TraX Documentation

Release 4.0.0

Luka Cehovin

May 04, 2023

Contents

Welcome to TraX reference library documentation. TraX stands for visual Tracking eXchange, the protocol was designed to make development and testing of visual tracking algorithms simpler and faster. This documentation describes the protocol specification but also documents the reference library that implements the protocol and can be used to add support to visual tracking programs quickly and without the need to understand the low-level details.

The source code for the reference library is availabe on GitHub under Lesser GPL v3.0 license.

Citing

If you are using TraX protocol in your research, please cite the following publication that describes the protocol and the library.

```
@article{cehovin2017trax,
        author = {{\v{C}}ehovin, Luka},
        doi = {http://dx.doi.org/10.1016/j.neucom.2017.02.036},
        issn = {0925-2312},
        journal = {Neurocomputing},
        title = {{TraX: The visual Tracking eXchange Protocol and Library}},
        year = {2017}
}
```

Index

2.1 TraX protocol specification

The TraX protocol is designed with simplicity of integration in mind, but is also flexible enough to allow extensions and custom use-cases. The protocol is primarily based on the a mechanism that all modern operating systems provide and requires no additional dependencies - standard input and output streams of a process. However, other media, such as TCP streams can also be used, the main idea is that the protocol communication is embedded the communication between the tracker process and the control process in these streams. The communication is divided into line-based messages. Each message can be identified by a prefix that allows us to filter out tracker custom output from the protocol communication.

2.1.1 Definitions

We adopt the standard client-server terminology when describing the interaction, although the definition is a bit counterintuitive in some aspects. We define the basic terms of the protocol as:

- **Server**: A server is a tracker process that is providing tracking information to the client that is supplying the server with requests – a sequence of images. Unlike traditional servers that are persistent processes that communicate with multiple clients, the server in our case is started by a single client and is only communicating with it.

- **Client**: A client is a process that is initiating tracking requests as well as controlling the process. In most cases this would be an evaluation software that would aggregate tracking data for performance analysis, however, additional use-cases can be accommodated in this scheme.

- **Message**: Server and client communicate with each other using messages. Each message begins in new line, is prefixed by an unique string and ends with the end of the line. Generic message structure is defined in *Message format* and types of messages are defined in *Protocol messages and states*.

2.1.2 Message format

Individual message in the protocol is a line, which means that it is separated from the past and future stream content by the new line (EOL) character. The format of all client or server messages is the same. To distinguish between arbitrary program outputs and embedded TraX messages a prefix `@@TRAX:` is used. The prefix is followed immediately (without white space character) by the name of the message, which is then followed by space-separated arguments. The format is illustrated in figure below. The message header is followed by a number of mandatory message arguments. This number depends on the type of the message and on the runtime configuration. The mandatory arguments are then followed by a variable number of optional named arguments which consist of a key and a value part and can be used to communicate additional data.
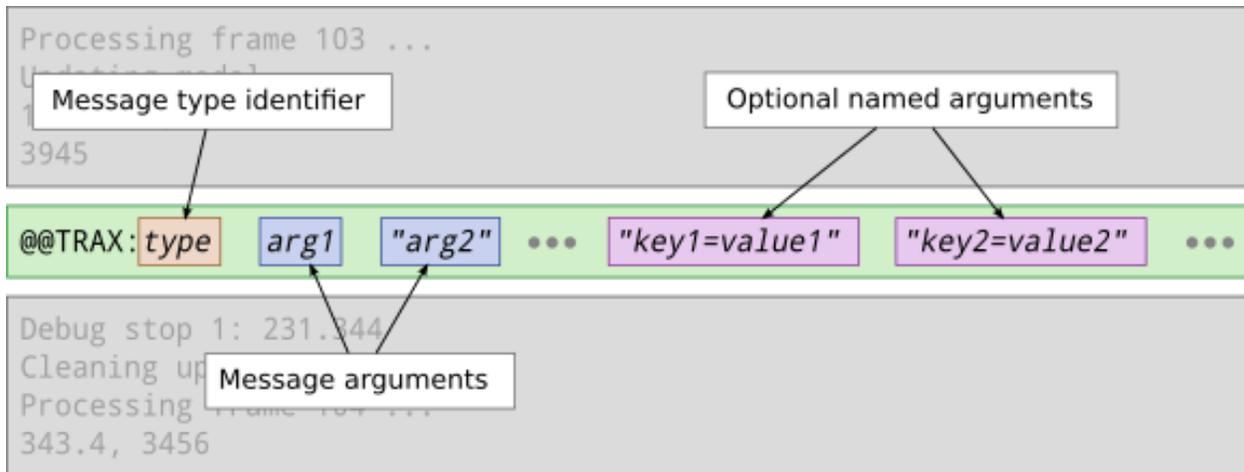
Escape sequences

All the arguments can contain spaces, however, they have to be enclosed by double-quote (`"`) symbols. If you want to use to use the same symbol inside the argument, it has to be prefixed by back-slash (`\`) symbol, i.e. you have to use an escape sequence. Similarly escape sequence is also used to denote a bach-slash character itself (written as `\\`) and a newline symbol, which has to be replaced by the `\n` character sequence.

Named arguments

Named arguments consist of a key, followed by character (=) and an arbitrary value character sequence. The key sequence may only include alphanumerical characters, dot (`.`) and underscore (_) and has to be at most 64 characters long. If a key is not valid the remote party may reject the argument or terminate the connection because of an illegal message format.

2.1.3 Protocol messages and states

Below we list the valid messages of the protocol as well as the states of the client and server. Despite the apparent simplicity of the protocol its execution should be strict. An inappropriate or indecipherable message should result in immediate termination of connection in case of both parties.

- `hello` (server): The message is sent by the server to introduce itself and list its capabilities. This message specifies no mandatory arguments, however, the server can report the capabilities using the optional named arguments. The official arguments, recognized by the first version of the protocol are:

- `trax.version` (integer): Specifies the supported version of the protocol. If not present, version 1 is assumed.

- `trax.name` (string): Specifies the name of the tracker. The name can be used by the client to verify that the correct algorithm is executed.

- `trax.identifier` (string): Specifies the identifier of the current implementation. The identifier can be used to determine the version of the tracker.

- `trax.image` (string): Specifies the supported image format. See Section *Image formats* for the list of supported formats. By default it is assumed that the tracker can accept file paths as image source.

- `trax.region` (string): Specifies the supported region format. See Section *Region formats* for the list of supported formats. By default it is assumed that the tracker can accept rectangles as region specification.

- `trax.channels` (string, version 2+): Specifies support for multi-modal images. See Section *Image channels* for more information.

- `initialize` (client): This message is sent by the client to initialize the tracker. The message contains the image data (for one or more images) and the region of the object. The actual format of the required arguments is determined by the image and region formats specified by the server.

- `frame` (client): This message is sent by the client to request processing of a new image (or multiple images). The message contains the image data. The actual format of the required argument is determined by the image format specified by the server.

- `state` (server): This message is used by the server to send the new region to the client. The message contains region data in arbitrary supported format (most commonly the same format that the server proposed in the introduction message).

- `quit` (client, server): This message can be sent by both parties to terminate the session. The server process should exit after the message is sent or received. This message specifies no mandatory arguments.

The state diagram of server and client is defined by a simple automata, shown in figure below. The state changes upon receiving appropriate messages from the opposite party. The client state automata consists of the following states:

- **Introduction**: The client waits for `hello` message from the server. In this message the server describes its capabilities that the client can accept and continue the conversation by moving to *initialization* state, or reject it and terminate the session by sending the `quit` message.

- **Initialization**: The client sends a `initialize` message with the image and the object region data. Then the client moves to *observing* state.

- **Observing**: The client waits for a message from the server. If the received message is `state` then the client processes the incoming state data and either moves to *initialization*, *termination* or stays in *observing* state. If the received message is `quit` then the client moves to *termination* state.

- **Termination**: If initiated internally, the client sends the `quit` message. If the server does not terminate in a certain amount of time, the client can terminate the server process.

The server state automata consists of the following states:

- **Introduction**: The server sends an introductory `hello` message where it optionally specifies its capabilities.

- **Initialization**: The server waits for the `initialize` or `quit` message. In case of `initialize` message a tracker is initialized with the given data and the server moves to {em reporting} state. The new state is reported back to the client with a `state` message. In case of the `quit` message the server moves to *termination* state.

- **Reporting**: The server waits for the `frame`, `initialize`, or `quit` message. In case of {tt frame} message the tracker is updated with the new image information and the new state is reported back to the client with a

state message. In case of `initialize` message a tracker is initialized with the given data and the new state is reported back to the client with a `state` message. In case of the `quit` message the server moves to *termination* state.

- **Termination**: If initiated internally, the server sends the `quit` message and then exits.
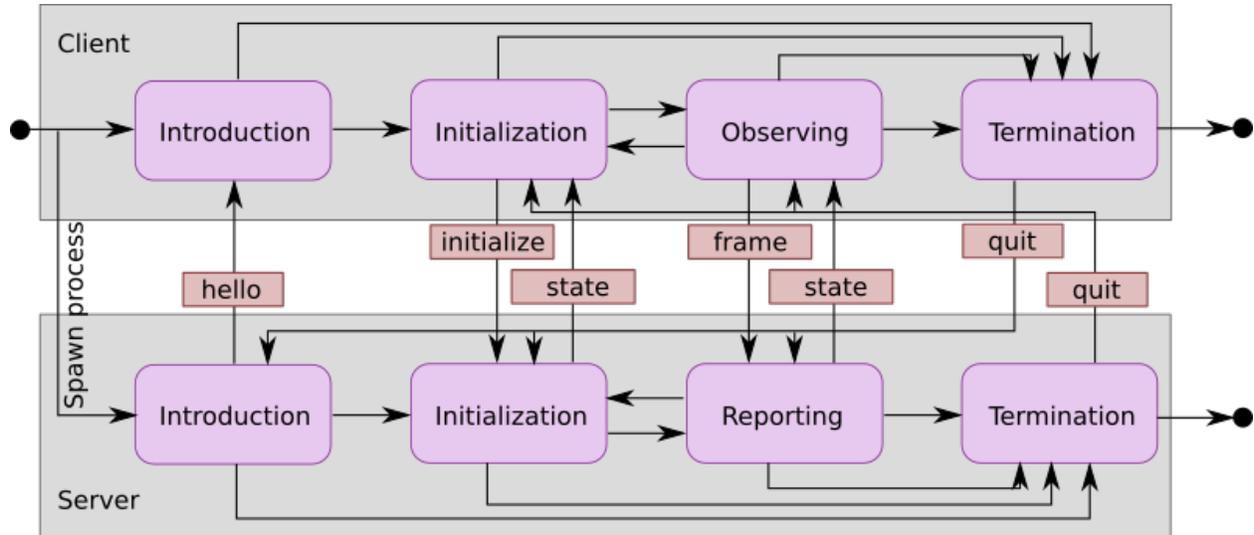

2.1.4 Region formats

The region can be encoded in two point-based formats. All two formats are comma-separated and illustrated graphically in figure below.

- **Rectangle** (`rectangle`): The simplest form of region format is the axis-aligned bounding box. It is described using four values, `left`, `top`, `width`, and `height` that are separated by commas.

- **Polygon** (`polygon`): A more complex and flexible region description that is specified by even number of at least six values, separated by commas that define points in the polygon (`x` and `y` coordinates).

2.1.5 Image formats

The image can be encoded in a form of Uniform Resource Identifiers. Currently the protocol specifies support for four types of resources.

- **File path** (`path`): Image is specified by an URL to an absolute path on a local file-system that points to a JPEG or PNG file. The server should take care of the loading of the image to the memory in this case. Some examples of image paths are `file:///home/user/sequence/00001.jpg` for Unix systems or `file://c:/user/sequence/00001.jpg`.

- **Memory** (`memory`): Raw image data encoded in an URI with scheme identifier {tt image:}. The encoding header contains information about width, height, and the pixel format. The protocol specifies support for the following formats: single channel 8 or 16 bit intensity image (`gray8` and `gray16`) and 3 channel 8-bit RGB image (`rgb`). Note that the intensity format can also be used to encode infra-red or depth information. The header is followed by the raw image data row after row using Base64 encoding. An example first part of the data for a 320 x 240 RGB image is therefore `image:320;240;rgb;...`.

- **Data** (`data`): The image is encoded as a data URI using JPEG or PNG format and encoded using Base64 encoding. The server has to support decoding the image from the memory buffer directly. An example of the first part of such data is `data:image/jpeg;base64;...`.
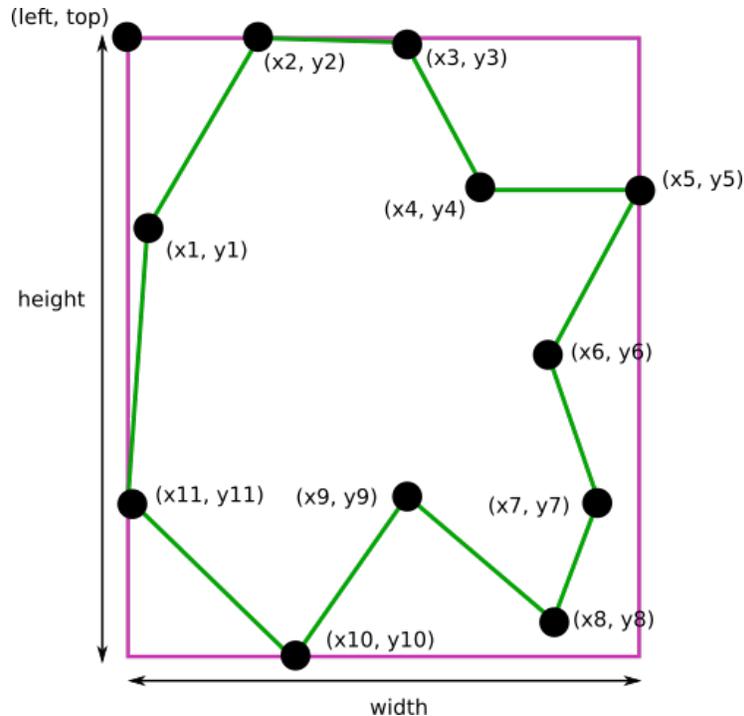
- **URL** (`url`): Image is specified by a general URL for the image resource which does not fall into any of the above categories. Tipically HTTP remote resources, such as `http://example.com/sequence/0001.jpg`.

2.1.6 Image channels

Version 2 of the protocol also specifies support for multi-modal images, encoded in multiple image planes. To use this feature, the server must sent the `trax.channels` argument in the introduction message. The content of this argument is a comma-separated list of channels that are required. Currently supported channels are:

- **Visible light** (`color`): Image is in visible light spectrum, by default only this channel is available.
- **Depth** (`depth`): Channel contains depth information, useful for RGBD data.
- **Infra Red** (`ir`): Infra red information, useful for IR sequences or for RGB + IR seqences.

Upon receiving the intorduction, the client evaluates if it can provide the data in requested format. If it proceeds, each channel is sent to the server as one or more arguments, encoded as specified in Section *Image formats*. The order of the images is always the same as the order of the identifier list elements.

2.2 API documentation

The TraX API documentation describes technical details of TraX library and bindings to other languages.

2.2.1 Reference C library

The TraX protocol can be implemented using the protocol specification, the protocol is quite easy to implement in many high-level languages. However, a reference C implementation is provided to serve as a practical model of implementation and to make the adoption of the protocol easier.

Requirements and building

The library is built using CMake build tool which generates build instructions for a given platform. The code is written in C89 for maximum portability and the library has no external dependencies apart from the standard C library. For more details on building check out the *tutorial on compiling the project*.

Documentation

All the public functionality of the library is described in the `trax.h` header file, below is a summary of individual functions that are available in the library.

Communication

TRAX_ERROR
> Value that indicates protocol error

TRAX_OK
> Value that indicates success of a function call

TRAX_HELLO
> Value that indicates introduction message

TRAX_INITIALIZE
> Value that indicates initialization message

TRAX_FRAME
> Value that indicates frame message

TRAX_QUIT
> Value that indicates quit message

TRAX_STATE
> Value that indicates status message

trax_logging
> Structure that describes logging handle.

trax_bounds
> Structure that describes region bounds.

trax_handle
> Structure that describes a protocol state for either client or server.

trax_image
> Structure that describes an image.

trax_region
> Structure that describes a region.

trax_properties
> Structure that contains an key-value dictionary.

trax_logging **trax_no_log**
> A constant to indicate that no logging will be done.

trax_bounds **trax_no_bounds**
> A constant to indicate that here are no bounds.

const char* **trax_version** ()
> Returns a string version of the library for debugging purposes. If possible, this version is defined during compilation time and corresponds to Git hash for the current revision.
>
> > **Returns** Version string as a constant character array

trax_metadata* **trax_metadata_create** (int *region_formats*, int *image_formats*, int *channels*, const char* *tracker_name*, const char* *tracker_description*, const char* *tracker_family*)
> **Parameters**
>
> > - **region_formats** – Supported regions formats bit-set.
> > - **image_formats** – Supported image formats bit-set.
> > - **channels** – Required image channels bit-set. If empty then only *TRAX_CHANNEL_VISIBLE* is assumed.
>
> Create a tracker metadata structure returning its pointer
>
> > **Returns** A pointer to a metadata structure that can be released using *trax_metadata_release()*.

void **trax_metadata_release** (trax_metadata** *metadata*)
> Releases a given metadata structure, clearing its memory.
>
> > **Parameters**
> >
> > > - **metadata** – Pointer of a pointer of tracker metadata structure.

trax_logging **trax_logger_setup** (trax_logger *callback*, void* *data*, int *flags*)
> A function that can be used to initialize a logging configuration structure.
>
> > **Parameters**
> >
> > > - **callback** – Callback function used to process a chunk of log data
> > > - **data** – Additional data passed to the callback function as an argument
> > > - **flags** – Optional flags for logger
> >
> > **Returns** A logging structure for the given data

trax_logging **trax_logger_setup_file** (FILE* *file*)
> A handy function to initialize a logging configuration structure for file logging. Internally the function calls *trax_logger_setup()*.
>
> > **Parameters**
> >
> > > - **file** – File object, opened for writing, can also be stdout or stderr
> >
> > **Returns** A logging structure for the given file

*trax_handle** **trax_client_setup_file** (int *input*, int *output*, *trax_logging* *log*)
> Setups the protocol state object for the client. It is assumed that the tracker process is already running (how this is done is not specified by the protocol). This function tries to parse tracker's introduction message and fails if it is unable to do so or if the handshake fails (e.g. unsupported format version).
>
> > **Parameters**
> >
> > > - **input** – Stream identifier, opened for reading, used to read server output
> > > - **output** – Stream identifier, opened for writing, used to write messages
> > > - **log** – Logging structure
> >
> > **Returns** A handle object used for further communication or NULL if initialization was unsuccessful

---

*trax_handle** **trax_client_setup_socket** (int *server*, int *timeout*, *trax_logging log*)

> Setups the protocol state object for the client using a bi-directional socket. It is assumed that the connection was already established (how this is done is not specified by the protocol). This function tries to parse tracker's introduction message and fails if it is unable to do so or if the handshake fails (e.g. unsupported format version).

> **Parameters**
>
> > - **server** – Socket identifier, used to read communcate with tracker
> >
> > - **log** – Logging structure
>
> **Returns** A handle object used for further communication or NULL if initialization was unsuccessful

int **trax_client_wait** (*trax_handle** *client*, *trax_region*** *region*, *trax_properties** *properties*)

> Waits for a valid protocol message from the server.

> **Parameters**
>
> > - **client** – Client state object
> >
> > - **region** – Pointer to current region for an object, set if the response is TRAX_STATE, otherwise NULL
> >
> > - **properties** – Additional properties
>
> **Returns** Integer value indicating status, can be either TRAX_STATE, TRAX_QUIT, or TRAX_ERROR

int **trax_client_initialize** (*trax_handle** *client*, *trax_image** *image*, *trax_region** *region*, *trax_properties** *properties*)

> Sends an initialize message to server.

> **Parameters**
>
> > - **client** – Client state object
> >
> > - **image** – Image frame data
> >
> > - **region** – Initialization region
> >
> > - **properties** – Additional properties object
>
> **Returns** Integer value indicating status, can be either TRAX_OK or TRAX_ERROR

int **trax_client_frame** (*trax_handle** *client*, *trax_image** *image*, *trax_properties** *properties*)

> Sends a frame message to server.

> **Parameters**
>
> > - **client** – Client state object
> >
> > - **image** – Image frame data
> >
> > - **properties** – Additional properties
>
> **Returns** Integer value indicating status, can be either TRAX_OK or TRAX_ERROR

*trax_handle** **trax_server_setup** (trax_metadata* *metadata*, *trax_logging log*)

> Setups the protocol for the server side and returns a handle object.

> **Parameters**
>
> > - **metadata** – Tracker metadata
> >
> > - **log** – Logging structure

> **Returns** A handle object used for further communication or NULL if initialization was unsuccessful

trax_handle* **trax_server_setup_file**(trax_metadata* *metadata*, int *input*, int *output*, *trax_logging* *log*)

> Setups the protocol for the server side based on input and output stream and returns a handle object.

> > **Parameters**

> > - **metadata** – Tracker metadata
> > - **input** – Stream identifier, opened for reading, used to read client output
> > - **output** – Stream identifier, opened for writing, used to write messages
> > - **log** – Logging structure

> > **Returns** A handle object used for further communication or NULL if initialization was unsuccessful

int **trax_server_wait**(*trax_handle** *server*, *trax_image*** *image*, *trax_region*** *region*, *trax_properties** *properties*)

> Waits for a valid protocol message from the client.

> > **Parameters**

> > - **server** – Server state object
> > - **image** – Pointer to image frame data, set if the response is not *TRAX_QUIT* or *TRAX_ERROR*, otherwise NULL
> > - **region** – Pointer to initialization region, set if the response is *TRAX_INITIALIZE*, otherwise NULL
> > - **properties** – Additional properties

> > **Returns** Integer value indicating status, can be either *TRAX_INITIALIZE*, *TRAX_FRAME*, *TRAX_QUIT*, or *TRAX_ERROR*

int **trax_server_reply**(*trax_handle** *server*, *trax_region** *region*, *trax_properties** *properties*)

> Sends a status reply to the client.

> > **Parameters**

> > - **server** – Server state object
> > - **region** – Current region of an object
> > - **properties** – Additional properties

> > **Returns** Integer value indicating status, can be either *TRAX_OK* or *TRAX_ERROR*

int **trax_terminate**(*trax_handle** *handle*)

> Used in client and server. Closes communication, sends quit message if needed. This function is implicitly called in *trax_cleanup()*.

> > **Parameters**

> > - **handle** – Server or client state object

> > **Returns** Integer value indicating status, can be either *TRAX_OK* or *TRAX_ERROR*

int **trax_cleanup**(*trax_handle*** *handle*)

> Used in client and server. Closes communication, sends quit message if needed. Releases the handle structure.

> > **Parameters**

- **handle** – Pointer to state object pointer

> **Returns** Integer value indicating status, can be either *TRAX_OK* or *TRAX_ERROR*

const char* **trax_get_error**(*trax_handle** *handle*)
> Retrieve last error message encountered by the server or client.

> > **Parameters**

> > - **handle** – Pointer to state object

> **Returns** Returns NULL if no error occured.

int **trax_is_alive**(*trax_handle** *handle*)
> Check if the handle has been terminated.

> > **Parameters**

> > - **handle** – Pointer to state object

> **Returns** Returns zero if handle is not alive and non-zero if it is.

int **trax_set_parameter**(*trax_handle** *handle*, int *id*, int *value*)
> Sets the parameter of the client or server instance.

int **trax_get_parameter**(*trax_handle** *handle*, int *id*, int* *value*)
> Gets the parameter of the client or server instance.

ImageList

TRAX_CHANNEL_VISIBLE
> Visible light channel identifier.

TRAX_CHANNEL_DEPTH
> Depth channel identifier.

TRAX_CHANNEL_IR
> IR channel identifier.

TRAX_CHANNELS
> Number of available channels.

TRAX_CHANNEL_INDEX(I)
> Convert channel identifier into index.

TRAX_CHANNEL_ID(I)
> Convert channel index into identifier.

trax_image_list* **trax_image_list_create**()

> > Create an emptry image list

> > **Returns** Pointer to image list container

void **trax_image_list_release**(trax_image_list** *list*)

> > Release image list structure, does not release any channel images

> > **Parameters**

> > - **list** – Image list container pointer

void **trax_image_list_clear**(trax_image_list* *list*)

Cleans image list, releases all allocated channel images

> **Parameters**
>
> > • **list** – Image list container pointer

trax_image* **trax_image_list_get** (const trax_image_list* *list*, int *channel*)

Get image for a specific channel

> **Parameters**
>
> > • **list** – Image list structure pointer
> >
> > • **channel** – Channel idenfifier
>
> **Returns** Image structure pointer

void **trax_image_list_set** (trax_image_list* *list*, *trax_image** *image*, int *channel*)

Set image for a specific channel

> **Parameters**
>
> > • **list** – Image list structure pointer
> >
> > • **image** – Image structure pointer
> >
> > • **channel** – Channel idenfifier
>
> **Returns** Pointer to null-terminated character array

int **trax_image_list_count** (int *channels*)

Count available channels in provided bit-set

> **Parameters**
>
> > • **channels** – Bit-set of channel identifiers
>
> **Returns** Number of on bits.

Image

TRAX_IMAGE_EMPTY
> Empty image type, not usable in any way but to signify that there is no data.

TRAX_IMAGE_PATH
> Image data is provided in a file on a file system. Only a path is provided.

TRAX_IMAGE_URL
> Image data is provided in a local or remote resource. Only a URL is provided.

TRAX_IMAGE_MEMORY
> Image data is provided in a memory buffer and can be accessed directly.

TRAX_IMAGE_BUFFER
> Image data is provided in a memory buffer but has to be decoded first.

TRAX_IMAGE_BUFFER_ILLEGAL
> Image buffer is of an unknown data type.

TRAX_IMAGE_BUFFER_PNG
  Image data is encoded as PNG image.

TRAX_IMAGE_BUFFER_JPEG
  Image data is encoded as JPEG image.

TRAX_IMAGE_MEMORY_ILLEGAL
  Image data is available in an unknown format.

TRAX_IMAGE_MEMORY_GRAY8
  Image data is available in 8 bit per pixel format.

TRAX_IMAGE_MEMORY_GRAY16
  Image data is available in 16 bit per pixel format.

TRAX_IMAGE_MEMORY_RGB
  Image data is available in RGB format with three bytes per pixel.

void **trax_image_release** (*trax_image*** *image*)
  Releases image structure, frees allocated memory.

> **Parameters**
>
>> • **image** – Pointer to image structure pointer (the pointer is set to `NULL` if the structure is destroyed successfuly)

*trax_image** **trax_image_create_path** (const char* *path*)
  Creates a file-system path image description.

> **Parameters**
>
>> • **url** – File path string, it is copied internally
>
> **Returns** Image structure pointer

*trax_image** **trax_image_create_url** (const char* *url*)
  Creates a URL path image description.

> **Parameters**
>
>> • **url** – URL string, it is copied internally
>
> **Returns** Image structure pointer

*trax_image** **trax_image_create_memory** (int *width*, int *height*, int *format*)
  Creates a raw in-memory buffer image description. The memory is not initialized, you have do this manually.

> **Parameters**
>
>> • **width** – Image width
>>
>> • **height** – Image height
>>
>> • **format** – Image format, see format type constants for options
>
> **Returns** Image structure pointer

*trax_image** **trax_image_create_buffer** (int *length*, const char* *data*)
  Creates a file buffer image description.

> **Parameters**
>
>> • **length** – Length of the buffer
>>
>> • **data** – Character array with data, the buffer is copied
>
> **Returns** Image structure pointer

int **trax_image_get_type** (const *trax_image** *image*)

> Returns a type of the image handle.

> > **Parameters**

> > > • **image** – Image structure pointer

> > **Returns** Image type code, see image type constants for more details

const char* **trax_image_get_path** (const *trax_image** *image*)

> Returns a file path from a file-system path image description. This function returns a pointer to the internal data which should not be modified.

> > **Parameters**

> > > • **image** – Image structure pointer

> > **Returns** Pointer to null-terminated character array

const char* **trax_image_get_url** (const *trax_image** *image*)

> Returns a file path from a URL path image description. This function returns a pointer to the internal data which should not be modified.

> > **Parameters**

> > > • **image** – Image structure pointer

> > **Returns** Pointer to null-terminated character array

void **trax_image_get_memory_header** (const *trax_image** *image*, int* *width*, int* *height*, int* *format*)

> Returns the header data of a memory image.

> > **Parameters**

> > > • **image** – Image structure pointer

> > > • **width** – Pointer to variable that is populated with width of the image

> > > • **height** – Pointer to variable that is populated with height of the image

> > > • **format** – Pointer to variable that is populated with format of the image, see format constants for options

char* **trax_image_write_memory_row** (*trax_image** *image*, int *row*)

> Returns a pointer for a writeable row in a data array of an image.

> > **Parameters**

> > > • **image** – Image structure pointer

> > > • **row** – Number of row

> > **Returns** Pointer to character array of the line

const char* **trax_image_get_memory_row** (const *trax_image** *image*, int *row*)

> Returns a read-only pointer for a row in a data array of an image.

> > **Parameters**

> > > • **image** – Image structure pointer

> > > • **row** – Number of row

> > **Returns** Pointer to character array of the line

const char* **trax_image_get_buffer** (const *trax_image** *image*, int* *length*, int* *format*)

> Returns a file buffer and its length. This function returns a pointer to the internal data which should not be modified.

Parameters

- **image** – Image structure pointer
- **length** – Pointer to variable that is populated with buffer length
- **format** – Pointer to variable that is populated with buffer format code

Returns Pointer to character array

Region

TRAX_REGION_EMPTY

Empty region type, not usable in any way but to signify that there is no data.

TRAX_REGION_SPECIAL

Special code region type, only one value avalable that can have a defined meaning.

TRAX_REGION_RECTANGLE

Rectangle region type. Left, top, width and height values available.

TRAX_REGION_POLYGON

Polygon region type. Three or more points available with x and y coordinates.

. . . c:macro:: TRAX_REGION_MASK

TRAX_REGION_ANY

Any region type, a shortcut to specify that any supported region type can be used.

void **trax_region_release** (*trax_region*** *region*)

Releases region structure, frees allocated memory.

Parameters

- **region** – Pointer to region structure pointer (the pointer is set to NULL if the structure is destroyed successfuly)

int **trax_region_get_type** (const *trax_region** *region*)

Returns type identifier of the region object.

Parameters

- **region** – Region structure pointer

Returns One of the region type constants

trax_region* **trax_region_create_special** (int *code*)

Creates a special region object.

Parameters

- **code** – A numerical value that is contained in the region type

Returns A pointer to the region object

void **trax_region_set_special** (*trax_region** *region*, int *code*)

Sets the code of a special region.

Parameters

- **region** – Region structure pointer
- **code** – The new numerical value

---

int **trax_region_get_special** (const *trax_region* region*)

> Returns a code of a special region object if the region is of *special* type.

> > **Parameters**

> > > • **region** – Region structure pointer

> > **Returns** The numerical value

trax_region* **trax_region_create_rectangle** (float *x*, float *y*, float *width*, float *height*)

> Creates a rectangle region.

> > **Parameters**

> > > • **x** – Left offset

> > > • **y** – Top offset

> > > • **width** – Width of rectangle

> > > • **height** – Height of rectangle

> > **Returns** A pointer to the region object

void **trax_region_set_rectangle** (*trax_region* region*, float *x*, float *y*, float *width*, float *height*)

> Sets the coordinates for a rectangle region.

> > **Parameters**

> > > • **region** – A pointer to the region object

> > > • **x** – Left offset

> > > • **y** – Top offset

> > > • **width** – Width of rectangle

> > > • **height** – Height of rectangle

void **trax_region_get_rectangle** (const *trax_region* region*, float* *x*, float* *y*, float* *width*, float* *height*)

> Retreives coordinate from a rectangle region object.

> > **Parameters**

> > > • **region** – A pointer to the region object

> > > • **x** – Pointer to left offset value variable

> > > • **y** – Pointer to top offset value variable

> > > • **width** – Pointer to width value variable

> > > • **height** – Pointer to height value variable

trax_region* **trax_region_create_polygon** (int *count*)

> Creates a polygon region object for a given amout of points. Note that the coordinates of the points are arbitrary and have to be set after allocation.

> > **Parameters**

> > > • **code** – The number of points in the polygon

> > **Returns** A pointer to the region object

void **trax_region_set_polygon_point** (*trax_region* region*, int *index*, float *x*, float *y*)

> Sets coordinates of a given point in the polygon.

> > **Parameters**

- **region** – A pointer to the region object

- **index** – Index of point

- **x** – Horizontal coordinate

- **y** – Vertical coordinate

void **trax_region_get_polygon_point** (const *trax_region** *region*, int *index*, float* *x*, float* *y*)
Retrieves the coordinates of a specific point in the polygon.

> **Parameters**
>
> - **region** – A pointer to the region object
>
> - **index** – Index of point
>
> - **x** – Pointer to horizontal coordinate value variable
>
> - **y** – Pointer to vertical coordinate value variable

int **trax_region_get_polygon_count** (const *trax_region** *region*)
Returns the number of points in the polygon.

> **Parameters**
>
> - **region** – A pointer to the region object
>
> **Returns** Number of points

trax_region* **trax_region_create_mask** (int *x*, int *y*, int *width*, int *height*)
Creates a mask region object of given size. Note that the mask data is not initialized.

> **Parameters**
>
> - **x** – Left offset
>
> - **y** – Top offset
>
> - **width** – Mask width
>
> - **height** – Mask height
>
> **Returns** A pointer to the region object

void **trax_region_get_mask_header** (const *trax_region** *region*, int* *x*, int* *y*, int* *width*, int* *height*)
Returns the header data of a mask region.

> **Parameters**
>
> - **region** – A pointer to the region object
>
> - **x** – Pointer to left offset value variable
>
> - **y** – Pointer to top offset value variable
>
> - **width** – Pointer to width value variable
>
> - **height** – Pointer to height value variable

char* **trax_region_write_mask_row** (*trax_region** *region*, int *row*)
Returns a pointer for a writeable row in a data array of a mask.

> **Parameters**
>
> - **region** – Region structure pointer
>
> - **row** – Number of row
>
> **Returns** Pointer to character array of the line

const char* **trax_region_get_mask_row** (const *trax_region** *region*, int *row*)

> Returns a read-only pointer for a row in a data array of a mask.
>
> > **Parameters**
> >
> > > • **region** – Region structure pointer
> > >
> > > • **row** – Number of row
> >
> > **Returns** Pointer to character array of the line

trax_bounds **trax_region_bounds** (const *trax_region** *region*)

> Calculates a bounding box region that bounds the input region.
>
> > **Parameters**
> >
> > > • **region** – A pointer to the region object
> >
> > **Returns** A bounding box structure that contains values for left, top, right, and bottom

trax_region* **trax_region_clone** (const *trax_region** *region*)

> Clones a region object.
>
> > **Parameters**
> >
> > > • **region** – A pointer to the region object
> >
> > **Returns** A cloned region object pointer

trax_region* **trax_region_convert** (const *trax_region** *region*, int *format*)

> Converts region between different formats (if possible).
>
> > **Parameters**
> >
> > > • **region** – A pointer to the region object
> > >
> > > • **format** – One of the format type constants
> >
> > **Returns** A converted region object pointer

float **trax_region_contains** (const *trax_region** *region*, float *x*, float *y*)

> Calculates if the region contains a given point.
>
> > **Parameters**
> >
> > > • **region** – A pointer to the region object
> > >
> > > • **x** – X coordinate of the point
> > >
> > > • **y** – Y coordinate of the point
> >
> > **Returns** Returns zero if the point is not in the region or one if it is

float **trax_region_overlap** (const *trax_region** *a*, const *trax_region** *b*, const *trax_bounds bounds*)

> Calculates the spatial Jaccard index for two regions (overlap).
>
> > **Parameters**
> >
> > > • **a** – A pointer to the region object
> > >
> > > • **b** – A pointer to the region object
> >
> > **Returns** A bounds structure to contain only overlap within bounds or `trax_no_bounds` if no bounds are specified

char* **trax_region_encode** (const *trax_region** *region*)

> Encodes a region object to a string representation.
>
> > **Parameters**

- **region** – A pointer to the region object

   Returns A character array with textual representation of the region data

trax_region* **trax_region_decode** (const char* *data*)

   Decodes string representation of a region to an object.

   Parameters

   - **region** – A character array with textual representation of the region data

   Returns A pointer to the region object or NULL if string does not contain valid region data

Properties

trax_properties* **trax_properties_create** ()

   Create an empty properties dictionary.

   Returns A pointer to a properties object

void **trax_properties_release** (*trax_properties*** *properties*)

   Destroy a properties object and clean up the memory.

   Parameters

   - **properties** – A pointer to a properties object pointer

void **trax_properties_clear** (*trax_properties** *properties*)

   Clears a properties dictionary making it empty.

   Parameters

   - **properties** – A pointer to a properties object

void **trax_properties_set** (*trax_properties** *properties*, const char* *key*, const char* *value*)

   Set a string property for a given key. The value string is cloned.

   Parameters

   - **properties** – A pointer to a properties object

   - **key** – A key for the property, only keys valid according to the protocol are accepted

   - **value** – The value for the property, the string is cloned internally

void **trax_properties_set_int** (*trax_properties** *properties*, const char* *key*, int *value*)

   Set an integer property. The value will be encoded as a string.

   Parameters

   - **properties** – A pointer to a properties object

   - **key** – A key for the property, only keys valid according to the protocol are accepted

   - **value** – The value for the property

void **trax_properties_set_float** (*trax_properties** *properties*, const char* *key*, float *value*)

   Set an floating point value property. The value will be encoded as a string.

   Parameters

   - **properties** – A pointer to a properties object

   - **key** – A key for the property, only keys valid according to the protocol are accepted

   - **value** – The value for the property

char* **trax_properties_get** (const *trax_properties** *properties*, const char* *key*)

> Get a string property. The resulting string is a clone of the one stored so it should be released when not needed anymore.

> > **Parameters**
> >
> > > - **properties** – A pointer to a properties object
> > >
> > > - **key** – A key for the property
> >
> > **Returns** The value for the property or NULL if there is no value associated with the key

int **trax_properties_get_int** (const *trax_properties** *properties*, const char* *key*, int *def*)

> Get an integer property. A stored string value is converted to an integer. If this is not possible or the property does not exist a given default value is returned.

> > **Parameters**
> >
> > > - **properties** – A pointer to a properties object
> > >
> > > - **key** – A key for the property
> > >
> > > - **def** – Default value for the property
> >
> > **Returns** The value for the property or default value if there is no value associated with the key or conversion from string is impossible

float **trax_properties_get_float** (const *trax_properties** *properties*, const char* *key*, float *def*)

> Get an floating point value property. A stored string value is converted to an integer. If this is not possible or the property does not exist a given default value is returned.

> > **Parameters**
> >
> > > - **properties** – A pointer to a properties object
> > >
> > > - **key** – A key for the property
> > >
> > > - **def** – Default value for the property
> >
> > **Returns** The value for the property or default value if there is no value associated with the key or conversion from string is impossible

int **trax_properties_count** (const *trax_properties** *properties*)

> Get a number of all pairs in the properties object.

> > **Parameters**
> >
> > > - **properties** – A pointer to a properties object
> >
> > **Returns** Number of key-value pairs in the properties object

void **trax_properties_enumerate** (*trax_properties** *properties*, trax_enumerator *enumerator*, const void* *object*)

> Iterate over the property set using a callback function. An optional pointer can be given and is forwarded to the callback.

> > **Parameters**
> >
> > > - **properties** – A pointer to a properties object
> > >
> > > - **enumerator** – A pointer to the enumerator function that is called for every key-value pair
> > >
> > > - **object** – A pointer to additional data for the enumerator function

Integration example

The library can be easily integrated into C and C++ code (although a C++ wrapper also exists) and can be also linked into other programming languages that enable linking of C libraries. Below is an sripped-down example of a C tracker skeleton with a typical tracking loop. Note that this is not a complete example and servers only as a demonstration of a typical tracker on a tracking-loop level.

```c
#include <stdio.h>

int main( int argc, char** argv)
{
    int i;
    FILE* out;
    rectangle_type region;
    image_type image;

    out = fopen("trajectory.txt", "w");

    region = read_bounding_box();
    image = read_image(1);
    region = initialize_tracker(region, image);

    write_frame(out, region);

    for (i = 2; ; i++)
    {
        image = read_image(i);
        region = update_tracker(image);
        write_frame(out, region);
    }

    fclose(out);
    return 0;
}
```

The code above can be modified to use the TraX protocol by including the C library header and changing the tracking loop to accept frames from the protocol insead of directly reading them from the filesystem. It also requires linking the protocol library (`libtrax`) when building the tracker executable.

```c
#include <stdio.h>

// Include TraX library header
#include "trax.h"

int main( int argc, char** argv)
{
    int run = 1;
    trax_image_list* img = NULL;
    trax_region* reg = NULL;

    // Call trax_server_setup at the beginning
    trax_handle* handle;
    trax_metadata* meta = trax_metadata_create(TRAX_REGION_RECTANGLE, TRAX_IMAGE_PATH,
    ↪ TRAX_CHANNEL_VISIBLE, "Name", NULL, NULL);

    handle = trax_server_setup(meta, trax_no_log);

```

```
18          trax_metadata_release(&meta);

19

20      while(run)
21      {
22          int tr = trax_server_wait(handle, &img, &reg, NULL);

23

24          // There are two important commands. The first one is
25          // TRAX_INITIALIZE that tells the tracker how to initialize.
26          if (tr == TRAX_INITIALIZE) {

27

28              rectangle_type region = initialize_tracker(
29                  region_to_rectangle(reg), load_image(img));
30              trax_server_reply(handle, rectangle_to_region(region), NULL);

31

32          } else
33          // The second one is TRAX_FRAME that tells the tracker what to process next.
34          if (tr == TRAX_FRAME) {

35

36              rectangle_type region = update_tracker(load_image(trax_image_list_get(img,
    ↪ TRAX_CHANNEL_VISIBLE)));
37              trax_server_reply(handle, rectangle_to_region(region), NULL);

38

39          }
40          // Any other command is either TRAX_QUIT or illegal, so we exit.
41          else {
42              run = 0;
43          }

44

45          if (img) {
46              trax_image_list_clear(img); // Also delete individual images
47              trax_image_list_release(&img);
48          }
49          if (reg) trax_region_release(&reg);

50

51      }

52

53      // TraX: Call trax_cleanup at the end
54      trax_cleanup(&handle);

55

56      return 0;
57  }
```

2.2.2 Library C++ wrapper

The main functionality of the reference library is written in pure C, however, it also offers a C++ wrapper if used with a C++ compiler. This wrapper uses classes and objects as well as reference counting for memory management, making is more suitable choice when using the reference library in a C++ algorithm.

Requirements and building

No additional requirements are necessary for building the wrapper but a C++ compiler.

Documentation

The wrapper is composed of several classes, mostly following the underlying C functions. All the classes are contained in `trax` namespace.

class Metadata
:   A wrapper class for tracker metadata structure

    Metadata (int *region_formats*, int *image_formats*, std::string *tracker_name*, std::string *tracker_description*, std::string *tracker_family*)
    :   Creates a new metadata object.

    ~Metadata ()

    int **image_formats** ()
    :   Returns supported image formats as a bit field.

    int **region_formats** ()
    :   Returns supported region formats as a bit field.

    std::string **tracker_name** ()
    :   Returns tracker name string or empty string.

    std::string **tracker_description** ()
    :   Returns tracker description string or empty string.

    std::string **tracker_family** ()
    :   Returns tracker family string or empty string.

class Logging
:   A wrapper class for logging configuration structure

    Logging (trax_logging *logging*)

    Logging (trax_logger *callback* = NULL, void **data* = NULL, int *flags* = 0)

    ~Logging ()

class Bounds

    Bounds ()

    Bounds (trax_bounds *bounds*)

    Bounds (float *left*, float *top*, float *right*, float *bottom*)

    ~Bounds ()

class Handle

    ~Handle ()

    const *Metadata* **metadata** ()

    const bool **terminate** ()
    :   Terminate session by sending quit message. Implicitly called when object is destroyed.

    std::string **get_error** ()
    :   Return last error string or empty string if no error has occured in last call to handle.

    bool **is_alive** ()
    :   Check if the handle is opened or not.

class Client

    Client (int *input*, int *output*, *Logging logger*)
        Sets up the protocol for the client side and returns a handle object.

    Client (int *server*, *Logging logger*, int *timeout* = -1)
        Sets up the protocol for the client side and returns a handle object.

    ~Client ()

    int **wait** (*Region* &*region*, *Properties* &*properties*)
        Waits for a valid protocol message from the server.

    int **initialize** (**const** *Image* &*image*, **const** *Region* &*region*, **const** *Properties* &*properties*)
        Sends an initialize message.

    int **frame** (**const** *Image* &*image*, **const** *Properties* &*properties*)
        Sends a frame message.

class Server

    Server (Configuration *configuration*, *Logging log*)
        Sets up the protocol for the server side and returns a handle object.

    ~Server ()

    int **wait** (*Image* &*image*, *Region* &*region*, *Properties* &*properties*)
        Waits for a valid protocol message from the client.

    int **reply** (**const** *Region* &*region*, **const** *Properties* &*properties*)
        Sends a status reply to the client.

class Image

    Image ()

    Image (**const** *Image* &*original*)

    static *Image* **create_path** (**const** std::string &*path*)
        Creates a file-system path image description. See `trax_image_create_path()`.

    static *Image* **create_url** (**const** std::string &*url*)
        Creates a URL path image description. See `trax_image_create_url()`.

    static *Image* **create_memory** (int *width*, int *height*, int *format*)
        Creates a raw buffer image description.See `trax_image_create_memory()`.

    static *Image* **create_buffer** (int *length*, **const** char **data*)
        Creates a file buffer image description. See `trax_image_create_buffer()`.

    ~Image ()
        Releases image structure, frees allocated memory.

    int **type** () **const**
        Returns a type of the image handle. See `trax_image_get_type()`.

    bool **empty** () **const**
        Checks if image container is empty.

    const std::string **get_path** () **const**
        Returns a file path from a file-system path image description. This function returns a pointer to the internal
        data which should not be modified.

const std::string **get_url**() **const**
> Returns a file path from a URL path image description. This function returns a pointer to the internal data which should not be modified.

void **get_memory_header** (int *width*, int *height*, int *format*) **const**
> Returns the header data of a memory image.

char *****write_memory_row** (int *row*)
> Returns a pointer for a writeable row in a data array of an image.

const char *****get_memory_row** (int *row*) **const**
> Returns a read-only pointer for a row in a data array of an image.

const char *****get_buffer** (int *length*, int *format*) **const**
> Returns a file buffer and its length. This function returns a pointer to the internal data which should not be modified.

class Region

Region()
> Creates a new empty region.

Region (**const** *Region* &*original*)
> Creates a clone of region.

static *Region* **create_special** (int *code*)
> Creates a special region object. Only one paramter (region code) required.

static *Region* **create_rectangle** (float *x*, float *y*, float *width*, float *height*)
> Creates a rectangle region.

static *Region* **create_polygon** (int *count*)
> Creates a polygon region object for a given amout of points. Note that the coordinates of the points are arbitrary and have to be set after allocation.

static *Region* **create_mask** (int *x*, int *y*, int *width*, int *height*)
> Creates a mask region object of given size. Note that the mask data is not initialized.

~Region()
> Releases region, frees allocated memory.

int **type**() **const**
> Returns type identifier of the region object.

bool **empty**() **const**
> Checks if region container is empty.

void **set** (int *code*)
> Sets the code of a special region.

int **get**() **const**
> Returns a code of a special region object.

void **set** (float *x*, float *y*, float *width*, float *height*)
> Sets the coordinates for a rectangle region.

void **get** (float *x*, float *y*, float *width*, float *height*) **const**
> Retreives coordinate from a rectangle region object.

void **set_polygon_point** (int *index*, float *x*, float *y*)
> Sets coordinates of a given point in the polygon.

void **get_polygon_point** (int *index*, float **x*, float **y*) **const**
    Retrieves the coordinates of a specific point in the polygon.

int **get_polygon_count** () **const**
    Returns the number of points in the polygon.

void **get_mask_header** (int **x*, int **y*, int **width*, int **height*) **const**
    Returns the header data of a mask region.

char ****write_mask_row** (int *row*)
    Returns a pointer for a writeable row in a data array of a mask.

const char ****get_mask_row** (int *row*) **const**
    Returns a read-only pointer for a row in a data array of a mask.

Bounds **bounds** () **const**
    Computes bounds of a region.

Region **convert** (int *type*) **const**
    Convert region to one of the other types if possible.

float **overlap** (**const** *Region* &*region*, **const** *Bounds* &*bounds* = *Bounds*()) **const**
    Calculates the Jaccard index overlap measure for the given regions with optional bounds that limit the
    calculation area.

class Properties

Properties ()
    Create a property object.

Properties (**const** *Properties* &*original*)
    A copy constructor.

~Properties ()
    Destroy a properties object and clean up the memory.

int **size** ()
    Return the number of elements.

void **clear** ()
    Clear a properties object.

void **set** (**const** std::string *key*, **const** std::string *value*)
    Set a string property (the value string is cloned).

void **set** (**const** std::string *key*, int *value*)
    Set an integer property. The value will be encoded as a string.

void **set** (**const** std::string *key*, float *value*)
    Set an floating point value property. The value will be encoded as a string.

std::string **get** (**const** std::string *key*, **const** std::string &*def*)
    Get a string property.

int **get** (**const** std::string *key*, int *def*)
    Get an integer property. A stored string value is converted to an integer. If this is not possible or the
    property does not exist a given default value is returned.

float **get** (**const** std::string *key*, float *def*)
    Get an floating point value property. A stored string value is converted to an float. If this is not possible or
    the property does not exist a given default value is returned.

bool **get** (**const** std::string *key*, bool *def*)
> Get an boolean point value property. A stored string value is converted to an integer and checked if it is zero. If this is not possible or the property does not exist a given default value is returned.

void **enumerate** (Enumerator *enumerator*, void **object*)
> Iterate over the property set using a callback function. An optional pointer can be given and is forwarded to the callback.

void **from_map** (**const** std::map<std::string, std::string> &*m*)
> Adds values from a dictionary to the properties object.

void **to_map** (std::map<std::string, std::string> &*m*)
> Copies key-value pairs in the properties object into the given dictionary.

void **to_vector** (std::vector<std::string> &*v*)
> Copies keys from the properties object into the given vector.

Integration example

In C++ tracker implementations you can use either the C++ wrapper or basic C protocol implementation. The wrapper is more conveninent as it is object-oriented and provides automatic deallocation of resources via reference counting. Below is an sripped-down example of a C++ tracker skeleton with a typical tracking loop. Note that this is not a complete example and servers only as a demonstration of a typical tracker on a tracking-loop level.

```
1   #include <iostream>
2   #include <fstream>
3
4   using namescpace std;
5
6   int main( int argc, char** argv)
7   {
8       int i;
9       FILE* out;
10      Rectangle region;
11      Image image;
12      Tracker tracker;
13
14      ofstream out;
15      output.open("trajectory.txt", ofstream::out);
16
17      region = read_bounding_box();
18      image = read_image(1);
19      region = tracker.initialize(region, image);
20
21      out << region << endl;
22
23      for (i = 2; ; i++)
24      {
25        image = read_image(i);
26        region = tracker.update(image);
27        out << region << endl;
28      }
29
30      out.close();
31      return 0;
32  }
```

The code above can be modified to use the TraX protocol by including the C/C++ library header and changing the

tracking loop to accept frames from the protocol insead of directly reading them from the filesystem. It also requires linking the protocol library (`libtrax`) when building the tracker executable.

```cpp
#include <stdio.h>

// Include TraX library header
#include "trax.h"

using namespace std;

int main( int argc, char** argv)
{
    int run = 1;

    // Initialize protocol
    trax::Server handle(trax::Metadata(TRAX_REGION_RECTANGLE,
                                    TRAX_IMAGE_PATH), trax_no_log);

    while(run)
    {
        trax::Image image;
        trax::Region region;
        trax::Properties properties;

        int tr = handle.wait(image, region, properties);

        // There are two important commands. The first one is
        // TRAX_INITIALIZE that tells the tracker how to initialize.
        if (tr == TRAX_INITIALIZE) {

            cv::Rect result = tracker.initialize(
                    trax::region_to_rect(region), trax::image_to_mat(image));

            handle.reply(trax::rect_to_region(result), trax::Properties());

        } else
        // The second one is TRAX_FRAME that tells the tracker what to process next.
        if (tr == TRAX_FRAME) {

            cv::Rect result = tracker.update(image_to_mat(image));
            handle.reply(trax::rect_to_region(result), trax::Properties());

        }
        // Any other command is either TRAX_QUIT or illegal, so we exit.
        else {
            run = 0;
        }

    }

    return 0;
}
```

2.2.3 Python wrapper

Requirements and support

The Python wrapper requires only core Python libraries and numpy for handling mask data. The Python code itself is a CTypes wrapper arround the trax C library.

Documentation

Main module

Server-side communication

Client-side communication

Image module

Region module

Integration example

Below is a simple example of a Python code skeleton for a tracker, exposing its tracking loop but hidding all tracker-specific details to keep things clear.

```python
1   tracker = Tracker()
2   trajectory = [];
3   i = 1
4
5   rectangle = read_bounding_box()
6   image = read_image(i)
7   rectangle = tracker.initialize(rectangle, image)
8
9   trajectory.append(rectangle)
10
11  while True:
12      i = i + 1
13      image = read_image(i)
14      rectangle = tracker.update(image)
15      trajectory.append(rectangle)
16
17  write_trajectory(trajectory)
```

To make the tracker work with the TraX protocol you have to modify the above code in the following way and also make sure that the `trax` module will be available at runtime.

```python
1   import trax.server
2   import trax.region
3   import trax.image
4   import time
5
6   tracker = Tracker()
7
8   with trax.server.Server(trax.region.RECTANGLE, trax.image.PATH) as server:
9       while True:
10          request = server.wait()
```

```python
11      if request.type in ["quit", "error"]:
12        break
13      if request.type == "initialize":
14        rectangle = tracker.initialize(get_rectangle(request.region),
15              load_image(request.image))
16      else:
17        rectangle = tracker.update(load_image(request.image))
18
19      server.status(get_region(rectangle))
```

2.2.4 Matlab/Octave wrappers

Matlab is a multi-paradigm numerical computing environment and a programming language, very popular among computer vision researchers. Octave is its open-source counterpart that is sometimes used as a free alternative. Both environments are by themselves quite limiting in terms of low-lever operating system access required for the TraX protocol to work, but offer integration of C/C++ code in the scripting language, tipically via MEX mechanism (dynamic libraries with a predefined entry point). Using this mechanism a reference C implementation can be wrapped and used in a Matlab/Octave tracker.

Requirements and building

To compile `traxserver` MEX function manually you need a MEX compiled configured correctly on your computer.

Since the MEX function is only a wrapper for the C library, you first have to ensure that the C library is compiled and available in a subdirectory of the project. Then go to Matlab/Octave console, move to the `support/matlab/` subdirectory and execute `compile_trax` script. If the script finishes correctly you will have a MEX script (extension varies from platform to platform) available in the directory. If the location of the TraX library is not found automatically, you have to verify that it exists and possibly enter the path to its location manually.

Documentation

The `traxserver` MEX function is essentially used to send or receive a protocol message and parse it. It accepts several parameters, the first one is a string of a command that you want execute.

```
[response] = traxserver('setup', region_formats, image_formats, ...);
```

The call setups the protocol and has to be called only once at the beginning of your tracking algorithm. The two mandatory input arguments are:

- **region_formats**: A string that specifies region format that is supported by the algorithm. Any other formats should be either converted by the client or the client should terminate if it is unable to provide data in correct format. Possible values are: **rectangle** or **polygon**, see protocol specification for more details.

- **image_formats**: A string or specifies the image format that are supported by the algorithm. Any other formats should be either converted by the client or the client should terminate if it is unable to provide data in correct format. Possible values are: **path**, **url**, **memory** or **data**, see protocol specification for more details.

Additionally, you can also specify tracker name, description and family taxonomy as strings using named arguments technique. A single output argument is a boolean value that is true if the initialization was successful.

```
[image, region, parameters] = traxserver('wait');
```

A call blocks until a protocol message is received from the client, parses it and returns the data. Based on the type of message, some output arguments will be initialized as empty which also helps determining the type of the message.

- **image**: Image data in requested format. If the variable is empty then the termination request was received and the tracker can exit.

- **region**: Region data in requested format. If the variable is set then an initialization request was received and the tracker should be (re)initialized with the specified region. If the variable is empty (but the image variable is not) then a new frame is received and should be processed.

```
[response] = traxserver('state', region, parameters);
```

A call sends a status message back to the client specifying the region for the current frame as well as the optional parameters. The two input arguments are:

- **region**: Region data in requested format.

- **parameters** (optional): Arbitrary output parameters used for debugging or any other purposes. The parameters are provided either in a single-level structure (no nested structures, just numbers or strings for values) or a N x 2 cell matrix with string keys in the first column and values in the second. Note that the protocol restricts the characters used for parameter names and limits their length.

```
[response] = traxserver('quit', parameters);
```

Sends a quit message to the client specifying that the algorithm wants to terminate the tracking session. Additional parameters can be specified using an input argument.

- **parameters** (optional): Arbitrary output parameters used for debugging or any other purposes, formats same as above.

Image data

Image data is stored in a matrix. For file path and URL types this is a one-dimensional char sequence, for in-memory image this is a 3-dimensional matrix of type `uint8` with raw image data, ready for processing and for data type it is in a structure with fields `format` and `data` that contain encoding format (JPEG or PNG) and raw file data.

Region data

Region data for rectangle and polygon types is stored in a one-dimensional floating-point matrix. For rectangle the number of elements is 4, for polygon it is an even number, greater or equal than 6 (three points). In all cases the first coordinate is in the horizontal dimension (columns) and not the way Matlab/Octave usually addresses matrices.

Internals

Additionally the function also looks for the `TRAX_SOCKET` environmental variable that is used to determine that the server has to be set up using TCP sockets and that a TCP server is opened (the port or IP address and port are provide as the value of the variable) and waiting for connections from the tracker. This mechanism is important for Matlab on Microsoft Windows because the standard streams are closed at startup and cannot be used.

Integration example

As with all tracker implementations it is important to identify a tracking loop. Below is a very simple example of how a typical tracking loop looks in Matlab/Octave with all the tracker specific code removed and placed in self-explanatory functions.

```matlab
1    % Initialize the tracker
2    region = read_bounding_box('init.txt');
3    image = imread('0001.jpg');
4    region = initialize_tracker(region, image);
5
6    result = {region};
7    i = 2;
8
9    while true
10           % End-of-sequence criteria
11           if ~exist(sprintf('%04d.jpg', i), 'file')
12                   break;
13           end;
14           i = i + 1;
15
16           % Read the next image.
17           image = imread(sprintf('%04d.jpg', i));
18
19           % Run the update step
20           region = update_tracker(image);
21
22           % Save the region
23           result{end+1} = region;
24    end
25
26    % Save the result
27    save_trajectory(result);
```

To enable tracker to receive the images over the protocol you have to change a few lines. First, you have to initialize the protocol at the begining of the script and tell what kind of image and region formats the tracker supports. Then the initialization of a tracker has to be placed into a loop because the protocol

```matlab
1    % Initialize the protocol
2    traxserver('setup', 'rectangle', 'path', 'Name', 'Example');
3
4    while true
5    % Wait for data
6    [image, region] = traxserver('wait');
7
8    % Stopping criteria
9    if isempty(image)
10           break;
11    end;
12
13    % We are reading a given path
14    mat_image = imread(image);
15
16    if ~isempty(region)
17           % Initialize tracker
18       region = initialize_tracker(region, mat_image);
19    else
20           region = update_tracker(mat_image);
21    end
22
23           % Report back result to advance to next frame
24    traxserver('status', region);
25
```

```
26  end
27
28  % Quit session if that was not done already
29  traxserver('quit');
```

2.2.5 Client library

The client support library provides a C++ client class that uses C++ protocol API to communicate with the tracker process, besides communication the class also takes care of launching tracker process, handling timeouts, logging, and other things. The class also supports setting up communication over streams as well as over TCP sockets. A detailed tutorial on how to use the client library is available *here*. To compile the module you have to enable BUILD_CLIENT flag in CMake build system (check out the *tutorial on compiling the project* for more details).

CONNECTION_DEFAULT
> Default connection mode, standard input and output streams.

CONNECTION_EXPLICIT
> Non-standard input and output streams are created, passed to tracker via environment variables TRAX_IN and TRAX_OUT.

CONNECTION_SOCKETS
> Using TCP socket for communication, connection parameters passed to tracker via environment variable TRAX_SOCKET.

class TrackerProcess

> **TrackerProcess**(**const** std::string &*command*, **const** std::map<string, string> *environment*, int *timeout* = 10, trax::ConnectionMode *connection* = trax::CONNECTION_DEFAULT, VerbosityMode *verbosity* = trax::VERBOSITY_DEFAULT)
>
> > **Parameters**
> >
> > - **command** – The name of tracker program followed by its input arguments
> >
> > - **environment** – A map of environmental variables that have to be set for the tracker process
> >
> > - **timeout** – Number of seconds to wait for tracker's response before terminating the session
> >
> > - **connection** – Type of connection, supported are either *CONNECTION_DEFAULT*, *CONNECTION_EXPLICIT*, or *CONNECTION_SOCKETS*.
> >
> > - **verbosity** – How verbose should the output be

> **~TrackerProcess**()
> > Destroys the process and cleans up data.

> int **image_formats**()
>
> > **Returns** A bitset of image formats supported by the tracker.

> int **region_formats**()
>
> > **Returns** A bitset of region formats supported by the tracker.

> bool **ready**()
>
> > **Returns** True if the tracker process is alive.

bool **tracking**()

> **Returns** True if the tracker process is tracking an object (was initalized).

bool **initialize**(**const** trax::*Image* &*image*, **const** trax::*Region* &*region*, **const** trax::*Properties* &*properties* = trax::*Properties*())
Request tracker initialization with a given image and region.

> **Parameters**
>
> - **image** – Input image object
>
> - **region** – Input region object
>
> - **properties** – Optional properties passed to the tracker
>
> **Returns** True on success, if the function returns false check *trax::TrackerProcess::ready()* if the process has crashed or simply requested termination

bool **frame**(**const** trax::*Image* &*image*, **const** trax::*Properties* &*properties* = trax::*Properties*())
Request tracker update with a given image.

> **Parameters**
>
> - **image** – Input image object
>
> - **properties** – Optional properties passed to the tracker
>
> **Returns** True on success, if the function returns false check *trax::TrackerProcess::ready()* if the process has crashed or simply requested termination

bool **wait**(trax::*Region* &*region*, trax::*Properties* &*properties*)
Waits for tracker's response to the previous request (either initialization or update).

> **Parameters**
>
> - **region** – Output region object, on success populated with output region
>
> - **properties** – Optional output properties object, populated by returned optional values, if any
>
> **Returns** True on success, if the function returns false check *trax::TrackerProcess::ready()* if the process has crashed or simply requested termination

bool **reset**()
Restarts the tracker process. The function terminates the tracker process and starts a new one.

int **load_trajectory**(**const** std::string &*file*, std::vector<*Region*> &*trajectory*)
Utility function to load a trajectory (a sequence of object states) form a text file.

> **Parameters**
>
> - **file** – Filename string
>
> - **trajectory** – Empty vector that will be populated with region states
>
> **Returns** Number of read states

void **save_trajectory**(**const** std::string &*file*, std::vector<*Region*> &*trajectory*)
Utility function to save a trajectory (a sequence of object states) to a text file.

> **Parameters**
>
> - **file** – Filename string

   - **trajectory** – Vector that contains the trajectory

2.2.6 OpenCV conversions library

OpenCV is one of most frequently used C++ libraries in computer vision. This support library provides conversion functions so that protocol image and region objects can be quickly converted to corresponding OpenCV objects and vice-versa.

Requirements and building

To compile the module you have to enable the `BUILD_OPENCV` flag in CMake build system (check out the *tutorial on compiling the project* for more details). You will of course need OpenCV library available on the system, version 2.4 or higher is supported.

Documentation

cv::Mat **image_to_mat** (**const** *Image* &*image*)
>   Converts a protocol image object to an OpenCV matrix that represents the image.

>   > **Parameters** **image** – Protocol image object

>   > **Returns** OpenCV matrix object

cv::Rect **region_to_rect** (**const** *Region* &*region*)
>   Converts a protocol region object to an OpenCV rectangle structure.

>   > **Parameters** **image** – Protocol region object

>   > **Returns** OpenCV rectangle structure

std::vector<cv::Point2f> **region_to_points** (**const** *Region* &*region*)
>   Converts a protocol region object to a list of OpenCV points.

>   > **Parameters** **image** – Protocol region object

>   > **Returns** List of points

Image **mat_to_image** (**const** cv::Mat &*mat*)
>   Converts an OpenCV matrix to a new protocol image object.

>   > **Parameters** **mat** – OpenCV image

>   > **Returns** Protocol image object

Region **rect_to_region** (**const** cv::Rect *rect*)
>   Converts an OpenCV rectangle structure to a protocol region object of type rectangle.

>   > **Parameters** **rect** – Rectangle structure

>   > **Returns** Protocol region object

Region **points_to_region** (**const** std::vector<cv::Point2f> *points*)
>   Converts a list of OpenCV points to a protocol region object of type polygon.

>   > **Parameters** **rect** – List of points

>   > **Returns** Protocol region object

void **draw_region** (cv::Mat &*canvas*, **const** *Region* &*region*, cv::Scalar *color*, int *width* = 1)
>   Draws a given region to an OpenCV image with a given color and line width.

Parameters

- **canvas** – Target OpenCV image to which the region is drawn
- **region** – Protocol region object
- **color** – Color of the line as a an OpenCV scalar structure
- **width** – Width of the line

2.3 Support utilities

Besides protocol implementation and utility libraries the repository also contains several tools that can be used for testing and performing simple experiments these tools are part of the client support module as they utilize the client library.

2.3.1 CLI interface

Client support module also provides a *traxclient*, a simple CLI (command line interface) to the client that can be used for simple tracker execution. If the OpenCV support module is also compiled then the CLI interface uses it for some extra conversions that are otherwise not supported (e.g. loading images and sending them in their raw form over the communication channel if the server requests it).

The *traxclient* utility is in fact quite powerful and can be used to perform many batch experiments if run multiple times with different parameters. It is internally used by the VOT toolkit and can be used to perform other batch experiments as well. If we assume that image sequence is stored in *images.txt* file (one absolute image path per line) and that groundtruth are stored in *groundtruth.txt* file (one region per line, corresponding to the images), we can run a tracker on this sequence by calling:

```
traxclient -g groundtruth.txt -I images.txt -o result.txt -- <tracker command>
```

This command will load the groundtruth and use it to initialize the tracker, then it will monitor the tracker until the end of the sequence and storing the resulting trajectory to *result.txt*. More complex behavior can be achieved by adding other command-line flags (run `traxclient -h` to get their list).

2.3.2 Protocol testing

The *traxtest* utility can be used for protocol support testing even if no sequence is available. The tool can produce a static image stream to test if the tracker is responding properly.

2.3.3 OpenCV player

If the OpenCV support module is also compiled then a *traxplayer* utility is available. This utility can be used to interactively test trackers with the data from video or camera stream.

2.4 Tutorials

This is a list of official tutorials on TraX, covering a wide set of topics form compiling the library to its integration on various platforms and in various languages and even creating a specialized TraX client. The list does not cover all the possible use cases so contributions (as well as corrections) are welcome and will could be included in the documentation.

2.4.1 General remarks and guidelines

Before starting implementing a new tracker or trying to adapt existing implementation to support TraX protocol, please read the following guidelines on the structure of the code. A lot of these remarks are very common and are not important only for easy integration of the protocol but for general code readability which is an important aspect of reproducibility of results.

Tracking loop

From the implementation point of view, all on-line trackers should implement the *tracking loop* pattern in some way, however, the loop is much harder to identify in some project than in the other. The tracking loop describes the general state of the tracker. A tracker is initialized (is given and image and the state of the object to initialize its internal state), is updated in a loop for a given amount of frames for which it produces a predicted state of the tracked object, and is eventually terminated.

A clear separation of the tracking loop means that it is clear where these three stages are located in the source code. This kind of organization makes integration of the TraX protocol much easier.

Algorithm abstraction

Clear abstraction of the tracking algorithm is related to the tracking loop pattern. It means that the boilerplate code that handles input data retrieval, manages tracking loop, and eventually takes care of result storage is clearly separated from the main algorithm. How this separation is achieved depends greatly on the programming language used, the most common approach is to use object-oriented structure and present the algorithm as a class with functions to initialize and update the state.

2.4.2 Compiling TraX library

The TraX library and supporting modules are built using CMake build tool. If you have not used CMake before the best way to familiarize with it is by going through the official tutorial. CMake does not build the code by itself but generates build instructions for a given platform, e.g. Make on Unix systems, Visual Studio on Windows, and Xcode on OSX.

Note: In addition to CMake you will also have to install Python distutil if you want to build a Python package (although you do not have to do this to use the Python implementation) and Octave development packages if you want to build Octave-compatible MEX file (this can also be done from *within the Octave or Matlab console*).

If you want to build optional OpenCV support, you will have to install OpenCV library (together with development header files). Depending on your OpenCV installation location you may have to tell the CMake where to find the library.

Compiling using Make (Unix)

Building the project using Make tool is the typical for Unix systems (e.g. Linux and Max OSX). After installing all the dependencies and obtaining TraX source code you can open the terminal, go to the library source root directory and type the following commands:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

With this you will create a subdirectory named *build* that will contain all the build artifacts, move to the directory, use CMake to generate build instructions and run Make to build the code.

Note: On Unix systems it is also possible to install the library and all public files directly from the build using *make install* command. The result of the command is system-depended but it will very likely install the library and other files in the */usr/local* directory where they will be available to all users. While this kind of installation is not recommended it is a fast way of getting library available without having to manually configure search paths.

By default only the C library is built, if you want to enable additional modules you have to tell this to CMake by adding flags:

```
$ cmake -DBUILD_OPENCV=ON -DBUILD_CLIENT=ON ..
```

This way both OpenCV module and client module will be built when running the Make command.

Compiling using Visual Studio (Windows)

On Windows CMake provides a GUI interface that can be used to generate a Visual Studio solution. Open the interface and set the source and build paths. According to the CMake best practices we perform build in a separate sub-directory to separate the generated binaries that can be safely removed and the source code. An example is shown in the screenshot of the interface below.
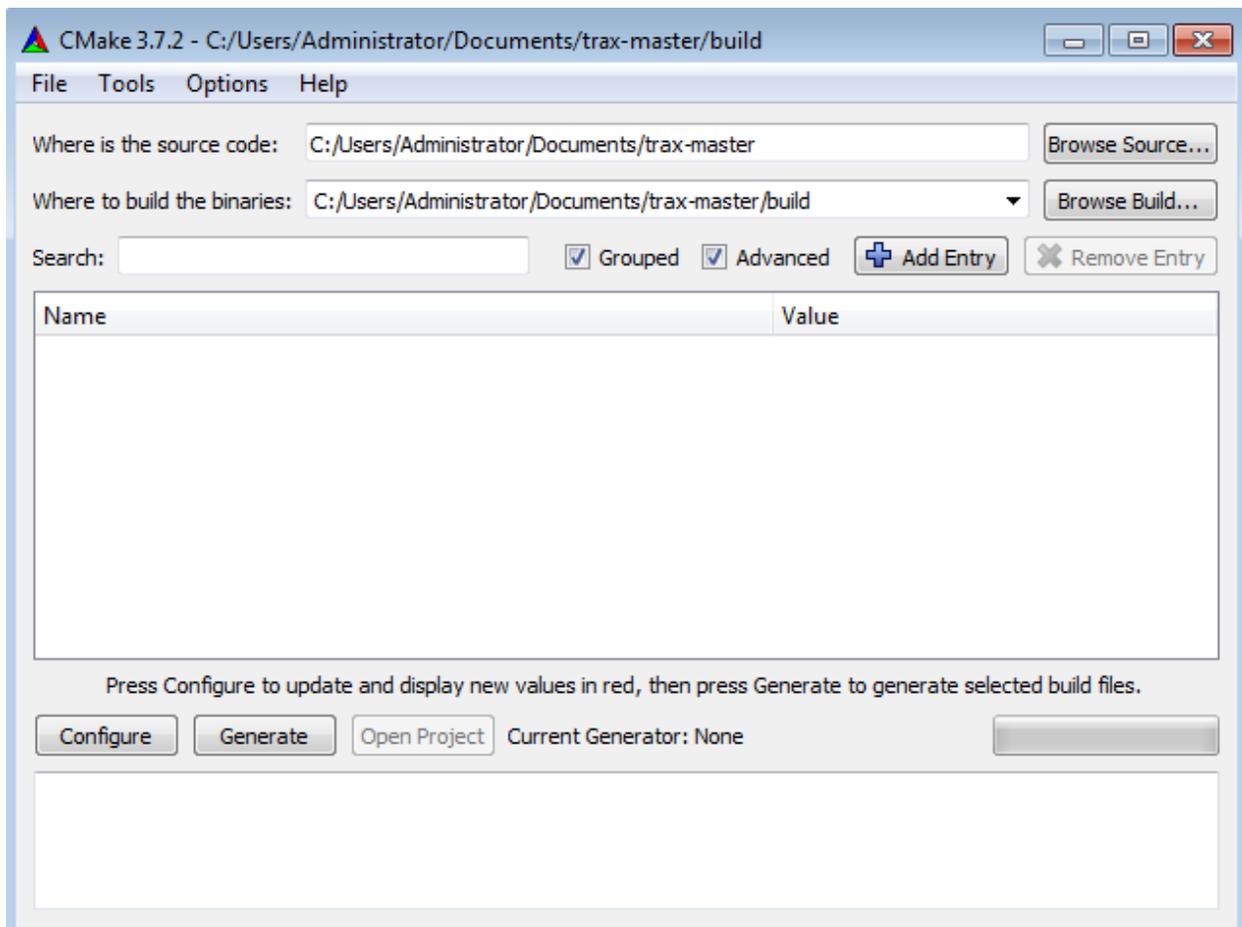
When configuring the project, first select the appropriate generator for you version of Visual Studio. After first generation stage, the variables table will be populated with default variable values. Optional modules, such as client support library or OpenCV extensions can be turned on using variables (OpenCV extensions will also require OpenCV 2.4 or higher to be available on the computer, you may have to tell CMake where to find it).
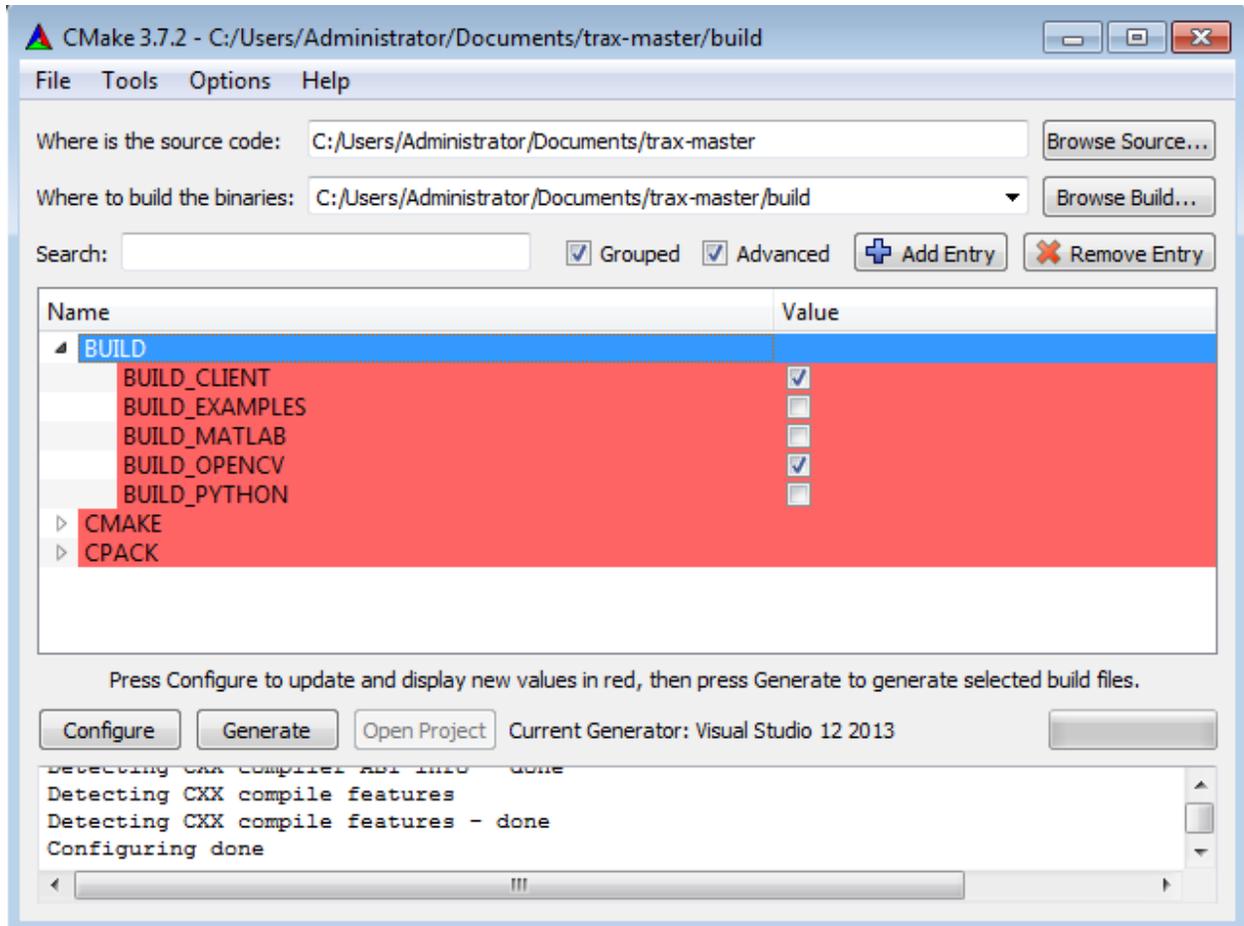
After the configuration stage is successful, you can generate Visual Studio solution, open it can compile the project by compiling the target *ALL_BUILD*. The compiled files will be available in either *Debug* or *Release* subdirectory of your build directory. Alternatively you can also compile target *INSTALL* to install the libraries to the folder, specified by variable *CMAKE_INSTALL_PREFIX* or targer *PACKAGE* to generate a ZIP archive of compiled artifacts.

2.4.3 Using TraX on Linux

This tutorial will show you how to compile C and C++ trackers on Linux systems using CMake build system. It is assumed that you have read the *general remarks on tracker structure*.

There are many Linux distributions, this tutorial will focus on the most popular one, Ubuntu, and its derivatives (e.g. Linux Mint). For these systems a PPA repository is available so that you do not have to compile the TraX library yourself. For other distributions you have to *compile and install* the library before continuing with the second step of this tutorial.

Step 1: Installing library from PPA

To install the library from PPA you have to add the PPA to your package repository list:

```
sudo add-apt-repository ppa:lukacu/trax
sudo apt-get update
```

Then install the required packages:

```
sudo apt-get install libtrax0 libtrax-dev
```

Note: The project is separated in multiple packages in the PPA, if you want to install OpenCV support you will have to install additional packages

```
sudo apt-get install libtrax-opencv0 libtrax-opencv-dev
```

Step 2: Preparing your project

For the purposes of this tutorial we will use a simple NCC tracker that uses OpenCV library (because of this we will also use OpenCV support library to make integration easier and more general). The original source code for the tracker is available in the project repository in directory *docs/tutorials/tracker*. All the source code of the tracker is contained in a single file called *ncc.cpp*.

Inspect the source code to see that it is composed of three parts. The first one is the tracker implementation which contains the actual tracking algorithm. The second one contains utility functions that take care of loading the input data and writing the results. The final part contains the main tracking loop. Since this code is properly structured, all the adaptation work has to be done on this final part of the code. The initial tracking loop looks like:

```cpp
int main( int argc, char** argv) {

  // Reading
  std::vector<std::string> images = readsequence("images.txt");
  cv::Rect initialization = readrectangle("region.txt");
  std::vector<std::Rect> output;

  NCCTracker tracker;

  // Initializing the tracker with first image
  cv::Mat image = cv::imread(images[0]);
  tracker.init(image, initialization);
  // Adding first output to maintain equal length of outputs and
  // input images.
  output.push_back(initialization);

  // The tracking loop, iterating through the rest of the sequence
  for (size_t i = 1; i < images.size(); i++) {

    cv::Mat image = cv::imread(images[i]);
    cv::Rect rect = tracker.update(image);
    output.push_back(rect);

  }

  // Writing the tracking result and exiting
  writerectangles("output.txt", output);

}
```

To modify the source code to use the TraX protocol we have to remove explicit loading of the initialization data and let the protocol take care of this.

```cpp
int main( int argc, char** argv) {

  NCCTracker tracker;

  trax::Server handle(trax::Metadata(TRAX_REGION_RECTANGLE, TRAX_IMAGE_PATH |
                         TRAX_IMAGE_MEMORY | TRAX_IMAGE_BUFFER), trax_no_log);

  while (true) {

    trax::Image image;
    trax::Region region;
    trax::Properties properties;

    int tr = handle.wait(image, region, properties);
    if (tr == TRAX_INITIALIZE) {

      tracker.init(trax::image_to_mat(image), trax::region_to_rect(region));
      handle.reply(region, trax::Properties());

    } else if (tr == TRAX_FRAME) {

      cv::Rect result = tracker.update(image_to_mat(image));
      handle.reply(trax::rect_to_region(result), trax::Properties());

    }
    else break;
  }

}
```

Lets now look at individual modifications. The creation *handle* object provides a protocol server handle that initializes the protocol and sends the introductory message. The handle is given protocol configuration structure that specifies what kind of data the server can handle as well as the optional output log that can be used for debugging:

```cpp
trax::Server handle(trax::Configuration(TRAX_IMAGE_PATH | TRAX_IMAGE_MEMORY |
                    TRAX_IMAGE_BUFFER, TRAX_REGION_RECTANGLE), trax_no_log);
```

When using OpenCV support library's function *trax::image_to_mat()*, the conversion from file path, raw memory and image buffer types happens automatically so supporting them all is really easy. Without this function you have to convert the image yourself.

The tracking loop has been modified to accept commands from the client. This happens with the call to the *trax::Server::wait()* function. The function populates the provided variables: new image, object state (on initialization), and optional parameters.

Since this kind of client-guided session means that the server does not know in advance how long will the tracking session be, the loop is only broken when a quit message is received from the client. The other two options are the initialization and new frame (which can always only follow successful initialization). The re-initialization can happen at any time throughout the session so the server should be capable of reinitializing the tracker (note that in untrusted setups the client may also terminate the session and start a new one).

Requests for initialization or update must be answered with a state message generated by *trax::Server::reply()*. This function accepts the object state as predicted by the tracker as well as any additional parameters that can be accumulated by the client for development or debugging purposes.

Because all the results are processed and stored by the client, we can remove the explicit results storage at the end of

the loop. All these modifications also make all the utility functions in from the initial tracker implementation (second part of the source code) obsolete, they can be removed as their function is handled by the client.

Step 3: Compiling the project

Finally we will modify the example's CMake file. Properly installed TraX library supports CMake discovery mechanism, the only line that we have to add is therefore:

```
FIND_PACKAGE(trax REQUIRED COMPONENTS core opencv)
TARGET_LINK_LIBRARIES(ncc_tracker ${TRAX_LIBRARIES})
INCLUDE_DIRECTORIES(AFTER ${TRAX_INCLUDE_DIRS})
LINK_DIRECTORIES(AFTER ${TRAX_LIBRARY_DIRS})
```

This command will locate the TraX core library and the OpenCV support library and configure several variables.

The project is compiled as a standard CMake project. First, we call CMake to generate a Makefile, then we call the *make* tool to build the project. According to the CMake best practices we perform build in a separate sub-directory to separate the generated binaries that can be safely removed and the source code:

```
$ mkdir bulild
$ cd build
$ cmake ..
$ make
```

Step 4: Testing integration

A successful build results in a binary program (in the case of this tutorial the program is called *ncc_tracker*. To test if the program correctly supports TraX protocol we can use the client *traxtest* provided by the client support module of the project. This program tries to run the tracker on a sequence of static images to see if the protocol is correctly supported. Note that this test does not discover all the logical problems of the implementation as they may only occur during very specific conditions; it only tests the basic TraX compliance.

To run the test move to the build directory and type:

```
$ traxtest -d -- ncc_tracker
```

If the integration is successful this command should output something like:

```
CLIENT: Starting process  "ncc_tracker"
CLIENT: Setting up TraX with standard streams connection
@@TRAX:hello "trax.image=path;memory;buffer;" "trax.region=rectangle;" "trax.version=1
↪"
CLIENT: Tracker process ID: 13019
CLIENT: Connection with tracker established.
@@TRAX:initialize "data:image/...
Tracker initialized
@@TRAX:state "130.0000,80.0000,70.0000,110.0000"
@@TRAX:frame "data:image/...
Tracker updated
@@TRAX:state "130.0000,80.0000,70.0000,110.0000"
@@TRAX:frame "data:image/@@TRAX:frame "data:image/...
Tracker updated
@@TRAX:state "130.0000,80.0000,70.0000,110.0000"
@@TRAX:frame "data:image/...
Tracker updated
```

(continues on next page)

```
...
...
```

2.4.4 Using TraX on Windows

This tutorial will show you how to compile C and C++ trackers on Windows systems using CMake build system. It is assumed that you have read the *general remarks on tracker structure*.

This tutorial assumes that you are using Visual Studio 2012 or 2015 for your development and are reasonably familiar with Visual Studio in general. For other build systems and version of Visual Studio you have to *compile and install* the library before continuing with the second step of this tutorial.

Step 1: Downloading binaries

Prebuilt binaries are available here. They are compiled using the AppVeyor Continious Integration service. The binaries are compiled for Visual Studio 2012 (32bit and 64bit versions, linked with OpenCV 2.4) and 2015 (64bit version, linked with OpenCV 3.1).

Download the appropriate archive and unzip it to the desired location. Then add path to the *bin* subdirectory of the extracted bundle to the system PATH variable so that you will be able to use client tools and that programs will find the TraX dynamic library.

Step 2: Preparing your project

For the purposes of this tutorial we will use a simple NCC tracker that uses OpenCV library (because of this we will also use OpenCV support library to make integration easier and more general). The original source code for the tracker is available in the project repository in directory *docstutorialstracker*. All the source code of the tracker is contained in a single file called *ncc.cpp*.

Inspect the source code to see that it is composed of three parts. The first one is the tracker implementation which contains the actual tracking algorithm. The second one contains utility functions that take care of loading the input data and writing the results. The final part contains the main tracking loop. Since this code is properly structured, all the adaptation work has to be done on this final part of the code. The initial tracking loop looks like:

```cpp
1   int main( int argc, char** argv) {
2
3     // Reading
4     std::vector<std::string> images = readsequence("images.txt");
5     cv::Rect initialization = readrectangle("region.txt");
6     std::vector<std::Rect> output;
7
8     NCCTracker tracker;
9
10    // Initializing the tracker with first image
11    cv::Mat image = cv::imread(images[0]);
12    tracker.init(image, initialization);
13    // Adding first output to maintain equal length of outputs and
14    // input images.
15    output.push_back(initialization);
16
17    // The tracking loop, iterating through the rest of the sequence
18    for (size_t i = 1; i < images.size(); i++) {
19
```

(continued from previous page)

```
20      cv::Mat image = cv::imread(images[i]);
21      cv::Rect rect = tracker.update(image);
22      output.push_back(rect);
23
24    }
25
26    // Writing the tracking result and exiting
27    writerectangles("output.txt", output);
28
29 }
```

To modify the source code to use the TraX protocol we have to remove explicit loading of the initialization data and let the protocol take care of this.

```
1  int main( int argc, char** argv) {
2
3    NCCTracker tracker;
4
5    trax::Server handle(trax::Metadata(TRAX_REGION_RECTANGLE, TRAX_IMAGE_PATH |
6                             TRAX_IMAGE_MEMORY | TRAX_IMAGE_BUFFER), trax_no_log);
7
8    while (true) {
9
10      trax::Image image;
11      trax::Region region;
12      trax::Properties properties;
13
14      int tr = handle.wait(image, region, properties);
15      if (tr == TRAX_INITIALIZE) {
16
17        tracker.init(trax::image_to_mat(image), trax::region_to_rect(region));
18        handle.reply(region, trax::Properties());
19
20      } else if (tr == TRAX_FRAME) {
21
22        cv::Rect result = tracker.update(image_to_mat(image));
23        handle.reply(trax::rect_to_region(result), trax::Properties());
24
25      }
26      else break;
27    }
28
29 }
```

Lets now look at individual modifications. The creation *handle* object provides a protocol server handle that initializes the protocol and sends the introductory message. The handle is given protocol configuration structure that specifies what kind of data the server can handle as well as the optional output log that can be used for debugging:

```
trax::Server handle(trax::Configuration(TRAX_IMAGE_PATH | TRAX_IMAGE_MEMORY |
                    TRAX_IMAGE_BUFFER, TRAX_REGION_RECTANGLE), trax_no_log);
```

When using OpenCV support library's function *trax::image_to_mat()*, the conversion from file path, raw memory and image buffer types happens automatically so supporting them all is really easy. Without this function you have to convert the image yourself.

The tracking loop has been modified to accept commands from the client. This happens with the call to the *trax::Server::wait()* function. The function populates the provided variables: new image, object state (on

initialization), and optional parameters.

Since this kind of client-guided session means that the server does not know in advance how long will the tracking session be, the loop is only broken when a quit message is received from the client. The other two options are the initialization and new frame (which can always only follow successful initialization). The re-initialization can happen at any time throughout the session so the server should be capable of reinitializing the tracker (note that in untrusted setups the client may also terminate the session and start a new one).

Requests for initialization or update must be answered with a state message generated by *trax::Server::reply()*. This function accepts the object state as predicted by the tracker as well as any additional parameters that can be accumulated by the client for development or debugging purposes.

Because all the results are processed and stored by the client, we can remove the explicit results storage at the end of the loop. All these modifications also make all the utility functions in from the initial tracker implementation (second part of the source code) obsolete, they can be removed as their function is handled by the client.

Step 3: Compiling the project

Finally we will modify the example's CMake file. Properly installed TraX library supports CMake discovery mechanism, the only line that we have to add is therefore:

```
FIND_PACKAGE(trax REQUIRED COMPONENTS core opencv)
TARGET_LINK_LIBRARIES(ncc_tracker ${TRAX_LIBRARIES})
INCLUDE_DIRECTORIES(AFTER ${TRAX_INCLUDE_DIRS})
LINK_DIRECTORIES(AFTER ${TRAX_LIBRARY_DIRS})
```

The project is compiled as a standard CMake project. First, we open CMake GUI tool, select the source and build directories as shown in the example below.

During the configuration phase we select the appropriate generator. In case CMake is unable to find TraX library automatically we have to set the *trax_DIR* variable manually (set the variable to the *share* subdirectory of the local TraX installation root since this is the directory where the CMake configuration files are stored). When there are no more configuration errors we can generate the Visual Studio solution in the build folder open it and compile the project.

Step 4: Testing integration

A successful build results in a binary program (in this tutorial the program is called *ncc_tracker.exe* and is available in the *Debug* or *Release* subdirectory of your build directory). To test if the program correctly supports TraX protocol we can use the client *traxtest.exe* provided by the client support module of the project. This program tries to run the tracker on a sequence of static images to see if the protocol is correctly supported. Note that this test does not discover all the logical problems of the implementation as they may only occur during very specific conditions; it only tests the basic TraX compliance.

To run the test move to the build directory and type:

```
$ traxtest.exe -d -- ncc_tracker.exe
```

If the integration is successful this command should output something like:

```
CLIENT: Starting process  "ncc_tracker.exe"
CLIENT: Setting up TraX with standard streams connection
@@TRAX:hello "trax.image=path;memory;buffer;" "trax.region=rectangle;" "trax.version=1
↪"
CLIENT: Tracker process ID: 13019
CLIENT: Connection with tracker established.
```
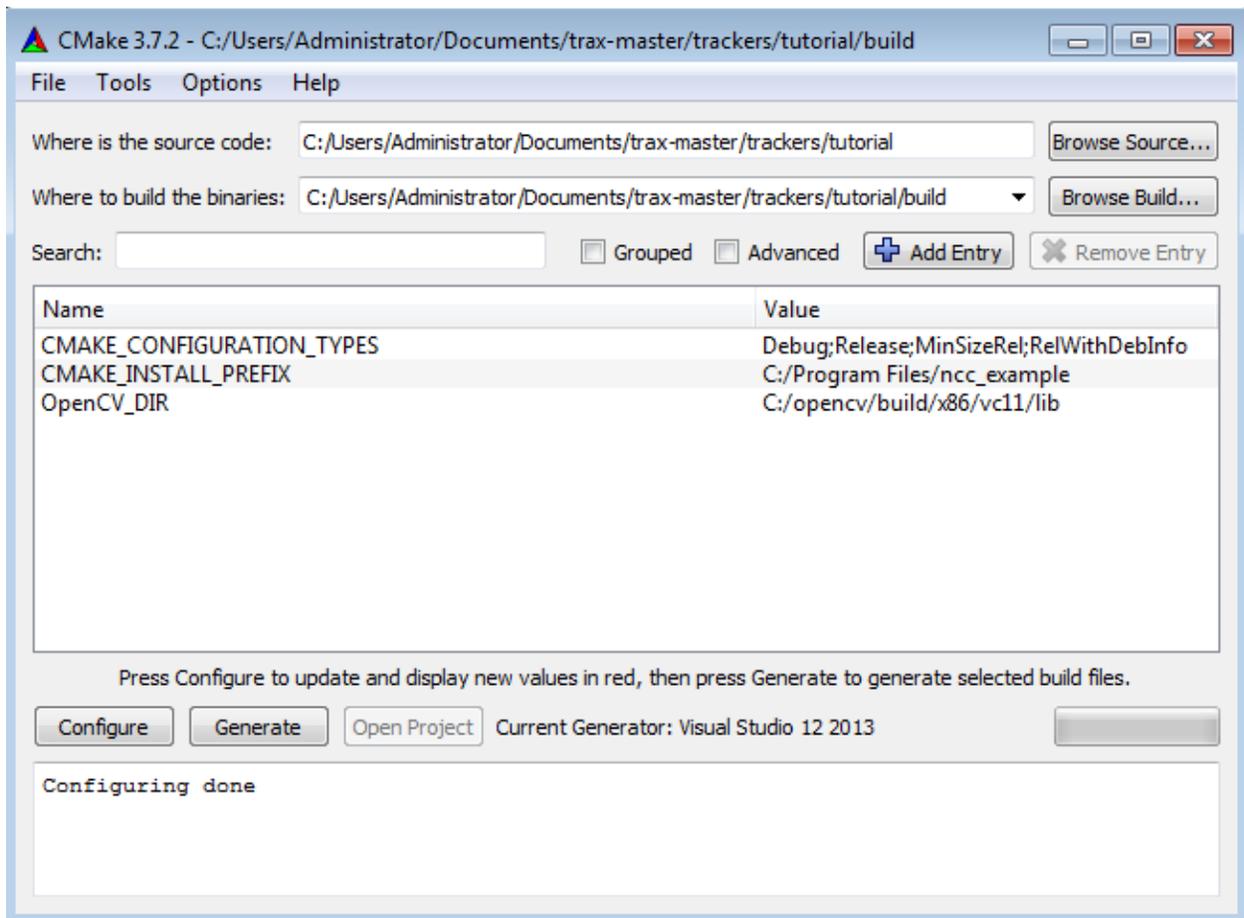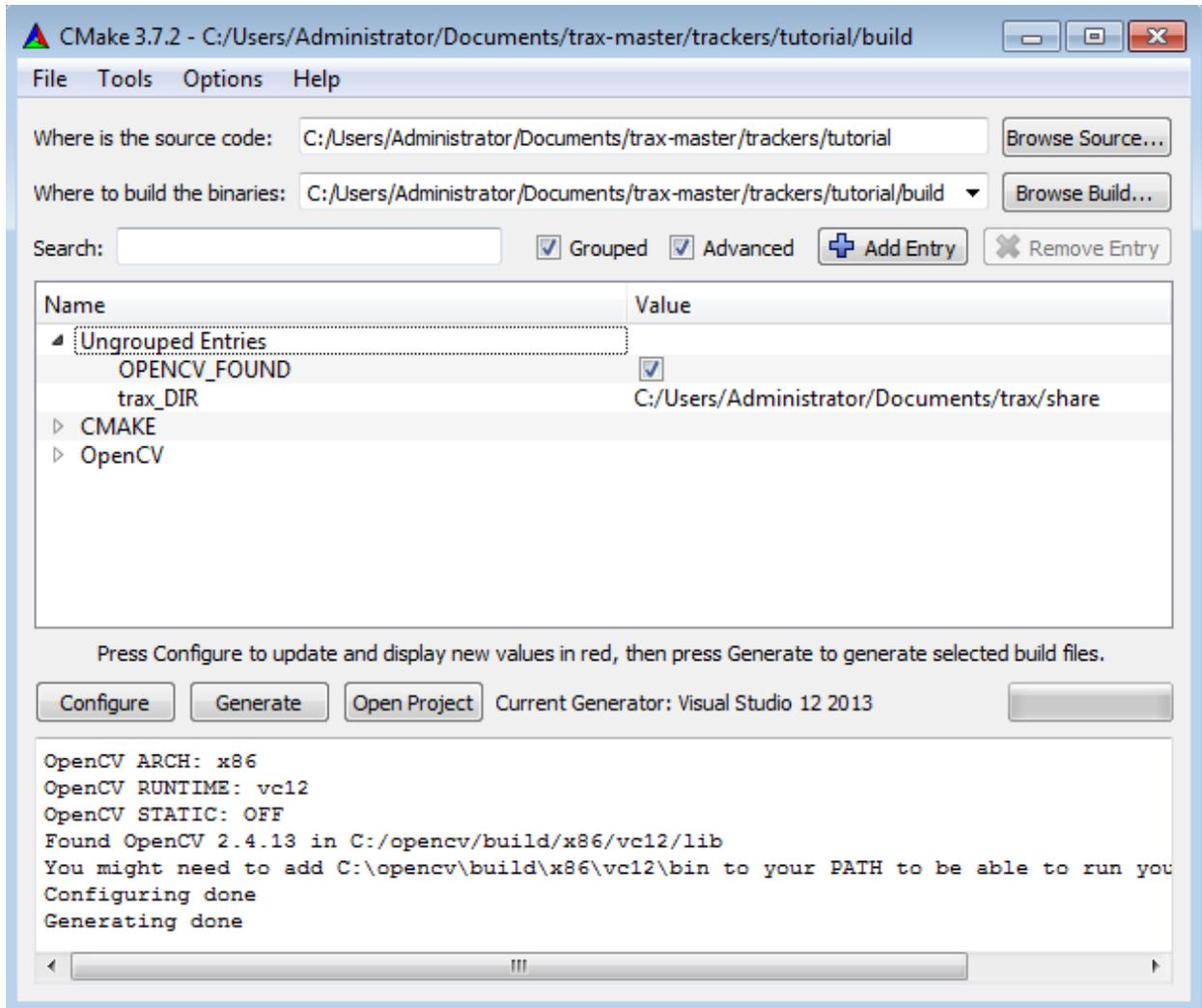
(continues on next page)

```
@@TRAX:initialize "data:image/...
Tracker initialized
@@TRAX:state "130.0000,80.0000,70.0000,110.0000"
@@TRAX:frame "data:image/...
Tracker updated
@@TRAX:state "130.0000,80.0000,70.0000,110.0000"
@@TRAX:frame "data:image/@@TRAX:frame "data:image/...
Tracker updated
@@TRAX:state "130.0000,80.0000,70.0000,110.0000"
@@TRAX:frame "data:image/...
Tracker updated
...
...
```

2.4.5 TraX in Matlab and Octave

This tutorial will guide you through integration of support for TraX protocol to your Matlab/Octave tracker using the reference library. It is assumed that you already have TraX *compiled* or installed on your computer and that you have read the *general remarks on tracker structure*.

Step 1: Enabling TraX support

To enable TraX support you have to compile the *traxserver* MEX file that provides a bridge between the library and scripting environment.

Note: On Ubuntu systems you can install a pre-compiled Octave package named *octave-trax* from the PPA repository. Then all you have to do is type *pkg load trax* in Octave console to get access to *traxserver* function.

The source files for *traxserver* are located in *support/matlab* subdirectory of the source code repository. In the same directory there is also an utility script, *compile_trax.m*, that compiles the source using the available MEX compiler (If you have never compiled MEX files before you will probably have to configure the compiler first.).

If you have compiled the TraX library yourself then all you have to do is to run the utility script. If you are using a pre-built version then you will have to pass the script the path to the root directory where TraX libraries are located:

```
compile_trax('< ... path to TraX ... >');
```

Step 2: Preparing your project

In this tutorial we will use a simple NCC tracker written in Matlab/Octave language. The original source code for the tracker is available in the project repository in directory *docs/tutorials/matlab*. All the code of the tracker is contained in a single file called *ncc_tracker.m*. In the main function we can observe a typical tracking loop. The data is loaded from the files, image list from *images.txt* and initialization region from *region.txt*. The result is stored to *output.txt*.

```
1    % Load initalization data
2    fid = fopen('images.txt','r');
3    images = textscan(fid, '%s', 'delimiter', '\n');
4    fclose(fid); images = images{1};
5    region = dlmread('region.txt');
6
```

```matlab
7        % Initialize the tracker
8        [state, ~] = ncc_initialize(imread(images{1}), region);
9
10       output = zeros(length(images), 4);
11       output(1, :) = region;
12
13       % Iterate through images
14       for i = 2:length(images)
15
16               % Perform a tracking step, obtain new region
17               [state, region] = ncc_update(state, imread(images{i}));
18               output(i, :) = region;
19
20       end;
21
22       % Write the result
23       csvwrite('output.txt', output);
```

To modify the source code to use the TraX protocol we have to remove explicit loading of the initialization data and let the protocol take care of this. Then we modify the tracking loop to wait for client requests and respond to them.

```matlab
1        traxserver('setup', 'rectangle', {'path'});
2
3        while true
4
5                [image, region] = traxserver('wait');
6
7                if isempty(image)
8                        break;
9                end;
10
11               if ~isempty(region)
12                       [state, ~] = ncc_initialize(imread(image), region);
13               else
14                       [state, region] = ncc_update(state, imread(image));
15               end
16
17               traxserver('status', region);
18       end
19
20       traxserver('quit');
```

All interactions with TraX protocol are done using the `traxserver` function. The function is first called at line 1 to setup the protocol, then at line 5 wait for instructions. The request type is specified by the first two output arguments (image and region). If the first argument is empty this means that the client requested termination of session. If first and second arguments are not empty then tracker initialization is requested, if only the second argument is empty tracker update is requested. The tracker state is reported back to the client at line 17. Outside the tracking loop the function is called again at line 20 to terminate the tracking session.

It is also recommended that the main tracking function contains an onCleanup handle which triggers on exit from the method and ensures that the program closes. This is especially needed in Matlab since by default the interpreter returns to interactive mode at the end. This way we guarantee that the program will exit no matter the outcome (even in case of exceptions).

```matlab
cleanup = onCleanup(@() exit() );
```

Step 3: Testing integration

To test if the tracker correctly supports TraX protocol we can use the client *traxtest* provided by the client support module of the project. This program tries to run the tracker on a sequence of static images to see if the protocol is correctly supported and the tracker correctly responds to requests. Note that this test does not discover all the logical problems of the implementation as they may only occur during very specific conditions; it only tests the basic TraX compliance.

Matlab

To run the tracker using Matlab interpreter, Matlab has to be run in command line mode (no GUI). This is easy to achieve on Linux and partially on OSX, but a bit harder on Windows so we will address this operating system separately.

On Linux and OSX Matlab has to be run in headless mode to avoid redirecting input and output to GUI. We achieve this with flags `-nodesktop` and `-nosplash`. We tell Matlab what code to evaluate with `-r` flag. The inline script should call the main tracker file, in our case this is *ncc_tracker*. Additionally the script can also be used to add directories to search path list using the `addpath` command. The final call to *traxtest* therefore looks something like:

```
$ traxtest -d -- matlab -nodesktop -nosplash -r "ncc_tracker"
```

On Windows, the process is a bit trickier since Matlab forks a process and prevents communication over system streams. Because of this we have to switch to TCP stream by adding `-X` flag to *traxtest*. We also have to add flag `-wait` to Matlab command to tell Matlab not to quit the original process (otherwise the client will think that the tracker has terminated) as well as flag `-minimize` to minimize the command window that is opened by Matlab in any case. The final call to *traxtest* therefore looks something like:

```
$ traxtest -X -d -- matlab -nodesktop -nosplash -wait -minimize -r "ncc_tracker"
```

Octave

In contrast to previous versions Octave 4.0 has integrated GUI which has to be disabled to run the process in the background. We achieve this by passing `--no-gui` flag. Then, we tell Octave where to find tracker code using `-p` flag which adds path to Octave search path list. Finally we tell Octave what code to evaluate with `--eval` flag. The inline script should call the main tracker file, in our case this is *ncc_tracker*. Since packages have to be explicitly loaded in Octave this can also be done inline (unless you add these commands to the tracker script). Finally, we can also disable writing of octave-workspace file with `crash_dumps_octave_core` command. The final call to *traxtest* therefore looks something like:

```
$ traxtest -d -- octave --no-gui --eval "pkg load trax; pkg load image; crash_dumps_
→octave_core(0); ncc_tracker"
```

2.4.6 Creating a TraX client

According to TraX specification a client is an application that provides tracking task to the tracker, i.e. the server. The client typically has to start the server process, initialize the tracker, provide image sequence data and monitor the tracker's progress. Examples of fully functional clients in the TraX project are the CLI utility *traxclient* which can be used to perform simple experiments on a given sequence and *traxplayer* which can be used to interactively initialize the tracker and run it on the data from a video or even camera.

In this tutorial we will create a client that runs a tracker for 50 frames (the sequence is loaded from a given directory) and automatically reinitializes it after that. We will utilize the TraX client support library that takes care of common

and platform specific tasks like managing the tracker process. The complete source code of the example is available in *docs/tutorials/client* directory of the repository.

To include the client support we first have to include its header file (besides all other header files):

```cpp
// System utilities
#include <sstream>

// TraX headers
#include <trax.h>
#include <trax/client.hpp>
```

We will begin by parsing the input arguments, provided by the user. For simplicity sake we will assume that the input sequence is defined by the *groundtruth.txt* file which defines the annotated groundtruth state and that the image files are also available in the same folder and follow a eight digit zero-padded number, starting with 1: *00000001.jpg, 00000002.jpg, . . .*. The only remaining input is the tracker command which we have to parse from the input arguments and encode it as a string:

```cpp
int main( int argc, char** argv) {

        // Quit if tracker command not given
        if (argc < 2)
                return -1;

        std::vector<trax::Region> groundtruth;
        trax::load_trajectory("groundtruth.txt", groundtruth);

        int total_frames = groundtruth.size();

        std::stringstream buffer;
        for (int i = 1; i < argc; i++)
                buffer << " \"" << std::string(argv[i]) << "\" ";
        std::string tracker_command = buffer.str();
```

To obtain tracker handle we create a *TrackerProcess* object. This object internally starts a new process using the given command parameter. The construction of the object will faill if the tracker will not send an introduction TraX message which will also provide client with the information about tracker's capabilites (what kind of image formats does it support). We can also check if the process is still alive by calling *ready* method.

```cpp
trax::TrackerProcess tracker(tracker_command);
```

For now we will leave these advanced options and look at the client tracking loop. The client is aware of the length of the sequence so the loop iterates for a known number of steps. At each step we generate a path image descriptor to the new image. The client has complete control over what is going on which we can use to re-initialize the tracker every 50 frames.

```cpp
for (int i = 0; i < total_frames; i++) {

        char buff[32];
        snprintf(buff, sizeof(buff), "%08d.jpg", i + 1);
        trax::Image image = trax::Image::create_path(string(buff));

        // Initialize every 50 frames
        if (i % 50 == 0) {
                if (!tracker.initialize(image, groundtruth[i]))
                        // Something went wrong, break the loop
                        break;
```

```
        } else {
                if (!tracker.frame(image))
                        // Something went wrong, break the loop
                        break;
        }
```

After issuing the command we wait for the reply. Unless the tracker terminates execution, a valid reply will contain the status of the object as well as optional properties. All of this information can be simply stored or used to adjust tracking session.

```
                trax::Region status; // Will receive object state predicted by tracker
                trax::Properties properties; // Will receive any optional data

                // Wait for tracker response to the request
                bool success = tracker.wait(status, properties);

                if (success) {
                        // The tracker returns a valid status.
                        // Process the result ...
                } else {
                        if (tracker.ready()) {
                                // The tracker has requested termination.
                                break;
                        } else {
                                // In case of an error ...
                        }
                }

        }

}
```

And all we have to do now is to compile the code. We will use CMake to find the TraX library and configure the build. Notice that we have to tell CMake to include ´core´ and ´client´ TraX components:

```
ADD_EXECUTABLE(sample_client client.cpp)
FIND_PACKAGE(trax REQUIRED COMPONENTS core client)
TARGET_LINK_LIBRARIES(sample_client ${TRAX_LIBRARIES})
INCLUDE_DIRECTORIES(AFTER ${TRAX_INCLUDE_DIRS})
LINK_DIRECTORIES(AFTER ${TRAX_LIBRARY_DIRS})
```

2.5 Contributions and development

Contributions to the TraX protocol and library are welcome, the preferred way to do it is by submitting issues or pull requests on GitHub.

2.6 Protocol adoption

On this site we list public projects that have adopted the TraX protocol (either using the reference library or custom implementations). If you want to be on the list please write an email to the developers.

2.6.1 Trackers

- LGT and ANT: The repository contains Matlab implementations of LGT (TPAMI 2013), ANT (WACV 2016), IVT (IJCV 2007), MEEM (ECCV 2014), and L1-APG (CVPR 2012) trackers.

- KCF: A C++ re-implementation of the KCF (TPAMI 2015) tracker.

- ASMS: A C++ implementation of the ASMS (PRL 2014) tracker.

- MIL: An adapted original C++ implementation of the MIL (CVPR 2009) that works with OpenCV 2.

- Struck: A fork of the original Struck (ICCV 2011) implementation with protocol support and support OpenCV 2.

- CMT: A fork of the original CMT (CVPR 2015) implementation.

2.6.2 Applications

- VOT toolkit: the toolkit uses the TraX protocol as the default integration mechanism.