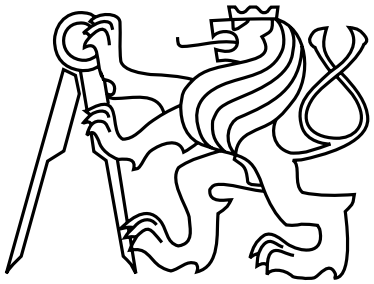
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



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 15, 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 the Timepix Detectors

1.2 The Timepix Network at ATLAS

1.3 The Problem of Efficient Data Manipulation

1.4 Structure of This Document

1.5 Raw Timepix Output

Timepix 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 256×256 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.

Timepix 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 the camera analogy, the acquisition time resembles the 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.

Provided that every Timepix detector installed in the ATLAS-TPX network has 2 layers of 256×256 pixel matrices, every captured frame consists of 131,072 integer values in total.

Interpretation of these values depends on another parameter, *the operation mode*. While it is technically possible to configure every pixel to operate in a different mode, for the desired application all pixels are set to the same mode of operation, making this essentially not a parameter of a pixel, but that of a frame. The following operation modes are available:

Hit Counting Mode (also known as the Medipix Mode) In this mode, the counter is incremented upon every transition from a state below the threshold to a state above the threshold. The result is an integer value representing the number of particles which have interacted with the pixel.

Time over Threshold Mode (TOT) In this mode, the counter is incremented by every clock cycle spent above the threshold. The result is an integer value corresponding to the energy of the interacting particle. Energy calibration methods are described in [4].

Time of Arrival Mode (TOA) In this mode, the counter is incremented by every clock cycle after the threshold is first exceeded. The result is an integer value corresponding to the time interval before the end of the measurement.

1.5.1 Read-out Interface

A read-out interface is a special dedicated hardware device that reads data and controls acquisition of the detector. [6] 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 server room, protected 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 1.1.

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.

1.5.2 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, if the frame has been captured in TOT mode and calibration data are available, the automated processing script can convert counter values to energy approximations.

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

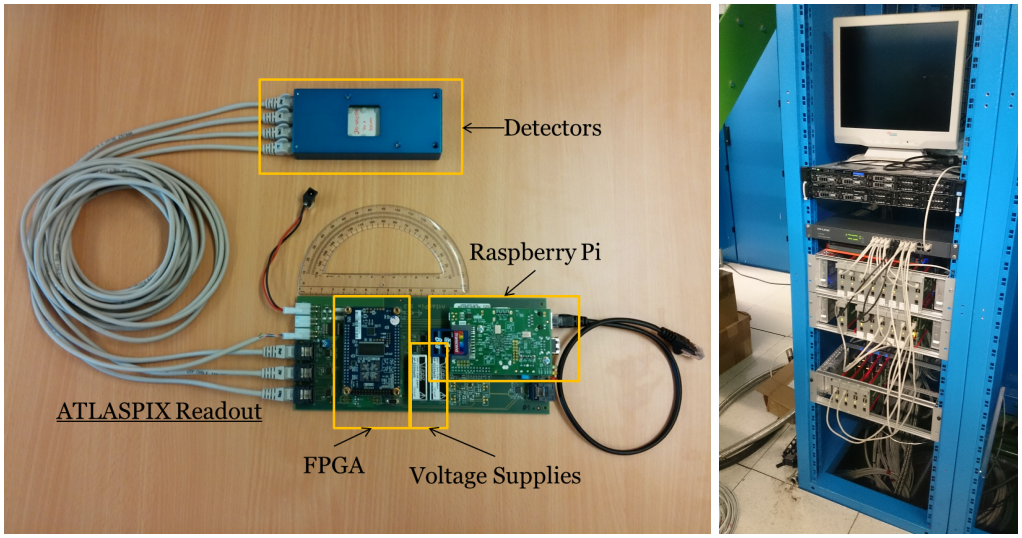


Figure 1.1: The ATLASPIX read-out interface installed at CERN.

non-zero counter value. Consequently, 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 approach produces 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 lost at the end of the 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 average of pixel coordinates in the cluster. In analogous way, the volumetric centroid is the very same average weighted by corresponding counter values.

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

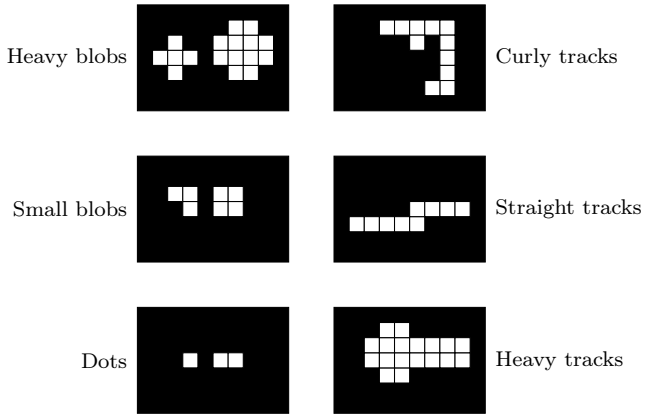


Figure 1.2: Different cluster types classified by their shapes.

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

1.6 Common Storage Formats

1.6.1 Plain Text

The most straight-forward way of storing data acquired by Timepix detectors is to use plain text files. Such output, referred to as the single-frame or the multi-frame format, commonly stores 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 by annotating 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 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 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.

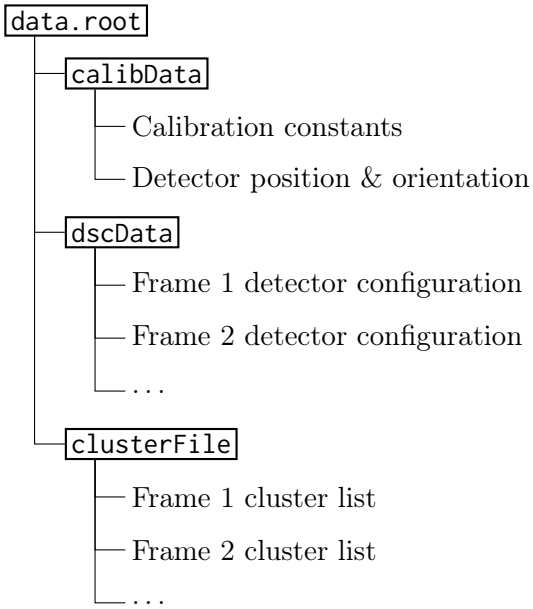


Figure 1.3: Structure of a ROOT file containing Timepix footage.

1.6.2 ROOT Framework

Another storage option is 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 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 caveats 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 1.3) 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 full tree data structure might be overly-complicated

and too generalized for a simple output described in previous sections. Lastly, ROOT framework has quite a complex object structure, making it hard to learn for first-time users.

Chapter 2

Data Structure and Storage

In this chapter, a data scheme capable of archiving Timepix footage for longer time periods is proposed. The primary concern is to minimize access latency in queries based on the time of measurement and the device of origin. For that reason, several auxiliary data structures, such as the index database, are introduced.

2.1 Formal Requirements

Requirements on such a data structure are as common as database requirements can get. It should 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 Timepix 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, our database should 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 will be primary storage site for all research data, there should be multiple independent copies of it as backups and the database structure should be designed with logic to enable timely synchronization of these copies.

Apart from all the requirements already listed, it is important 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 or by device. This is indeed a very natural approach, provided that every device 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 involved only a specific group of detectors in the network.

2.2 Database

In this section, the Timepix 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 based on a UNIX file system. This approach enables many users, not necessarily only those using UNIX-based operating systems, to access it directly by means of widely-used and standardized 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.

TODO

The file system has two directories named `processed` and `downloading`, corresponding to the respective sections of the database. In these directories, data is further grouped 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).

If it is not practical to group files by the day of acquisition, footage can be grouped by an alternative 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

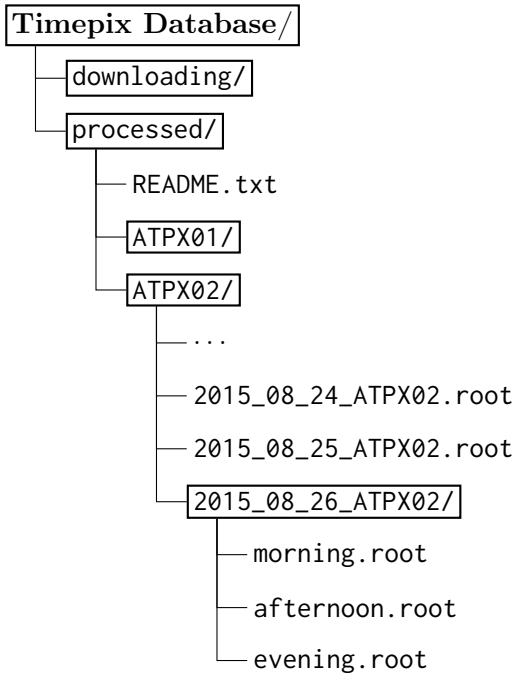


Figure 2.1: Example of the database file system structure.

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

To preserve storage space, various compression methods. Supported compression formats are ZIP, GZIP and TAR, or any combination of them. 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.

Lastly, it is forbidden to store any recursively compressed structures (such as archive within other archive, etc.). The recommended alternative is to increase compression level in already existing archives instead of constructing new ones. Note that this rule also applies for all data formats which use compression inherently, such as ROOT.

TODO

2.2.2 Expected Volume of Data

With our definition in mind, we will now perform a simple extrapolation to obtain an upper bound on the size of our 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), we have 96 gigabytes per

sensor per day. Accounting for the longest possible time of operation, our database will store up to 2,437 days of footage simultaneously recorded by up to 20 detectors. That means that our 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 this compression on our multi-frame data files, our database would have to hold *only* about 1.1 petabyte of archives.

We will now perform analogous calculation for the ROOT file format. Since the file structure already utilizes its own proprietary compression algorithms, we expect the overall volume to decrease significantly in comparison with the raw uncompressed multi-frame data. From the data recorded by the ATLAS Timepix network in the fall of 2015, we observe that a single day of footage stored in the ROOT format may take up to 18 gigabytes in size. Using the same constants as before, we arrive at the conclusion that our database will have to hold about 877 terabytes of information. This result is in agreement with our expectations.

Please note that neither of these upper bounds is by any means, since we intentionally over-estimated the number of detectors in our network and the length of the operation period in our assumptions. In addition, it is likely that some of our detectors will be configured to capture data with frequencies lower than the maximum possible frequency as every device is configured separately to observe particles at different speeds. For all these reasons, our estimation only gives us vague information about the orders of magnitude of storage space required to operate our database and its subsequent backups. In spite of this limitation, the estimation suffices to design and rate other components of our system.

2.3 Index Database

So far, we have established a set of rules for our 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, our rules do not define any conventions regarding retrieval of specific frames from files in the ROOT data format. Due to this limitation, users seeking individual frames will have to download bulks of data from 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, the file structure inside of such directories is undefined, and may require additional decisions on the user side. And what do we do when we want to retrieve frames based on different criteria than time and device of origin? At the moment, we have no option other than to directly enumerate frames stored in all files in our file system, which (considering their potential size) might not be a preferrable solution. To resolve all these issues, we will introduce one more element to our design—an index database.

This database will be contain information which can be recalculated at any instant from the primary data files, hence it will not need to be backed up. The information stored in our database will mostly include, as the name suggests, index of all files and frames on the record

and addresses pointing to the them on our file system. In addition, the index database will also store some commonly requested aggregated values.

2.3.1 Definition

The index database will be compliant to the SQL standard, so that users may design their own queries. For the reasons of simplicity, we will define only three basic entities in our database. 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 Timepix 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 Timepix 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 Timepix 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 Timepix 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 Timepix 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 the definition, it follows that our index database will help deterministically resolve all time-based queries, even in situations when frames are stored in an undefined directory structure. Apart from this optimization, the database will also provide file validation primitives to ensure that any corrupted files are discovered as soon as possible. But there is one more significant performance optimization we have so far neglected to mention.

When retrieving frames by the time and device of origin, we can use the predefined naming patterns to obtain a path in the file system. In case the path points to a directory, we can consult the index database to quickly scan for a file containing the information we need. Still, we are left with an unoptimized task since in order to retrieve the specific frame (or frames) we are looking for, we will have to scan the entire file, which may be several gigabytes in size.

This issue may be in part resolved by sorting all frames in our data files consistently by their start time, allowing us to use a binary search algorithm instead of regular one, thus reducing the complexity of the operation from linear to logarithmic. But we can still do better. Since we already perform lookups in the index database, we can us them to retrieve data addresses, which will point us to specific locations in the file. This way, no search will be needed at all.

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 in a single frame. Entries belonging to the same frame can be identified by having equal value of the `Start_time` leaf. If we agree to sort all entries in both trees by this leaf value, the `clusterFile` entries will consequently form continuous bulks of data corresponding to individual frames. This means that once we discover the bulk belonging to the frame we want, we only need to read entries as long as the start time remains the same (or we reach the last entry). For every frame, we can then store the index of the corresponding `dscData` entry and the index of the first `clusterFile` entry in the bulk to achieve constant-time search operation, as illustrated in Figure 2.2. This significant benefit comes at the price of 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 our database may want to calculate aggregated statistics. Since these types of requests are hard to anticipate and do not constitute a significant portion

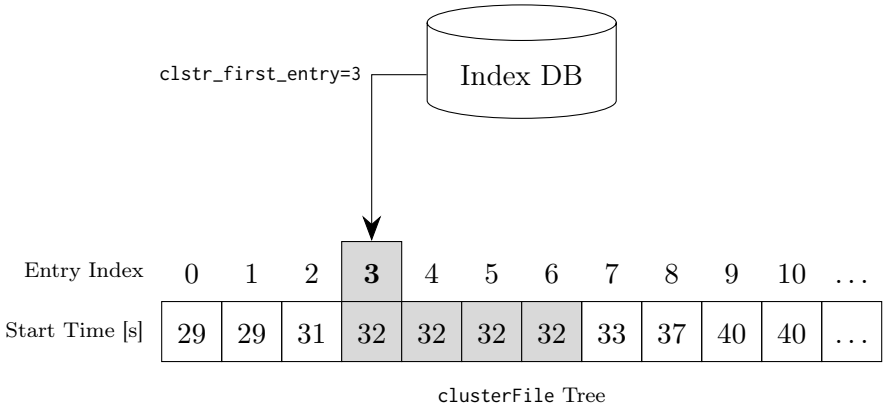


Figure 2.2: Illustration of the optimization mechanism provided by the index database.

of all user queries, it is not worth our effort to create separate data structures in order to accelerate their processing. We can, however, make use of data structures we already have in place. For instance, our index database makes a great candidate in particular since it already contains data associated with individual files and frames, and is easily accessible and queriable using SQL. We will therefore include several statistical values, such as count of clusters differentiated by individual cluster types and frame occupancy encoded as number of non-zero pixels.

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 we used to accelerate our access to individual frames within data files. Using their own tree indices built from various columns of data tables, servers can speed up certain queries containing predicates or orderings based on such columns. We can utilize this mechanism to make our data access procedure even faster, in a sense indexing the index database. To see how exactly we make use of this technique, examine Listings A.2, A.3 and A.4.

Chapter 3

Communication Protocol

In this chapter, we move our focus from the data itself to the JSON Timepix Protocol, a communication protocol we use to transmit Timepix footage for the purposes of visualization. We describe overall scheme of communication and define JSTP in a formal way.

3.1 Remote Access

In the previous chapter, we have defined a database capable of storing footage captured by the ATLAS-TPX network at CERN. Since this database 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[1] network protocols to interact with it remotely.

Since our database already supports concurrent network access for multiple users, defining another dedicated communication protocol such as JSTP seems rather redundant. So, what advantages does this approach offer? The primary motivation for the existence of JSTP is the web visualization UI, which is the subject of Chapter 5. In this application, our users want to observe recorded footage frame by frame. If we do not define our own protocol to facilitate transmissions of individual frames, we are bound to use one of storage formats listed in section 1.6, none of which is particularly suitable for this task. For instance, the Multi-Frame format stores data in multiple files, implying that several parallel downloads would be required just to display even a single frame, possibly putting a strain on user's network connection in the process. The ROOT format uses its own compression algorithms, making it non-trivial to deflate in a website context. Lastly, since both ROOT and Multi-Frame store data in bulks, the information overhead to transmit a single frame would be unbearable, especially considering that files in question may be several gigabytes in size.

With this motivation in mind, let us now state several assumptions about our web visualization, and by extension JSTP. We expect to have multiple users connecting to our server over a local area network or through the Internet. We assume that our users want to see and possibly further inspect some of the frames captured by the ATLAS-TPX network, transmitted in units at a time. We do not wish to transmit all information from our data files,

[1] Recall that in section 2.2.1 we define that our database supports FTP, SMB, SSH, AFP and HTTP access.

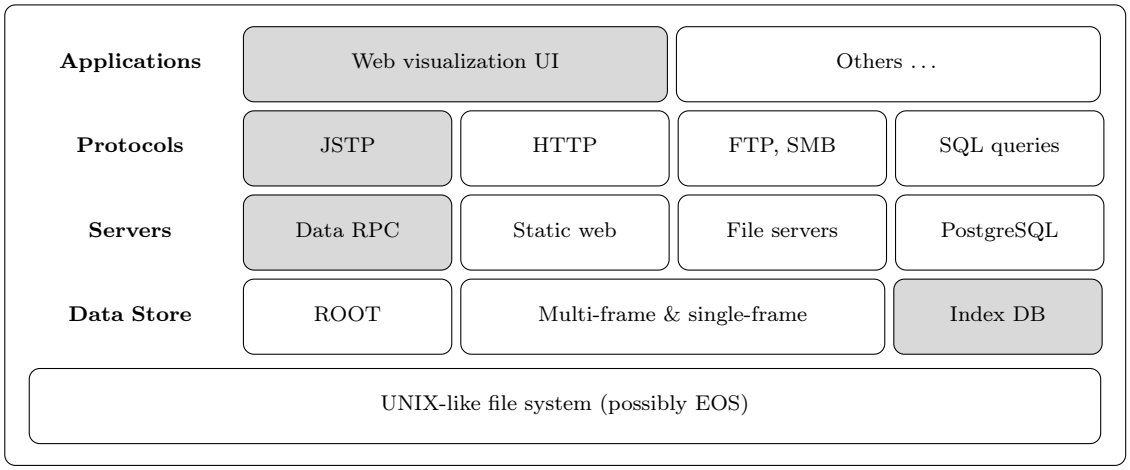


Figure 3.1: A multi-layered system. Proprietary components are emphasized by gray color.

nor do we want to send continuous footage at streaming speeds. Instead, we define JSTP to enable simple access to the most important detector data, and to provide a brief overview of recent detector operation with emphasis on any irregular or pattern-defying events.

3.1.1 Considerations

Note that in designing our system, we would like to uphold a multi-layered architecture. This way, we maintain 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 of our system will always have freedom to choose a component, with which they wish to interact, in turn choosing the level of offered services, processing speed and algorithmic complexity.

We may imagine this as follows. 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 utilizable 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, are welcome to 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.

We should also consider extensibility of JSTP in the future. With multiple concurrent

projects such as MoEDAL-TPX[2], SATRAM[3] and RISESat[4], it is likely that JSTP will be used for compatibility 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

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

In the first version, we require JSTP only to transmit results of the 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 will have to consist 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, we require that JSTP provides an exhaustive list of network elements along with information about their availability in the system. This might seem a bit redundant at first, but consider that JSTP needs to be ready for situations when detectors malfunction, are replaced, or new detectors are installed in the network. These 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, we require JSTP to offer us 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 we listed earlier, we do not place any 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 web protocol and as such, it utilizes HTTP as its underlying standard. This allows us to abstract ourselves from caveats of data transmission and compression, and to focus more on the transmitted data instead. By this declaration, we also indirectly imply 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 these calls.

[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.

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). 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.

Observant readers may 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 instance, 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 data structures and calling conventions.

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 1.5.2).

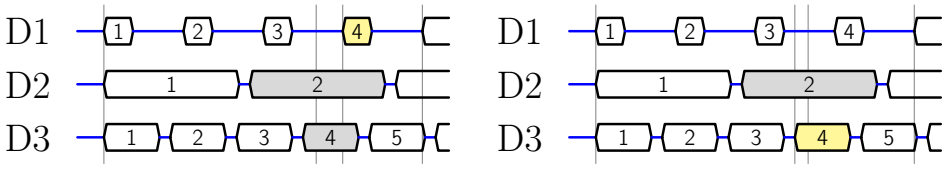
Frame Occupancy Total number of non-zero pixels in all frames in the interval, indicating their levels of saturation.

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.



(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).

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:

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 1.5.2). 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`.

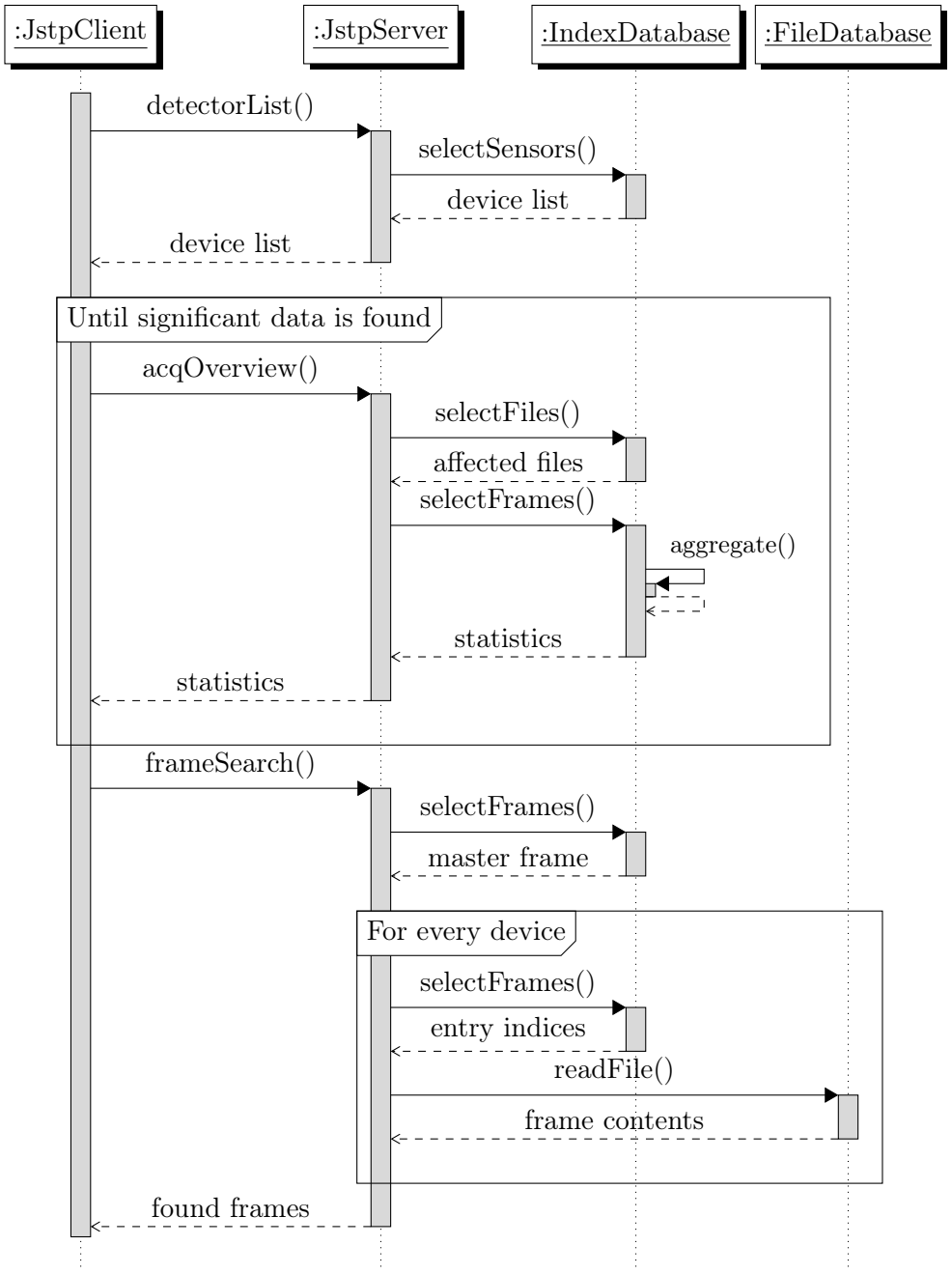


Figure 3.3: UML diagram depicting expected interactions between JSTP client and server, hinting levels of processing complexity at server-side.

Chapter 4

Data Server

This chapter describes implementation of a JSTP server in C++. While sections in the beginning focus on the role of the server in the entire visualization system and its connection to other components of the application, the sections in the end give details on its operation and propose performance optimizations.

4.1 Role of the Application

It has been already mentioned in the definition of JSTP that the primary purpose of the protocol is to facilitate connection between the web visualization UI and the ATLAS-TPX footage database. Since the database is located at the server side, the main responsibility of the JSTP server is to decode stored TPX frames from data files and encode them in JSTP messages as efficiently as possible.

To achieve this goal, the server is expected to interact with not only with its underlying file system (which may be replaced by EOS in the future), but also with the index database. Using the index, the server can accelerate time-based queries, as described in section 2.3.2.

TODO

4.2 Decomposition

The server consists of two major components, an HTTP thread pool and a ROOT transcoder. As the name suggests, the thread pool is responsible mainly for handling outbound HTTP connections, whereas the transcoder ensures fast consumption of data stored in the ROOT file format.

4.2.1 HTTP Thread Pool

The thread pool is a standard component in many other server applications. 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 thread returns to suspended state, awaiting further instructions. This way, threads are recycled within the pool throughout server operation.

Each worker thread manages its own separate set of resources, allowing it process requests autonomously. Since such processing occupies a single core of server's CPU, it follows that the total number of threads corresponds[1] with the number of cores in the CPU.

Apart from reading and writing to the HTTP socket, worker threads are responsible for choosing the correct behavior in order to respond to requests in compliance with the JSTP specification. For instance, every web method listed in section 3.3 is represented by a separate *request handler*. Upon request, a 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.

4.2.2 ROOT Transcoder

The primary purpose of the transcoder is to access detector footage stored within ROOT data files and convert it to JSTP messages. Unlike the thread pool, the transcoder is not a single object but rather a dedicated set of tools and objects designed to efficiently handle large amounts of data.

Recall from section 1.6.2 that the ROOT format stores information in tree structures, separating detector configuration from cluster lists. Due to possibly overwhelming sizes of data files, it is nontrivial to devise 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 In sequential reading, the ROOT data format implicitly uses a file cache to prefetch all buffers for the selected data in the memory.

Tree Switching 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.

[1] The correspondence need not be exact. For instance, when testing the application on machines with 2, 4 or 8 cores, the most effective number of threads was equal to the number of cores multiplied by 4.

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

4.2.3 Auxiliary Components

Apart from the thread pool and the transcoder, the server has multiple auxiliary components:

JSON Serialization In order to produce valid JSTP messages, the server utilizes JSON parser to read method parameters and JSON writer to construct response bodies. Both components are provided by the RapidJSON open source library.

Since the number of request parameters is defined to be constant, the parser operates in a stateless DOM mode, which is memory-inefficient but easy to use. And because produced responses are expected to be large, the writer is configured to operate in more efficient SAX mode, which is harder to work with.

Logging The server produces status messages with multiple levels of severity. To persist these messages, the Google Logging library is used. The library is capable of managing multiple open files, differentiated by the minimal severity of log messages and the date of logging.

Configuration Configuration of the server is provided by the Google Flags library. It allows various configuration settings (most notably server IP, port and behavior) to be customized by passing command-line arguments to the process at startup.

Index Database Connection The server interacts with the index database using the official PostgreSQL connector library for C++. This library essentially provides the ability to execute arbitrary SQL commands. It also conveniently abstracts all type conversions and low-level data transmissions between both server applications, and enables operation in transaction mode.

4.3 Object-Oriented Design

4.4 A Note on Parallelism

4.5 Performance Optimizations

Chapter 5

Web Visualization

5.1 Naive Decomposition

5.2 Final Decomposition

5.3 Underlying Standards

5.4 Dependencies

5.5 Website Structure

Chapter 6

Conclusion

6.1 System Deployment

6.2 Data Import

6.3 Automating Data Acquisition

6.4 Future of the Application

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] 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

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

CRUD

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

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.

HTTP Hypertext Transfer Protocol.

JSON

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

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

MPX Medipix, a semiconductor pixel detection chip.

OS

ROOT An object oriented data analysis framework. [1]

RPC

SAX

SLS

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 acquisition mode. For more information, see section 1.5.

TOT Time of Threshold acquisition mode. For more information, see section 1.5.

TPX Timepix, a semiconductor pixel detection chip successing Medipix2.

UI

UNIX A family of computer operating systems.

URI

URL

UTC

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 [**?**]):

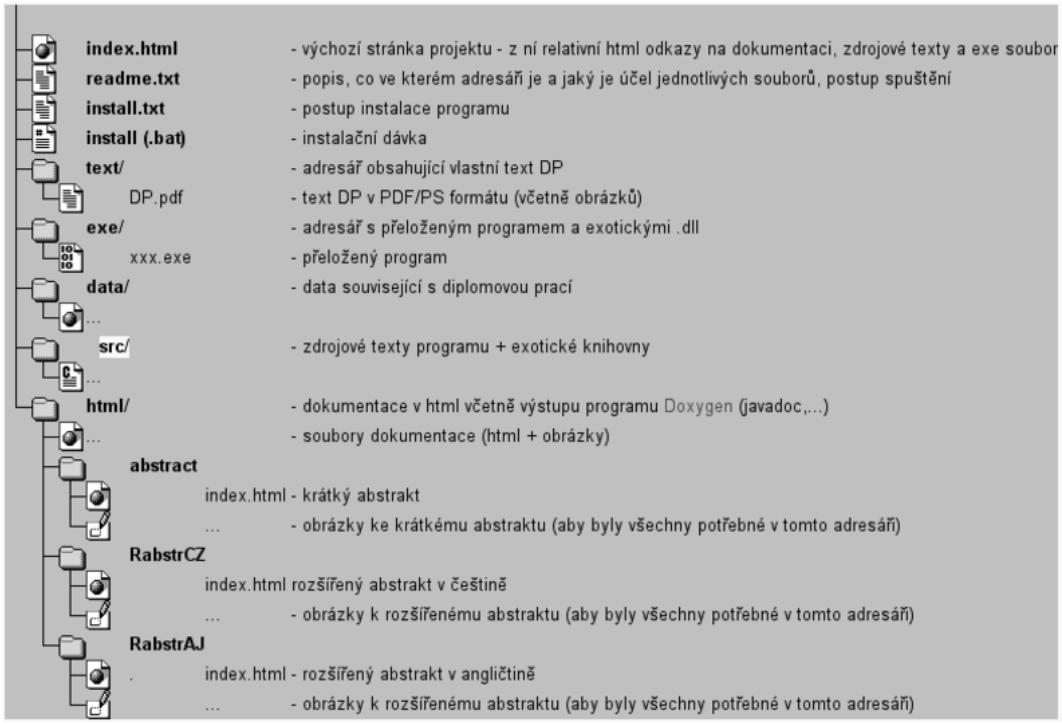


Figure D.1: Seznam přiloženého CD — příklad