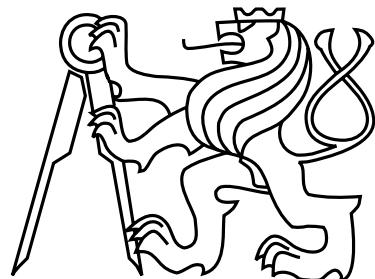


České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

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

Bc. Jakub Begera

Vedoucí práce: Ing. Štěpán Polanský

Studijní program: Otevřená informatika

Obor: Softwarové inženýrství

27. prosince 2018



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., Llopard, 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, <http://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

TODO

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 27. 5. 2016

.....

Abstract

TODOabstrakt anglicky

Abstrakt

TODOabstrakt česky

Obsah

1	Úvod	1
1.1	Motivace	1
1.2	Timepix3 detektor	1
1.3	Struktura práce	2
2	Úvod do hybridních čáстicových pixelových detektorů	3
2.1	Hardwarová architektura	3
2.2	Princip detekce	4
2.3	Operační mody detektoru	5
2.4	Vyčítání naměřených dat	6
2.5	Kalibrace	7
2.5.1	Treshold equalizace	8
2.5.2	Energetická kalibrace	8
2.5.3	Time-Walk korekce	10
2.5.4	ToA offset korekce	11
2.6	Přehled detektorů rodiny Medipix	11
2.7	Vyčítací rozhraní	13
2.7.1	FITPix	13
2.7.2	AdvaDAQ	14
2.7.3	ATLAS Pix	14
2.7.4	Katherine	15
2.7.4.1	Komunikace s nadřazeným PC a řízení akvizice dat	16
3	Návrh architektury	19
3.1	Motivace	19
3.2	Softwarová architektura	20
3.2.1	Detektor	20
3.2.2	Datové úložiště	21
3.2.3	Handler	21
3.2.4	Master	22
3.3	Hardwarová architektura	22
3.4	Škálovatelnost	24
3.5	Použité technologie	24

4 Handler	29
4.1 Vrstvy softwarové architektury	30
4.1.1 Detektorová vrstva	30
4.1.1.1 Zavádění modulů	31
4.1.1.2 Komunikační rozhraní	31
4.1.1.3 Datové rozhraní	36
4.1.2 Vrstva managementu detektorů	36
4.1.3 Spring vrstva	37
4.1.3.1 Detectors REST API controller	37
4.1.3.2 Handler status REST API controller	38
4.1.3.3 Swagger API doc REST controller	39
4.1.3.4 Webové rozhraní handleru	40
4.2 Konfigurace a nasazení	41
5 Katherine rozhraní	43
6 Master	45
7 Testování	47
8 Závěr	49
A Seznam použitých zkratek	55
B Obsah přiloženého CD	59

Seznam obrázků

2.1	Struktura hybridního polovodičového pixelového detektoru Timepix3, skládající se z vyčítacího čipu a polovodičového senzoru [19].	4
2.2	Princip detekce ionizujícího záření detektorem Timepix3 [19].	4
2.3	Zpracování signálu pixelem detektoru dle nastaveného módu (<i>Medipix</i> , <i>ToT</i> a <i>ToA</i>) [19].	5
2.4	Doba vyčítání detektoru za použití <i>Frame-based</i> (non-sparse) a <i>Data-driven</i> (sparse) módu [20].	7
2.5	Kalibrační funkce, udávající závislost mezi energií v keV a ToT [10], vzniklá proložením získaných kalibračních bodů funkcí 2.2 a sestávající se ze dvou částí - (i) nelineární částí pro oblast nižších energií (hyperbola) a (ii) lineární částí pro vyšší energie (příjímkou).	8
2.6	Příklad energetického spektra jednoho pixelu Timepix detektoru s proloženou funkcí 2.3 [3].	9
2.7	<i>Time-walk</i> efekt: příklad interakce jedné částice se čtyřmi pixely detektoru, kde v každém pixelu byla deponována jiná energie, což na výstupu zesilovačů pixelů způsobilo jiné hodnoty napětí. Díky rozdílné charakteristice náběžné hrany pulzů byl threshold překročen v různých časech (t_{1-4}) [26].	11
2.8	Schéma pixelu detektoru <i>Timepix3</i> se společnou elektronikou pro 8 pixelů (tzv. <i>Super-pixel</i>) [20].	12
2.9	Vyčítací rozhraní <i>Katherine</i> s připojeným detektorem <i>Timepix3</i> vlevo a popisem vstupních a výstupních portů vpravo[5].	14
2.10	Katherine: příklad requestu a response řídícího datagramu pro vyčtení biasu.	17
3.1	<i>Pixnet</i> : softwarová architektura.	20
3.2	Sekvenční diagram znázorňující příklad přidání detektoru do systému a jeho přiřazení handleru.	23
3.3	<i>Pixnet</i> : hardwarová architektura s příkladem realizace sítě o dvou podsítích, kde v každé jsou dva handlery a pět detektorů, mastera (s <i>SQL</i> databází pro persistenci konfigurace a frontend server poskytující webovou aplikaci) a centrálního datového úložiště naměřených dat.	24
4.1	<i>Pixnet</i> - handler: příklad instance handleru s pěti připojenými detektory, poskytujícího REST API pro své řízení a webové uživatelské rozhraní s přehledem připojených detektorů.	29

4.2	Pixnet - handler: softwarová architektura s vrstvami pro (i) rozhraní detektoru (<i>Detector layer</i>), (ii) management detektorů (<i>Detectors management layer</i>),(iii) Spring vrstvu (<i>Spring layer</i>) a (iv) výstupní vrstvou pro REST API a webové uživatelské rozhraní.	30
4.3	Asynchronní blokující fronta naměřených dat s příkladem produkujících vláken v komunikačním modulu a přijímacím vláknu v datovém modulu.	35
4.4	Springfox online dokumentace handleru.	40
4.5	Screenshot webového rozhraní handleru.	41
B.1	Obsah přiloženého CD	60

Seznam tabulek

2.1	Katherine: závislost maximálního datového toku na typu propojení mezi detektorem a vyčítacím rozhraním [5].	15
4.1	Endpointy komponenty <i>Detectors REST API controller</i>	38

Seznam zdrojových kódů

4.1	Příklad obsahu souboru <code>MANIFEST.MF</code> , obsaženého v <code>jar</code> archívu modulu.	31
4.2	Komunikační interface detektoru, napsané v jazyce Kotlin (viz 3.5)).	32
4.3	Příklad definice <i>ValueCommand</i> detektoru pro příkaz s názvem " <i>Bias</i> ", id 42, jednotkou Volt, modifikátorem přístupu <i>Setter & Getter</i> a reálným modelem hodnot, omezeným intervalm $<-300, 300>$	33
4.4	Příklad definice <i>ExecutionCommand</i> pro nastavování akvizičního módu detektoru s vyčítacím rozhraním <i>Katherine</i> (viz 2.7.4). Z příkladu je patrné, že vstupní model je tvořen dvěma hodnotami a výstupní model je prázdný.	34
4.5	Datový interface detektoru, napsané v jazyce Kotlin (viz 3.5)).	36
4.6	Příklad volání API komponenty pro správu detektorů. V příkladu na řádcích 1 až 12 je <i>request</i> pro vykonání <i>ExecutionCommand</i> pro nastavení akvizičního módu detektoru a na řádcích 14 až 18 je <i>response</i> serveru.	38
4.7	Příklad volání API komponenty pro poskytování stavu handleru, resp. těla odpovědi endpointu <code>/status</code>	39
4.8	YAML konfigurační soubor handleru.	41

Kapitola 1

Úvod

TODO

1.1 Motivace

TODO

1.2 Timepix3 detektor

Hybridní čáстicový pixelový detektor Timepix3[20] je nástupcem detektoru Timepix[16] a je vyvíjen v rámci Medipix¹ kolaborace v CERN, mezi jejíž členy patří od roku 1999 i ÚTEF ČVUT v Praze.

Detektor se skládá z matice 256×256 nezávislých pixelů, každý o hraně $55 \mu m$. Jednotlivé pixely se skládají z citlivého polovodičového senzoru (nejčastěji *Si*, nebo *GaAs*) 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 uzávěrkou (tzv. *shutter signal*). Po tuto dobu pak citlivý polovodičový objem detektoru zaznamenává interakce s nabitymi částicemi a dále je zpracovává dle nastaveného modu. V kapitole 2 bude na příklad popsán *Time-Over-Threshold* mód, kde hodnota čítače pixelu na konci akvizice odpovídá deponované energii interagovaných častic s daným pixelem (mezi energií a TOT je nelineární závislost, která je dána fyzikálními vlastnostmi každého pixelu a je předmětem energetické kalibrace detektoru [10]).

Timepix3 detektor přináší oproti svému předchůdci několik výhod. Je schopný operovat i 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 zvyšuje datový tok z detektoru, jehož maximální teoretická hodnota je až $5.12Gb/s$.

¹<<http://medipix.web.cern.ch/>>

1.3 Struktura práce

TODO

Kapitola 2

Úvod do hybridních částicových pixelových detektorů

Ionizující záření je lidskými smysly nedetectovatelné, avšak jeho studie nám umožnuje pochopit podstatu hmoty, její vlastnosti a interakce. To lidstvu umožnilo mnohé aplikace, jako je například protonová terapie [18], defektoskopie nebo zkoumání pravosti uměleckých děl. První pokusy o detekci ionizujícího záření sahají do počátku 20. století, kde pomocí mlžné komory se prvně podařilo zachytit trajektorii nabitych častic. Rozvoj polovodičové technologie dal vzniku novým detekčním technologiím až po v současné době nejpokrokovějším - pixelovým detektorům.

Existuje celá řada částicových pixelových detektorů, ale v této kapitole budou popsány jen 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*, *CaTe* apod.) a tloušťek (vetšinou $300\mu m$, nebo $500\mu m$).

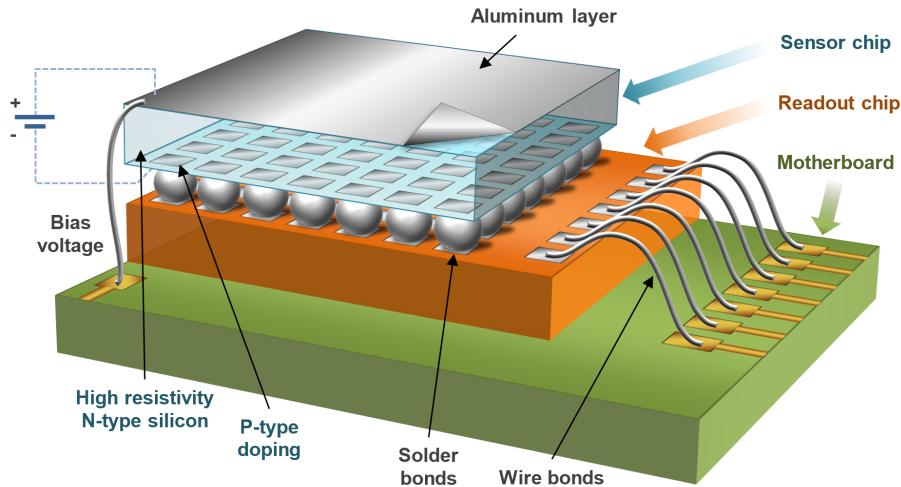
Na tomto místě je třeba zmínit, že existuje více druhů těchto detektorů (*AGH Fermilab*, *Pilatus*, *Philips Chromaix* apod.)[2], v této práci budou použity pouze detektory z rodiny detektorů Medipix.

2.1 Hardwarová architektura

Většina hybridních částicových pixelových detektorů rodiny Medipix obsahuje matici 256×256 pixelů. Každý z nich má stanu o délce $55\mu m$, takže senzor čítající 65536 má plochu $1.4 \times 1.4 cm^2$.

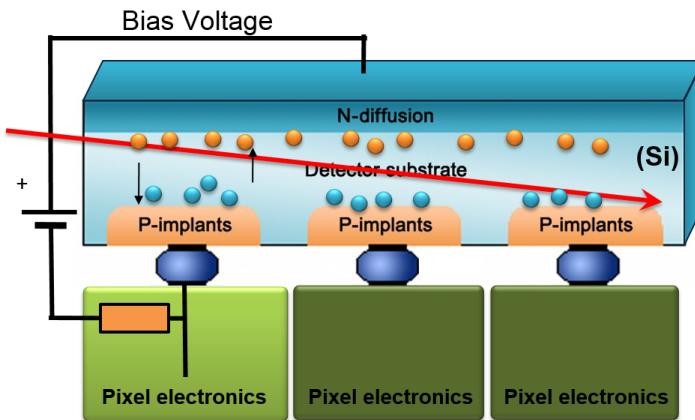
Na obrázku 2.1 je znázorněna struktura detektoru Timepix3. Vrchní část detektoru tvoří polovodičový senzor, který je nejčastěji vyroben z křemíku, ale výjimkou není také *GaAs* nebo *CaTe*. Jednotlivé pixely senzoru jsou spojeny s integrovaným ASIC¹ vyčítacím čipem pomocí technologie zvané *Bump-Bounding*. Vyčítací čip je pak propojen se základní deskou pomocí *wire-bound*, z které je ještě přivedeno měřící napětí na senzor detektoru (tzv. *bias*).

¹z angl. Application Specific Integrated Circuit



Obrázek 2.1: Struktura hybridního polovodičového pixelového detektoru Timepix3, skládající se z vyčítacího čipu a polovodičového senzoru [19].

2.2 Princip detekce



Obrázek 2.2: Princip detekce ionizujícího záření detektorem Timepix3 [19].

Princip detekce ionizujícího záření pixelovými detektory je založen na známém jevu detekce ionizujícího záření v polovodiči.

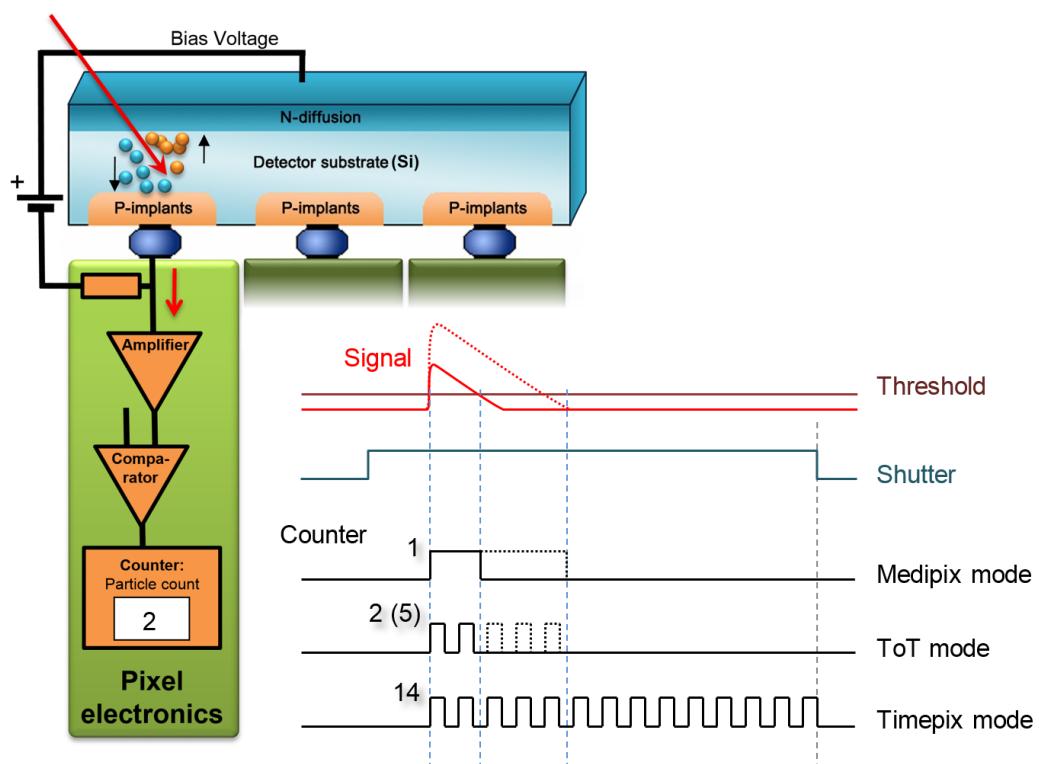
Jako náhradní schéma jednoho pixelu si lze představit diodu zapojenou v závěrném směru, kterou bez přítomnosti ionizujícího záření protéká minimální proud. Vnikne-li do senzoru ionizující částice a dojde k její interakci se senzorem, resp. část její energie je deponována do polovodičového objemu senzoru,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

znázorňuje interagující částici, elektrony jsou znázorněny žlutě, modře díry).

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

Na rozdíl od CCD technologie, CMOS readout *Timepix/Medipix* detektorů negeneruje temný proud², díky odstínění signálu od šumu pomocí komparačního napětí. To znamená, že doba jedné akvizice je teoreticky neomezena, protože detektor je schopný detektovat jen ty částice, jejichž deponovaná energie (resp. amplituda vzniklého napěťového pulzu) je větší, než *threshold*.

2.3 Operační módy detektoru



Obrázek 2.3: Zpracování signálu pixelem detektoru dle nastaveného módu (*Medipix*, *ToT* a *ToA*) [19].

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

Jak už bylo popsáno v předchozí kapitole, interagovaná částice vyvolá napěťový impulz, jehož tvar koreluje s deponovanou energií. Pro účely analýzy se ale používá pouze binární

²Termín charakterizující vyčítací šum u CCD snímačů. Obvykle je udáván v elektronech za sekundu při konstantní teplotě a ve tmě.

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 módů:

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 jeho čítač je inkrementován 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 [24] nebo [18].

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.

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*, byl prvně použit v detektoru *Timepix3*. Pracuje-li detektor v tomto módu, pak v průběhu akvizice dat (resp. když *shutter* signál na 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 použít *Frame-Based* módu, protože není třeba přenášet souřadnice zasažených pixelů. Podle [20] vyčítací čas může být definován následovně:

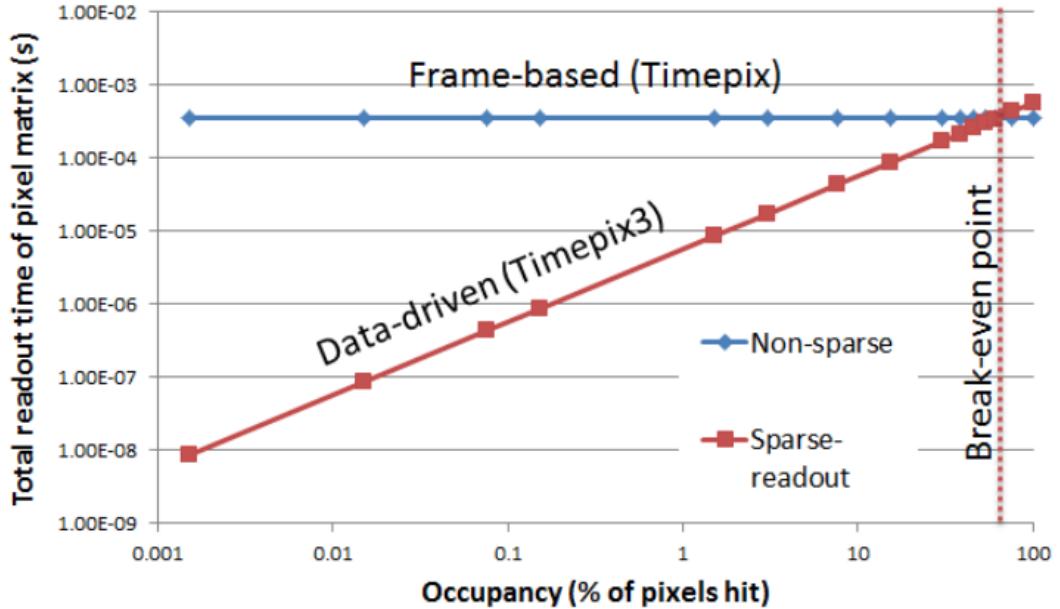
$$T_{readout} = N_{pixels} * bits_{pixel} / BW \quad (2.1)$$

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 [20].

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 minimizace systematické chyby měření. Z pohledu aplikace získaných kalibračních dat je možné kalibrační metody rozdělit do dvou kategorií:

- Použití v průběhu akvizice dat - jedná se o data, která jsou použita pro nastavení akvizice dat v detektoru a mají přímý vliv na naměřená data, která danou metodou není možné dodatečně kalibrovat. Do této kategorie spadá například *threshold equalizace* (viz 2.5.1).
- 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. To této kategorie spadá například *Energetická kalibrace* (viz 2.5.2) nebo *Time-Walk korekce* (viz 2.5.3).

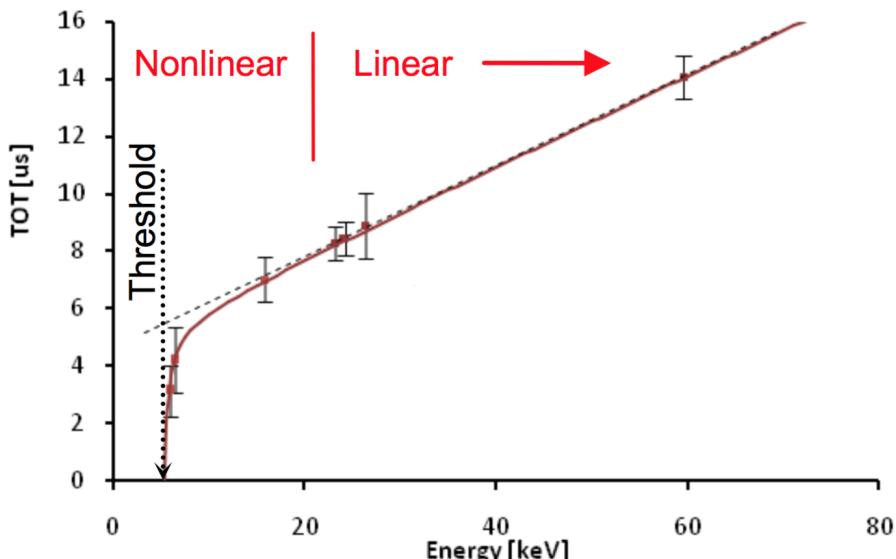
2.5.1 Threshold equalizace

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 $4b$ hodnoty thresholdu (viz obr. 2.8).

Vlastní proces equalizace probíhá tak, že se udělá threshold scan přes všechny hodnoty, přičemž je třeba minimalizovat interakce detektoru z částicemi. V běžné praxi stačí detektor dostatečně odstínit. Výstupem tohoto procesu je pak globální threshold a jeho 4-bitové korekce pro jednotlivé pixely.

Jako vedlejší produkt tohoto procesu je rovněž maskovací matice detektoru, které obsahuje šumějící, nebo jinak poškozené pixely. To jsou například takové pixely, které bez přítomnosti interagujících částic hlásí překročení thresholdu.

2.5.2 Energetická kalibrace



Obrázek 2.5: Kalibrační funkce, udávající závislost mezi energií v keV a ToT [10], vzniklá proložením získaných kalibračních bodů funkcí 2.2 a sestávající se ze dvou částí - (i) nelineární částí pro oblast nižších energií (hyperbola) a (ii) lineární částí pro vyšší energie (příjemka).

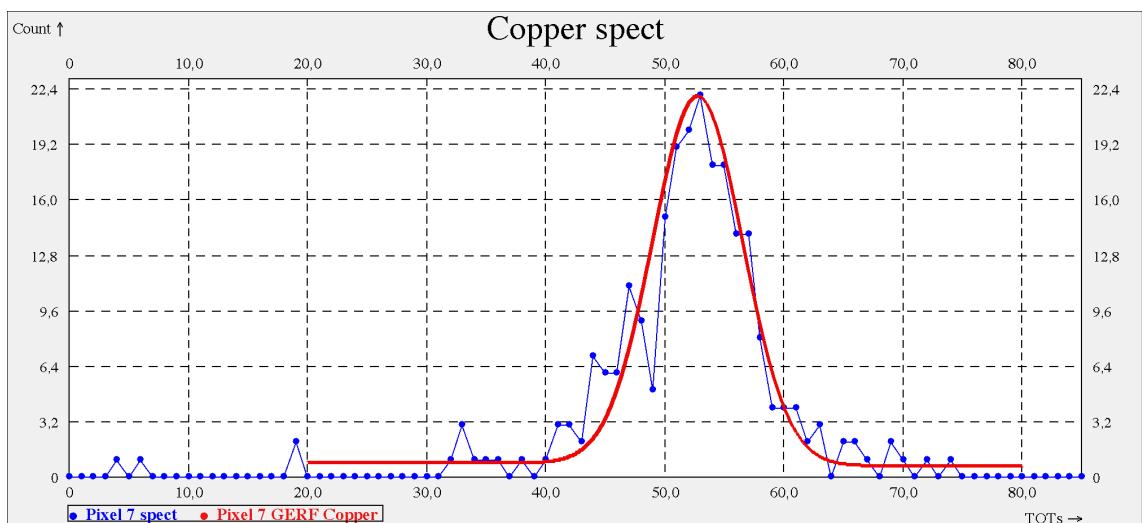
V předchozí části práce byl již představen *Time-Over-Threshold* mód (viz 2.3), ve kterém je detektor schopný měřit deponovanou energii interagovaných v částic, která je udávána v ToT. Jak již bylo ukázáno, vztah mezi energií v keV a ToT je nelineární závislost a závisí na

fyzikální vlastnostech daného pixelu, což je předmětem energetické kalibrace, která bude v této podkapitole popsána.

Tato metoda [10] spočívá v provedení několika sad měření se zdroji ionizujícího 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.

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

Pro měření kalibračních dat se jako efektivní řešení v praxi ukázalo použití rentgenové fluorescence³ [9]. Pro zajištění dobré kvality kalibrace je třeba naměřit takový počet událostí, aby spektra ve snímcích byla dobře rozeznatelná. Z naměřených dat jsou vyfiltrovány pouze tzv. *Single-hit* události⁴, aby se minimalizovaly negativní vlastnosti *Charge-sharing* efektu (díky společné elektrodě senzoru jsou díky interagující částici vzniklé elektrony zpracovány více pixely najednou a část deponované energie nemusí být ASIC čipem zpracována, protože vzniklý signál může být nižší než threshold daného pixelu).



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

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 střední hodnoty ToT a tabulkové hodnoty energie fluorescenčního záření mědi. Střední hodnota je získána proložením spektra funkcí 2.3 - jedná se o součet

³Děj ke kterému dochází při ozářování materiálu (nejčastěji 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 úrovni obsadí jeho místo a přebytečnou energii emituje formou vyzářeného fotonu - fluorescenčního záření, jehož charakteristické monoenergetické spektrum je pro většinu prvků dobře známé.

⁴Události, ve kterých částice interagovala pouze s jedním pixelem detektoru

Gaussovy funkce a Gaussovy chybové funkce (kvůli levé nesymetrii vzniklé *Charge-sharing* efektem)

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

kde:

A je amplituda,

μ je stření hodnota hledané energie,

σ je rozptyl střední hodnoty energie μ , který je možné vypočítat ze vzorce 2.4, kde FWHM⁵ udává šířku gausiánu v polovině jeho výšky a

avg_{right} , avg_{left} je průměrná hodnota spektra na pravém (resp. levém) úpatí gausiánu.

$$\sigma = \frac{2\sqrt{2\ln 2}}{FWHM} \quad (2.4)$$

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 napěťového pulzu na zesilovači pixelu. Při interakci ionizující částice s více pixely je pak díky tomuto jevu interagovanými pixely zaznamenána jiná hodnota *Time-of-Arrival* (ToA), i přes to že událost byla způsobena stejnou částicí a ToA by měl být stejný. Viz obrázek 2.7, kde jsou pro interakci stejné částice čtyřmi sousedními pixely zaznamenány různé hodnoty ToA.

S použitím detektoru *Timepix3*[20] je možné tuto energeticky závislou chybu eliminovat za použití kalibrační metody [26], protože tento detektor umožňuje měřit v ToA a ToT módu současně. Tato kalibrační metoda spočívá v analytickém zpracování dat získaných z měření se zdrojem alfa častic (v [26] použito ^{241}Am), které generuje clustery o velikosti maximálně čtyři pixely. Nejprve je však třeba potřeba energeticky zkalibrovat 2.5.2.

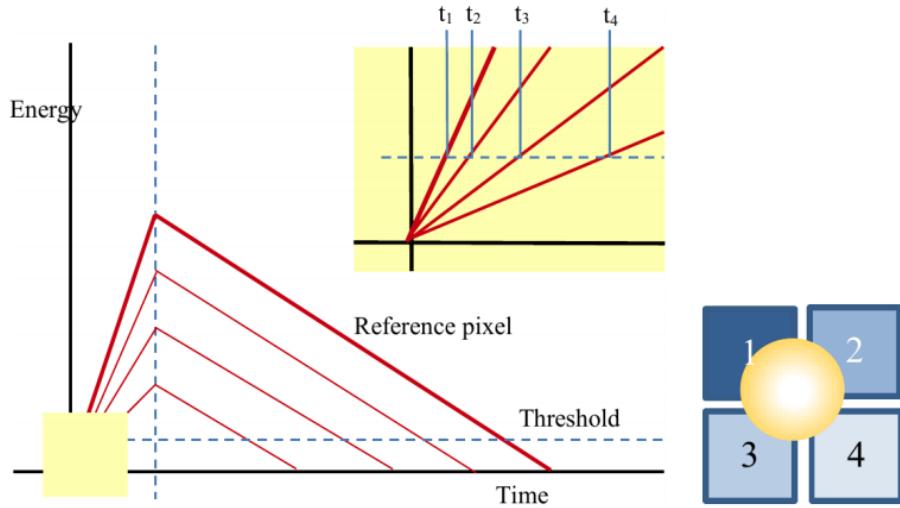
O vybraných clusterech o velikosti 3 a 4 pixely pro ^{241}Am víme, že součet jejich energií je 59.5keV . Z clusteru je vybraný pixel s energií 30keV , který je použit jako referenční (na 2.5.3 jako t_1), zbylá energie je náhodně rozdělena mezi ostatní pixely (na 2.5.3 jako t_{2-4}). Jednotlivé rozdíly $t_i + t_1$ jsou analytickými metodami, popsanými v [26], zpracovány a výsledkem tohoto procesu jsou konstanty c, d pro každý pixel detektoru, kterými lze vypočítat jeho *Time-Walk offset* ΔT :

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

kde:

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

⁵z angl. Full Width at Half Maximum



Obrázek 2.7: *Time-walk* efekt: příklad interakce jedné částice se čtyřmi pixely detektoru, kde v každém pixelu byla deponována jiná energie, což na výstupu zesilovačů pixelů způsobilo jiné hodnoty napříč. Díky rozdílné charakteristice náběžné hrany pulzů byl threshold překročen v různých časech (t_1-t_4) [26].

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

2.5.4 ToA offset korekce

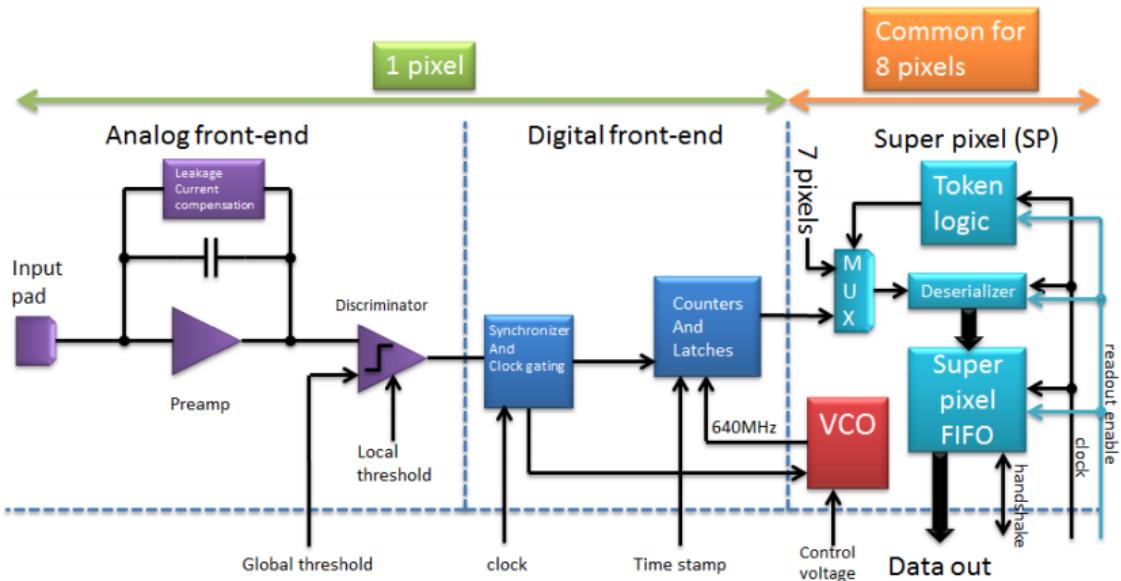
Jedná se o jev, ke kterému dochází u *Timepix3* detektorů z důvodu chyby hardware, který způsobuje špatnou synchronizaci časového razítka napříč sloupcí detektoru [5]. Tato chyba vzniká jen při zapnutí detektoru a poté je ToA offset již stabilní, takže je možné vytvořit korekci.

Odstranění tohoto jevu spočívá ve vytvoření kompenzační tabulku pomocí středních hodnot interních testovacích pulzů [20]. 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 [11].

Medipix1 V roce 1998 byl vyvinut detektor *Medipix1* [22], také zvaný *Photon-Counting-Chip*, byl první velkoplošný detektor používající CMOS technologie. S maticí 64×64 pixelů, každým o hraně $170 \mu\text{m}$, má celkovou aktivní plochu $1,1 \text{ cm}^2$. I přes malý



Obrázek 2.8: Schéma pixelu detektoru *Timepix3* se společnou elektronikou pro 8 pixelů (tzv.*Super-pixel*) [20].

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

Detektor je vyrobený pomocí $1 \mu m$ 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* [15], 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 $55 \mu m$. Navíc má dva thresholdy s diskriminací na 4 pixely.

Stejně jako svůj 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* [16], 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 $55 \mu m$), 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 integraci náboje do clusteru 4×4 pixelů.

Každý pixel má dva 14-bitové čítače. Jednotlivé pixely můžou být naprogramovány tak, aby vždy měřily za pomocí 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 $55 \mu\text{m}$) na senzor s pixely o hraně $110 \mu\text{m}$, takže výsledný super-pixel má k dispozici 8 čítačů a pomocí různých úrovní thresholdů jej lze použít jako 8-kanálový spektrometr.

Stejně jako svůj 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* [20], jako nástupce *Timepix* detektoru. Rozměry byly zachovány, ale bylo použita $130 \mu\text{m}$ 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 kontinuálně z detektoru vyčítána (viz 2.4). Takto navržená architektura umožňuje vyčítat data až do $40M_{\text{hits}} * \text{cm}^{-2} * \text{s}^{-1}$ 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 obrázku vpravo je pak společná elektronika, sdílená vždy 8 sousedními pixely (tzv. *Superpixel*).

2.7 Vyčítací rozhraní

V této podkapitole bude popsáno několik vyčítacích rozhraní, které byly vyvinuty (v rámci *Medipix kolaborace*⁶) ÚTEF ČVUT v Praze a jeho spinoff společností ADVACAM s.r.o. Pro komunikaci s *Timepix3* detektorem bude v rámci této práce bude použito nově vyvinuté vyčítací rozhraní **Katherine** [5] 2.7.4.

2.7.1 FITPix

FITPix [13] vyčítací rozhraní bylo vyvinuto v roce 2010 a skládá ze z programovatelného hradlového pole (FPGA)⁷, USB 2.0 rozhraní, DAC⁸ a ADC⁹ převodníků a obvodů pro generování měřícího napětí (*bias*).

Zařízení je s detektorem propojeno pomocí *LVDS*¹⁰ 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.

⁶<<https://medipix.web.cern.ch>>

⁷Z angl. Field Programmable Gate Array

⁸Převodník digitálního signálu na analogový.

⁹Převodník analogového signálu na digitální.

¹⁰Z angl. Low-voltage differential signaling



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

FITPix je možné použít pouze jako nízkoúrovňové vyčítací rozhraní a veškerá business logika musí být implementována v připojeném počítači (na př. deserializace a derandomizace naměřených dat apod.). Jako řídící software je použit *Pixelman*[25]. Zařízení je kompatibilní se všemi detektory uvedenými v 2.6, kromě *Timepix3*.

2.7.2 AdvaDAQ

AdvaDAQ [26] je nástupcem vyčítacího rozhraní FITPix a kompatibilní se všemi detektory uvedenými v 2.6. Principem funkce vychází ze svého předchůdce, ale díky použitímu 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 [17, 3] je zařízení vyvinuté v rámci experimentu *ATLAS-TPX*, síti 16 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 počítače *Raspberry Pi*, ve kterém je implementován software, poskytující API pro své vzdálené řízení přes *TPC* protokol. Komunikace mezi *FPGA* a počítačem je realizována pomocí *SPI*¹¹ 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 na experimentech, kde je nutná větší vzdálenost mezi detektorem a vyčítacím rozhraním (u *Katherine* až 100 m). Toto omezení je dáno především nedostatečnou radiační a elektromagnetickou odolností použité elektroniky. To umožňuje aplikace v blízkosti jaderných reaktorů nebo na částicových urychlovačích (na příklad měření luminosity v rámci experimentu ATLAS na LHC v CERN [23]).

Zařízení je kompatibilní s *Timepix3* detektory, které jsou osazeny na CERN chipboard, nebo na kompatibilní PCB¹² s VHDCI¹³ 68-pinovým konektorem. S detektorem může být propojeno na přímo (viz obr. 2.9 vlevo), prodlužovacím VHDCI kabelem o maximální délce až 10 m, nebo speciálním extenderem na bázi ethernetu pro vzdálenost až 120 m. S rostoucí vzdáleností ale klesá maximální datový tok (viz tabulka 2.1).

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].

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

Pro úplnost výčtu vstupních a výstupních portů je třeba doplnit, že *Katherine* také disponuje GPIO¹⁴ konektorem (viz obr. 2.9 vpravo), zahrnující 4 obecné signály pro integraci s dalšími zařízeními a další rozšíření, jako na příklad pro trigger synchronizaci (pro možnost zapojení na příklad do detektorového teleskopu [6]). Zařízení dále disponuje LEMO konektorem (viz 2.9), poskytujícím vysokonapěťový zdroj ($\pm 300 V$) 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

¹¹Z anglicky Serial Peripheral Interface.

¹²Z anglicky Printed Circuit Board.

¹³Z anglicky Very-High-Density-Cable-Interconnect.

¹⁴Z anglicky General Purpose Input/Output.

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.

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*¹⁵ na server, kde jsou data 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 každé 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*¹⁶ protokolu. Protokol využívá dvou *UDP* portů - **řídícího** a **datového**.

Komunikační port je vyhrazen pro pro přenos řídících a konfiguračních paketů. Komunikace je implementována pomocí 36 příkazů, kde každý z nich se skládá 64-bitového datagramu pro request a 64-bitového datagramu pro response, takže se jedná o synchronní potvrzovanou komunikaci¹⁷. 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 **TODOref**) bude komunikační rozhraní vyčítacího rozhraní *Katherine* popsáno podrobněji.

¹⁵Z anglicky Secure File Transfer Protocol (protokol pro zabezpečený přenos souborů mezi počítači pomocí TPC protokolu).

¹⁶Z anglicky User Datagram Protocol.

¹⁷Potvrzování je realizováno na aplikační úrovni ISO/OSI modelu.

63	56	48	40	32	24	16	8	0
Command data								
0x0C	-	bias id	-					

} Odchozí datagram

63	56	48	40	32	24	16	8	0
Command data								
0x0C	-		bias (float)					

} Příchozí datagram

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

*KAPITOLA 2. ÚVOD DO HYBRIDNÍCH ČÁSTICOVÝCH PIXELOVÝCH
DETEKTORŮ*

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 vzájemné interakce. 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 vysoký 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 síř o více detektorech je schopná generovat.

Zde je možné namítnout, že každý systém je možné škálovat vertikálně¹. Zatímco cena škálování horizontálně škálovatelného systému je lineární závislost 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 rozhraní (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 [[17](#), [3](#)] pro řízení sítě detektorů *Timepix* [2.6](#), prostřednictvím vyčítacího rozhraní *ATLAS Pix* [2.7.3](#). Software však nevyhovuje požadavkům zmíněných výše:

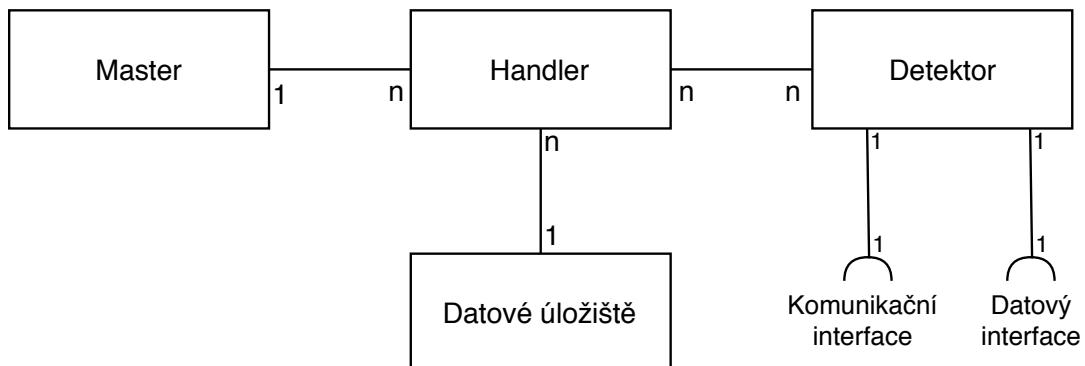
¹Škálováním v kontextu počítačových systémů rozumíme změnu vlastností daného systému za účelem zvýšení, nebo snížení jeho výpočetního výkonu (ev. jiného sledovaného parametru). Zatímco u vertikálního škálování měníme vlastnosti jednoho uzlu systému (například přidáváním procesorů, pamětí, kapacity úložiště apod.), u horizontálního škálování přidáváme jednotlivé uzly - samostatné jednotky (na př. 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 zpracovávány. 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 můžou nabývat hodnot omezených zadáným 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 můžou být ukládána do datového úložiště handleru a následně asynchronně nahrána do centrálního datového úložiště, nebo můžou být rovnou synchronně nahrávána do datového úložiště.

3.2.2 Datové úložiště

Datové úložiště by mělo být škálovatelné, protože pro sítě o větším počtu detektorů by ukládání dat do jednoho uzlu znamenalo omezení maximálního datového toku, dané kapacitou daného uzlu.

Jednou z možností implementace by mohlo být využití distribuovaného souborového systému, například *Hadoop Distributed File System* [21], který se v praxi² používá pro distribuované ukládání dat s možností jejich redundance na více uzlech (pro potřeby zálohování).

Další možnosti může být použití nějaké NoSQL distribuované databáze, 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 uloženy do datového úložiště, nebo zpracovány 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 detektorům homogenním způsobem, jak již bylo zmíněno v 3.1.

Aby mylo možné handlery, potažmo detektory, řídit centralizovaně, handler poskytuje API³. Pomocí API jsou jednotlivé handlery připojovány do systému, handlerům jsou přiřazovány a odebrány detektory, inicializuje se konfigurace detektorů a jí řízena akvizice dat.

²například v roce 2010 internetová společnost *Yahoo!* používala HDFS pro persistenci 25 PB podnikových dat [21].

³Z angl. *Application programming interface* (aplikativní 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 například uživatel přidává nový detektor do systému, tak nejprve skrze uživatelské rozhraní frontend aplikace zadá parametry detektora, včetně implementace komunikačního a datového rozhraní, jak je znázorněno na obr. 3.2. Poté pomocí je detektor nahrán do mastera pomocí jeho API, kde je následně persistentně 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 detektörů, informace o jejich stavu a o stavu jednotlivých handlerů bude navržena relační SQL⁴ 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, které budou popsány v této podkapitole.

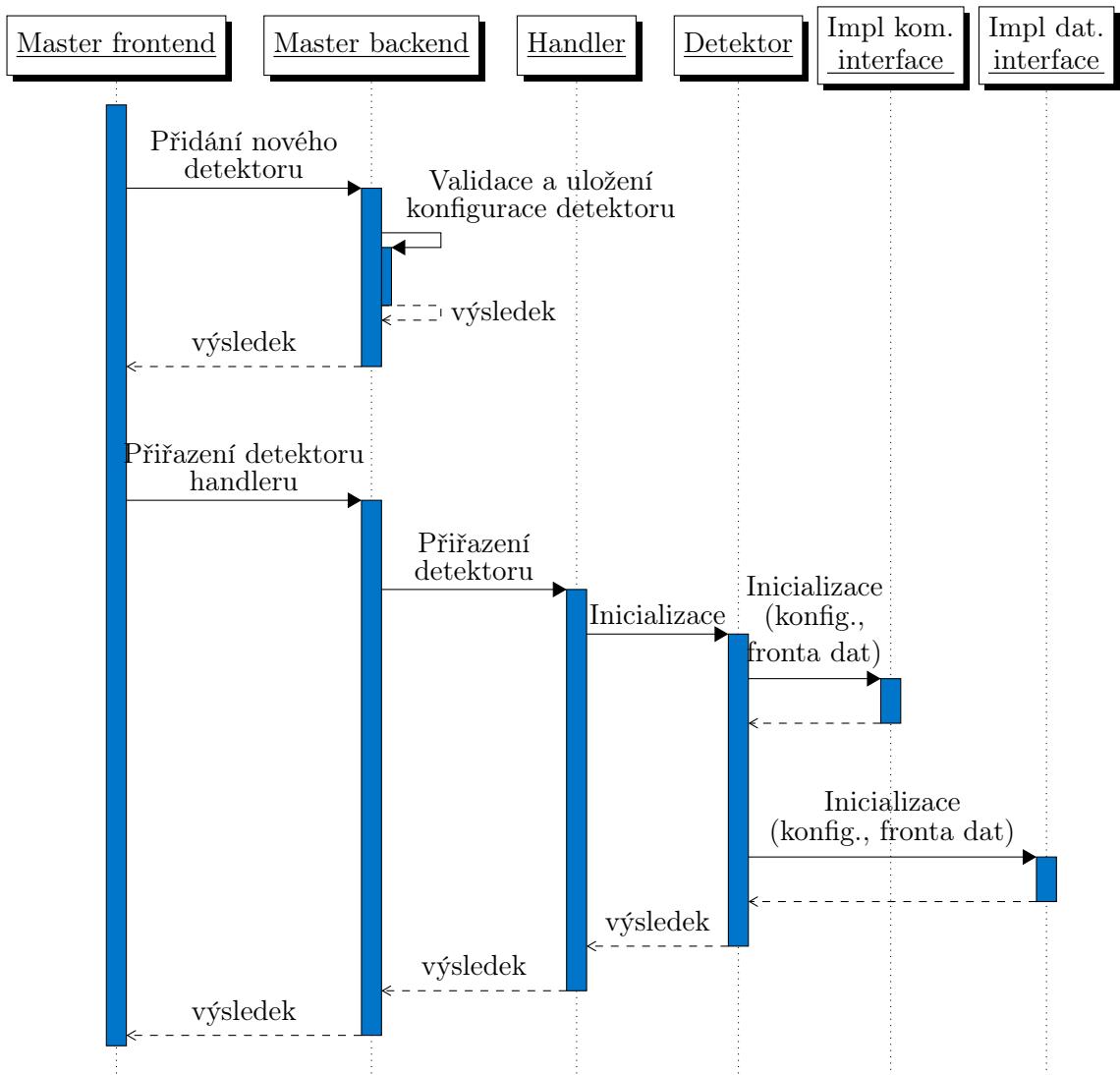
Na obrázku 3.3 je příklad sítě sestávající se ze dvou podsítí (tzv. *subnetwork*). Pod síť rozumíme takovou podmnožinu detektörů a handlerů, ve které libovolný detektor může být přiřazen libovolnému handleru. Pro úplnost je třeba doplnit, že nutnou podmínkou je, aby všechny handlery měly spojení s masterem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

Master se skládá ze tří komponent:

Backend serveru implementujícího business logiku pro komunikaci s handlery (jako například přiřazování detektörů, řízení akvizice dat apod.). Server zároveň poskytuje API pro možnou integraci s webovým frontend serverem se systémy třetích stran⁵, kterými může být plnohodnotně řízen.

⁴Z anglicky *Structured Query Language* (struktuovaný dotazovací jazyk).

⁵Například pro experiment ATLAS TPX v CERN je plánována integrace s DAC [4] (*Detector control system*), který umožňuje centrální řízení detektorových systémů, včetně integrace do bezpečnostního systému DSS (*Detector safety system*), datového akvizičního systému DAQ (*Data Acquisition System*) apod.

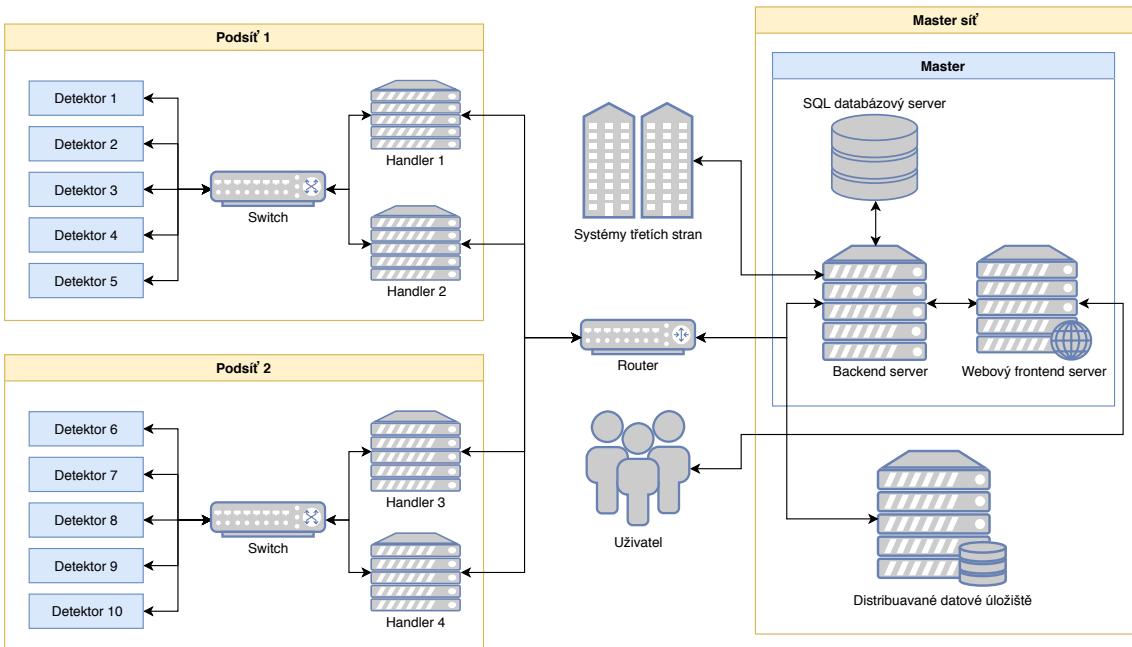


Obrázek 3.2: Sekvenční diagram znázorňující příklad přidání detektoru do systému a jeho přiřazení handleru.

Databázového serveru poskytující relační SQL databázi pro persistenci konfigurace systému, včetně implementace potřebných rozhraní, stavových informací sítě apod.

Frontend serveru poskytujícího 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 podsítí



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

Možnost škálování datového úložiště je závislá na instanci hardwarové architektury popsané výše. V [8] bylo ukázáno, že NoSQL databázové systémy (jako na příklad *MongoDB* nebo *Apache Cassandra*) jsou dobře horizontálně škálovatelné a jsou schopny zajistit dostatečnou rychlosť 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 budou stručně shrnutы 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 díky své jednoduchosti, výkonosti a přenositelnosti.

Java je interpretovaný programovací jazyk - při komplikaci kód není komplikovaný 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 interpretorem Javy - *JVM (Java Virtual Machine)*. Java není čitě interpretovaný programovací jazyk, protože v pozdějších verzích byla přidána podpora pro JIT (tzv. *Just-in-time*) komplikátoru, který umožňuje dynamickou komplikaci *bytecode* do strojového kódu dané platformy, díky čemuž můžu Java ve srovnání s neinterpretovaný my 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 komplikaci do Java *bytecode* může používat Java knihovny, takže část komplikovaného 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 uložena funkce), *extension functions* (podobně jako v C#, je možné přidat funkci do dané třídy bez nutnosti její modifikace, nebo vytvořené poddělené třídy) a další vylepšení.

Především díky výhodám popsaným výše, bylo v pozdější fázi vývoje rozhodnuto o použití Kotlinu, aby primárního programovacího jazyku pro handler a backend mastera. Také bylo experimentování 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 komplikátoru Kotlin kódů 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 jiné aplikace. 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 build⁶ systém, který vychází z *Apache Ant* a *Apache Maven* a používá DSL⁷ založeném na *Groovy* a *Kotlin*. Umožňuje automatizované stahování

⁶Sestavování počítačových programů.

⁷Z 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 J2EE⁸ aplikací a v rámci systému Pixnet bude použit v rámci implementace handlera a mastera. Spring framework je sada modulů, které nabízí 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 WAR¹⁰ na webový server, protože Spring Boot aplikace je zkompilovatelná do spustitelné JAR¹¹ 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 BSON¹² dokumentů. Mezi hlavní výhody *MongoDB* patří:

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

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

¹⁰Z angл. *Web Application Archive* (archív se zkompilovanou webovou J2EE aplikací).

¹¹Z angл. *Java Archive* (archív se zkompilovanou Java aplikací).

¹²Binární 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* [12]. *Sharded cluster* je pak množina uzlů s jednotlivými *shardy*, kde každý obsahuje podmnožinu *shardovaných* dat.

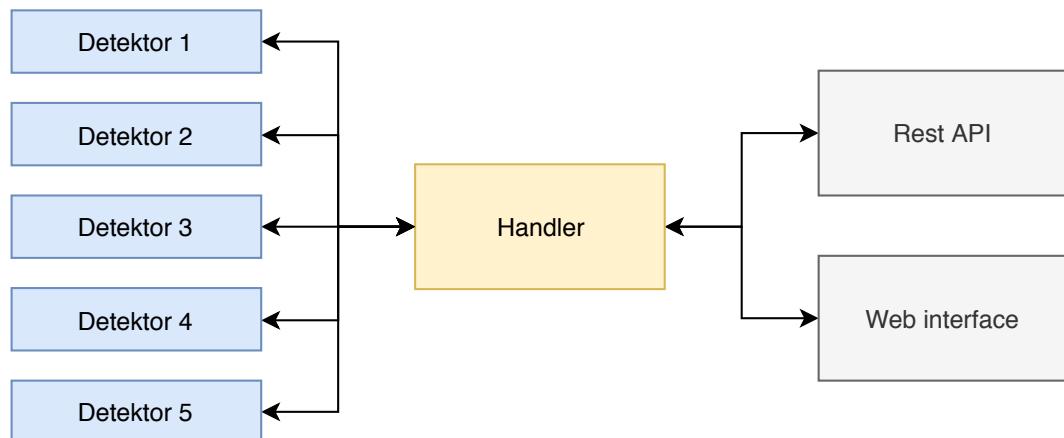
Kapitola 4

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.

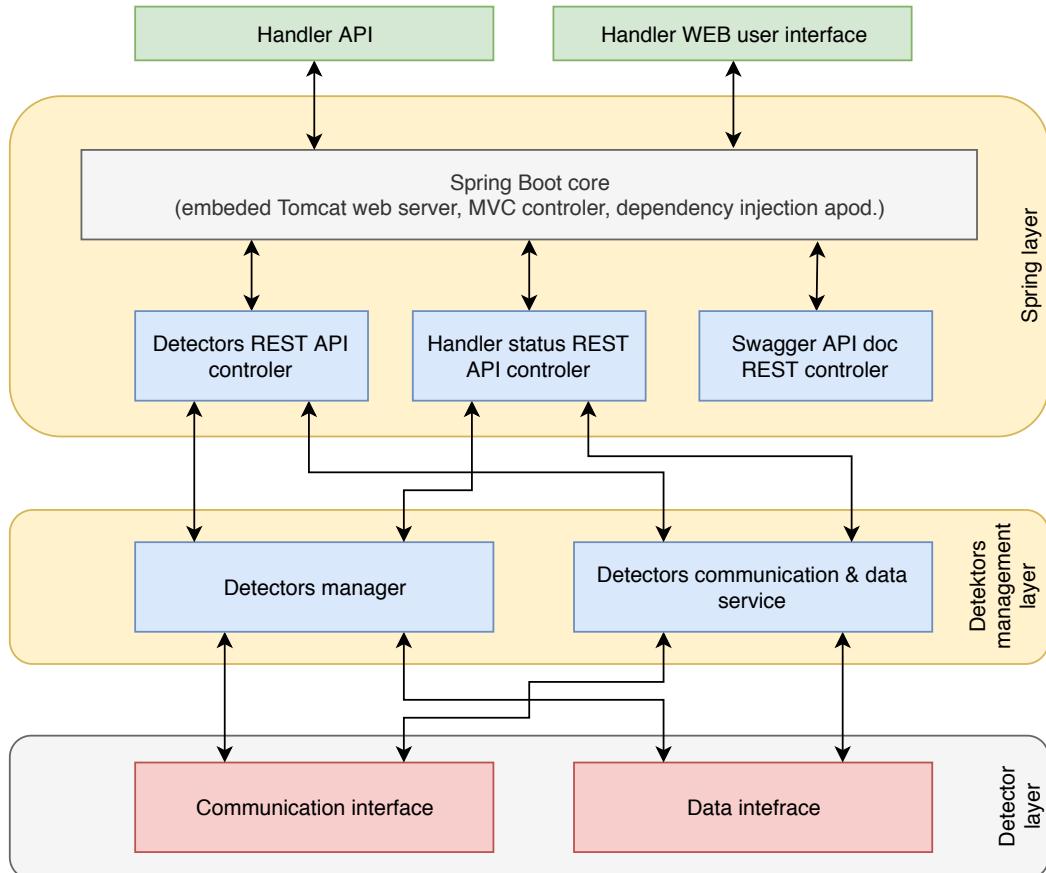
Na obrázku 4.1 je znázorněn příklad instance handleru z pohledu vstupu (pět přiřazených detektorů) a pohledu výstupu (poskytované REST API a webové rozhraní.)



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. Budou zde představeny jednotlivé její 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 zmíněných 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 má čistě informativní charakter 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 být do něj 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. Avšak je doporučeno, aby zvolený formát byl strojově i lidsky čitelný, z důvodu jeho snazší editace². Takový formát může být na příklad `JSON`³, nebo `YAML`⁴, který je použit pro serializace konfigurace komunikačního modulu Katherine (viz **TODO**přidat ref.).

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:

¹Datový typ obsahující textový řetězec

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

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

⁴<<http://yaml.org/>>

```

1  interface DetectorComm {
2
3      fun getDetectorType(): DetectorType
4      fun getReadoutName(): String
5      fun getSensorsCount(): Int
6      fun getDetectorWidth(): Int
7      fun getDetectorHeight(): Int
8      fun setDetectorConfig(config: String)
9      fun getDetectorConfig(): String?
10     fun getSupportedValueCommands(): List<AbstractValueCommand>
11     fun getSupportedExecutionCommands(): List<AbstractExecutionCommand>
12     fun getAcceptedFilesKeys(): List<String>
13     fun getDataFrameQueue(): BlockingQueue<AbstractDataFrame>
14
15    fun isConnected(): Boolean
16    fun connect(): Boolean
17    fun disconnect(): Boolean
18
19    fun executeSetValueCommand(commandID: Int, payload: ValuePayload)
20    fun executeGetValueCommand(commandID: Int): ValuePayload
21    fun executeExecutionCommand(commandID: Int, input: Map<String,
22        ↪ ValuePayload>): Map<String, ValuePayload>
22    fun uploadFile(fileKey: String, file: ByteArray)
23    fun setCallback(callback: Callback)
24
25    interface Callback {
26        val classLoader: ClassLoader?
27    }
28
29 }
```

Zdrojový kód 4.2: Komunikační interface detektoru, napsané v jazyce Kotlin (viz [3.5](#)).

(i) ValueCommand obsahuje atributy pro 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 spojité (hodnota může nabývat jakékoli hodnoty ze zadáного 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 intervalom $-300, 300$.

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 vykonať 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 příkazů 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) a ani jejich množstvím. Tento přístup tedy umožňuje definovat příkazy, které mají 0 až m vstupních hodnot a 0 až n výstupních hodnot. Obdobně jako *AbstractValueCommand*, i *AbstractExecutionCommands* má v poskytované knihovně dvě implementace:

- (i) *ExecutionCommand* je definován obdobně jako *ValueCommand* - má své ID, jméno, ale 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 *Fast VCO* přepínač) a že žádné hodnoty nevrací.

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

```
1 executionCommands.add(ExecutionCommand(
2     KatherineExecutionCommands.SET_ACQ_MODE.internalID, // ID (Int)
3     "Set acquisition mode", // name
4     // model vstupních hodnot
5     arrayOf(
6         ValueModelVerbose(
7             "Acquisition mode", // název hodnoty
8             ValueId.acq_mode.name, // ID hodnoty (String)
9             // celočíselný diskrétní model hodnot
10            IntValueModel(mapOf(
11                Pair("ToA & ToT", 0),
12                Pair("ToA", 1),
13                Pair("Event & iToT", 2)
14            )),
15            ValueUnit.DIMENSIONLESS // jednotka hodnoty
16        ),
17        ValueModelVerbose(
18            "Fast vco enabled",
19            ValueId.fast_vco_en.name,
20            BooleanValueModel(mapOf(
21                Pair("Enable", true),
22                Pair("Disable", false)
23            )),
24            ValueUnit.DIMENSIONLESS
25        )
26    ),
27    null // model výstupních hodnot
28))
```

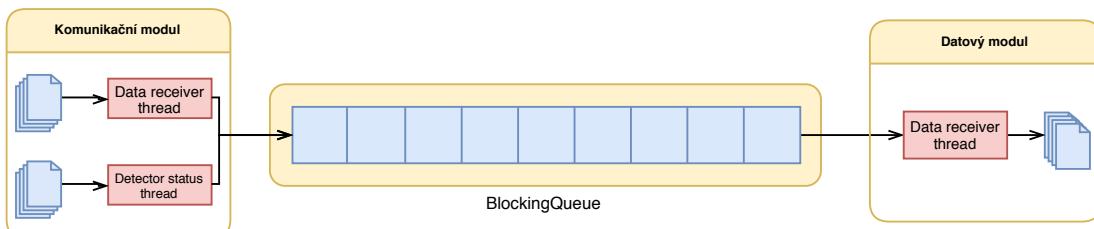
Zdrojový kód 4.4: Příklad definice `ExecutionCommand` pro nastavování akvizičního módu detektoru s vyčítacím rozhraním *Katherine* (viz 2.7.4). Z příkladu je patrné, že vstupní model je tvořen dvěma hodnotami a výstupní model je prázdný.

Pro vykonání `ExecutionCommand` slouží metoda `executeExecutionCommand()`, viz řádek 21 komunikačního interface (zdrojový kód 4.2). Metoda akceptuje ID příkazu a mapu (tzn. seznam páru 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. L0B⁵). 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 fonty je implementováno jako blokující operace (tzn. že zablokuje čtecí vlákno, než bude ve frontě nějaký element v 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).

⁵Z anglicky Large Object.

⁶Z anglicky Java Collections Framework.

⁷Implementace `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 poli) 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, které 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.

⁸Z angл. *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 Service⁹.

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 endpointy¹⁰ 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 použil 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](#).

⁹Dle [7] *Service* je množina operací, která není součástí modelu a zároveň není nositelem stavu.

¹⁰Označení 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	Odstaní detektor podle zadaného ID
POST	/connect	Prověde připojení zvoleného detektoru
POST	/disconnect	Prověde 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.

```

1  {
2      "data": {
3          "version": "0.0.0.1",
4          "status": "ONLINE",
5          "detectors": [
6              {
7                  "id": "emulator_katherine",
8                  "name": "katherine emulator",
9                  "status": {
10                      "connection": "ONLINE",
11                      "measurement": "UNKNOWN"
12                  }
13              },
14              {
15                  "id": "katherine",
16                  "name": "katherine",
17                  "status": {
18                      "connection": "UNKNOWN",
19                      "measurement": "UNKNOWN"
20                  }
21              }
22          ]
23      },
24      "success": true
25  }

```

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.

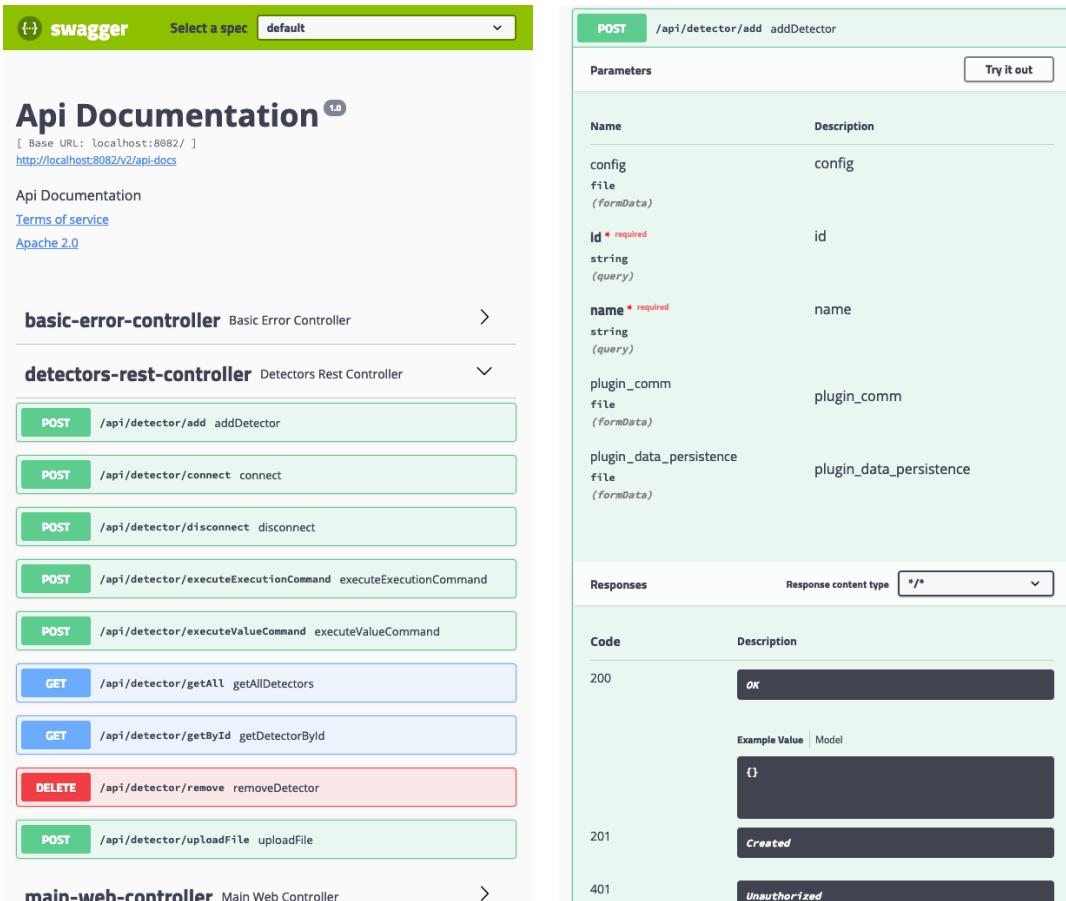
Za tímto účelem byla implementována knihovna *Springfox* [14] - nástroj pro generování strojově i lidsky čitelné dokumentace. *Springfox* je založen na *OpenAPI*¹¹ specifikaci a *Swagger*¹² frameworku. *Springfox* po spuštění aplikace automaticky analyzuje její komponenty a jednotlivé její třídy pomocí Java reflexe. Na základě analýzy potom vytvoří sémantický model jednotlivých endpointů.

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

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.

dpointu (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.

The screenshot shows the main API documentation interface. It features a sidebar with sections like 'basic-error-controller' and 'detectors-rest-controller'. Under 'detectors-rest-controller', there are several listed endpoints: POST /api/detector/add addDetector, POST /api/detector/connect connect, POST /api/detector/disconnect disconnect, POST /api/detector/executeExecutionCommand executeExecutionCommand, POST /api/detector/executeValueCommand executeValueCommand, GET /api/detector/getAll getAllDetectors, GET /api/detector/getById getDetectorById, DELETE /api/detector/remove removeDetector, and POST /api/detector/uploadFile uploadFile. The 'removeDetector' endpoint is highlighted with a red background.

(b) Detail endpointu.

The screenshot shows a detailed view of the 'addDetector' endpoint under the 'detectors-rest-controller'. It includes a table for parameters, a table for responses, and a section for example values.

Name	Description
config file (formData)	config
Id * required string (query)	id
name * required string (query)	name
plugin_comm file (formData)	plugin_comm
plugin_data_persistence file (formData)	plugin_data_persistence

Code	Description
200	OK
201	Created
401	Unauthorized

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>>).

#	Connection status	Measurement status	Readout
emulator_katherine	ONLINE	UNKNOWN	katherine
katherine	UNKNOWN	UNKNOWN	katherine

Obrázek 4.5: Screenshot webového rozhraní handleru.

4.2 Konfigurace a nasazení

Pro spuštění handleru je třeba mít nainstalované JRE 8¹³, 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 pojmenování handleru.

Aplikaci je tedy možné spustit například takto:

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

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

Zdrojový kód 4.8: YAML konfigurační soubor handleru.

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

Kapitola 5

Katherine rozhraní

TODO

Kapitola 6

Master

TODO

Kapitola 7

Testování

TODO

Kapitola 8

Závěr

TODO

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] EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. New Jersey, USA : Pearson Education (US), 2003. ISBN 978-0321125217.
- [8] GANDINI, A. et al. Performance evaluation of NoSQL databases. In *European Workshop on Performance Engineering*, s. 16–29. Springer, 2014.
- [9] 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 2008 Proceedings of the 10th International Workshop on Radiation Imaging Detectors.
- [10] 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).
 - [11] Kolektiv autorů Medipix kolaborace. *Web Medipix* [online]. 2018. [cit. 1. 10. 2018]. Dostupné z: <<https://medipix.web.cern.ch>>.
 - [12] Kolektiv autorů MongoDB. *MongoDB at Scale* [online]. 2018. [cit. 16. 12. 2018]. Dostupné z: <<https://www.mongodb.com/mongodb-scale>>.
 - [13] 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>>.
 - [14] KRISHNAN, D. – KELLY, A. *Springfox Reference Documentation* [online]. 2018. [cit. 27. 12. 2018]. Dostupné z: <<http://springfox.github.io/springfox/docs/current>>.
 - [15] 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.
 - [16] 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} 2007 Proceedings of the 11th International Vienna Conference on Instrumentation.
 - [17] 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.
 - [18] 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.
 - [19] PLATKEVIC, M. *Signal Processing and Data Read-Out from Position Sensitive Pixel Detectors*. PhD thesis, Czech Technical University in Prague, Czech Republic, 2014.

- [20] 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>>.
- [21] 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.
- [22] 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](https://doi.org/10.1016/S0168-9002(03)01648-6). Dostupné z: <<http://www.sciencedirect.com/science/article/pii/S0168900203016486>>. Proceedings of the 4th International Workshop on Radiation Imaging Detectors.
- [23] 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.
- [24] 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.
- [25] 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>>.
- [26] 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>>.

LITERATURA

Příloha A

Seznam použitých zkratek

ADC Analogově digitální převodník

Al Aluminium

API Application Programming Interface

ASIC Application-specific Integrated Circuit

ATLAS A Toroidal LHC Apparatus

B byte

b bite

BPMN Business Process Model and Notation

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

CSM Charge Summing Mode

DAC Digitálně analogový převodník

DCS Detector Control Systems

DPS Deska plošného spoje

eV elektronvolt

FITPix Fast Interface for Timepix Pixel Detectors

FPGA Field Programmable Gate Array

FSM Finite State Machine

PŘÍLOHA A. SEZNAM POUŽITÝCH ZKRATEK

FWHM Full width at half maximum
GaAs Arsenid gallitý
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure
HW Hardware
IP Internet Protocol
JSON JavaScript Object Notation
LED Light Emitting Diode
LHC Large Hadron Collider
LiF Lithium fluoride
LS Long Shutdown - dlouhodobá technologická přestávka LHC
LVDS Low-voltage differential signaling
MPX Medipix
PC Personal Computer
PCC Photon Counting Chip
PE Polyethylen
PN Přechod polovodiče typu P a polovodiče typu N
REST Representational State Transfer
RS232 Standart sériové linky
SMD Surface Mount Technology
SPI Serial Peripheral Interface
SPM Single Pixel Mode
SQL Structured Query Language
SSH Secure Shell
SW Software
TCP Transmission Control Protocol
TDAQ Trigger and Data Aquisition
TOA Time of Arrival - mód detektoru (viz ??)

TOT Time Over Threshold - mód detektoru (viz ??)

TPX Timepix

URL Uniform Resource Locator

USA15 Serverová místnost ATLAS experimentu

USB Universal Serial Bus

UX15 Označení prostor s ATLAS detektorem (tzv. cavern)

ÚTEF Ústav technické a experimentální fyziky

PŘÍLOHA A. SEZNAM POUŽITÝCH ZKRATEK

Příloha B

Obsah přiloženého CD

```
CD/
└── atlas_tpx/
    ├── dokumentace/ – adresář obsahující popis API
    ├── exe/
    │   └── emulator/
    │       ├── readme.txt – README soubor
    │       └── AtlasPixEmulator-0.1.jar – spustitelný jar soubor emulátoru
    └── atlas_tpx_server/
        ├── readme.txt – README soubor
        ├── AtlasTPX.server-0.1.jar – spustitelný jar soubor serveru
        ├── server-configuration.yml – soubor s konfigurací serveru
        └── detectors.csv – tabulka s detektory

    └── src/ – adresář se zdrojovými kódy

└── kalibrace/
    ├── exe/ – spustitelné binární soubory
    ├── libs/ – jar knihovny
    ├── test_data/ – vstupní data kalibrace (spektra)
    ├── readme.txt – README soubor
    └── X-rayTimepixCalibration.jar – spustitelný jar soubor

    └── src/ – adresář se zdrojovými kódy

└── text/
    ├── LaTeX/ – adresář se zdrojovými soubory tohoto dokumentu
    ├── thesis-bejerjak-2016.pdf – tato práce ve formátu PDF
    ├── abstract_cz.txt – abstrakt česky
    └── abstract_en.txt – abstrakt anglicky
```

Obrázek B.1: Obsah přiloženého CD