2011, 6, 01, s. C01079. Dostupné z: <http://stacks.iop.org/1748-0221/6/i=01/a=C01079>.

[18] KRISHNAN, D. – KELLY, A. *Springfox Reference Documentation* [online]. 2018. [cit. 27. 12. 2018]. Dostupné z: <http://springfox.github.io/springfox/docs/current>.

[19] LLOPART, X. et al. Medipix2: A 64-k pixel readout chip with 55-/spl mu/m square elements working in single photon counting mode. *IEEE Transactions on Nuclear Science.* Oct 2002, 49, 5, s. 2279–2283. ISSN 0018-9499. doi: 10.1109/TNS.2002.803788.

[20] LLOPART, X. et al. Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment.* 2007, 581, 1-2, s. 485 – 494. ISSN 0168-9002. doi: http:

//dx.doi.org/10.1016/j.nima.2007.08.079. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S0168900207017020>. {VCI} 2007Proceedings of the 11th International Vienna Conference on Instrumentation.

[21] MANEK, P. et al. Software system for data acquisition and analysis operating the ATLAS-TPX network. In *2017 International Conference on Applied Electronics (AE)*, s. 1–4, Sept 2017. doi: 10.23919/AE.2017.8053593.

[22] MARTISIKOVA, M. et al. Study of the capabilities of the Timepix detector for Ion Beam radiotherapy applications. *IEEE Nuclear Science Symposium Conference Record*. 10 2012, s. 4324–4328. doi: 10.1109/NSSMIC.2012.6551985.

[23] PLATKEVIC, M. *Signal Processing and Data Read-Out from Position Sensitive Pixel Detectors*. PhD thesis, Czech Technical University in Prague, Czech Republic, 2014.

[24] POIKELA, T. et al. Timepix3: a 65K channel hybrid pixel readout chip with simultaneous ToA/ToT and sparse readout. *Journal of Instrumentation*. 2014, 9, 05, s. C05013. Dostupné z: <http://stacks.iop.org/1748-0221/9/i=05/a=C05013>.

[25] SHVACHKO, K. et al. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, s. 1–10, May 2010. doi: 10.1109/MSST.2010.5496972.

[26] SINOR, M. et al. Charge sharing studies with a Medipix1 pixel device. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*. 2003, 509, 1, s. 346 – 354. ISSN 0168-9002. doi: https://doi.org/10.1016/S0168-9002(03)01648-6. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S0168900203016486>. Proceedings of the 4th Internatonal Workshop on Radiation Imaging Detectors.

[27] SOPCZAK, A. et al. Precision Luminosity of LHC Proton–Proton Collisions at 13 TeV Using Hit Counting With TPX Pixel Devices. *IEEE Transactions on Nuclear Science*. March 2017, 64, 3, s. 915–924. ISSN 0018-9499. doi: 10.1109/TNS.2017.2664664.

[28] TREMSIN, A. et al. High Resolution Photon Counting With MCP-Timepix Quad Parallel Readout Operating at $>$ 1 KHz Frame Rates. *IEEE Transactions on Nuclear Science*. 04 2013, 60, s. 578–585. doi: 10.1109/TNS.2012.2223714.

[29] TURECEK, D. et al. Pixelman: a multi-platform data acquisition and processing software package for Medipix2, Timepix and Medipix3 detectors. *Journal of Instrumentation*. 2011, 6, 01, s. C01046. Dostupné z: <http://stacks.iop.org/1748-0221/6/i=01/a=C01046>.

[30] TURECEK, D. – JAKUBEK, J. – SOUKUP, P. USB 3.0 readout and time-walk correction method for Timepix3 detector. *Journal of Instrumentation*. 2016, 11, 12, s. C12065. Dostupné z: <http://stacks.iop.org/1748-0221/11/i=12/a=C12065>.

Příloha A

Přílohy vyčítacího rozhraní Katherine

A.1 Přehled DAC vyčítacího rozhraní Katherine

Název	ID pro čtení	ID pro zápis
Ibias_Preamp_ON	1	0
Ibias_Preamp_OFF	2	1
VPreamp_NCAS	3	2
Ibias_Ikrum	4	3
Vfbk	5	4
Vthreshold_fine	6	5
Vthreshold_coarse	7	6
Ibias_DiscS1_ON	8	7
Ibias_DiscS1_OFF	9	8
Ibias_DiscS2_ON	10	9
Ibias_DiscS2_OFF	11	10
Ibias_PixelDAC	12	11
Ibias_TPbufferIn	13	12
Ibias_TPbufferOut	14	13
VTP_coarse	15	14
VTP_fine	16	15
Ibias_CP_PLL	17	16
PLL_Vcntrl	18	17
BandGap output	28	-
BandGap_Temp	29	-
Ibias_dac	30	-
Ibias_dac_cas	31	-

Tabulka A.1: Přehled DAC (digitálně analogových převodníků) vyčítacího rozhraní Katherine.

A.2 Přehled HW příkazů vyčítacího rozhraní Katherine

ID	Název
0	Sensor Config Registers Update
1	Internal DAC Update
2	Internal DAC Back Read
3	Timer Read
4	Timer Set
5	Reset Matrix Sequential
6	Stop Matrix Command
7	Load Column Test Pulse Register
8	Read Column Test Pulse Register
9	Load Pixel Register Configuration
10	Read Pixel Register Configuration
11	Read Pixel Matrix Sequential Setting
12	Read Pixel Matrix Data-Driven Setting
13	Chip ID Read
14	Output Block Config Update
15	Digital Test

Tabulka A.2: Přehled HW příkazů vyčítacího rozhraní Katherine.

A.3 Přehled registrů detektoru Timepix3 v rámci vyčítacího rozhraní Katherine

ID	Název
0	Test Pulse Period
1	Number Test Pulses
2	Out Block Config
3	PLL Config
4	General Config
5	SLVS Config
6	Power Pulsing Pattern
7	SetTimer 15..0
8	SetTimer 31..16
9	SetTimer 47..32
10	Sense DAC Selector
11	Ext DAC Selector

Tabulka A.3: Přehled registrů detektoru Timepix3 v rámci vyčítacího rozhraní Katherine.

A.4 Konfigurace pixelů detektoru

```
fun bmcToMatrixConfig(bmc: ByteArray): ByteArray {
    assert(bmc.size == 65536)
    val buff = IntArray(16384)

    for (i in bmc.indices) {
        var y = i / 256
        val x = i % 256
        val tmp = bmc[i]
        y = 255 - y
        buff[64 * x + (y shr 2)] = buff[64 * x + (y shr 2)] or (tmp.toInt() shl
        ↪  8 * (3 - y % 4))
    }

    val out = ByteArray(65536)
    var k = 0
    for (mem in buff) {
        val b = ByteBuffer.allocate(4).putInt(mem).array()
        for (i in 0..3) {
            out[k + 3 - i] = b[i]
        }
        k += 4
    }

    return out
}
```

Zdrojový kód A.1: Ukázka kódu pro převod pixelové konfigurace detektoru z formátu `BMC` do formátu podporovaného vyčítacím rozhraním *Katherine*.

A.5 Příklad výstupních dat datového modulu.

```
# Start of measurement: 2018.12.11 08:51:18.363+0000
# Acq start timestamp: 1544518278053
# Chip ID: M7-W0005
# Acq mode: ToA & ToT with FastToA
# Fast VCO enabled: true
# Readout mode: Data driven
# Bias: 230,000000
# Temperature readout: 52,125000
# Temperature sensor: 82,522804
# Payload older: PIX_ID        ToA        ToT
# NEW_FRAME_ESTABLISHED
21541         54657847        1
22060         54657847        3
22310         54657840        4
36548         9197790130        8
36810         9197790132        75
35266         9197790130        14
36549         9197790129        16
36553         9197790125        67
35522         9197790132        23
36809         9197790130        8
35778         9197790135        8
...
3973         9674803477        9
4239         9674803471        10
3983         9674803465        24
4226         9674803479        19
5262         9674803485        27
4495         9674803477        6
4751         9674803474        23
5007         9674803479        22
29064         9846817974        5
29319         9846817971        10
29320         9846817966        33
# Start of frame timestamp: 2
# End of frame timestamp: 400000002
# Lost pixels: 0
# Frame 0 finished
# Sent pixels: 859
```

Zdrojový kód A.2: Příklad výstupních dat datového modulu.

Příloha B

Seznam použitých zkratek

ACQ Datagram typu acknowledge

ADC Analogově digitální převodník

API Application Programming Interface

ASCII American Standard Code for Information Interchange

ASIC Application-specific Integrated Circuit

ATLAS A Toroidal LHC Apparatus

B byte

b bite

BSON Binární forma JSON

CdTe Cadmium telluride

CERN Evropská organizace pro jaderný výzkum (Originální název: *Conseil Européen pour la Recherche Nucléaire*), se sídlem v Ženevě, ve Švýcarsku.

CMOS Complementary Metal Oxide Semiconductor

CRUD Create, read, update and delete

CSM Charge Summing Mode

CSS Cascading Style Sheets

Cu Hliník

DAC Digitálně analogový převodník

DAQ Data Acquisition System

DCS Detector Control Systems

DOM Document Object Model

DSL Domain-specific language

DSS Detector Safety System

ESA The European Space Agency

eV elektronvolt

Fe Železo

FITPix Fast Interface for Timepix Pixel Detectors

FPGA Field Programmable Gate Array

FWHM Full width at half maximum

GaAs Arsenid gallitý

GPIO General Purpose Input/Output

HDFS Hadoop Distributed File System

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

HW Hardware

IaaS Infrastructure as a Service

ID Identifikátor

In Indium

IP Internet Protocol

ISO International Organization for Standardization

J2EE Java Enterprise Edition

JAR Java Archive

JPA Java Persistent API

JRE Java Runtime Environment

JS JavaScript

JSON JavaScript Object Notation

LHC Large Hadron Collider

LOB Large Object

LS Long Shutdown - dlouhodobá technologická přestávka LHC

LVDS Low-voltage differential signaling

MPX Medipix

MVC Model View Controller

ORM Object-relational mapping

OSI Open Systems Interconnection model

PC Personal Computer

PCB Printed Circuit Board

PCC Photon Counting Chip

PN Přechod polovodiče typu P a polovodiče typu N

REST Representational State Transfer

SATRAM Space Application of Timepix based Radiation Monitor

SFTP Secure File Transfer Protocol

Si Křemík

SPI Serial Peripheral Interface

SQL Structured Query Language

SSH Secure Shell

SW Software

TCP Transmission Control Protocol

ToA Time of Arrival - mód detektoru

TOT Time Over Treshold - mód detektoru

TPX Timepix

UDP User Datagram Protocol

URL Uniform Resource Locator

USB Universal Serial Bus

UTC Koordinovaný světový čas (anglicky Coordinated Universal Time)

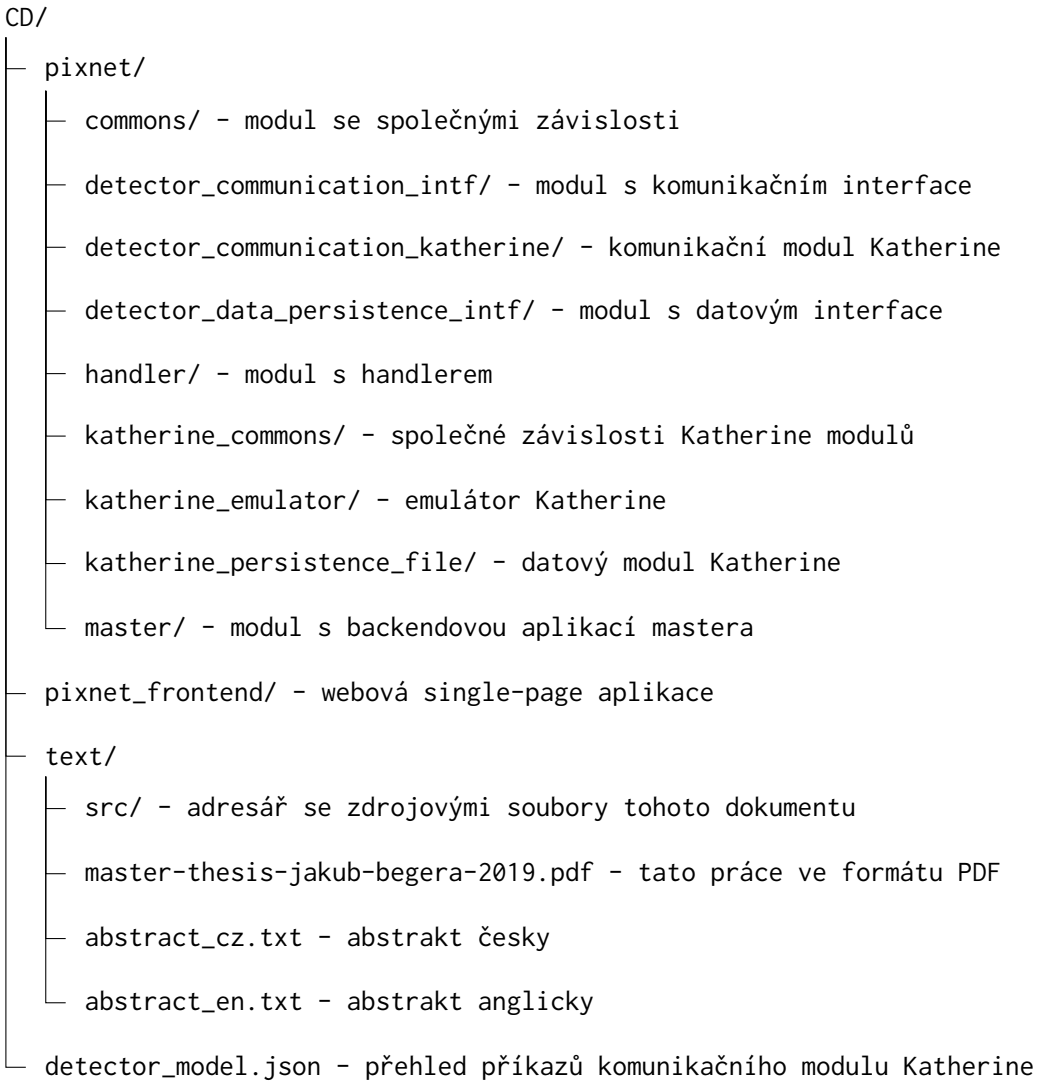
VCPU Virtuální procesor

WAR Web Application Archive

ÚTEF Ústav technické a experimentální fyziky

Příloha C

Obsah přiloženého CD



Obrázek C.1: Obsah přiloženého CD