TraX Documentation

Release 1.0

Luka Cehovin

January 19, 2017

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.

CHAPTER 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.

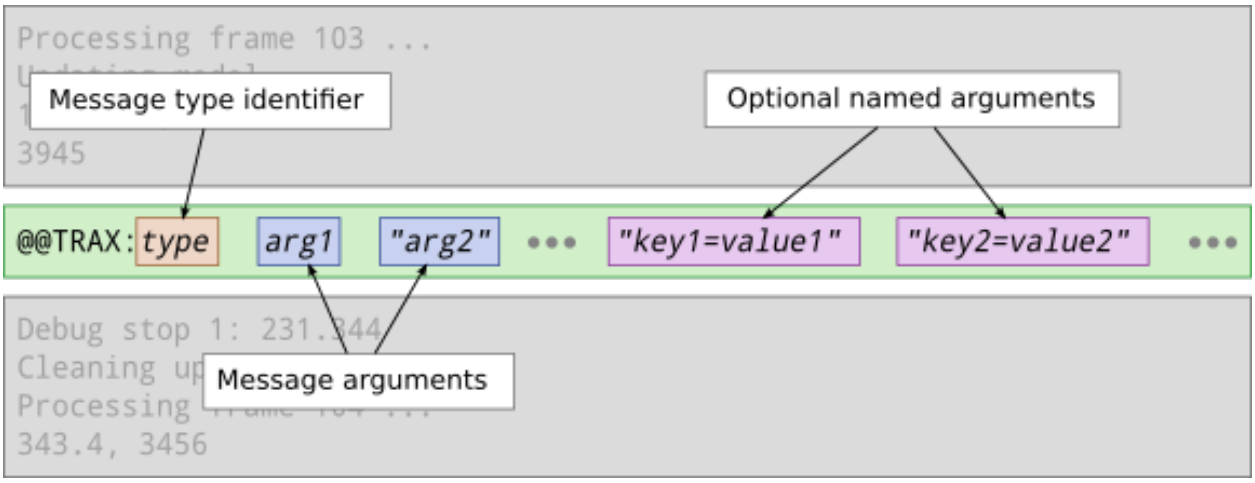
1.1 Definitions

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*.

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.



1.2.1 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.

1.2.2 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.

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 *Region 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 *Image formats* for the list of supported formats. By default it is assumed that the tracker can accept rectangles as region specification.
- `initialize` (client): This message is sent by the client to initialize the tracker. The message contains the image data 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. 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.

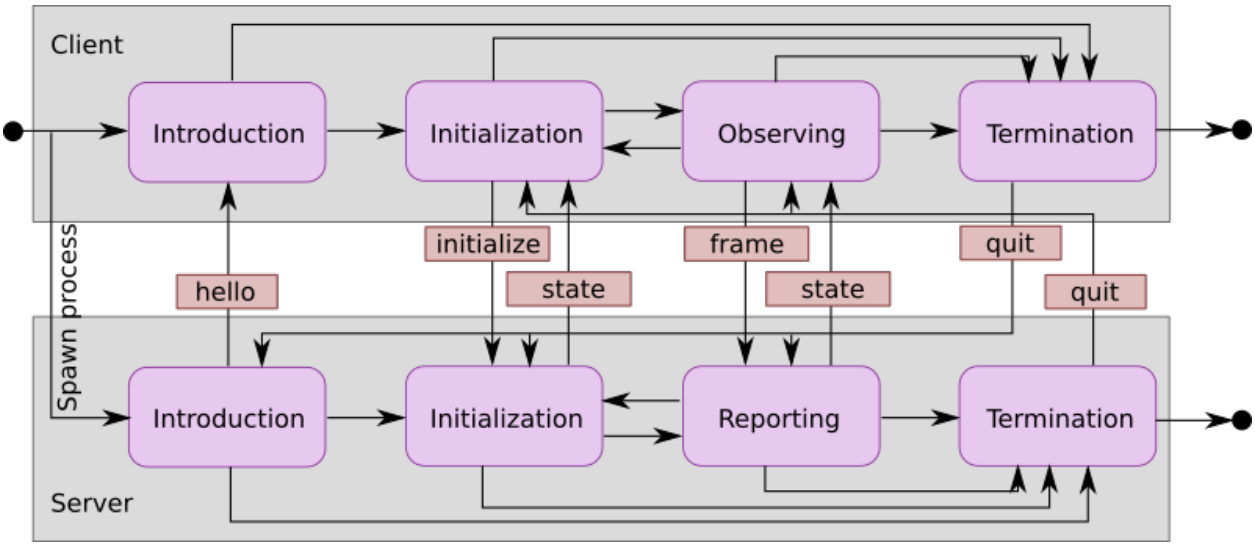
The server state automata consists of the following states:

- **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.

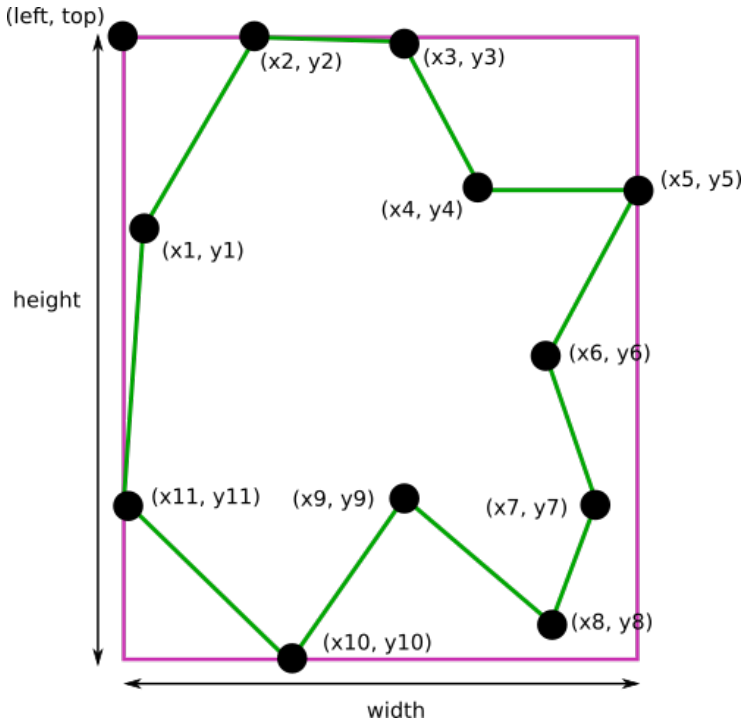
1.4 Region formats

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).



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`.

CHAPTER 2

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.

2.1 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.

2.2 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.

2.2.1 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_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_configuration *config*, *trax_logging* *log*)

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

Parameters

- **config** – Configuration structure
- **log** – Logging structure

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

*trax_handle** **trax_server_setup_file**(trax_configuration *config*, 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

- **config** – Configuration structure
- **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_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*

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.

2.2.2 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

2.2.3 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_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

2.2.4 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

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

2.3 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.

```
#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.

```
#include <stdio.h>

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

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

    // Call trax_server_setup at the beginning
    trax_handle* handle;
    trax_configuration config;
    config.format_region = TRAX_REGION_RECTANGLE;
    config.format_image = TRAX_IMAGE_PATH;

    handle = trax_server_setup(config, trax_no_log);

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

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

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

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

            rectangle_type region = update_tracker(load_image(img));
            trax_server_reply(handle, rectangle_to_region(region), NULL);

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

        if (img) trax_image_release(&img);
        if (reg) trax_region_release(&reg);

```

```
48      }
49
50      // TraX: Call trax_cleanup at the end
51      trax_cleanup(&handle);
52
53      return 0;
54  }
```

CHAPTER 3

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.

3.1 Requirements and building

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

3.2 Documentation

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

class `Configuration`
A wrapper class for

`Configuration`(trax_configuration *config*)

`Configuration`(int *image_formats*, int *region_formats*)

`~Configuration`()

class `Logging`
A wrapper class for

`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 `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.

const *Configuration* `configuration` ()

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.

const *Configuration* `configuration` ()

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.

~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.

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.

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 values in the properties object into the given dictionary.

3.3 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.

```
#include <iostream>
#include <fstream>

using namescpace std;

int main( int argc, char** argv)
{
    int i;
    FILE* out;
    Rectangle region;
    Image image;
    Tracker tracker;

    ofstream out;
    output.open("trajectory.txt", ofstream::out);

    region = read_bounding_box();
    image = read_image(1);
    region = tracker.initialize(region, image);

    out << region << endl;

    for (i = 2; ; i++)
    {
      image = read_image(i);
      region = tracker.update(image);
      out << region << endl;
    }

    out.close();
    return 0;
}
```

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.

```
#include <stdio.h>

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

using namespace std;

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

```
    int run = 1;
    trax_image* img = NULL;
    trax_region* reg = NULL;

    // Initialize protocol
    trax::Server handle(trax::Configuration(TRAX_IMAGE_PATH,
          TRAX_REGION_RECTANGLE), 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) {

          rectangle_type region = tracker.initialize(
               region_to_rectangle(region), load_image(image));

          handle.reply(rectangle_to_region(region), trax::Properties());

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

          rectangle_type region = tracker.update(load_image(image));
          handle.reply(rectangle_to_region(region), trax::Properties());

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

    }

    return 0;
}
```

CHAPTER 4

Python implementation

4.1 Requirements and support

4.2 Documentation

4.2.1 Main module

Implementation of the TraX protocol. The current implementation is written in pure Python and is therefore a bit slow.

class `trax.`**`MessageType`**
The message type container class

exception `trax.`**`TraXError`**
A protocol error class

4.2.2 Server-side communication

Implementation of the TraX sever. This module provides implementation of the server side of the protocol and is therefore meant to be used in the tracker.

class `trax.server.`**`Request`**
A container class for client requests. Contains fileds type, image, region and parameters.

class `trax.server.`**`Server`**(*options*, *verbose=False*)
TraX server implementation class.

`quit`()
Sends quit message and end terminates communication.

`status`(*region*, *properties=None*)
Reply to client with a status region and optional properties.

Parameters

- **`region`** (`trax.region.Region`) – Resulting region object.
- **`properties`** (*`dict`*) – Optional arguments as a dictionary.

`wait`()
Wait for client message request. Recognize it and parse them when received .

Returns A request structure

Return type *trax.server.Request*

class trax.server.**ServerOptions**(*region*, *image*, *name=None*, *identifier=None*)
TraX server options

4.2.3 Image module

Image description classes.

trax.image.**BUFFER = 'buffer'**
Constant for encoded memory buffer image

class trax.image.**BufferImage**(*data=None*, *format='unknown'*)
Image encoded in a memory buffer stored in JPEG or PNG file format

class trax.image.**FileImage**(*path=None*)
Image saved in a local file

Variables **path** – Path to the image file

class trax.image.**Image**(*type*)
Base class for all image containers

Variables **type** – Type constant for the image

trax.image.**MEMORY = 'memory'**
Constant for raw memory image

class trax.image.**MemoryImage**(*image*)
Image saved in memory as a numpy array

trax.image.**PATH = 'path'**
Constant for file path image

trax.image.**URL = 'url'**
Constant for remote or local URL image

class trax.image.**URLImage**(*url=None*)
Image saved in a local or remote resource

Variables **url** – URL of the image

trax.image.**parse**(*string*)
Parses string image representation to one of the containers

4.2.4 Region module

Region description classes.

trax.region.**POLYGON = 'polygon'**
Constant for polygon region type

class trax.region.**Polygon**(*points*)
Polygon region

Variables

- **points** (*list*) – List of points as tuples [(x1,y1), (x2,y2),...,(xN,yN)]
- **count** (*int*) – number of points

trax.region.**RECTANGLE = 'rectangle'**
Constant for rectangle region type

class trax.region.**Rectangle**(*x=0*, *y=0*, *width=0*, *height=0*)
Rectangle region

Variables

- **x** – top left x coord of the rectangle region
- **y** (*float*) – top left y coord of the rectangle region
- **w** (*float*) – width of the rectangle region
- **h** (*float*) – height of the rectangle region

class trax.region.**Region**(*type*)
Base class for all region containers

Variables **type** – type of the region

trax.region.**SPECIAL = 'special'**
Constant for special region type

class trax.region.**Special**(*code*)
Special region

Variables **code** – Code value

trax.region.**convert**(*region*, *to*)
Perform conversion from one region type to another (if possible).

Parameters

- **region** (Region) – original region
- **to** (*str*) – type of desired result

Result converter region or None if conversion is not possible

4.3 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.

```
tracker = Tracker()
trajectory = [];
i = 1

rectangle = read_bounding_box()
image = read_image(i)
rectangle = tracker.initialize(rectangle, image)

trajectory.append(rectangle)

while True:
  i = i + 1
  image = read_image(i)
  rectangle = tracker.update(image)
  trajectory.append(rectangle)

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.

```
import trax.server
import trax.region
import trax.image
import time

tracker = Tracker()

options = trax.server.ServerOptions(trax.region.RECTANGLE, trax.image.PATH)

with trax.server.Server(options) as server:
  while True:
    request = server.wait()
    if request.type in ["quit", "error"]:
      break
    if request.type == "initialize":
      rectangle = tracker.initialize(get_rectangle(request.region),
            load_image(request.image))
    else:
      rectangle = tracker.update(load_image(request.image))

    server.status(get_region(rectangle))
```

CHAPTER 5

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.

5.1 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 correcty you will have a MEX script (exension varies from platform to platform) available in the directory. If the location of the TraX library is not found automatically, you have to verfy that it exists and possibly enter the path to its location manually.

5.2 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.

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 hepls 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.

5.2.1 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.

5.2.2 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.

5.2.3 Internals

Additionaly 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 sockers and that a TCP server is opened (the port or IP address and port are proviede 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.

5.3 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.

```
    % Initialize the tracker
    region = read_bounding_box('init.txt');
    image = imread('0001.jpg');
    region = initialize_tracker(region, image);

    result = {region};
    i = 2;

    while true
            % End-of-sequence criteria
            if ~exist(sprintf('%04d.jpg', i), 'file')
                    break;
            end;
            i = i + 1;

            % Read the next image.
            image = imread(sprintf('%04d.jpg', i));

            % Run the update step
            region = update_tracker(image);

            % Save the region
            result{end+1} = region;
    end

    % Save the result
    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

```
% Initialize the protocol
traxserver('setup', 'rectangle', 'path');

    while true
    % Wait for data
    [image, region] = traxserver('wait');

    % Stopping criteria
    if isempty(image)
            break;
    end;

    % We are reading a given path
    mat_image = imread(image);

    if ~isempty(region)
            % Initialize tracker
        region = initialize_tracker(region, mat_image);
    else
            region = update_tracker(mat_image);
    end

            % Report back result to advance to next frame
    traxserver('status', region);

end
```

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

CHAPTER 6

Support modules

Besides protocol implementation the repository also contains supporting libraries that help with some frequently needed functionalities.

6.1 Client utilities

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. To compile the module you have to enable `BUILD_CLIENT` flag in CMake build system.

6.1.1 CLI interface

Client support module also provides a simple CLI (command line interface) to the client that can be used for simple tracker execution and protocol testing. If the OpenCV support module (below) 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).

6.2 OpenCV conversions

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.

The module is automatically built if the OpenCV library is found on the system, additionally you can also enable it by turning on the `BUILD_OPENCV` flag in CMake build system (but you may have to set the OpenCV location manually in this case).

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

CHAPTER 7

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.

CHAPTER 8

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.

8.1 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.

8.2 Applications

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

t

B

C

F

I

M

P

Q

R

S

T

U

W