VideoTerror Service

Extensible tool for development and deployment of computer vision applications

Technical Report - FIT - VG20102015006 - 2015 – 01

Ing. Vojtěch Fröml
Ing. Tomáš Volf
doc. Ing. Jaroslav Zendulka, CSc.

Faculty of Information Technology, Brno University of Technology

September 11, 2015

Abstract

This technical report describes VTServer, which is a RPC server providing services that can be used by computer vision applications. It is based on VTApi, a framework for image and video data and metadata management, a useful tool for computer vision application development. The objective of both is to make the development of applications easier and more effective.

Contents

1 Introduction

The VideoTerror project was focused on development of image and video processing methods and tools. Several methods have been developed and implemented in the project. (e. g. video summarization or face tracking). These implementations employ a data storage, which contains raw image and video data, metadata related to and/or extracted from this data by them. To simplify this data management and to support operations with it, a framework VTApi (VideoTerror Application Programming Interface) has been developed. It provides developers higher-level operations to access data, supports querying and other more advanced operations. It is based on OpenCV library and PostgreSQL database.

No matter of the fact that the main objective of the project was the development of methods, we decided to extend VTApi to support not only implementation of individual computer vision methods, but also to support integration of several processing modules into more complex applications. As a result VTApi has been extended by processing modules metadata and analytical results management. In addition to this extended functionality of VTApi, another superstructure framework which employs VTApi called VTServer has been developed. It adds further controlling mechanisms and functionality that allows developers to combine processing modules implementing computer vision and other (e.g. analytical ones) into more complex and advanced applications. It designed as a RPC server and in its basic form its service acts primarily as a mechanism for developing and running event-based computer-vision applications. As a sample application based on VTApi and VTServer, an Event-based Video Analysis Tool (EVIDANT) has been implemented.

This technical report is structured as follows. Chapter 2 describes the basic concepts, including VTApi fundamentals. Chapter 3 shows briefly how VTApi can be used in computer vision applications. VTApi and VTServer deployment is described in Chapter 4. Chapter 5 presents the EVIDANT analysis tool.

2 Concepts

2.1 System overview

VideoTerror Service (VTServer) is an application which encapsulates various computer-vision algorithms and data storage mechanisms. It provides easy modular interface for the developmment of both low-level analytical methods and defining complex processing tasks using those; and allows end user to manage data and these tasks with a simple controlling interface. Overview of the service's components is shown in Figure 1.

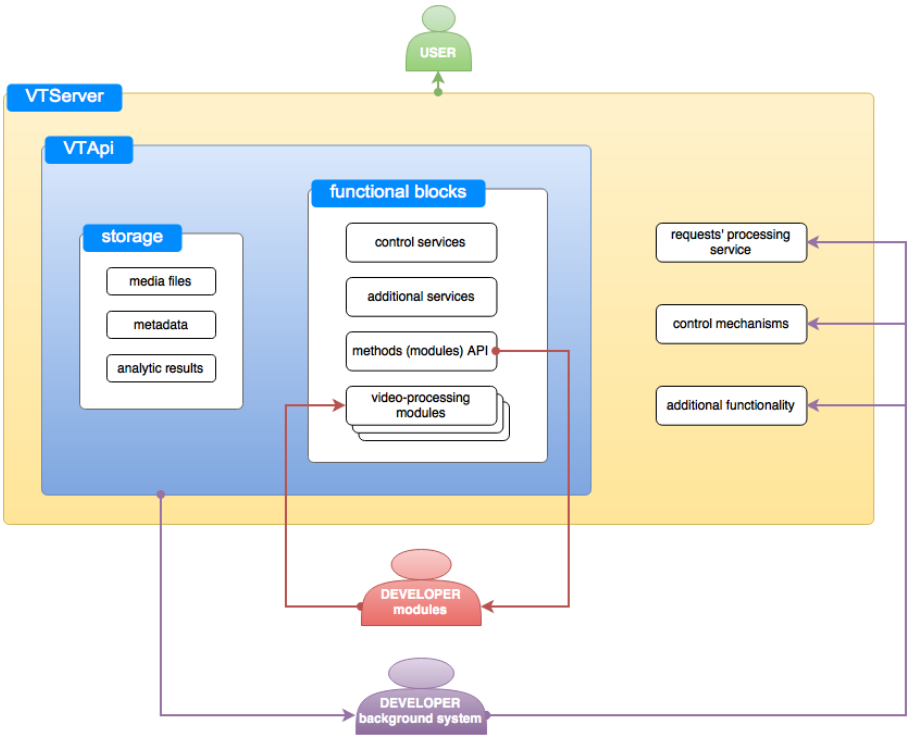

Figure 1: Concept of VTApi + VTServer: user and developer usage

The service is implemented as a convenient extension of VideoTerror API (VTApi). This framework automatically manages manipulation with image data, metadata and computation results. Analytical algorithms may be implemented as VTApi video-processing modules using internal modules API by a programmer. Running, stopping and querying results of these modules is achieved via process control services.

Because VTServer is intended as extension of an existing framework, its functionality may easily be extended or modified according to specific analytical needs. In its basic form, the service acts primarily as a mechanism for developing

and running Event-Based computer-vision applications. It processes requests for data manipulation (add a video, change information about a video...), processing task definitions, control of computation processing instances and querying of result data in a specific videodata event format.

2.2 Application development

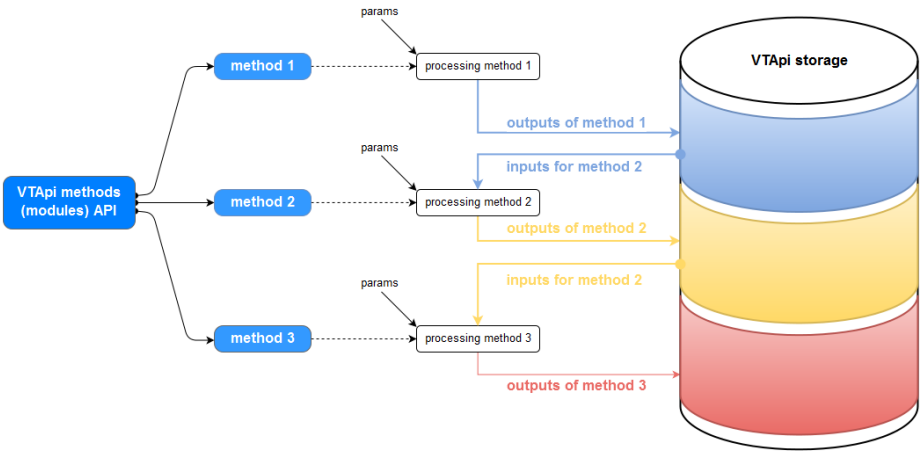

Figure 2: VTApi methods concept

Figure 2 shows an example of an analytical application consisting of three specifically chained algorithms. VTApi provides unified interface for implementations of methods (algorithm). Each method is defined by format of its input/output computed data, input parameters and implementation provided by programmer. In this example above, defining input requirements for methods 2 and 3 as (a subset of) outputs of methods 1 and 2 respectively achieves desired chaining of these partial methods into one application. Intermediary and final results are automatically stored and retrieved from VTApi database using internal methods, no direct database programming is necessary. Running these individuals methods as processes is done in sequence. Each of these computational instances accepts additional algorithm parameters specified by the user. Together with method definition, these parameters define processing task which could be assigned to afore-mentioned process along with video data chosen for analysis. Developer may choose to implement module API directly or use an Event-Based module API which simplifies it further into a need to implement only one video-processing function.

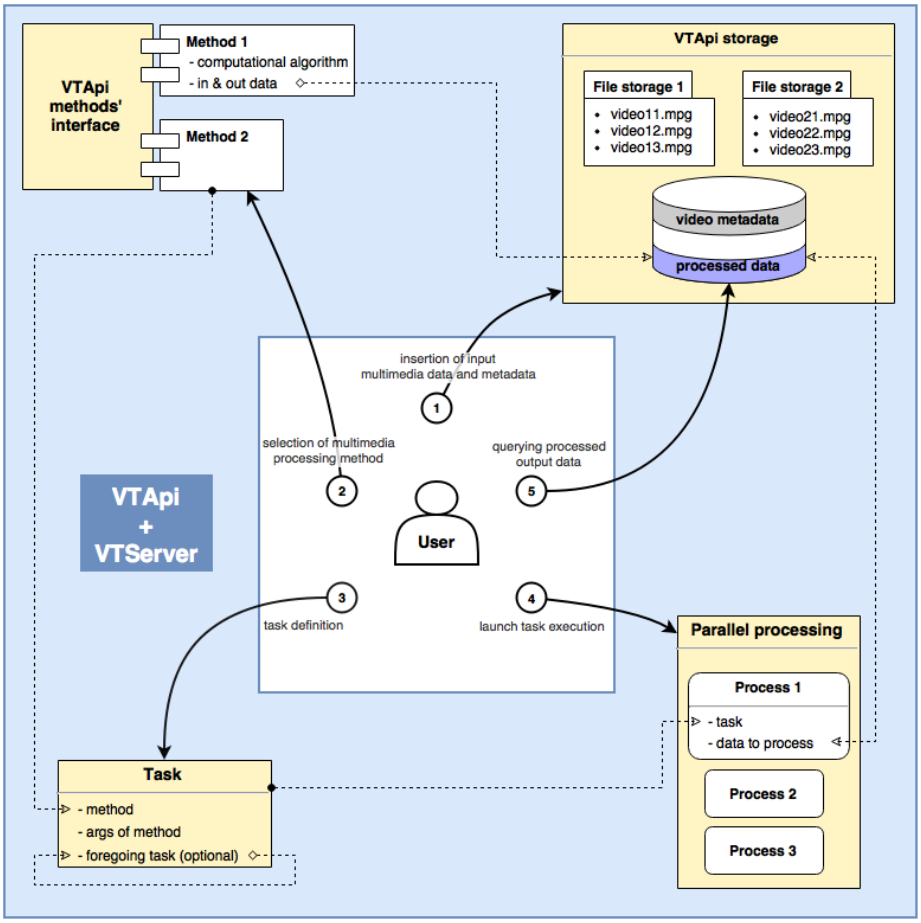

Figure 3: Use case from the viewpoint of the user

2.3 Application deployment

In this section we describe typical usage of VTApi and VTServer from the viewpoint of the user as is shown in figure 3.

In a first step, the user inserts selected multimedia (video or images) data with their metadata to the VTApi storage. In a second step, user chooses one of the available methods (modules) and subsequently defines a task (step 3), which comprises chosen method, desired input arguments of method and optionally also foregoing task. Next in a fourth step, the processes which ensure task execution are launched. Each process has assigned task and certain data to process - so parallel processing can be achieved. Last, user can query output (step 5) data processed by using chosen method (in step 2).

2.4 VTServer service

VTServer is designed as a RPC server utilizing VTAPI abilities and adding further controlling mechanisms and functionality. Its interface is defined using Google Protocol Buffers and messages are passed through ZMQ[1]. Implementation is based on RPCZ library combining previously mentioned technologies, having C++ and Python API. Overview of available methods is grouped into subsets of the API servicing different stages of the event-detection process.

Brief overview of VTServer funcionality follows. Detailed technical specification for all supported RPC methods may be found on project results server[2]

- **Dataset API** allows user to create or manage datasets for different projects or simply subdivide video data files according to analytical needs. Datasets encapsulate all project-related information - video metadata, defined processing tasks, processing units, computed or otherwise processed data and various metrics to query the event detection process and results.
- **Videos API** allows a user to load or delete video data to previously created dataset.
- **Processing tasks API** provides methods for definition and querying the processing tasks. Definition consists of selecting pre-installed computation method and specifying values of its input parameters. Prerequisite processing task may also be specified to utilize its previously computed outputs as inputs for the new task, thus allowing process-chaining (Video-processing - Event-detection). During processing task creation, VTServer automatically allocates necessary system resources for output data and checks if all required inputs are available, notifying user of possible errors. Created tasks may be queried for their progress or successfully completed computing operations on certain videos.
- **Processes API** allows launching and control of system process instances performing assigned processing tasks. Each process instance is assigned previously defined processing task and a set of video data for which it is responsible to complete the processing. Process parallelization is supported, multiple process instances responsible for same computation will correctly prevent themselves from performing redundant work.
- **Events API** methods handle retrieving computed events - outputs of Event-detection tasks. For faster analysis of events relevancy, it provides built-in statistical and filtering functionality.

2.5 VTApi framework

VTApi, Video Terror Application programming interface, is an open source API for computer-vision applications, which need to store data and metadata

[1] distributed messaging; http://zeromq.org/
[2] http://vidte.fit.vutbr.cz/vtapi.html

acquired and computed from a video, that was analysed by any computer-vision algorithm and also to retrieve data and metadata for other purposes (for example further processing, reporting and so on).

VTApi consists of a database and media file operations API and a framework for custom modules development. It was designed to unify and accelerate a development of computer-vision applications.

VTApi also allows to centralize data, metadata and computer-vision algorithms into a single server which can provide these data, metada and algorithms for a wide range of users. They can use them for their own analysis or development new alghorithms building on any alghoritm using its outputs.

VTApi is written in C++. VTApi uses OpenCV as the primary computer-vision framework and PostreSQL database as primary data and metadata storage (except multimedia files, which are stored in filesystem). Five versions of VTApi was released during its development (between years 2010 and 2015). At this moment 4 of them are obsolete and we bring VTApi 3.0. Unless otherwise noted descriptions in this technical report are related to latest VTApi 3.0.

2.5.1 VTApi terminology

- **Dataset** is a named set of multimedia data along with their metadata (descriptive data). Each dataset contains sequences.
- **Sequence** is a named ordered set of frames (time-based ordering) referred to as **Video** or a named ordered set of images referred to as **ImageFolder** *(or **images**)*.
- **Method** defines a structure of metadata consumed and produced by the custom computer-vision algorithm.
- **Task** is selected method together with preset input parameters (encapsulation of method with specific input parameters together) and optionally also with outputs of preceding task (this approach allows task chaining in a relatively simple way).
- **Process** is executive unit of task, which enables parallelization. Process performs the task over the assigned data.
- **Interval** is any subsequence of Video or ImageFolder (images) whose elements share the same metadata. For example, it can be a video shot or any sequence of frames containing the monitored object in the video or scene. Metadata of an interval are created by a process.
- **Selection** is a subset of logically related metadata, appropriately chosen, so that operations (processes) are effective and allow the natural chaining of tasks (some of outputs of one task may be inputs for other task). Common examples of selection is Interval. This concept is related to the effective implementation and access to the metadata in the database.

2.5.2 KeyValue mechanism and general functions

Key-Value mechanism is the basic way of metada organization in VTApi. It is a generic data structure (associative array) that allows to store data as `<key, value>` pairs, so changes in data definition do not imply changes of the VTApi code.

The simplified class diagram of VTApi is illustrated in figure 4. Most classes inherit from the class KeyValues, that provides the basic operations needed to manage key-value pairs, on which the VTApi model is based.

The KeyValues class is crucial to ensure the functionality and generality of the framework by the main function `next()`, which includes not only navigation over data structures, but also executes database queries, commits changes made by setters and also commits new data added by setters.

VTApi is strongly typed, so KeyValues class also include general getters and setters for each supported type. In the following description is used notation of `X` referring to any supported data type (all supported data types you can see in section 2.5.6, vector of data types are in the form `XVector`), `k` reffering to key and `v` reffering to its value of type `X`. Getters are in the form `getX(k)`, whereas setters are in the form `updateX(k, v)`.

2.5.3 Specialized functions of specific objects

Classes derived from KeyValues contain only functionality related to the consistency of data and to make some operations easier for VTApi users and factory methods. For instance, function `getLocation()` returns the physical data location (e.g., a dataset or a directory with pictures). The method `loadSequences()` of the Dataset class object is an example of a factory method (these methods are marked as `loadO()` in figure 4, where `O` is particular object). It creates anew object of the class Sequence with all necessary parameters. So, then it is possible to access all the current dataset's sequences identified by `getName()` by calling the `next()` method.

In addition to general functions, some VTApi objects contain also specialized functions, whose presence are necessary from the nature of specific objects. Main representatives of these specific objects are objects Image and Video.

The differences can be found in process of acquiring the image or the video: function `getImageData()` from class Image returns opencv matrix (`cv::Mat`) representing appropriate image, whereas function `openVideo()` from class Video returns opencv capture (`cv::VideoCapture`) representing appropriate video. While image is static, video is a variable set of images, where we need to find out for example FPS rate *(rate of frames per second)* or a speed of video. For these purposes there are another specialized functions, concretely a function `getFPS()` and a function `getSpeed()`.

2.5.4 Specialized database functions

VTApi contains also a set of specialized database functions, that ensure consistency of more difficult operations – since the database functions are performed

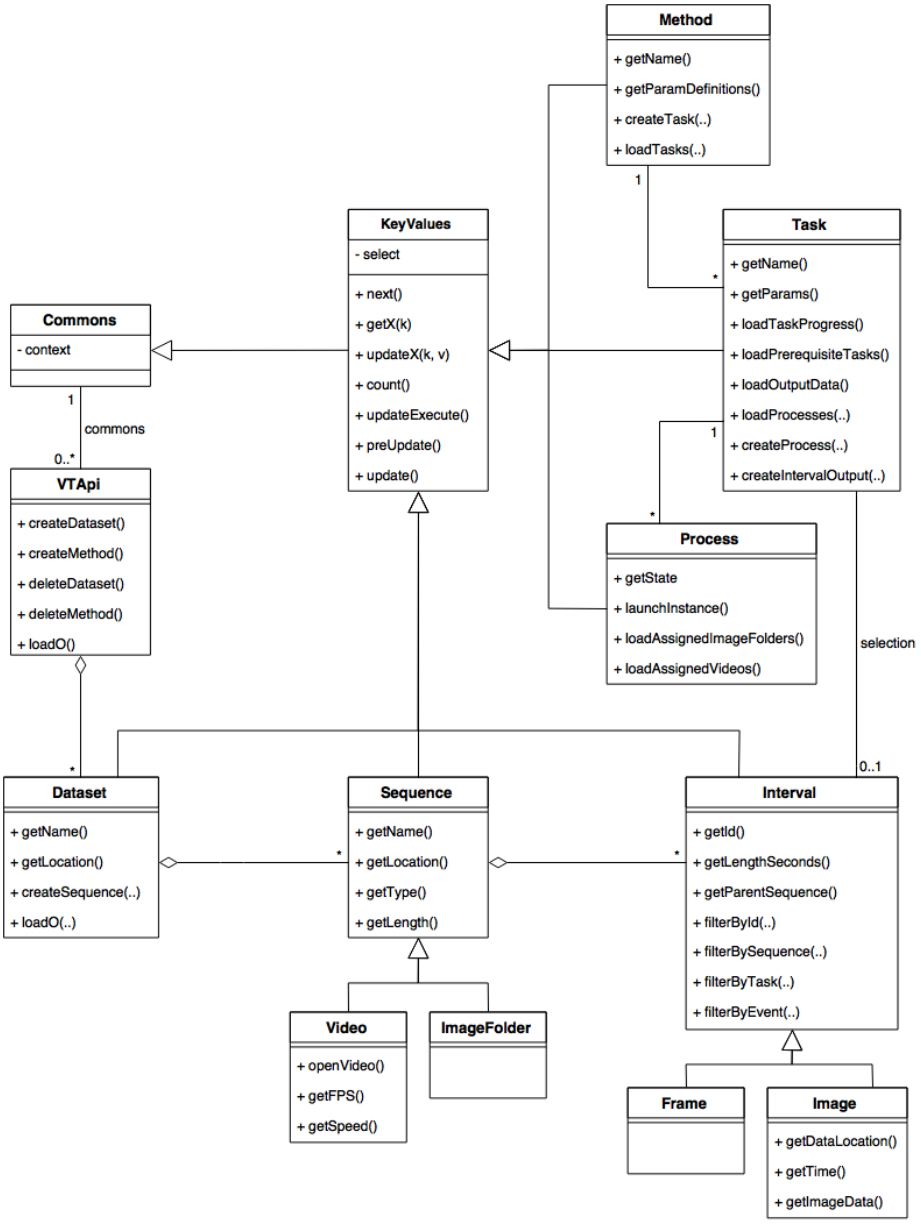

Figure 4: Simplified VTApi class diagram

in a single database transaction. There are also some specialized functions that complement missing functionality in PostgreSQL.

These functions can be divided into five groups:

1. **Support functions for datasets**
   These functions encapsulate creation (function `VT_dataset_create()`) or deletion (function `VT_dataset_drop()`) of datasets, where is needed to add (or delete) the row from datasets table in public schema and also create (or drop) appropriate schema designed for data of dataset. Dataset may be also truncated – this operation can be done by using function `VT_dataset_truncate()`. The last function offered in this group is `VT_dataset_drop_all()`, which is useful for deletion of all datasets and their data.

2. **Support functions for methods**
   Functions from this group encapsulate registration of some method in database by using function `VT_method_add()` or its unregistration by function `VT_method_delete()`. Function for registration of method is crucial for developers of methods.

3. **Support functions for tasks**
   These functions encapsulate creation (function `VT_task_create()`) or deletion (function `VT_task_delete()`) of methods‘ tasks. Function `VT_task_create()` is important for dynamic creation of output data infrastructure. It creates table for storing task‘ output data and creates indexes for more efficient output data searching; all this functionality is based on method data given by registration proces of method.

4. **Support functions for events filtering**
   `VT_task_out_filter_event()` is a function, which simplifies and encapsulates filtering of events. There are four available filters, which can be combined: duration filter, realtime filter, daytime filter and spatial filter.

   - **Duration filter** is given by minimal or/and maximal duration in seconds.
   - Similarly, the **realtime filter** is given by minimal or/and maximal real date and time.
   - **Daytime filter** is little bit special filter – it is given by minimal or/and maximal time within each day; using this filter may be filtered all events which occurred for example between 11 and 12 o’clock.
   - **Spatial filter** is given by region, in which the event occurred.

5. **Other support functions**
   VTApi enriches PostgreSQL database by 3 conversion functions. First one is function `tsrange()` that returns tsrange from given timestamp and number of seconds. Last two are functions `daytimenumrange()`,

that return numrange - one from two given times, next from given timestamp and number of seconds. As the name of last function implies, it is designed for daytime filter (numrange of seconds within the day).

2.5.5 Indexing

We can find two types of indexes in VTApi database schema. First of them are statically created indexes (fixed indexes), the other one are dynamically created indexes (custom definable indexes).

Fixed indexes are created above the tables, where is needed by VTApi for searching in specific columns or for joining tables on specific columns. For example, you can see index above columns `taskname` (index `processes_taskname_idx`) or `state.status` (index `processes_status_idx`) in table processes.

Second approach, which is used in VTApi, is dynamically created indexes based on used method. These indexes are created above task's output table (table generated based on method and its keys definition). There can be defined for each key of method, whether index is to be created above whole column or only its certain part (or more parts) in the appropriate task's output table. Whole column is indexed if `indexedkey` property of `methodkeytype` is set to `TRUE`. Certain part of column (which is of user-defined type) is indexed in case, that its position within user-defined type is defined in `indexedparts` array property of `methodkeytype`; similarly for more certain parts.

2.5.6 Supported data types

This chapter briefly clarify, which data types you can use for VTApi applications and its computer-vision algorithms. VTApi suports perphaps all common data types. Simple supported boolean, character and numeric data types are:

- boolean
- char
- string (character array)
- integer
- long integer (`int8`)
- float
- double

Furher, VTApi supports also an array data types – concretely a vector of most of the aforementioned simple data types:

- Vector of strings
- Vector of integer

- Vector of long integer (`int8`)
- Vector of float
- Vector of double

There are also supported some other special data types as:

- Timestamp (`std::chrono::system_clock::time_point`)
- OpenCV matrix (`cv::Mat`)
- PostgreSQL 2D spatial point (`Point`)
- Vector of these points
- `IntervalEvent` (special data type for video application event descriptor)
- `processState` (special data type for state of the process)

2.5.7 Is VTApi extensible?

Yes, VTApi can be extended in several ways. It can be extended to support more other (unsupported by official release) common, specific or even custom data types as well as to support other SQL database as a data and metadata storage.

2.5.7.1 Datatype extending

In addition to using already supported datatypes (see section 2.5.6) you can extend VTApi to support other common data types like other PostgreSQL spatial types (only 2D point is currently supported by VTApi). You can find other specific data types like PostGIS geometry types, which can also extend supported data types by VTApi. You can even create your own data types, which you can then also include to support by VTApi.

In case you would like to develop a support for other data types, let yourself be inspired by, for example, `getCvMat()` and `updateCvMat()` methods declaration in a file `./include/vtapi/data/keyvalues.h` and their implementations in a file `./src/backends/postgresql/pg_resultset.cpp`.

2.5.7.2 Database storage extending

You can also extend VTApi to support other database as a storage of data and metadata. VTApi supports variable database storage, however no support to any other database is fully implemented. There is only a stub for SQLite database, but no one for any other databases.

In case you would like to develop a support for other database, let yourself be inspired by this SQLite stub – see `./src/backends/sqlite/*.h` files for methods declaration and `./src/backends/sqlite/*.cpp` files for their implementations.

3 Computer vision application development

VTApi may be integrated into a specific analytical applications in various degrees. The most typical type of use is implementing the provided Method interface and compiling the implementation into a dynamic library for loading and running by a loader application (vtmodule). Example of such application is shown an described in section 2.2. using Method interface is the best option for application which don't conform to Event-based format of results.

Another interface which may be used for the development is an Event-based module interface. This is suitable for applications that use the Event-based format of results as it hides much of manipulation with VTApi library and thus reduces learning curve and saves time. The interface is designed to be as minimalistic as possible requiring only the video processing function alone to be implemented.

It is also possible to use VTApi as standalone library without using any of its external interfaces. If used this way VTApi primarily functions as a useful database API for video data management.

3.1 Usage of the VTApi Method interface

Modules examples for demonstration purposes may be found in src/modules folder in Git repository. All such modules developed using Method interface requires implementation of following functions:

```
    /**
     * @brief Module initialization
     * Called ALWAYS on plugin initialization
     * Throw vtapi::RuntimeModuleException on failure
     * @param vtapi main vtapi object to access VTApi
        ↪ for initialization purposes
     * @throws vtapi::RuntimeModuleException
        ↪ initialization error
     */
    virtual void initialize(VTApi & vtapi) = 0;

    /**
     * @brief Module uninitialization
     * Called ALWAYS on plugin uninitialization
     */
    virtual void uninitialize() noexcept = 0;

    /**
     * @brief Main processing function
     * Called after initialization ended without error
     * Throw vtapi::RuntimeModuleException on failure
```

```
     * Throw vtapi::ModuleUserAbortException on user
        ↪ abort
     * Proper processing may get a bit complicated,
        ↪ check demo modules
     * for example
     * @param process process object representing
        ↪ processing to be done
     * @throws vtapi::RuntimeModuleException
        ↪ processing error
     * @throws vtapi::ModuleUserAbortException
        ↪ processing error
     */
    virtual void process(Process & process) = 0;

    /**
     * @brief Call to this function should cause
        ↪ currently active processing
     * to throw a vtapi::ModuleUserAbortException
     * It is called during process() function from a
        ↪ different thread (!)
     * It should return ASAP and not wait for
        ↪ processing end
     */
    virtual void stop() noexcept = 0;
```

3.2 Usage of the Event-based module interface

Modules examples using the Event-based interfaces are located in vtapi_modules folder in Git repository. These must be compiled with provided sources of intermediate layer between Method API (which is already implemented) and Event-based modules interfaces. There are two such interfaces that represent two phases of a analytical process and must be implemented:

Video-Processing computes time/resource intensive processing tasks and stores its data in raw format (eg. feature vectors). Following interface must be implemented:

```
    /**
     * @brief Returns reference to singleton interface
        ↪  implementation
     * @return singleton instance
     */
    static IVideoProcessing &instance();

    /**
     * @brief Called ALWAYS before processing
     * @param params parameters of processing task
```

```
     * Throw vtapi::RuntimeModuleException on error
     * @throws vtapi::RuntimeModuleException on failed
        ↪ initialization
     */
    virtual void initialize(const ::vtapi::TaskParams
       ↪ & params) = 0;

    /**
     * @brief Called ALWAYS after processing or failed
        ↪ initialization
     */
    virtual void uninitialize() noexcept = 0;

    /**
     * @brief Main video processing function
     * Throw vtapi::RuntimeModuleException on failure
     * Throw vtapi::ModuleUserAbortException on user
        ↪ abort
     * @param video processed video object
     * @param output object for outputting intervals
     * @param progress object for updating video
        ↪ progress
     * @throws vtapi::RuntimeModuleException on failed
        ↪ processing
     * @throws vtapi::ModuleUserAbortException
        ↪ processing error
     */
    virtual void processVideo(const ::vtapi::Video &
       ↪ video,
        ::vtapi::IntervalOutput & output,
        ::Modules::IVideoProgressUpdater & progress) =
           ↪ 0;

    /**
     * @brief Call to this function should cause
        ↪ currently active processing
     * to throw a vtapi::ModuleUserAbortException
     * It is called during processVideo() function
        ↪ from a different thread (!)
     * It should return ASAP and not wait for
        ↪ processing end
     */
    virtual void stop() noexcept = 0;
```

Event-Detection uses the results of Video-Processing task and performs lightweight on-the-fly interpretation of these data according to parameters spec-

ified by user. It should save its output in Event format and implement following interface:

```
    /**
     * @brief Returns reference to singleton interface
        ↪  implementation
     * @return singleton instance
     */
    static IEventDetection &instance();

    /**
     * @brief Called ALWAYS before processing
     * @param params parameters of processing task
     * Throw vtapi::RuntimeModuleException on error
     * @throws vtapi::RuntimeModuleException on failed
        ↪  initialization
     */
    virtual void initialize(const ::vtapi::TaskParams
       ↪ & params) = 0;

    /**
     * @brief Called ALWAYS after processing or failed
        ↪  initialization
     */
    virtual void uninitialize() noexcept = 0;

    /**
     * @brief Main event detection function
     * Throw vtapi::RuntimeModuleException on failure
     * Throw vtapi::ModuleUserAbortException on user
        ↪ abort
     * @param video video object
     * @param input input intervals from video
        ↪ processing
     * @param output object for outputting intervals
     * @param progress object for updating video
        ↪ progress
     * @throws vtapi::RuntimeModuleException on failed
        ↪  processing
     * @throws vtapi::ModuleUserAbortException
        ↪ processing error
     */
    virtual void processVideo(const vtapi::Video &
       ↪ video,
        ::vtapi::Interval & input,
        ::vtapi::IntervalOutput & output,
```

```
        ::Modules::IVideoProgressUpdater & progress) =
            ↪  0;

    /**
     * @brief Call to this function should cause
        ↪ currently active processing
     * to throw a vtapi::ModuleUserAbortException
     * It is called during processVideo() function
        ↪ from a different thread (!)
     * It should return ASAP and not wait for
        ↪ processing end
     */
    virtual void stop() noexcept = 0;
```

4 Service deployment

4.1 VTApi deployment

4.1.1 Dependencies

VTApi uses some third party libraries and executables. There is a list of them with their minimum required versions, in parenthesis is mentioned usual package name:

- Cmake 2.8.9 (`cmake`)
- pkg-config (`pkg-config`)
- POCO 1.61[3] (`libpoco-dev`)
- OpenCV 2.4[4] (`libopencv-dev`)
- PostgreSQL 9.3[5] (`libpq-dev`)
- libpqtypes 1.5[6] (`libpqtypes-dev`)
- SQLite 3.8[7] (`libsqlite3-dev`)

4.1.2 VTApi specific and useful common CMake preferences

VTApi uses CMake for building. Following table desctibes useful common CMake preferences:

preference with type of value	description	default value
`CMAKE_INSTALL_PREFIX=<path>`	A path, where will be installed VTApi.	`/usr/local`
`CMAKE_BUILD_TYPE=<type>`	Through this preference, you may set build type to `Debug`, if you need it.	`Release`

Table 1: Common CMake preferences

In addition to common CMake preferences, VTApi extends CMake with its own preferences related to used dependencies.

[3] http://pocoproject.org/
[4] http://opencv.org/
[5] http://www.postgresql.org/
[6] http://libpqtypes.esilo.com/
[7] https://www.sqlite.org/

Dependencies are searched in standard system path(s) by default. In case, they are installed in non-standard path, you will need to specify some of VTApi specific CMake preferences. Similarly, `pkg-config` looks in the directory `<prefix>/lib/pkgconfig/` for `*.pc` files. In case, that considered `*.pc` file is not located in aforementioned directory, base directory of this file can be added to the `PKG_CONFIG_PATH` environment variable or it can be used again some of VTApi specific CMake preferences.

These VTApi specific CMake preferences are described in the following table, all of them are of type path *(<preference>=<path>)*.

preference	description
`OPENCV_PC_PATH`	A path, where is located `opencv.pc` file for `pkg-config`.
`PG_CONFIG_PATH`	A path, where is located `pg_config`.
`LIBPQTYPES_PATH`	A path, where is located `libpqtypes` library *(this path must contain `include/` and `lib/` subdirectories, where are located header files and libraries).*
`POCO_PATH`	A path, where is located POCO *(this path must contain `include/` and `lib/` subdirectories, where are located header files and libraries).*
`POCO_INCLUDE_PATH`	A path, where are located POCO header files *(it is useful in case, that header and library directories are located separately in different paths).*
`POCO_LIBRARY_PATH`	A path, where are located POCO library files *(it is useful in case, that header and library directories are located separately in different paths).*
`POCOFOUNDATION_INCLUDE_PATH`	A path, where are located POCO Foundation header files *(it is useful in case, that different parts of POCO project are on different locations).*
`POCOJSON_INCLUDE_PATH`	A path, where are located POCO JSON header files *(it is useful in case, that different parts of POCO project are on different locations).*

POCOUTIL_INCLUDE_PATH	A path, where are located POCO Util header files *(it is useful in case, that different parts of POCO project are on different locations).*
POCOXML_INCLUDE_PATH	A path, where are located POCO XML header files *(it is useful in case, that different parts of POCO project are on different locations).*
POCOFOUNDATION_LIBRARY_PATH	A path, where are located POCO Foundation library files *(it is useful in case, that different parts of POCO project are on different locations).*
POCOJSON_LIBRARY_PATH	A path, where are located POCO JSON library files *(it is useful in case, that different parts of POCO project are on different locations).*
POCOUTIL_LIBRARY_PATH	A path, where are located POCO Util library files *(it is useful in case, that different parts of POCO project are on different locations).*
POCOXML_LIBRARY_PATH	A path, where are located POCO XML library files *(it is useful in case, that different parts of POCO project are on different locations).*
SQLITE3_PC_PATH	A path, where is located `sqlite3.pc` file for `pkg-config`.

Table 2: VTApi specific CMake preferences

4.1.3 Building VTApi

For build of VTApi you can use following steps:

```
cd <VTApi directory>
mkdir build
cmake .. [-D<CMake preference>] [-D<CMake preference>] [...]
make
sudo make install
```

Or you can also use prepared script `build_release.sh` for easy build of VTApi.

4.1.4 Configuration file

For easy configuration of VTApi you can use script `./vtapi-postinstall.sh`, which serves as a post-installation guide of VTApi.

4.2 VTServer deployment

4.2.1 Dependencies

VTServer uses some third party libraries and executables. There is a list of them with their minimum required versions, in parenthesis is mentioned usual package name:

- VTApi 3.0 *(package not available - must be installed from the source)*
- Boost 1.54[8] (`libboost-thread-dev libboost-program-options-dev`)
- Protocol Buffers 2.5[9] (`libprotobuf-dev libprotoc-dev`)
- ZeroMQ 4.0.4[10] (`libzmq3-dev`)
- libpqtypes 1.5[11] (`libpqtypes-dev`)
- rpcz[12] *(package not available - must be installed from the source - see footnote)*

4.2.2 Install VTServer

Installation of VTServer is performed by the following steps:

1. compilation of VTApi, VTApi modules and VTServer using scripts:
   - `vtapi/build_release.sh`
   - `vtserver/build_release.sh`
   - `vtapi_modules/build_release.sh`
2. environment settings
   - add `usr/local/bin` to `PATH`
   - add `usr/local/lib` to dynamic library search path

[8] http://www.boost.org/
[9] https://developers.google.com/protocol-buffers/
[10] http://zeromq.org/
[11] http://libpqtypes.esilo.com/
[12] https://github.com/thesamet/rpcz

4.2.3 Run VTServer

To run VTServer follow the subsequent steps:

1. creation of database using these scripts:
   - `vtapi/sql/pg_createdb.sql`
   - `vtapi_modules/sql/pg_modules.sql`
2. configuration settings
   - create `vtapi.conf` in the folder from which you're running `vtserver` *(use `vtapi/vtapi_example.conf` as template)*
   - set `datasets_dir` to some empty folder *(do not accidentaly delete your stuff)*
   - set `modules_dir` to `/usr/local/lib`
   - unset `default_dataset`
   - set `connection` to database
   - make sure `logfile` is set
   - log at least errors and warnings
3. run
   - `vtserver [--config=/some/other/vtapi.conf]`

5 Event-based Video Analysis Tool (EVIDANT)

The EVIDANT is an application based on VTServer and VTAPI solution and designed as client-server application using web technologies. The client implements GUI to manage user projects and video datasets (see Fig. 5), configure and control the functional blocks' execution, analyze detected video events and prepare graphical or textual reports. The client is thin front-end of the web server application that implements data model for user-specific data, database interface and application-dependent services. The web-server video-processing services are connected to VTServer and provide the EVIDANT processing functionality.

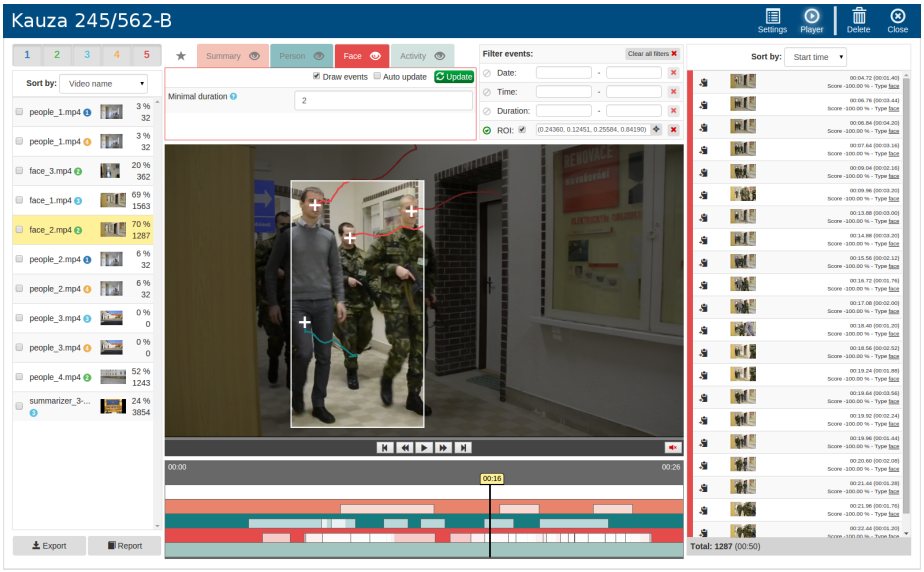

Figure 5: EVIDANT user interface

EVIDANT currently incorporates applications for these Event-based modules:

- Activity - activity detector based on temporal changes,
- Summary - compact video-parts with changing image content and constrained by required maximal length of final video,
- Face - face detection and tracking,
- People - people detection and tracking,
- Matcher - block comparing the image content of query video with recordings in video dataset, and

- VideoType - classification of the scenes by visual content.

6 Conclusion

The VTServer is a RPC server that can be employed as a service in computer vision applications. Currently it directly supports event-based computer vision applications development because its general controlling mechanisms and functionality is extended to process events. But its functionality can easily be extended or modified according to specific analytical needs.

References

[1] Beran, V. et al: VideoTerror demonstrator. Event-based Video Analytic Tool (EVIDANT). Technical Report FIT - VG20102015006 - 2015 – 02. FIT BUT in Brno. 2015. 38 p.

[2] Chmelař, P. a další: VTApi v. 2.0. Technická zpráva FIT - VG20102015006 - 2013 – 01. FIT VUT v Brně. 2013. 46 s. (in Czech)