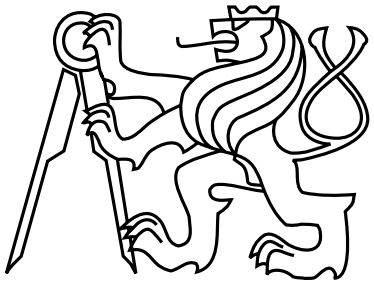
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Bachelor's Project

# Interactive visualization system for hybrid active pixel detectors within the ATLAS experiment at CERN

*Petr Mánek*

Supervisor: Ing. Stanislav Pospíšil, DrSc.

Study Programme: Open Informatics

Field of Study: Computer and Information Science

April 23, 2016

# Aknowledgements

Zde můžete napsat své poděkování, pokud chcete a máte komu děkovat.

# Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on May 15, 2016 .....

# Abstract

TODO

# Abstrakt

TODO

# Contents

# List of Figures

# List of Listings

# Chapter 1

# Introduction

## 1.1 About Timepix Detectors

TPX detectors are hybrid active pixel detectors, developed within the MPX collaboration at CERN. They consist of an active sensor layer bump-bonded to a readout ASIC. The ASIC divides the active sensor area into a square matrix of $256 \times 256$ pixels with a pixel-to-pixel distance of 55 µm. Each pixel has its own readout chain and can be controlled indenpendently. While the sensor layer material in the presented work was silicon, other sensor materials are available, most notably CdTe and GaAs, which are used e.g. for imaging applications.

TPX detectors are operated in a way that is similar to commercially available cameras. What would be a picture in photography, is referred to as *a frame*. Every pixel is equipped with a 14-bit integer register called *the counter*. When acquisition starts, registers is set to zero, and then possibly incremented upon every interaction. A frame thus represents the status of each pixel after the set *acqusition time*. Returning to camera analogy, acquisition time resembles exposure time of a photograph—when increased, more particles are to be expected interacting with detector's pixels.

Since pixels may not be identical due to material irregularities and manufacturing errors, every pixel has adjustable *threshold* parameter, which is subject to calibration. In a calibrated state, analog input measured from the pixel's semiconductor should exceed this threshold only when the pixel is interacting with a particle.

### 1.1.1 Operation Modes

Depending on their application, TPX detectors can be manufactured to contain multiple active sensor areas. After acquisition is finished, every area produces a matrix of $256 \times 256$ integer values. Interpretation of these values depends on another parameter, *the operation mode*. The following operation modes are available:

**Hit Counting Mode (also known as the Medipix Mode)** The counter is incremented upon every transition from a state below the threshold to a state above the threshold. Pixel value represents the number of particles which have interacted with the pixel.

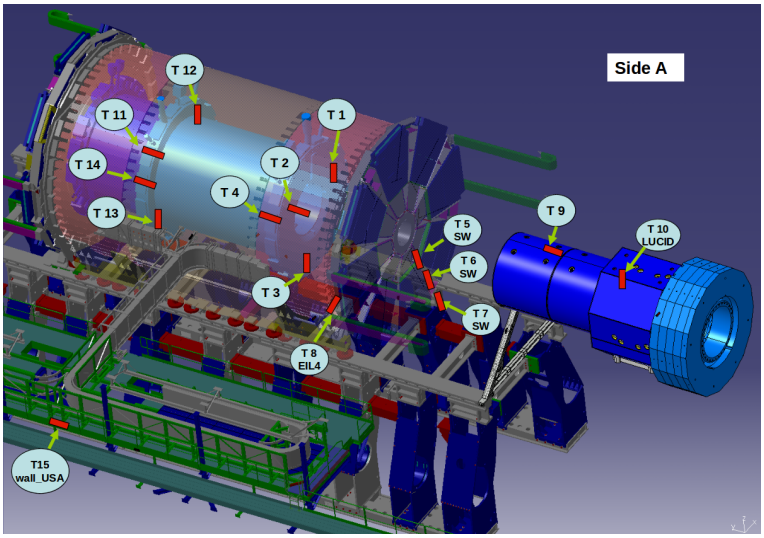



Figure 1.1: TPX detectors installed within the ATLAS machine at CERN.

**Time over Threshold Mode (TOT)** The counter is incremented in every clock cycle spent above the threshold. Pixel value corresponding to the energy of the interacting particle. Energy calibration methods are described in [4].

**Time of Arrival Mode (TOA)** The counter is incremented in every clock cycle after the threshold is first exceeded. Pixel value corresponds to the length of time interval before the end of the measurement.

## 1.2 ATLAS-TPX Network

In the ATLAS experiment at CERN, a network of 15 TPX detectors (see Figure 1.1) has been installed during the recent shutdown (LS1). It is a successor of a MPX detector network of similar architecture, which has been operated for several years by IEAP researchers.

Its upgraded capabilities include improved measurements of the ATLAS machine luminosity and activation of materials surrounding the detectors, better characterization of the radiation field at various locations withing the cavern an 2-layer design with different modes of operation allowing to reliably separate charged particles from neutral particles.

### 1.2.1 Read-out Interface

A read-out interface is a special dedicated hardware device that reads data and controls acquisition of the detector. [7] Given the harsh radiation environment within the ATLAS machine, the ATLASPIX interface was developed by modifying a regular FITPix interface.

The interface has two parts connected by four cables. The detector itself is positioned and oriented within the ATLAS machine, whereas the rest of the interface is placed in a nearby

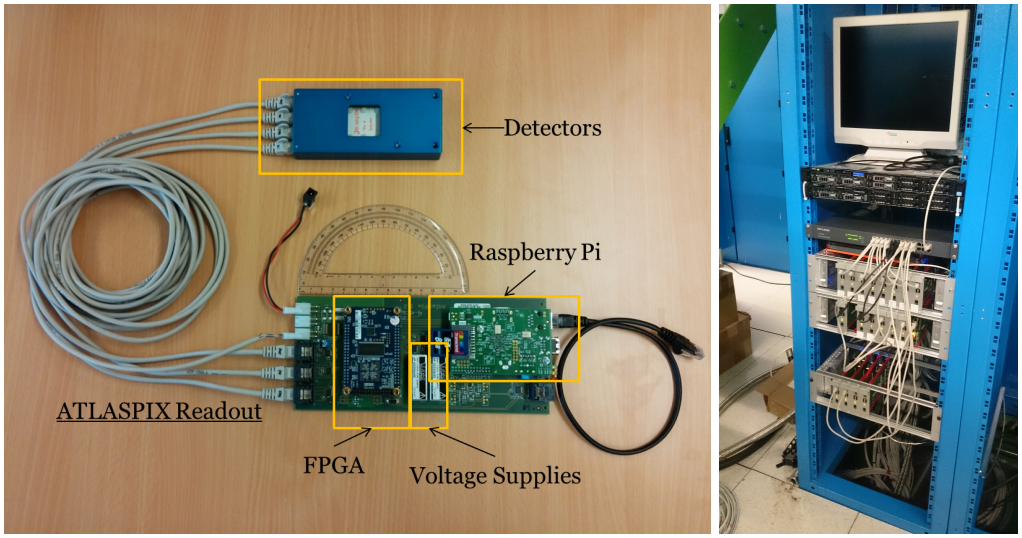



Figure 1.2: The ATLASPIX read-out interface installed at CERN.

server room, shielded against ionizing radiation. Cables connect both parts, allowing protected hardware to control detectors remotely during operation of the machine. To manage multiple detectors simultaneously, a computer is directly connected to all read-out interfaces. This computer, also known as *the control PC*, gathers all measured data and forwards commands from the system operator to the detectors through the ATLASPIX interface. This configuration is shown in Figure 1.2.

At the time of writing this work, the control PC is being operated manually from a remote location. The automation of the operation is under investigation (for more information, see section 5.1.2).

### 1.2.2 Cluster Analysis

In ATLAS-TPX detector footage, components of various shapes and sizes can be observed, depending on the experiments performed at the time of acquisition. These components, commonly known as *clusters*, are discovered and evaluated in an automated process called *the cluster analysis*. This procedure involves a connectivity-checking algorithm, such as *flood-fill*, operating on pixel matrices to distinguish individual clusters. In later stages, clusters are processed, measured and classified in various categories with regards to their shape. In addition, counter values in frames captured in the TOT mode can be combined with earlier calibrations to produce energy approximations. [4]

The output of cluster analysis consists of two separate lists of clusters, one per every sensor layer. It follows from the definition of a cluster that any pixel contained in it has a non-zero counter value. Conversely, all pixels unreferenced by any cluster are assumed to be equal to zero. The utilized technique of data encoding is well-known as it offers efficient compression rate for sparse pixel matrices. It is however worth noting at this point that in certain cases (represented most notably by saturated or nearly saturated frames), this

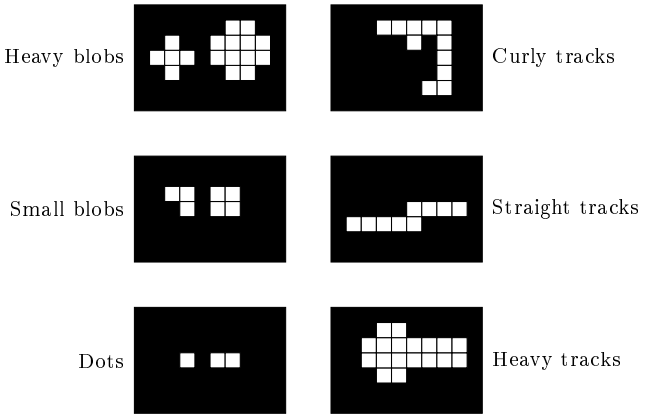



Figure 1.3: Different cluster types classified by their shapes.

approach generates voluminous data structures, which may take long time to enumerate, and in turn slow down other algorithms operating on them.

In a cluster list, pixels are stored as tuples of their Carthessian coordinates and their respective counter values. From this information, the pixel matrix can be reconstructed at any time. The original pixel matrix is therefore discarded without data loss at the end of cluster analysis, in order to minimize occupied space. For every cluster, several properties are calculated in the automated processing, most notable of which are:

**Shape Classification** By measuring geometric properties of a cluster (such as radius or size), it is possible to estimate whether the cluster resembles more a line segment or a circular blob. Similarly, an algorithm can ascertain if the cluster looks thin or thick. From that information, type of interacting particle can be determined, along with direction of its movement relative to the plane of incidence.

**TODO**

**Size, Volume** The size of a cluster is equal to the number of connected pixels which constitute it. The volume is a sum of counter values of those pixels.

**Centroid, Volumetric Centroid** The centroid is defined as an unweighted mean of pixel coordinates in the cluster. In analogous way, the volumetric centroid is the very same value weighted by corresponding counter values.

**Minimum and Maximum Cluster Height** These two figures refer to the lowest and the greatest counter values of pixels within the cluster.

**Energy-based Properties** ***(available only in TOT mode)*** If energy approximations are available, many of the values mentioned above can be also calculated with the energy substituted for counter values.

## 1.3 Common Data Storage Formats

Provided that every detector in the ATLAS-TPX network contains 2 active sensor layers, each with a square pixel matrix of size $256 \times 256$, a single captured frame consists of 131,072 integer values in total[1]. Given the fact that in the network, 15 detectors continuously perform acquisition as fast as 10 or 12 times per second, large amounts of data are produced. This section lists the most common file formats, in which such data is stored and archived for purposes of further research.

### 1.3.1 Plain Text

The most straightforward way of storing data acquired by TPX detectors is to use plain text files. Such output, referred to as *the single-frame format* (or possibly *the multi-frame format* depending on the number of frames stored), encodes data in three files per unit of acquisition.

**Data File** Data files contain captured data from individual pixels of the detector. The data is encoded as a simple list of tuples containing pixel positions and their respective counter values. All pixels which are not mentioned are assumed to be of zero value.

**Description File** Description files contain configuration of the detector at the time of acquisition. While there is no exhaustive definition listing every serialized parameter, description files allow to be easily extended, as they directly annotate all values of parameters they store.

To store a configuration parameter, three lines of text are required. The display name of the parameter (along with the unit or any other notes) is written on the first line. The second line describes the data type of the value and its range. The third line contains the actual value.

**Index File** Index files contain binary information, which binds data files and description files together. In an index file, every frame is represented by a tuple of data addresses pointing to the first entry of the frame in the corresponding data file and the first entry in the description file.

Even though the plain text format has the advantage of being easily accessible with any text editor, it is disk-inefficient in terms of access speed and disk space.

### 1.3.2 ROOT Framework

Another storage option is to use a proprietary data file format defined by the ROOT Data Analysis Framework [1]. Originally concieved at CERN in 1995, the framework provides a set of powerful tools with various applications in data mining, manipulation and visualization. Unlike other similar toolkits, ROOT comes with its own machine-independent binary file format (identified by the `.root` extension). This format is designed to store enormous

[1] Not including configuration information of the detector.

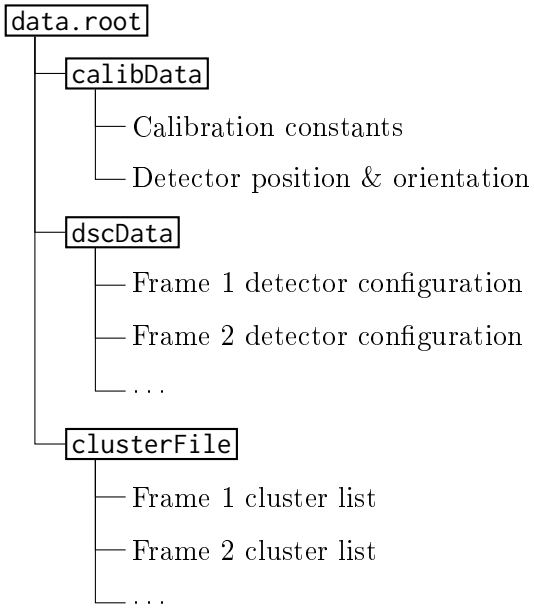



Figure 1.4: Structure of a ROOT file containing TPX footage.

amounts of data within various types of data structures efficiently, while maintaining good overall performance by employing low-level memory optimization techniques and multi-tier content caching.

Used by many physicists at CERN for several years now, ROOT was chosen as the data archivation format as many researchers have already learned its properties and know well how to operate it despite often lacking deeper background in Computer Science. For the purposes of programmatic access, ROOT also does well with documented APIs in Python, R and C++.

Should data be stored in ROOT, a basic relational database concept comes to mind. ROOT however offers even more abstract data structures with standard tables generalized in the form of *trees* and their columns in the form of *leaves*. One such tree would suffice for information about captured frames (such as acquisition time, operation mode, etc.) and other for a list of clusters for every frame. This schema (showcased in Figure 1.4) would efficiently abstract the entire storage structure, allowing for multiple frames to be stored in a single file, grouped for instance by a common time interval, similarly to the text file format.

In spite of being over 20 years in development, ROOT is not perfect. Using memory monitoring tools such as [5], we have confirmed that the C++ implementation of the ROOT framework is riddled with various memory leaks, making it unsuitable for time-extensive operations. Some might also argue that a its tree data structure offers too much versatility, thus being an overly-complicated storage solution for a simple output described in the previous sections. Lastly, ROOT framework has quite a complex object structure, making it difficult to learn for first-time users.

### 1.3.3 Data Manipulation Problem

With large amounts of data generated periodically by the ATLAS-TPX network, an obvious question is raised: How to efficiently read, process and display captured TPX footage while not being overwhelmed by the magnitude of data in question? Answering it entails addressing several subsidiary concerns, mostly regarding the choice of data structures, their arrangement and roles within the system.

There exist many computing approaches and techniques to deal with big data, as well as myriad of peculiar data strucutres, each optimizing different of its properties and operations. With respect to the architecture of the network, the format of data it produces, other works done on its predecessor and goals set forth by its users, this work tackles the problem by standard means. Its purpose is to design and implement a network-based system capable of archiving, accessing and visualizing data, while optimizing data retrieval speeds for certain types of user queries.

## 1.4 Structure of This Document

This section is intended as a brief outline of the rest of this work.

Chapter 2 formally defines all data structures, on which the system operates. Besides stating requirements on the storage facilities and listing assumptions about user queries, it introduces an auxiliary data structure, capable of accelerating access procedures for a certain types of data requests.

Chapter 3 is dedicated to the definition of the JSON Timepix protocol (JSTP), which is later used to transmit archived data to the visualization system over HTTP. The protocol is however not limited only to data visualization, but offers API for other applications as well.

Chapter 4 describes implementation and integration of concepts defined by the previous two chapters in the form of a server application. Apart from commenting on its internal structure, it also provides insights in its operation and contains UI documentation.

In the last chapter of this work, the procedure of data import is described. The last section outlines several applications of the software outside the ATLAS-TPX project and discusses its possible future.

The appendices mostly contain descriptive sections of technical nature, such as code listings, API documentation or the description of contents of CD attached to the printed version of the work.

# Chapter 2

# Data Structure and Storage

In this chapter, a data structure capable of archiving TPX footage for longer time periods is proposed. The motivation is to minimize access time in queries based on certain parameters. To help in that accomplishing that goal, the index database is introduced.

## 2.1 Formal Requirements

Requirements on the data storage system are as common as database requirements can get. It shall be a reliable permanent storage element, accessible for reading from multiple workstations at a same time and robust enough to withstand minor hardware failures. With 15 detectors already installed at ATLAS, and possible option of installing another 5, the database should be designed to hold frames from up to 20 TPX devices for the entire expected time period of their operation at LHC (that from June 2015 to LS3 in 2021).

**TODO**

As more and more frames arrive from the detector network, the database must allow to be periodically extended with new data, possibly processing and converting pixel matrices into cluster lists, as described in the previous sections. Since the database is going to be primary storage site for all research data, it shall have multiple independent copies and its structure should be designed with logic to enable their periodic synchronization.

Apart from all requirements already listed, it is important to mention that the anticipated structure of the majority of user queries is known. With regards to this information, data storage and retrieval procedures may be optimized to accelerate such queries.

It was determined that the most queries are going to filter data by time of acquisition and by device of origin. This is indeed a very natural approach, provided that every detector in the network is positioned and oriented in way allowing only for a certain type of particles to be observed. Researchers looking for signs of specific particles might often request data based on other experiments, which were conducted in a determinate time period and only affected detectors at specific locations within the ATLAS machine.

## 2.2 Database

In this section, the TPX footage database is defined. Accounting for the ever-growing nature of our data, the database is separated into two parts. The first part is to contain data which has already been processed by the cluster analysis, and is ready to be accessed by users. The second part is to contain data which has arrived from CERN in its raw form but hasn't been processed yet. As one might observe, this separation of data serves a fundamental purpose, that is to distinguish intermediate products from finished ones.

### 2.2.1 Definition

For the sake of compatibility, database is to be based on a UNIX file system. Many users, not necessarily only those using UNIX-based operating systems, may be to access it directly by any of widely-used and standardized network protocols such as FTP, SMB, SSH, AFP or HTTP. Utilization of these protocols contributes not only to the universality of the database, it also resolves shared resource access and other data concurrency issues. Some of these features may prove to be useful later on when synchronizing various storage sites in order to back up or restore data. In addition, UNIX file systems also offer fundamental security features, allowing administrators to grant read-write privileges to a certain group of users, while limiting others to a mere read-only access.

The stored data shall be grouped in two directories named `processed` and `downloading`, corresponding to the respective sections of the database. In these directories, data is further divided in subdirectories by the device of origin. To make navigation easier, device directory names use numeric identifiers in compliance with already published literature. For instance, all data originating from the detector no. 7 would be stored in a directory named `ATPX07`. In such directory, footage would be stored in time-coded files (or directories, should multiple files be grouped under single time code) according to the naming pattern: `[yyyy]_[mm]_[dd]_ATPX[id]` (where `[id]` represents the device identifier and `[yyyy]`, `[mm]`, `[dd]` represent year, month and day of acquisition respectively).

Should it be impractical to group frames by the day of acquisition, file and directory names may follow an alternate naming pattern with hourly granularity: `[yyyy]_[mm]_[dd]_ATPX[id]_[hh]` (where `[hh]` represents the hour of acquisition and other entities are treated as in the previous pattern). Note that in spite of grouping data files in separate subdirectories by the device of origin, the device identifier is intentionally included in the naming pattern for reasons of redundancy.

The directory structure described so far satisfies all requirements from the previous section. What's more, it optimizes access to data generated from specific devices at specific times, so that the majority of user requests is handled in timely manner.

All data files are stored at the lowest level of the directory structure under time-coded names according to our naming patterns (see Figure 2.1). Should more files fall under the same time code (marginal scenario), they are to be grouped in a directory with a time-coded name. File structure in such a directory is undefined. Although it is not required, it is expected that files in the `processed` directory are encoded in the ROOT format, whereas file in the `downloading` directory are encoded in plain text, as that is the initial format of all

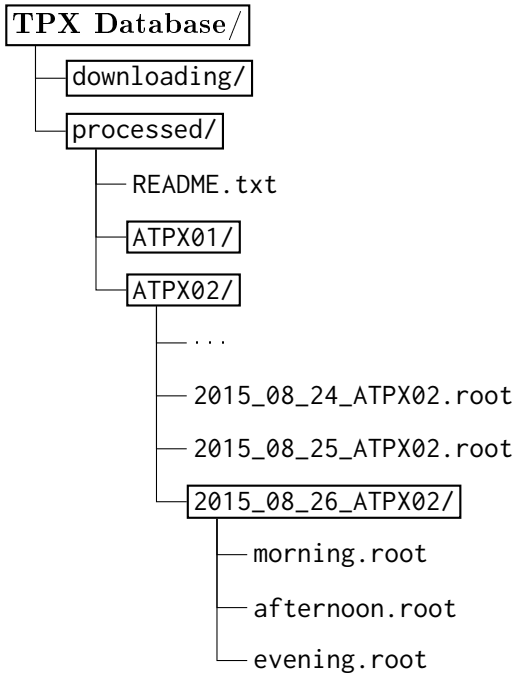



Figure 2.1: Example of the database file system structure.

unprocessed footage. Apart from these two formats, files of different types will be tolerated, but regarded as secondary.

To preserve storage space, the database must support file compression. The supported compression algorithms are ZIP or a combination of TAR and GZIP. Note that the definition explicitly forbids usage of any recursive compression structures of the same kind (e.g. ZIP archive within other ZIP archive, etc.). A recommended alternative in such case is to increase compression level of already existing file archives instead.

As individual data files are expected to grow quite large in size, compression is to be utilized only at the lowest level of the directory structure, that is in the time-coded data files and directories. Individual archives are allowed to store at most one time-coded file (or directory), thus being able to overtake file's time-coded name while remaining unique in the file listing. It is preferred but not required that all data files stored in a single directory are either all compressed, or none of them is, as any deviation from this scheme might point to an incomplete or broken data transaction.

### 2.2.2 Expected Volume of Data

This section includes a simple calculation to obtain an upper bound on the size of the database.

Assuming that one hour of footage stored in the multi-frame format may take up to 4 gigabytes in size (depending on the frequency of acquisition), acquisition from one detectore generates at most 96 gigabytes per day. Accounting for the longest possible time of

operation, the database will store up to 2,437 days of footage simultaneously recorded by up to 20 detectors. That means that the database will have to hold about 4.7 petabytes worth of uncompressed information. If we use Collin's compression algorithm benchmark from [2] as baseline, it is possible to estimate that a common variant of GZIP algorithm will reduce the file size in average by 75.9%. Applying such compression on the multi-frame data files, the database would have to hold *only* about 1.1 petabyte of archives.

An analogous calculation can be performed for the ROOT file format as well. Since the file structure already utilizes its own proprietary compression algorithms, the expectation is that the overall volume decreases significantly in comparison with the raw uncompressed multi-frame data. From the data recorded by the ATLAS-TPX network in the fall of 2015, it is possible to ascertain that a single day of footage stored in the ROOT format takes up to 18 gigabytes in size. The same extrapolation as before yields that the database will have to hold about 877 terabytes of information. This result is in agreement with the expectations stated before.

This estimation shows that the ROOT data format seems to be more space-efficient than the plain text format. For long-term archivation, it is recommended that the amount of footage kept in plain text is minimized either by compressing or erasing data files after cluster analysis is finished.

## 2.3 Index Database

So far, a set of rules has been established for the file system in order to quickly obtain data from a specific device captured at a specific time. These facilities are sufficient for navigating and accessing data in rudimentary manner, but are certainly not optimal. For instance, the definition does not contain any conventions regarding retrieval of specific frames from files in the ROOT data format. Due to this limitation, users seeking individual frames would have to download bulks of data corresponding to longer time periods (their length can vary from an hour to a day in time and from hundreds of megabytes to several gigabytes in size), which may induce unnecessary processing overhead and memory shortages.

There is also no guarantee that time-coded nodes in our directory structure will be individual files. If such nodes happen to be directories, file structure inside of such directories is undefined, and may require additional decisions on the user side. And what do users do when they want to retrieve frames based on different criteria than time and device of origin? At the moment, there is no other option than directly enumerating frames stored in all files on the file system, which (considering their potential size) might not be a preferrable solution. To resolve all these issues, one more element to our design is introduced—an index database.

This database is to contain information, which can be recalculated at any instant from the primary data files, and thus will need no backups or redundancies. The information stored in the index database shall mostly include, as the name suggests, index of all files and frames on record and addresses pointing to the them on the file system. In addition, the index database is to store commonly requested aggregated values.

### 2.3.1 Definition

The index database is compliant with the SQL standard. For the reasons of simplicity, only three basic entities are defined. The relationships between these entities are depicted in Figure **??**, whereas the meaning of their members is defined in this section.

**Sensor** Sensor represents a single ATLAS-TPX device, from which data can be acquired. For full definition of the SQL table, see Listing A.2.

- **Sensor Identifier (`sid`)** Identifier of the device, unique within the index database.
- **Name (`name`)** Readable name of the sensor, consistent with the other literature.
- **Calibration Constants (`calibration_layer1`, `calibration_layer2`)** Constants used for luminosity calculation, available only for some devices.

**ROOT File** File represents a single file in the ROOT data format, containing data acquired from a single ATLAS-TPX device in a determinate time period. For full definition of the SQL table, see Listing A.3.

- **File Identifier (`fid`)** Identifier of the file, unique within the index database.
- **Device of Origin (`sid`)** Identifier of the ATLAS-TPX device, which acquired all data stored within this file.
- **File Path (`path`)** Absolute path to the file in the server's file system.
- **Date of Addition (`date_added`)** Date and time, when the file was added to the database.
- **Covered Time Interval (`start_time`, `end_time`)** Minimum and maximum start time of the TPX frames stored within this file.
- **Statistics (`count_frames`, `count_entries`)** The total number of frames and clusters stored in within this file.
- **Validation Data (`checksum`, `date_checked`)** SHA1 checksum of the file and the latest date and time, when the file was validated against it to prevent data corruption.

**Frame** Frame represents a single event of data acquisition from a ATLAS-TPX device. Every frame is stored in some file (and a file can contain multiple frames). For full definition of the SQL table, see Listing A.4.

- **Frame Identifier (`frid`)** Identifier of the frame, unique within the index database.
- **File Identifier (`fid`)** Identifier of the file, in which the frame is stored.
- **Sensor Identifier (`sid`)** Identifier of the device, which captured this frame (must match `sid` of the file).
- **Start Time (`start_time`)** Start time of the acquisition.
- **Acquisition Time (`acquisition_time`)** Duration of the acquisition.
- **Data Addresses (`dsc_entry`, `clstr_first_entry`)** Index values pointing directly to entries within the ROOT file's internal structure, where the frame data is stored.
- **Statistics (`occupancy`, `clstr1_count`, ..., `clstr6_count`)** Total number of non-zero pixels in the frame, and numbers of clusters of different types in the frame.

### 2.3.2 Performance Optimization

By definition, it follows that the index database helps deterministically resolve all time-based queries, even in situations when frames are stored in an undefined directory structure. Apart from this optimization, the it also provides file validation primitives to ensure that any corrupted files are discovered as soon as possible. But there is one more significant performance optimization.

When retrieving frames by the time and device of origin, predefined naming patterns can be used to obtain a path in the file system. In case the path points to a directory, the contents of the index database specify, which file in the directory contains the information sought. Still, this approach leaves one more operation unoptimized since in order to retrieve a specific frame, a sequential scan the entire data file is required.

This issue is in part resolved by sorting all frames in data files consistently by their start time, enabling utilization of binary search algorithm, thus reducing complexity of the operation from linear to logarithmic. Furthermore, the index database maps every frame to a tuple of data addresses, which will point to specific locations within the data file, rendering any enumeration redundant.

Recall that ROOT files contain two trees of interest, the `dscData` tree with information about detector configuration, and the `clusterFile` tree, which contains concatenated lists of clusters from every frame. There is only one entry per frame in the `dscData` tree, whereas the `clusterFile` tree may contain anywhere from zero to hundreds of thousands of entries corresponding to a single frame. Entries belonging to the same frame can be identified by having equal value of the `Start_time` leaf.

If all entries in both trees are sorted by the value of this leaf, the `clusterFile` entries consequently form continuous bulks of data corresponding to individual frames. This means that once the first entry in the bulk of the sought frame is discovered, consecutive entries can be read as long as their start time remains the same (or the last entry is reached).

The indices of entries in the `dscData` and `clusterFile` trees are stored as data addresses for every frame in the index database, making search a constant-time[1] operation, as illustrated in Figure 2.2. This significant benefit comes with a trade off in increased complexity of the insert operation due to additional sorting of entries in our files, and slightly increased space occupied by the index database because of stored entry indices.

### 2.3.3 Data Aggregation and Metaindexing

In some cases, users of the database may want to calculate aggregated statistics. Since these types of requests are hard to anticipate and do not constitute a significant portion of all user queries, it is not worth the effort to create separate data structures in order to accelerate their processing. Existing data structures are however be extended to support some of such requests. For example, the index database makes a great candidate in particular since it contains data associated with individual files and frames, and is easily accessible and queriable using SQL. For that reason, several statistical values are included with every frame, such as count of clusters differentiated by individual cluster types and frame occupancy encoded as number of non-zero pixels.

[1] This statements neglects the complexity of lookup in the SQL table.

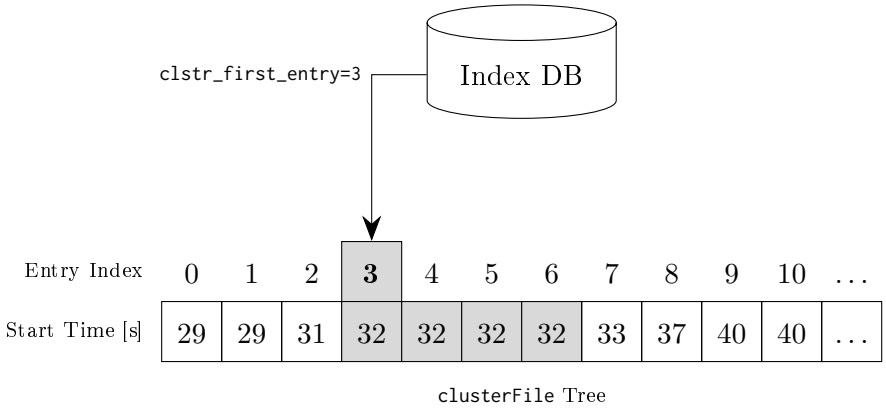



Figure 2.2: Illustration of the optimization mechanism provided by the index database.

Users can utilize filtering and aggregation features of SQL to quickly find files and frames in the index database, and if required, analyze their contents more thoroughly.

Lastly, SQL implementations include an analogy to the index mechanism used to accelerate access to individual frames within data files. Using their own tree indices built from various columns of data tables, SQL servers can speed up certain queries containing predicates or orderings based on such columns. This makes the data access procedure even faster, in a sense indexing the index database. To see detailed application of this technique, examine Listings A.2, A.3 and A.4.

# Chapter 3

# Communication Protocol

This chapter describes the JSON Timepix Protocol, a communication protocol used to transmit TPX footage for the purposes of visualization.

## 3.1 Remote Access

Since the database defined in the previous chapter is based on a UNIX file system, multiple users can access it simultaneously by either directly interacting with the computer responsible for its operation, or by using some of the supported protocols[1] to interact with it remotely over a network.

Due to this capability, one might argue that defining another dedicated communication protocol such as JSTP seems rather redundant. What advantages does this approach offer? The primary motivation for the existence of JSTP is the web visualization UI, which is described in the Chapter 4. It is expected that users of such application would want to observe recorded footage frame by frame. If no protocol is defined to facilitate transmissions of individual frames from the database to the visualization UI, data has to be transferred in one of the formats listed in section 1.3, none of which is particularly suitable for this task.

For instance, the plain text format stores data in multiple files implying that several parallel downloads would be required, possibly putting strain on user's network connection in the process. The ROOT format on the other hand uses its own compression algorithms, making it non-trivial to deflate in a website context. Lastly, since both ROOT and plain text formats store data in bulks, the information overhead to transmit units of frames would be nearly unbearable, especially considering that data files in question may be several gigabytes in size.

With this motivation in mind, JSTP is defined to effectively replace both formats in such situations. It is expected that multiple users would connect to a JSTP server over a local area network or through the Internet, possibly at the same time. Every user is assumed to have intentions to browse through or further inspect some of the frames captured by the ATLAS-TPX network, transmitted in units at a time. Note that JSTP is not designed to transmit all information from the data files, nor send continuous footage at streaming speeds. Instead,

[1] Recall that the section 2.2.1 mentions access over FTP, SMB, SSH, AFP and HTTP.

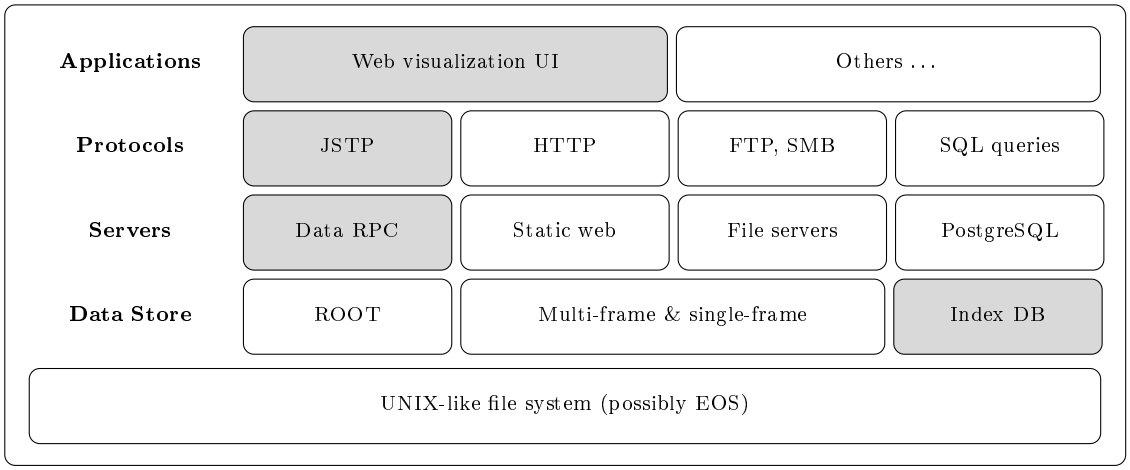



Figure 3.1: A multi-layered system. Proprietary components are emphasized by gray color.

JSTP enables simple access to the most important detector data, and provides brief overview of recent detector operation with emphasis on any irregular or pattern-defying events.

### 3.1.1 Considerations

In definition, a multi-layered system architecture is upheld. This primarily serves to create strict distinctions between individual components of the system and the tasks they perform, making them in effect easily extensible, substitutable and perhaps even portable to other applications. Other benefit of this approach is that users always have freedom to choose a component, with which they wish to interact, in effect choosing the level of offered services, processing speeds and algorithmic complexity.

This may be illustrated on a practical application. Users who want a quick peek at the detector operation without any effort might decide to use the visualization UI in their web browser. The website is quite easy to use, does not require any particular skills to operate, and is capable of displaying frames captured by the detectors as well as overview of their operation. In contrast, users who want to retrieve data for experimentation or statistical aggregation might utilize SQL or JSTP as these two protocols are not designed to interact with humans, but with other applications, most notably usable in scripts designed for custom data processing. Lastly, users in need of information, which is not displayed by the visualization UI nor transmitted by any of the mentioned protocols, can connect to the database storage facility remotely and directly download data files by means of some of the supported network transfer protocols. This concept is illustrated in Figure 3.1.

JSTP is built with extensibility in mind. With multiple concurrent projects such as MoEDAL-TPX[2], SATRAM[3] and RISESat[4], it is likely that JSTP will be used for compati-

[2]Similarly to ATLAS-TPX, MoEDAL-TPX is a network of Timepix devices installed within the MoEDAL experiment at CERN.

[3]SATRAM is a technology demonstration device carrying Timepix position-sensitive semiconductor pixel detector on board ESA's Proba V satellite. <http://satram.utef.cvut.cz/>

[4]RISESat is a microsatellite mission carrying several scientific instruments including a Timepix detector.

bility reasons in other applications as well. It should therefore allow for limited variability, gracefully handling minor alterations in transmitted data structures.

### 3.1.2 Requirements

This section lists all formal requirements on JSTP. The most basic requirement is that the protocol allows to retrieve frames captured by the ATLAS-TPX network by their start time and device of origin. This might remind observant readers of a similar requirement stated in the database definition (see section 2.3.1), as it is the most likely user request. However unlike the database, JSTP must be able to transmit only those frames, which satisfy the user predicate, effectively reducing information overhead in transmitted messages to zero.

In the first version, JSTP is required only to transmit results of cluster analysis, leaving door open for pixel matrix transmissions in the future. This indirectly implies that every message transmitted through JSTP containing a captured frame consists of two parts: a header (containing detector configuration, position, orientation, etc.) and a body (containing a list of clusters, or possibly a pixel matrix).

To efficiently reference detectors in the ATLAS-TPX network, it is required that JSTP provides an exhaustive list of network elements along with information about their availability in the system. This might seem redundant at first, but consider that JSTP needs handle situations when detectors malfunction, are replaced, or new detectors are installed. Such events might not be that uncommon, especially given the experimental nature of the project.

Lastly, in order to aid with navigation in large amounts of detector footage, JSTP has to offer a mechanism to generate statistics over larger periods of time. This information will help users find events of interest in overwhelming quantities of white noise, resembling the proverbial *needle in a haystack*.

Apart from various file management network protocols listed earlier, there are no data manipulation requirements on JSTP, implying that the protocol cannot be used for other than read-only access to detector footage.

## 3.2 Underlying Standards

JSTP is a web protocol and as such, it utilizes HTTP as its underlying standard, serving to abstract physical data transmission and compression. By this declaration, it is implied that JSTP is a request-response communication protocol between two types of agents: *a server* and *a client*.

In its architecture, JSTP consists of two parts: a web service providing API for remote procedure calls (RPC) and a data format built atop of it to facilitate such calls. Since JSTP does not include any universal service description mechanism such as WSDL or WADL, all clients need to know its capabilities and calling conventions prior to initiating communication with the server. For data serialization, JSTP utilizes JavaScript Object Notation (JSON) (hence its name). This format has been selected for various reasons. It is simple to parse, offers an extensible tree structure and is very common among web services of this kind, as it is directly supported by the JavaScript client-side runtime used in the web visualization

UI. Apart from JSON, JSTP does not offer nor accept communication in any other data formats.

One might ask whether JSTP web methods meet the standards of RESTful web services. While it is true that the protocol shows many traits often attributed to RESTful services (client-server model, stateless protocol, cacheability, layered system), it certainly does not satisfy all of them. For example, JSTP does not uniquely identify resources by their URI because it does not offer any of the common CRUD operations. Moreover, in referencing entities, JSTP uses arbitrary identifiers (recall members `fid`, `frid` and `sid` of entities defined in section 2.3.1), which are not passed in the URI but through an array in the request body. Moreover, JSTP does not offer a uniform interface, capable of negotiating data format according to client limitations. Instead, it forces clients to communicate strictly in JSON, adhering to its own arbitrary data structures and calling conventions.

**TODO**

## 3.3 Web Methods

The main component of JSTP is a web service, which can be described as a set of proprietary web methods. For the purpose of simplicity, in this section we provide only a semantical description of each method. Readers interested in full technical documentation of the methods are referred to Appendix B.

### 3.3.1 Detector List

The first method is dedicated to providing an updated list of operational detectors in the ATLAS-TPX network.

As we mentioned earlier, we need to be ready for situations when the physical structure of the network changes due to malfunctions or upgrades. For these reasons, any client intending to retrieve frames from a specific detector must first consult the list provided by this method to verify, whether the device is still connected and operational. In addition, other clients unaware of the network's architecture may use this method to obtain an exhaustive listing of all currently available data sources.

Execution of this method requires no parameters. The server responds by transmitting a list of devices, from which data can be retrieved at the time of request. For detailed documentation of this method including examples of requests and responses, see section B.2 of the Appendix.

### 3.3.2 Overview of Acquisition

To satisfy demands on navigation in voluminous amounts of data, we dedicate the second web method to providing an overview on detector acquisition. This is achieved by uniformly dividing a specific time period into finitely many time intervals, in which all relevant frames are gathered with respect to their start time (acquisition time is not considered). In every interval, frames are subsequently processed to produce aggregate statistics, which might

indicate time points, where frames of interest are located. This approach is in its essence very similar to the binning procedure used when constructing histograms.

Clients calling this method are required to transmit five parameters in their request:

**Detector Predicate** A group of detector identifiers, restricting all processed frames by their device of origin.

**Start Time, End Time** These parameters define the time period, in which we generate statistics. Obviously, the first parameter must be an earlier point of time than the latter.

**Group Period** The duration of every interval in the partitioning of the time period. Should an imperfect partitioning occur, the number of intervals is always rounded up to the nearest integer, possibly exceeding the specified end time.

Longer durations obviously result in a lower number of intervals, and in turn a lower number of returned data points. Shorter durations yield more data points, but may result in lengthy processing at server-side. For stability reasons, the server therefore requires that the duration of the group period results in at least 1 and at most 1024 intervals.

**Normalized Mode** An option to compensate possible data distortions caused by variations in frame acquisition times. This setting is irrelevant in configurations, where users can be certain such variations do not occur.

If the server finds request parameters to be valid and succeeds in generating requested statistics, it responds by transmitting a list of data points, corresponding with intervals of the partitioning of the specified time period. Every data point includes three values:

**Cluster Counts** Sums of cluster counts from every frame in the interval, summed separately per every of the six cluster types (for type definitions, see section 1.2.2).

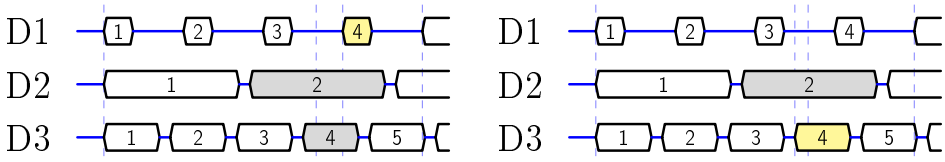
**Frame Occupancy** Total number of non-zero pixels in all frames in the interval, indicating their levels of saturation.

**Number of Frames** The count of frames aggregated in the interval.

For detailed documentation of this method including examples of requests and responses, see section B.3 of the Appendix.

### 3.3.3 Frame Search

The third method serves to retrieve frames captured at any given point in time by a detector (or a group of detectors). As the method's name might suggest, time need not be exact, resulting in a search for the nearest frame operating on the scope of the index database. There are several search modes available, each offering a different strategy to find *the master frame*. Once such frame is identified, its start time is then used to locate other frames from the remaining detectors, yielding at most one frame per every detector. In the first version of JSTP, we support two search modes:




(a) Forward search. (b) Backward search.

Figure 3.2: Time diagram of frame search illustrating behavioral differences between search modes. Individual blocks correspond with periods of detector acquisition. Emphasized blocks are returned as the search result (yellow marks the master frame).

**Sequential Forward Mode** The master frame is the frame with the start time nearest to, but greater or equal than the time parameter of the search.

**Sequential Backward Mode** The master frame is the frame with the start time nearest to, but lower or equal than the time parameter of the search.

Let us demonstrate operation of these modes on a real world example. Suppose that we are interested in frames captured by two devices. Detector 1 captures frames every 0.25 seconds with acquisition time of 0.05 seconds, whereas detector 2 captures frames every 0.33 seconds with acquisition time of 0.27 seconds. If we set the search time to 0.4 seconds and search in the forward mode, the third frame captured by detector 1 will be designated as the master frame. Since the start time of this frame is 0.5 seconds, the second chosen frame will be the second frame captured by detector 2 as its start time is 0.33 seconds and its end time is 0.6 seconds. This scenario is depicted in Figure 3.2a. If we to use the backward mode instead, the second frame captured by detector 2 will be designated as the master frame. Since its start time is 0.33 seconds and there is a gap in detector 1 footage between 0.3 seconds (end time of the second frame) and 0.5 seconds (start time of the third frame), the algorithm will return no frame for the other detector. This is illustrated in Figure 3.2b.

To summarize, clients calling this method are required to specify four parameters:

**Time of Search** The point in time used as a starting point of the search.

**Detector Predicate** A group of detector identifiers, restricting retrieved frames by their device of origin.

**Search Mode** A strategy to select the master frame based on the time of search and available detector footage.

**Integral Frames** Number of consecutive frames to be integrated for every device.

If the server finds request parameters to be valid and succeeds in locating at least one frame, it responds by transmitting the start time of the master frame, followed by headers and bodies of all found frames, corresponding with the order of identifiers in the detector predicate of the request. Frame bodies are transmitted in the form of cluster lists (for properties of clusters, see section 1.2.2). Detailed documentation of this method, including examples of requests and responses, is available in section B.4 of the Appendix.

## 3.4 Miscellaneous

JSTP has been originally designed to serve solely as a data transmission component of the visualization UI. Over time, it has however grown to be a more complex protocol, with applications in other projects than ATLAS-TPX and outside the conventional task of data visualization. It is the intention of author to continue development of this protocol with further releases in the future, eventually decoupling it from the Timepix chip and abstracting it to the point where it could be utilized in combinations with different hardware.

Since the amount of data in our database is expected to become rather overwhelming, the protocol itself is structured and meant to be used in a top-down model (see Figure 3.3), allowing clients to gradually refine parameters of their requests and locate the information they seek, while avoiding transmission of data in overly granular bulks. In other cases, the protocol minimizes information overhead by requiring strong usage of predicates operating on the index database.

Note that in the protocol definition, we do not specify whether the results of individual web method calls are cacheable by clients. This is due to the diversified nature of its applications. Since HTTP already contains its own caching logic[5], we encourage all clients to comply with strategies described in [3], section 13, as JSTP servers are permitted to use this mechanism to emply different caching policies for individual response messages. Analogous declaration is used for data compression (for HTTP specification, see section 3.5 of [3]).

[5] Caching in HTTP is controlled by values of headers provided in every response message. Relevant header names are: `Cache-Control`, `Expires` and `Pragma`.

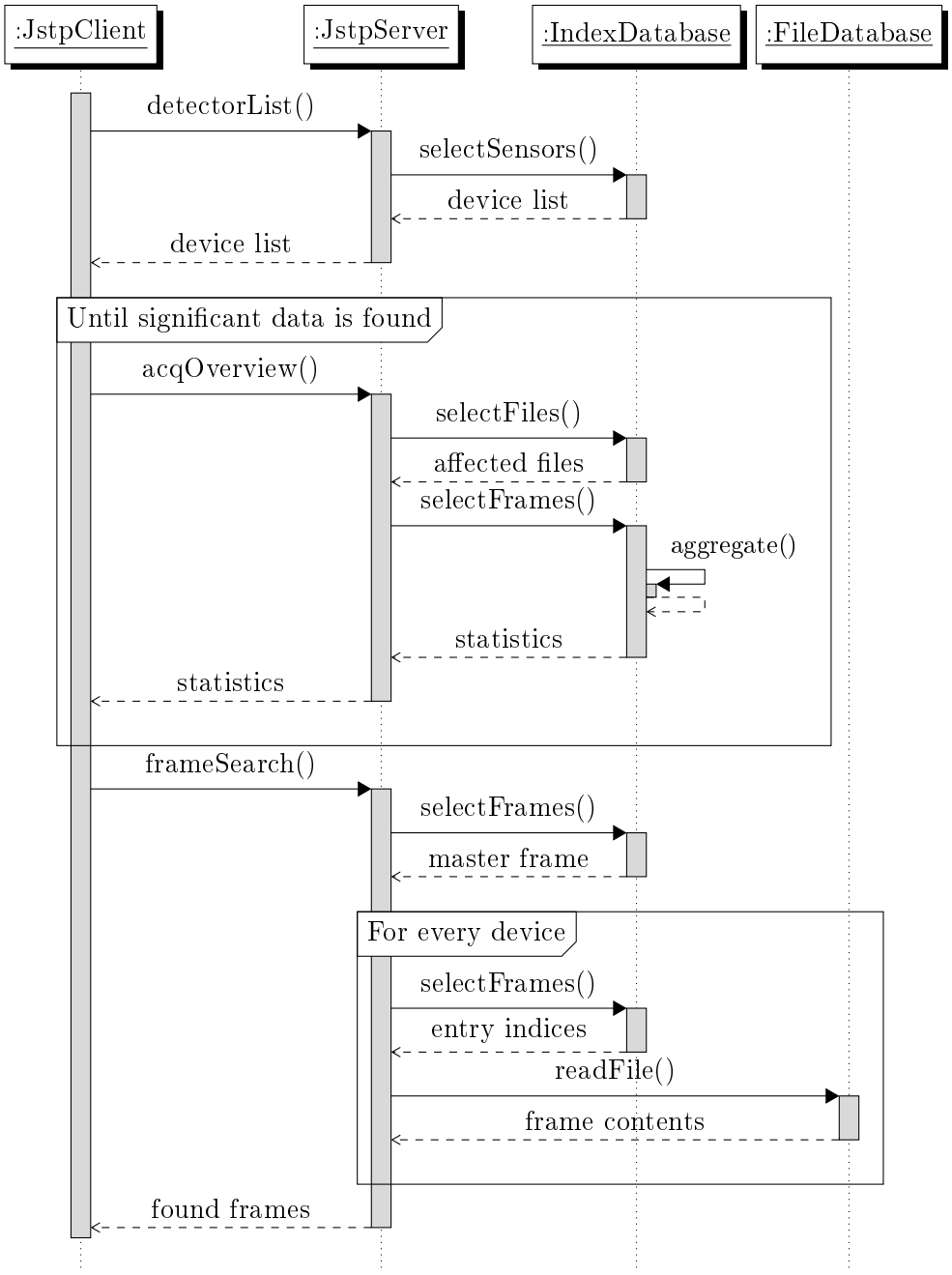



Figure 3.3: UML diagram depicting expected interactions between JSTP client and server, hinting levels of processing complexity at server-side.

# Chapter 4

# Server Implementation

This chapter describes implementation of a JSTP server with a web visualization UI. While sections in the beginning focus on caveats of efficient data transcoding and give details on some backend performance optimizations, the sections in the end describe the user interface composition and chart rendering.

## 4.1 Decomposition

The server application consists of two major components, a JSTP data server and a static web server. As their names suggest, the data server asynchronously delivers data to visualize in the form of JSTP messages, whereas the web server provides the visualization UI in the form of static-hosted files.

Both applications are operated simultaneously and independently of each other as Linux daemons or services in the systemd initialization system. Each application listens and responds to client requests on its own dedicated port.

### 4.1.1 JSTP Data Server

The JSTP data server is a C++ application built using the Facebook Proxygen open source library. It interacts with the ATLAS-TPX footage database and the index database and transcodes TPX data to the JSTP format.

The core component of the server is a thread pool. It allows simultaneous communication with multiple clients, provided that server's hardware offers parallel processing support. At the startup, multiple *worker threads* are created. These threads are immediately suspended to conserve server's resources. When a new request arrives, one of the suspended threads is awakened and notified to process the request, compose and send a response. During this operation, the thread is said to be *busy* and cannot receive new requests. Should such a request arrive at that time, the server would opt to awaken another of the suspended threads, gradually exhausting its pool. After the response is sent, the busy thread returns to a suspended state, awaiting further instructions. This way, threads are recycled within the pool throughout server operation.

### 4.1.2 Static Web Server

The web server is a standard server application implemented in Node.js. It stores all files and dependencies of the web visualization, such as HTML files with UI definition, style sheets written in CSS and client-side scripts written in JavaScript. Since these files are quite static in their essence, the web server uses standard HTTP caching mechanisms to speed up its operation.

For security reasons, the HTTP socket managed by the web server is the only socket accessible from the Internet. All JSTP traffic is routed through the this socket and then redirected to a private socket owned by the data server, thus eliminating the need to expose more than one port to the Internet.

## 4.2 Object-Oriented Design

In the JSTP data server, much emphasis was put on the object design and the use of standard design patterns. This section contains the most notable instances.

### 4.2.1 Request Handling

When a request arrives, a worker thread is assigned to process it and respond accordingly. To avoid keeping server logic within the implementation of worker threads, the process of producing a response to a single instance of request is generalized into *a request handler* object.

Upon request, the worker thread decides which method is being called, creates a corresponding handler, and gives it abstracted control over the HTTP socket. After the handler is finished processing the request, the worker thread sends the response to the client and destroys the handler, freeing up resources related to the communication session.

Using object polymorphism, multiple types of request handlers are implemented to service requests corresponding with various JSTP web methods listed in section 3.3.

### 4.2.2 Behavior Selection

Apart from producing server responses, worker threads are also responsible for choosing an appropriate behavior for every client request. In comparison to processing requests themselves, this logic consists mostly of picking the correct request handler for every request. It is made autonomous through application of the factory method design pattern (see Figure 4.1).

When started, every worker thread creates *a factory* object. This object is later called when requests arrive, and based on their parameters, determines which request handler is to be used to produce a response.

Since the JSTP specification uses URL to determine called web methods (and by extension request handlers), a factory subclass has been implemented to utilize regular expressions to perform decisions about requests. At the creation time of the subclass, all supported request handlers along with their respective regular expressions are registered using the standard builder design pattern. One more request handler is designated as *the default handler*.

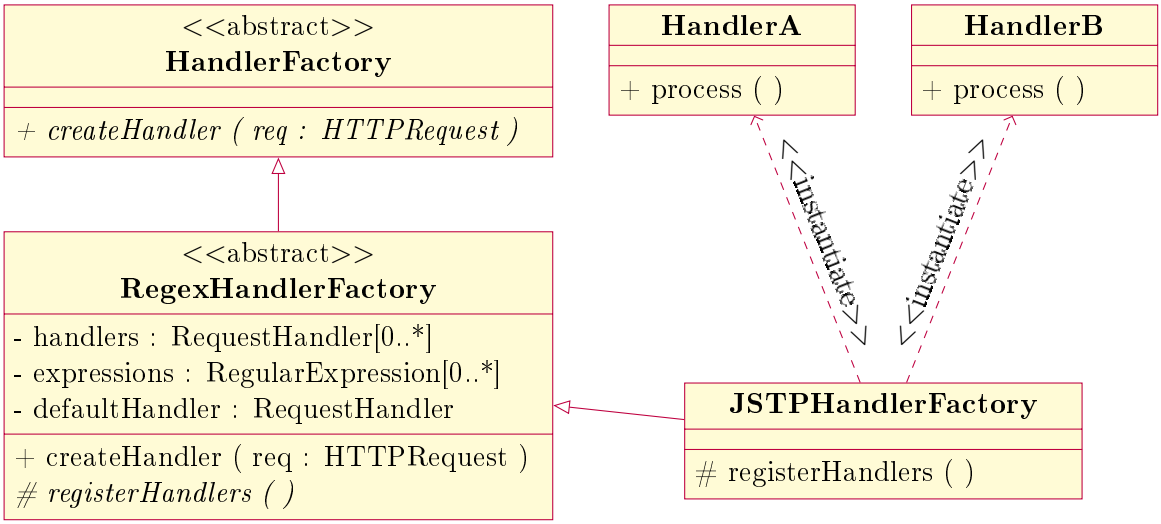



Figure 4.1: UML diagram illustrating the abstract factory design pattern applied in the context of request handler instantiation.

Upon request, all of the registered expressions are sequentially matched on its URL. Should one of them succeed, its corresponding request handler is selected. Otherwise, the default handler is used.

### 4.2.3 Content Abstraction

In order to produce valid JSTP messages, a JSON serialization component is required. All web methods are expected to read their parameters and compose their responses in the desired format. Since this behavior is shared amongst all request handlers, it is encapsulated in a separate subclass of a conventional request handler. To access properties of this subclass, all request handlers corresponding to JSTP web methods inherit from it.

The subclass interacts directly with the HTTP socket and uses a parser and a writer to retrieve and produce JSON strings. Its descendants can thus only call the parser and the writer to access and produce content, instead of reading and writing to the socket directly, as illustrated in Figure 4.2. This prevents bad object design by centralizing serialization logic in a single object. In addition, it allows for a limited degree of variability, since the subclass itself is the sole object responsible for communicating information to the socket. For instance, if the XML format was to be used instead of JSON, only internals of this subclass would have to be reimplemented, while its descendants could remain the same.

## 4.3 Performance Optimizations

In the JSTP server, various performance optimizations are used to minimize response latency. This section lists some of such optimizations.

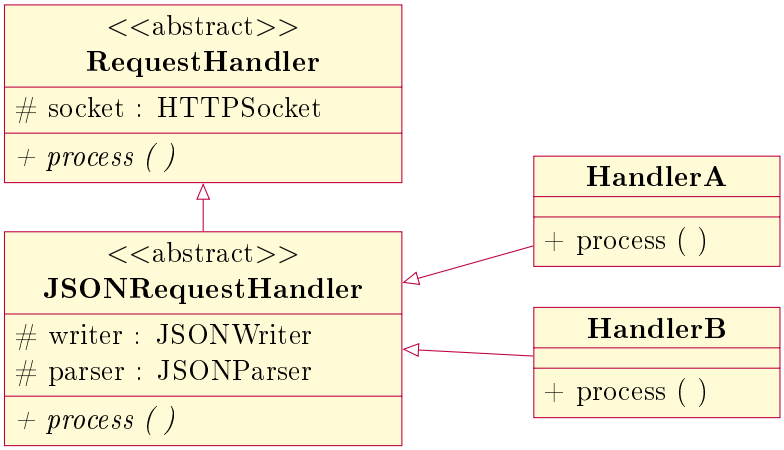



Figure 4.2: UML diagram illustrating the inheritance of request handler objects.

### 4.3.1 ROOT Reading Optimization

Recall from section 1.3.2 that the ROOT format stores information in tree structures, separating detector configuration from cluster lists corresponding to individual frames. Due to possibly overwhelming sizes of data files, it is nontrivial to devise a logic to minimize access time with respect to memory paging and L1 cache. Some of the most significant factors to consider are:

**ROOT Compression** The ROOT data format utilizes its own compression algorithm, roughly equivalent to the ZIP format in its efficiency. There are multiple levels of compression ranging from the best compression ratio to the fastest reading time. Choice of the compression level affects all subsequent processing required to encode and decode data.

**ROOT Cache** The ROOT data format implicitly uses a file cache to prefetch information in memory with assumption that the data will be read sequentially. For that reason, linear enumeration of data structures tends to be faster than a random access.

**Tree Locality** When retrieving frame data, switching from the `dscData` tree to the `clusterFile` tree and back might cause OS to swap memory every time a single frame is read, producing unnecessary overhead and slowing down the process.

**Data Demand** In many instances of JSTP messages, it is requested that only a portion of the stored data is read. ROOT allows applications to specify this information prior to initiating sequential reading, and in turn accelerate some procedures.

With respect to the these factors, all instances of objects requiring to read data from the ROOT file format, do so sequentially in strides. The application producing ROOT data files is configured to use the medium compression level, offering acceptable access speed while maintaining good compression ratio.

When reading frame data, the configuration information is first read from the `dscData` tree for all frames. After the information is processed, the server moves on to the `clusterFile` tree and repeats the procedure without returning to the `dscData` tree in the process. Lastly, all components of the server interacting with the ROOT file format exhaustively declare tree branches, which are subject to processing later on.

### 4.3.2 Centralized File Management

In the web visualization UI, users may often browse frames in the order of acquisition. On the server-side, this would imply that the same ROOT data file is opened, read from and closed multiple times over. Such behavior introduces unnecessary overhead, as the procedure of opening and closing file pointers to possibly large files may become somewhat inefficient, and in turn slow down server operation significantly. Note that it would be much better solution to only open the data file once, use it to extract multiple requested frames and then close it for good.

To resolve this problem, a centralized data structure is introduced into the server application. This structure is accessible to other components (such as request handlers) in compliance with the singleton design pattern. Its main responsibility lies in opening and closing ROOT files. Its implementation however does not forward these calls directly to the file system. Instead, the structure contains internal time-driven caching mechanism, which recycles open files between multiple consumers and closes their file pointers only when their contents are not requested for a greater period of time.

Please note that such a central structure shared amongst multiple components operating on different threads does not induce a race condition, because it allows multiple instances of the same file to be open at the same time.

## 4.4 User Interface Documentation

The web visualization UI consists of a single static HTML page, which is served to users by the web server. Apart from UI definitions and style sheets, it includes several client-side scripts, controlling the behavior of the website in the web browser. From design standpoint, UI is divided into multiple sections (depicted in Figure 4.3), each with a dedicated role and purpose.

### 4.4.1 Header Bar

The topmost section of the UI is dedicated to important descriptive and control elements of the screen. From the left to the right, it contains logos of the institutions involved in the acquisition of experimental data, detector control box, time control box and a frame stepper.

**Institution Logos** This section includes the logos of the ATLAS collaboration, Institute of Experimental and Applied Physics and the Czech Technical University in Prague. All pictures are linked to the respective institution websites.

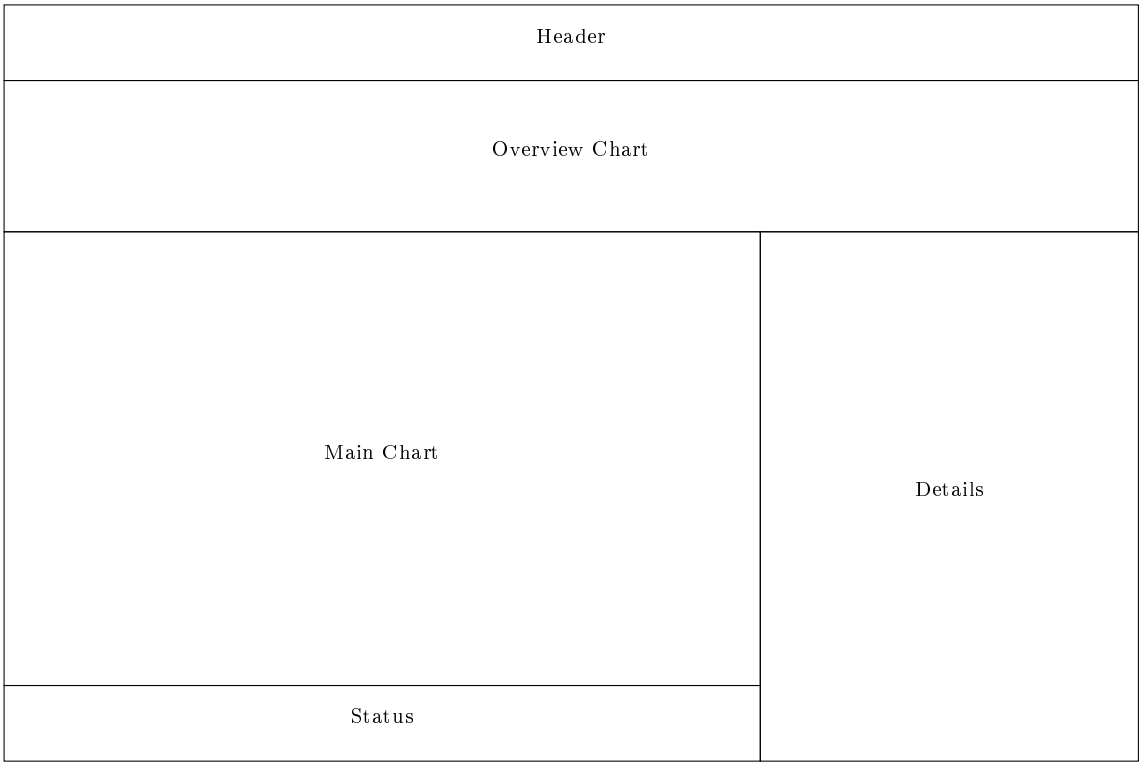



Figure 4.3: Wireframe showing the layout of UI sections.

**Detector Control** The detector control box allows user to specify a device (or a set of devices) in the ATLAS-TPX network to be used as a data source for the displayed frames. If only one device is selected (the default configuration), the box is optimized to allow quick switching using a dropdown control and incremental toggle buttons on its sides.

**Time Control** This box controls the start time of the displayed frames. It is essentially a simple date input element with additional incremental toggle buttons for every component of the date.

**Frame Stepper** This control is a simple extension of the time control box, allowing users to quickly browse frames sequentially in order of their acquisition. It consists of two big buttons with arrows pointing to both sides, signifying the direction of time movement. Next to these buttons, a number input element is located. This element is responsible for controlling the number of integral frames.

### 4.4.2 Overview Chart Area

The overview chart lies under the header bar and fills the entire width of the screen. Its purpose is to inform users about detector acquisition in a determinate time period, referred to as *the window*. To aid navigation in the data, the chart plots a vertical line at the point

**TODO**

(a) Absolute mode.

**TODO**

(b) Stacked mode.

Figure 4.4: Example of the same overview chart rendered in different modes.

corresponding with the current start time of the displayed frames. Upon every change of this parameter, the line moves horizontally in the chart to adjust. Should the line be plotted out of bounds by leaving the chart either on the left or the right side, the window is automatically updated to compensate. It is worth noting that this mechanism also works the other way around. Users can set the start time of the displayed frames to any time point from the window by clicking at its respective position on the horizontal axis.

In the chart, multiple series are plotted simultaneously. The horizontal axis always corresponds with time, whereas the vertical axis may correspond with the number of clusters, flux or frame occupancy, depending on the series in question. There are at most 8 plotted series at any instance. Their appearance is described by the legend located in the bottom left part of the chart area. Apart from providing description on the displayed series, the legend also allows users to turn plotted series on or off by clicking on the respective items of the legend. The series can be semantically divided in three groups:

**Cluster Counts** For every of the six types of clusters, clusters from frames are counted separately, producing six different series in the chart. These series have no unit as their values merely correspond with the number of clusters occupying frames, whose start time falls into a specific time interval.

If the normalized mode is active, contributions to these series from every frame is first divided by the acquisition time of the frame, producing flux values with unit $s^{-1}$.

**Total Sum** This series plots the sum of cluster counts over all cluster types. Its unit is same to that of the previous six series.

**Frame Occupancy** The values of this series correspond with the portion of frame area occupied by non-zero pixels. It has no unit and is distinguished from the other series by a dashed line.

The overview chart offers a number of custom settings. The length of the window can be reconfigured to any value from 30 seconds to 4 days by controls located in the bottom right corner of the chart area. The window itself can be also adjusted to align the vertical line corresponding to the current start time of displayed frames to the center of the screen. Furthermore, the chart offers two rendering modes to choose from, a comparable mode and a stacked mode.

**Absolute Mode** In this mode (seen in Figure 4.4a), all series of the chart are rendered as points in the plane with respect to the horizontal and vertical axis. The consecutive

points of every series are then connected by line segments of different color to indicate continuity in time.

This mode enables users to easily compare values of different series to each other. However, since the experimental data often includes one or two prevailent series, it also frequently overshadows the remaining series as they are rendered over each other, making it harder to read their values.

**Stacked Mode** In this mode (seen in Figure 4.4b), the six series corresponding to cluster counts are rendered as in a stacked chart, cumulatively adding to each other. Each cluster type series corresponds to a colored area in the chart, while the remaining series are rendered in the same way as in the absolute mode.

In comparison to the previous mode, this mode distorts the absolute values of the series. It however much better portrays the ratio of representation of one series to another.

### 4.4.3 Main Chart Area

As the name suggests, the main chart area is the primary section of the UI dedicated to plots of frames from the TPX detectors. By default, it shows only data from a single detector. It can however be configured to partition itself into multiple cells, each corresponding to data acquired by a different detector in the network. This feature is often useful on large screens and projectors.

Each cell consists of two square charts, corresponding with individual sensor layers of the detector. Both charts are identical in their layout and internal structure, differing only in the data visualized. Since frames are in their essence pixel matrices, charts visualize them in a standard way by mapping pixel values to different colors, which are later used to fill their respective rectangular areas. Frame charts can be customized in several ways:

**Visualized Values** If the visualized frame has been captured in the TOT mode, a calibration method described in [4] can be used to obtain energy values from raw counter values. In other operation modes, only counter values are available.

By default, both charts visualize energy values in the TOT mode and counter values other modes. This setting can however be overridden to always visualize counter values in all modes instead.

**Scale Bounds** In order to map pixel values to colors, a determinate interval must be defined to establish scale bounds. This interval is by default calculated from the visualized frame. Users can however configure its bounds to any fixed values by deactivating the *auto range* checkbox in the details panel.

**Scale Types** Given specific scale bounds and values of individual pixels, any function can be used to map absolute values to relative values from a $[0; 1]$ interval. By default, a simple linear mapping is used for this task. Users can change this setting to a logarithmic mapping, which better accentuates order differences between individual pixel values in some frames.

**Color Themes** The choice of color corresponding to a value between zero and one is purely arbitrary. The visualization UI offers three color themes commonly used in other research applications:

- The *Jet* theme ranges from blue to red, and passes through the colors cyan, yellow, and orange.
- The *Hot* theme varies smoothly from black through shades of red, orange, and yellow, to white.
- The *Gray* theme returns a linear grayscale from black to white.

Apart from the listed customizations, frame charts offer interactive data labels by responding to mouse movements over the chart area. When the mouse enters the frame, two perpendicular lines are drawn on the pixels underneath the mouse cursor. These lines track the cursor while it hovers over the chart. In addition, several rows of descriptive information are displayed next to the intersection of the lines, giving details on the pixel values and various properties of its associated cluster, should there be any.

Furthermore, frame charts offer a simple zooming feature. When hovering over the frame area, users can use the drag-and-drop mouse gesture[1] to highlight a square portion of the frame. Bounding line segments of this square are then set as the new bounds of the horizontal and vertical axis of the chart. To zoom back, users need to double-click their mouse when hovering over the frame area.

### 4.4.4 Details Panel

The details panel is located right of the main chart area and consists of multiple auxiliary screens, mostly dedicated to providing further details on the information displayed to the left. The panel is controlled by tabs on the top, each corresponding with a single screen. Only one tab can be selected at any instance, such tab is highlighted and the respective screen is displayed underneath it.

**Statistics Screen** The statistics screen provides a detailed statistical overview of the currently plotted frames. Should multiple detectors be selected for visualization, it offers an option to select one of the devices as a data source for the overview.

The statistics consist of two tables. The first table contains cluster counts, calculated on separate rows for every cluster type. Next to the number of clusters, a flux column is displayed. At the bottom of the table, values from both columns are summed together, producing a grand total.

The second table is displayed only in cases, when the frame has been captured in the TOT mode. It includes a sum of energies from all clusters in the frame and average evergy per cluster with a flux column. At the bottom of the table a calculation of instantaneous luminosity from the number of clusters is displayed. This calculation however makes only sense in frames, which are not fully saturated.

[1] The drag-and-drop gesture consists of depressing the left mouse button, dragging the mouse while still holding the button down and then releasing it at a desired position.

By default, data from both sensor layers are used in the statistical computations. User may however opt to differentiate statistics by individual sensor layers, doubling the amount of figures in both tables.

**Information Screen** The information screen displays configuration of the current detector. Similarly to the statistics screen, when multiple detectors are selected, it offers an option to select one of the devices as a data source.

The displayed information is grouped in several sections. The first section displays time information, such as the start time and the acquisition time of the frame. The following section displays technical information, for instance operation mode, frequency of the TPX clock signal and unique identification of the chip used to capture the frame.

The last two sections contain secondary information, such as the position and orientation of the detector within the ATLAS machine at the time of measurement, or index information about the ROOT data file, from which the displayed frame has been extracted.

**Filter Screen** The filter screen does not show any information. Instead, it allows users to modify displayed data by setting arbitrary predicates. Since most of such predicate operate on numerical properties of the displayed clusters, such as size, height, etc. (for detailed listing, see section 1.2.2), user can easily set a predicate by specifying the minimum and maximum value.

For convenience, each predicate has an additional switch, which determines if it is active on the current data set. Furthermore, user can activate *the warn mode*, in which a contrasting color is used to highlight all pixels violating the set predicate.

When any of the predicates is active, it is not only applied on the displayed charts in the main chart area, but also on the figures displayed on the statistics screen. This is convenient for many applications, e. g. to filter 1-pixel clusters out of noisy frames.

**Settings Screen** Similarly to the filter screen, the settings screen does not display any additional information to the user. It merely serves to configure the visualization of the data by enabling or setting various options of the charts.

### 4.4.5 Status Bar

As in most UI applications, the status bar is located at the very bottom of the screen. Its primary purpose is to inform the user of the current state of the visualization, most notably whether a data downloading is in progress, or whether the visualization is ready for new commands.

To display more information, the user can activate *the time profiling mode*, in which every procedure is measured and the time is displayed in the status bar.

## 4.5 Plotting Optimizations

Since rendering of all charts in the application occurs on the client side where hardware performance is not guaranteed, the web visualization UI attempts to improve the rendering

process as much as possible. This section lists few notable examples of methods used to minimize rendering latency and improve the overall appearance of the plotted output.

### 4.5.1 Prerendering

Both frame charts and the overview chart offer interactive features. For that reason, they often require to be redrawn upon various user-generated events, such as mouse cursor movements or mouse clicks. Since re-rendering of an entire chart could represent a time-consuming operation, especially on computers with weak hardware, optimization of this procedure is required.

Observant users of the visualization could have noticed that often enough, only a portion the charts in question needs to be redrawn. This can be for instance demonstrated on one of the frame charts. When the user's mouse is hovering over the chart, two perpendicular lines forming a cross are to appear and track its movements. That would imply that the entire chart is redrawn every time the mouse moves. However, the mouse movement does not affect the data plotted in the chart in any way. To exploit this observation, the chart, which was originally rendered as a whole on a single canvas, is divided into multiple auxiliary canvases, each of them responding to different events.

The auxiliary canvases are transparent, have the same size as the original chart and are laid on top of each other like layers in a photo editor. In addition, every auxiliary canvas maintains an additional bit value signifying whether its state is *valid*. The validity of a canvas can be defined in this context as the state of synchronization between the information drawn on the canvas and the data, which is used to generate such information. Upon different events, some of the auxiliary canvases are invalidated (meaning that their validity bit is set to the *invalid* value). Later on, when the chart is requested to be re-rendered, only those auxiliary canvases, which are invalid, are actually redrawn, saving the processor time, which would be otherwise spent drawing the remaining canvases.

This technique speeds up chart rendering significantly, as only portions of charts are redrawn due to UI events. It also implies that every event that affects rendering of UI elements has to come with additional information specifying, which auxiliary canvases need to be invalidated. However, with semantical division of the chart (such as partitioning into data area, scale labels and the mouse cross), this does not seem a complicated task.

### 4.5.2 Pixel Drawing

One of the UI bugs which proved quite tedious to resolve involved colored rectangles drawn to represent individual pixels of a TPX detector. Given the possibility of zooming, every frame chart has to be able to plot at least $2 \times 2$ and at most $256 \times 256$ of such pixel rectangles. Since dimensions of the charts adapt to the dimensions of the browser window, every time the window is resized, new dimensions of pixel rectangles need to be calculated.

This calculation is fairly straightforward, but since it utilizes floating-point arithmetic, its results may happen to be imprecise in some cases. And due to this imprecision, pixel rectangles in frame charts used to suffer from periodical fractional offsets, which manifested themselves in the form of a thin grid.

To resolve this issue, the auxiliary canvas responsible for data plots is automatically scaled to size, which is slightly greater than the actual amount of pixels available for rendering, but is a multiple of the number of detector pixels to plot. Consequently, the calculated pixel rectangle dimensions are integer values, and thus allow the usage of *the ImageData API*[2]. After the pixels are rendered in the canvas, it is scaled back down to the exact size of the plot area. At this point, antialiasing is performed. However, since the canvas has already been filled with colored rectangles, there are no transparent pixels from which a grid-like structure could be formed. For that reason, a conventional antialiasing algorithm will produce a satisfying picture.

### 4.5.3 Subpixel Rendering

To render all charts sharply, subpixel rendering is supported. Screens with standard pixel resolutions map logical pixels (referenced by the software) to physical pixels (present in the hardware) bijectively. In comparison, high resolution screens divide their logical pixels uniformly in a way, such that a single logical pixel is displayed by multiple physical pixels. The ratio between these two values is defined in the HTML 5 standard [6] as `devicePixelRatio`[3].

When determining chart sizes prior to rendering, the visualization UI reads the value of the pixel ratio of the current screen and uses it as a coefficient to upscale all canvases accordingly. Contents of charts are then drawn on the canvases, utilizing the enhanced resolution of the screen.

## 4.6 Deployment

Because the server application consists of two independent daemons, each operating on a diferent platform, a dedicated deployment method is used. In this section, such method is described in detail.

### 4.6.1 Extension Script Translation

For coding comfort, the website source makes use of extension languages such as LESS or TypeScript. These languages are respective extensions of the standard CSS and JavaScript, implementing somewhat useful features, which are missing or are not supported by their base counterparts. However, since ordinary web browsers are not capable of parsing their advanced syntax, all LESS and TypeScript source files need to be first translated by a third-party tool before the web application can be deployed.

The products of the translation process are further optimized for deployment by the process of minification, which removes source code comments, white spaces and obfuscates variable and method names in order to compress file size. Since the code is separated into multiple files, such files are concatenated to minimize the amount of HTTP requests needed

[2] The ImageData API is a part of HTML Canvas API, which allows its users to render pictures by setting values of color components of individual pixels in the bitmap. This approach is faster than the conventional API and guaranteed to not use antialiasing. For more information, see section 4.12.4.2.16 of [6].

[3] For instance, Retina display of Apple iPhone 4 has physical resolution $960 \times 640$ and logical resolution $480 \times 320$. Its pixel ratio is therefore 2.

to fully load the visualization UI. At the end of this procedure, two compressed files in CSS and JavaScript are generated.

### 4.6.2 Bower Dependencies

The web application utilizes various open source libraries, most notably jQuery and the Bootstrap Framework. To manage all such dependencies in a well arranged way, the Bower dependency management system is used. The web visualization specifies a *Bower manifest file* in the JSON format. This file lists all dependencies as well as their minimum compatible versions.

When the visualization is deployed, the Bower tool is executed to fetch the latest versions of the required libraries and add their redistributable resources to website contents. This process is deeply integrated with the extension script translation, as the redistributable resources are concatenated with other stylesheets and scripts and minified together. Some dependencies even support strongly-typed bindings in TypeScript. These bindings are downloaded along with the redistributable resources and are used to validate client-side scripts during the translation phase.

### 4.6.3 Grunt Build System

The Grunt system links all deployment phases together. In a way similar to Makefile, it allows target-driven execution of tasks with possible dependencies and variable arguments. This system is also integrated with the standard Node.js package manager application. To deploy the web visualization UI on the server, two main targets are defined in the application's *Gruntfile*:

**Production Target** The primary purpose of this target is to deploy the web visualization UI with maximum emphasis on speed optimization. All possible files including dependencies are concatenated and minified to minimize loading time of the website in the production environment.

**Debug Target** This target is not intended for production use. Its purpose is to maximize readability of the code for the purposes of debugging and development. To achieve this effect, several build phases such as minification and obfuscation are skipped.

# Chapter 5

# Conclusion

## 5.1 Data Import

Since the JSTP server uses the index database to locate data files based on any given start time, all data files subject to visualization must be stored in the ROOT format and referenced in the `rootfiles` SQL table.

Accounting for the ever-growing nature of ATLAS-TPX footage, a periodical prodcedure needs to be performed every time new data arrives from CERN, in order to keep the visualization UI up-to-date. At the time of writing this work, this procedure is semi-automated and initiated manually every day by the researchers at IEAP.

### 5.1.1 Processing Stages

At CERN, the control PC generates raw data files from the read-out interface of every detector in the network. Data is produced on a hourly basis and saved in the form of multi-frame files. Using FTP, these files are transferred into a temporary directory on the target hard drive. When all file transfers are completed and the validity of files is confirmed, several scripts designed to check consistency of detector acquisition are executed. These scripts analyze contents of received files, reading common configuration values such as the acquisition time or bias voltage, and attempt to find variations between individual frames.

Should these scripts succeed in detecting suspicious values, the files in question are moved into a separate directory and await further inspection by researchers. Otherwise, they move on to the next stage of processing. In this stage, captured frames are subjected to the cluster analysis (for more information about this process, see section 1.2.2). Should the frames be captured in the TOT mode, at this point calibration data is used in combination with the method described in [4] to calculate energy values. At the end of this process, ROOT files are produced.

Since the original multi-frame files are not needed anymore, they can be discarded without data loss (or compressed for the purposes of long-term archiving). The ROOT files are moved from the temporary directory to their final location on the hard drive containing the ATLAS-TPX footage database. Subsequently, a dedicated instance of the JSTP data server application is configured to generate information in the index database, in effect registering

them for retrieval of JSTP information. From this point onward, the JSTP server as well as the web visualization UI are able to read frames stored within newly added files.

### 5.1.2 Automation of the Procedure

At the present time, the procedure of extending TPX database with new footage is time-consuming as its processing stages often perform similar operations repeatedly for diferent purposes. For example, a single enumeration of multi-frame files should suffice both for consistency checking and cluster analysis. Furthermore, index data corresponding with produced ROOT files can be generated at the time of their production.

It is however worth noting that this procedure was originally designed in the fall of 2015 with the sole purpose of transferring data from CERN to IEAP. Its automation is currently under investigation.

## 5.2 Future of the Application

At the time of writing this work, the server application is hosted at IEAP. The web visualization UI is publicly accessible online[1], periodically updated with the latest ATLAS-TPX footage. Its internal components consitute a rudimentary data warehouse, which not only offers several terabytes of research data, but also serves to further promote scientific works done by IEAP, CTU, ATLAS and Medipix collaboration at CERN.

In the future, the physical machine hosting the application is expected to be upgraded and relocated to CERN. Consequently, the JSTP server would be updated for compatibility with EOS, which would be used to substitute a conventional file system. The PostgreSQL server used to manage the index database could be also replaced by CERN's OnDemandDb service. Moreover, while hosted at CERN, the JSTP server may be modified to directly receive data streams of captured information in the multi-frame format and process them automatically in a queue. This setup might reduce the time interval between data acquisition and visualization from days to hours or possibly even minutes, with respect to the limitations of hardware and local area network available at CERN.

A slightly modified version of the server application might also be utilized to offer access to footage captured by TPX detectors installed in different experiments operated in collaboration with IEAP. At the present time, the author of this work is investigating its possible deployment in the MoEDAL experiment CERN and the RISESat satellite collaboration. In the end, it is his belief that this application has the potential to become an data visualization software for any scientific experiment gathering data from TPX detectors, capable of efficiently operating with big amounts of research data.

[1] <http://atlastpx.utef.cvut.cz>

# Bibliography


[1] BRUN, R. – RADEMAKERS, F. {ROOT} — An object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*. 1997, 389, 1–2, s. 81 – 86. ISSN 0168-9002. doi: http://dx.doi.org/10.1016/S0168-9002(97)00048-X. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S016890029700048X>. New Computing Techniques in Physics Research V.

[2] COLLIN, L. *A Quick Benchmark: Gzip vs. Bzip2 vs. LZMA* [online]. 2005. Dostupné z: <https://web.archive.org/web/20150907021223/http://tukaani.org/lzma/benchmarks.html>.

[3] FIELDING, R. et al. Hypertext Transfer Protocol – HTTP/1.1, 1999.

[4] 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).

[5] NETHERCOTE, N. – SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, 42, s. 89–100. ACM, 2007.

[6] *HTML: A Living Standard* [online]. 2016. Dostupné z: <https://web.archive.org/web/20160304140644/https://html.spec.whatwg.org/>.

[7] TURECEK, D. Software for Radiation Detectors Medipix. Master's thesis, Czech Technical University in Prague, Czech Republic, 2011.

# Appendix A

# Database Creation Scripts

In this appendix, we include several PostgreSQL scripts used to create the index database.

```
CREATE ROLE tpx_readers
  NOSUPERUSER INHERIT NOCREATEDB NOCREATEROLE NOREPLICATION;

CREATE ROLE tpx_writers
  NOSUPERUSER INHERIT NOCREATEDB NOCREATEROLE NOREPLICATION;
```

Listing A.1: Definition of user access roles necessary to read and modify the database.

```
CREATE SEQUENCE seq_sid
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 15
  CACHE 1;
ALTER TABLE seq_sid
  OWNER TO tpx_writers;
GRANT ALL ON SEQUENCE seq_sid TO tpx_writers;

CREATE TABLE sensors
(
  sid integer NOT NULL DEFAULT nextval('seq_sid'::regclass),
  name text,
  calibration_layer1 double precision,
  calibration_layer2 double precision,
  CONSTRAINT pk_sid PRIMARY KEY (sid)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE sensors
  OWNER TO tpx_writers;
GRANT ALL ON TABLE sensors TO tpx_writers;
GRANT SELECT ON TABLE sensors TO tpx_readers;
```

Listing A.2: Definition of the *sensors* table.

```
CREATE SEQUENCE seq_fid
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 3688
  CACHE 1;
ALTER TABLE seq_fid
  OWNER TO tpx_writers;
GRANT ALL ON SEQUENCE seq_fid TO tpx_writers;

CREATE TABLE rootfiles
(
  fid integer NOT NULL DEFAULT nextval('seq_fid'::regclass),
  path text NOT NULL,
  date_added timestamp without time zone NOT NULL DEFAULT timezone('utc'::text, now()),
  start_time timestamp without time zone NOT NULL,
  end_time timestamp without time zone NOT NULL,
  count_frames integer NOT NULL,
  count_entries integer NOT NULL,
  date_checked timestamp without time zone NOT NULL DEFAULT timezone('utc'::text, now()),
  sid integer NOT NULL,
  checksum character varying(40),
  CONSTRAINT pk_fid PRIMARY KEY (fid),
  CONSTRAINT fk_sid FOREIGN KEY (sid)
      REFERENCES sensors (sid) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
  OIDS=FALSE
);
ALTER TABLE rootfiles
  OWNER TO tpx_writers;
GRANT ALL ON TABLE rootfiles TO tpx_writers;
GRANT SELECT ON TABLE rootfiles TO tpx_readers;

CREATE UNIQUE INDEX idx_path
  ON rootfiles
  USING btree
  (path COLLATE pg_catalog."default");

```

Listing A.3: Definition of the *rootfiles* table.

```
CREATE SEQUENCE seq_frid
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 178994830
  CACHE 1;
ALTER TABLE seq_frid
  OWNER TO tpx_writers;

CREATE TABLE frames
(
  start_time timestamp without time zone NOT NULL,
  fid integer NOT NULL,
  dsc_entry integer NOT NULL,
  clstr_first_entry integer,
  clstr1_count integer NOT NULL,
  clstr2_count integer NOT NULL,
  clstr3_count integer NOT NULL,
  clstr4_count integer NOT NULL,
  clstr5_count integer NOT NULL,
  clstr6_count integer NOT NULL,
  sid integer NOT NULL,
  acquisition_time interval NOT NULL,
  occupancy integer,
  frid bigint NOT NULL DEFAULT nextval('seq_frid'::regclass),
  CONSTRAINT pk_frid PRIMARY KEY (frid),
  CONSTRAINT fk_fid FOREIGN KEY (fid)
      REFERENCES rootfiles (fid) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE CASCADE,
  CONSTRAINT fk_sid FOREIGN KEY (sid)
      REFERENCES sensors (sid) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE CASCADE
)
WITH (
  OIDS=FALSE
);
ALTER TABLE frames
  OWNER TO tpx_writers;
GRANT ALL ON TABLE frames TO tpx_writers;
GRANT SELECT ON TABLE frames TO tpx_readers;

CREATE INDEX fki_fid
  ON frames
  USING btree
  (fid);

CREATE INDEX fki_sid
  ON frames
  USING btree
  (sid);

CREATE UNIQUE INDEX id_start_time_fid
  ON frames
  USING btree
  (start_time, fid);

CREATE UNIQUE INDEX idx_fid_dsc_entry_start_time
  ON frames
  USING btree
  (fid, dsc_entry, start_time);

CREATE INDEX idx_start_time
  ON frames
  USING btree
  (start_time);

CREATE UNIQUE INDEX idx_start_time_sid
```

# Appendix B

# Documentation of JSTP Web Methods

This chapter includes detailed documentation of the JSTP web service along with protocol conventions, parameter descriptions and examples of requests and responses.

## B.1 Conventions

Note that when referring to the service endpoint in method URLs, we use `<endpoint>` as a stand-in string. TODO

## B.2 Detector List

To execute this method, a client must initiate GET request to `<endpoint>/sensors` without any parameters. When successful, the server responds by returning an array of objects, each of which corresponds to a single device in the network. Example of such response is provided in Listing B.1. Every object in the array is guaranteed to contain:

**`sid`** Unique numeric identifier of the device retrieved from the index database.

**`name`** Readable name of the device.

## B.3 Overview of Acquisition

To execute this method, a client must initiate POST request to `<endpoint>/timeline`. The request body must contain a JSON object with *all* parameter values. You can examine an example request in Listing B.2.

When successful, the server responds by returning an array of objects, each of which responds to a single interval in the time period. For example response, see Listing B.3. Every object in the array is guaranteed to contain:

```
[
  {
    "sid": 1,
    "name": "tpx01"
  },
  {
    "sid": 2,
    "name": "tpx02"
  }
]
```

Listing B.1: Example response containing a list of two devices.

```
{
  "startTime": 1438052400,
  "endTime": 1438063200,
  "groupPeriod": 3600,
  "sensors": [1, 2],
  "normalize": true
}
```

Listing B.2: Example request body with time period starting at July 28, 2015 at 3:00 AM and ending at 6:00 AM. Data from 2 detectors is requested to be normalized and grouped by every hour. Response is expected to contain exactly 3 intervals.

**time** UNIX timestamp in UTC of the start time of the interval. End time of the interval can be calculated at by adding `groupPeriod` to this value.

**frames** Number of frames aggregated in the time interval.

**occupancy** Count of non-zero pixels in all aggregated frames, indicating the levels of saturation. The maximum possible occupancy is equal to the product of pixels in a single sensor layers, the number of sensor layers and the number of aggregated frames in the interval.

**counts** Array of counts of clusters in all aggregated frames, differentiated by their type classification. Counts are provided in the order: dots, small blobs, heavy blobs, heavy tracks, straight tracks, curly tracks.

If the calculations are normalized, individual contributions to these counts from every frame are divided by frame's acquisition time, yielding overall flux instead of counts.

## B.4 Frame Search

To execute this method, a client must initiate POST request to `<endpoint>/frame`. The request body must contain a JSON object with *all* parameter values:

**time** UNIX timestamp in UTC of the search time parameter.

```
[
  {
    "time": 1438052400,
    "frames": 2,
    "occupancy": 3,
    "counts": [0.15, 0, 0, 0, 0, 0]
  },
  {
    "time": 1438056000,
    "frames": 3,
    "occupancy": 2,
    "counts": [0.0666667, 0, 0, 0, 0, 0]
  },
  {
    "time": 1438059600,
    "frames": 2,
    "occupancy": 16,
    "counts": [0.1, 0.1, 0, 0, 0, 0.05]
  }
]
```

Listing B.3: Example response to the request from Listing B.2.

**sensors** Array of distinct `sid` values of the devices, from which we wish to retrieve data. This array must not be empty.

**searchMode** A non-negative integer value specifying the algorithm to be used in the search operation. Possible values are 0 for the Sequential Forward Mode and 1 for the Sequential Backward Mode.

**integralFrames** A positive integer not greater than 100 controlling the number of frames integrated in time. Value equal to 1 retrieves only a single frame per device.

```
{
  "time": 1438052400,
  "sensors": [1],
  "searchMode": 0,
  "integralFrames": 1
}
```

Listing B.4: Example request body with time parameter equal to July 28, 2015, 3:00 AM. A single frame captured by a single detector is requested to be located by the Sequential Forward Mode.

For an example request, see Listing B.4. In response, the server returns an object containing `foundTime`, the start time of the master frame, and `frames`, an array of objects corresponding with frames captured by every device in order, in which they were referenced in the `sensors` array. Every object is guaranteed to contain:

**rootFile** Path to the ROOT file, from which this frame was extracted (in the server's file system).

**rootFrameIndex** Index of the entry in ROOT file's `dscData` tree, containing information about detector configuration.

**rootFirstClusterIndex** Index of the first entry in ROOT file's `clusterFile` tree, corresponding with the first cluster in the frame. If no such entry exists, this value is null or negative.

**layers** Number of detector's sensor layers.

**startTime** UNIX timestamp in UTC of the start time of acquisition.

**acquisitionTime** The acquisition time (the length of acquisition) in seconds.

**biasVoltage** TODO

**mode** TODO

**chipboardId** TODO

**maskedPixels** TODO

**calibrationConstants** TODO (only TOT)

**clusters** TODO

# Appendix C

# Nomenclature

AFP Apple Filing Protocol, a network protocol mainly used for providing shared access to files on clients and servers compatible with operating systems developed by Apple Computer, Inc.

API Application Programming Interface, a set of routines, protocols and tools for building software and applications.

ASIC Application-specific Integrated Circuit.

ATLAS A Toroidal LHC Apparatus, one of particle detector experiments constructed at LHC.

CERN European Organization for Nuclear Research (French name: *Conseil Européen pour la Recherche Nucléaire*), based in Geneva, Switzerland.

CIFS Common Internet File System. See SMB.

CPU Central Processing Unit.

CRUD Create, Read, Update, Delete, four basic operations of persistent storage system.

CSS Cascading Style Sheets, a language used to describe presentation of web pages.

CTU Czech Technical University (Czech name: *České vysoké učení technické*), based in Prague, Czech Republic.

DCS Detector Control Systems, a system providing control of subdetectors and of common infrastructure of the experiment and communication with the services of CERN.

DOM Document Object Model, a family of XML parsers which generate a tree structure in memory from parsed content.

DPI Dots per Inch, a measure of screen pixel resolution.

EOS A primary storage system at CERN for LHC experiments.

FTP File Transfer Protocol, a network protocol mainly used for providing shared access to files.

HTML Hypertext Markup Language, a standard markup language used to create web pages. [6]

HTTP Hypertext Transfer Protocol.

IEAP Institute of Experimental and Applied Physics (Czech name: *Ústav technické a experimentální fyziky*), based in Prague, Czech Republic.

JSON JavaScript Object Notation, a data format derived from JavaScript.

JSTP JSON Timepix Protocol, a protocol used to transmit captured frames to the web visualization UI. For its description, see section 3.

LESS A dynamic style sheet language which can be compiled into CSS.

LHC Large Hadron Collider, an experimental factility built by CERN.

LS Long Shutdown, a period in CERN time schedule characteristic by temporary cesation of operation of particle accelerators and increased maintenance.

MIME A two-part file format identifier.

MPX Medipix, a semiconductor pixel detection chip.

OS Operating System.

ROOT An object oriented data analysis framework. [1]

RPC Remote Procedure Call, a mechanism used to execute computer subroutines on machines over a network.

SAX Simple API for XML, a family of XML parsers which generate various events while processing content sequentially.

SMB Server Message Block (also known as the Common Internet File System), a network protocol mainly used for providing shared access to files.

SQL Structured Query Language, a language designed to define, manage and query data in a relational database system.

SSH Secure Shell, a cryptographic network protocol commonly used for remote command-line access and remote command execution.

TOA Time of Arrival operation mode. For more information, see section 1.1.1.

TOT Time of Threshold operation mode. For more information, see section 1.1.1.

TPX Timepix, a semiconductor pixel detection chip successing Medipix2.

UI User Interface.

UNIX A family of computer operating systems.

URI Uniform Resource Identifier, a string of characters used to identify a resource.

URL Uniform Resource Locator, commonly known as web address.

UTC Coordinated Universal Time, a primary worldwide standard used to regulate clocks and time.

XML Extensible Markup Language, a standard markup language.

# Appendix D

# Obsah přiloženého CD

**Tato příloha je povinná pro každou práci. Každá práce musí totiž obsahovat přiložené CD. Viz dále.**

Může vypadat například takto. Váš seznam samozřejmě bude odpovídat typu vaší práce. (viz [**?** ]):

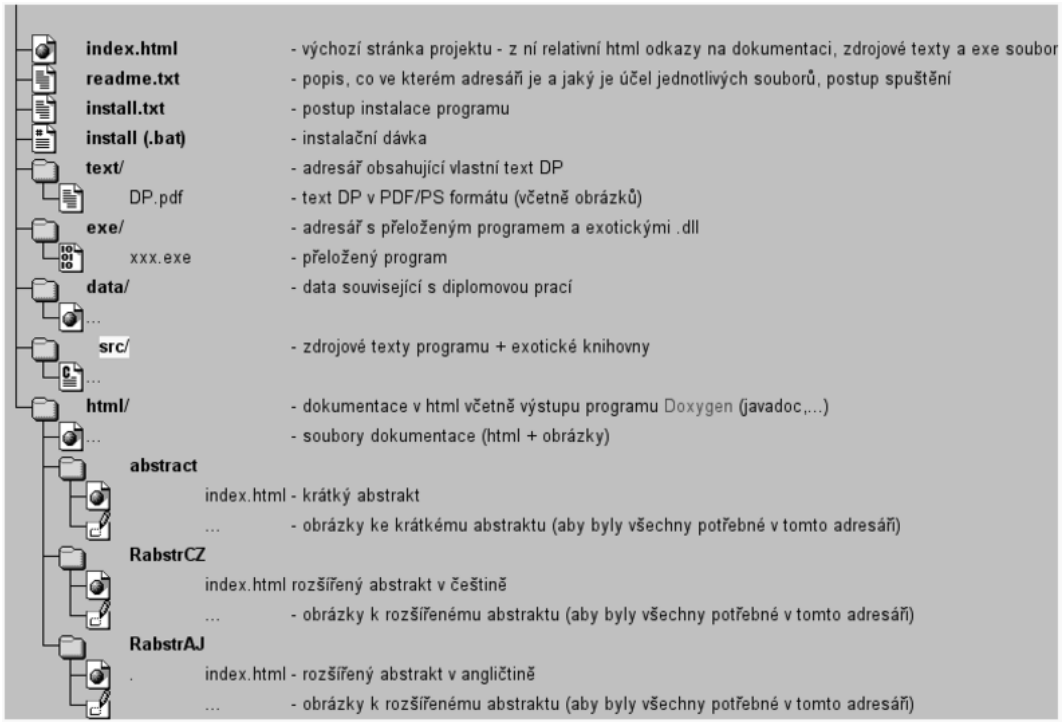



Figure D.1: Seznam přiloženého CD — příklad