TraX: Visual Tracking eXchange Protocol

Luka Čehovin

April, 2014

Abstract

This report motivates the TraX communication protocol as well as specifies its first iteration. TraX protocol is a simple protocol that was designed to make automatic evaluation of visual tracking algorithms quick, easy and independent of the choice of programming language, availability of source code or even the target operating system. It integrates with existing tracker implementations with little additional work and enables communication with external evaluation tools in order to perform objective evaluation experiments. In addition to the protocol specification we provide a reference implementation of the protocol in several popular programming languages that makes the protocol even easier to use.

1 Introduction

One of the problems in visual tracking research is fragmentation of evaluation methodology. While authors of new tracking methods provide comparative experimental results for their methods and the state-of-the-art in the papers, the evaluation procedures and datasets differ from one papers to another. Besides that the results are usually trimmed down to some summarizing performance scores due to paper length limitations. This makes it difficult for other researchers to simply reuse these results in their own evaluation. One way to overcome this problem is to share the tracker implementation. In the past years researchers tend to provide a binary form or even a source code of their implementation more frequently as a supplementary material to their papers. However, while these resources are indeed valuable as they encourage other people to repeat the experiments or perform new tests, the process of preparing such a tracker for such evaluation is still time consuming. In case of binary versions it can be even impossible to properly run the tracker on arbitrary testing image sequence and obtain results that can be then compared with other trackers.

A common challenge a computer vision researcher faces when designing a new visual tracking algorithm is how to perform comparative experiments without spending too much time on technical details of reference trackers. In this report we propose a simple communication protocol that allows researchers to quickly set up a second-party code in an evaluation environment or enable their

own trackers to be integrated across a variety of different evaluation or visualization tools. The protocol is called TraX which stands for Visual Tracking eXchange protocol.

The rest of the report is organized as follows: Section 2 provides the basic foundations of the protocol. Section 3 describes the message structure structure in Section 4 defines protocol states and Section 5 defines the supported data formats. In Section 6 we provide tips for protocol implementation and integration and we draw concluding remarks in Section 7, where we also describe the future plans for the protocol.

2 Overview and definitions

The TraX protocol is designed with simplicity of integration in mind as well as flexibility that allows extensions and custom use-cases. The protocol is based on the a mechanism that all modern operating systems provide: standard input and output streams of a process. The main idea is that we embed 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.

First we define the basic terminology of the protocol:

1. **Server**: We adopt the standard client-server terminology when describing the interaction. 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.

2. **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 determined.

3. **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. Types of messages are defined in Section 4 and the exact structure of a message is defined in Section 3.

3 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 1. 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 that can be used to communicate additional data.

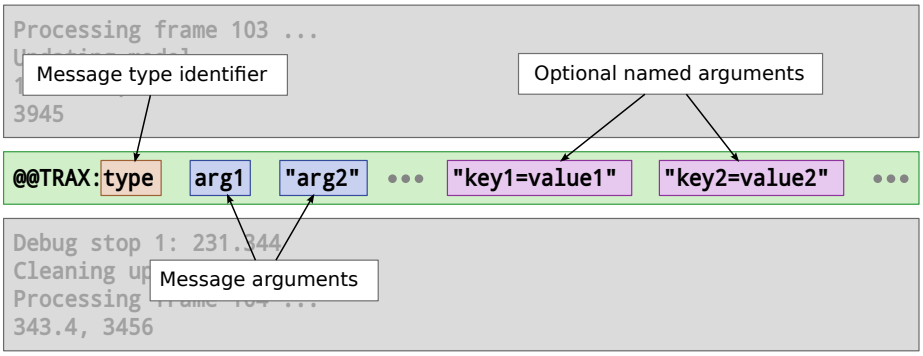

Figure 1: An illustration of a typical protocol message (green box) embedded within the process output stream (gray boxes).

All the arguments can contain spaces, however, they have to be enclosed by double-quote symbols. To use the same symbol inside the argument, it has to be prefixed by back-slash symbol. To use newline symbol inside the argument, it has to be replaced using `\n` symbol sequence.

4 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 5 for more details.
  - **trax.region** (string): Specifies the supported region format. See Section 5 for more details.

- `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 server. 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 2. The state changes upon receiving appropriate messages from the opposite party. The client state automata consists of the following states:

1. **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.

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

3. **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.

4. **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:

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

2. **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 *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.

3. **Reporting**: The server waits for the `frame`, `initialize`, or `quit` message. In case of `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.

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

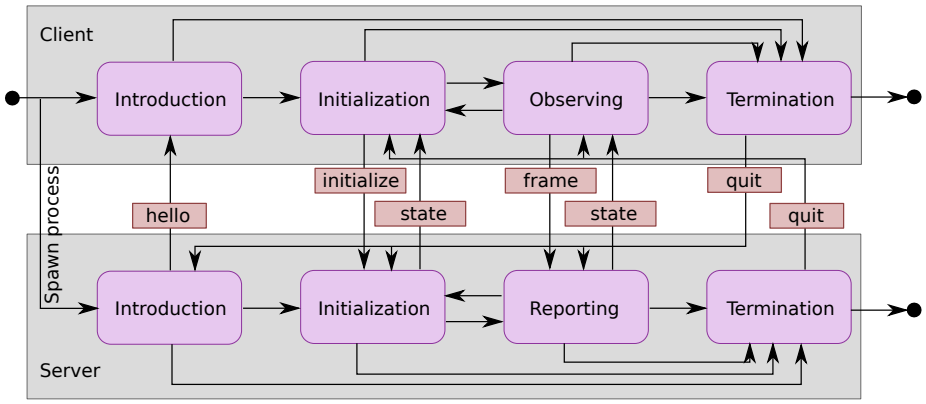

Figure 2: A graphical representation of client and server automata together with protocol states.

5 Data formats

The protocol is designed to be scalable, however, only a few basic data formats have been specified in this first iteration of the protocol. There are two region formats and one image format.

5.1 Region formats

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: 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).

Both region formats are shown graphically in Figure 3.

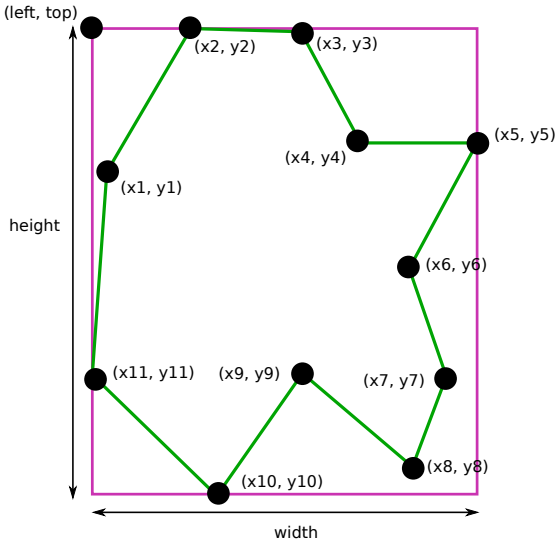

Figure 3: An illustration of rectangle and polygon region encoding.

5.2 Image format

Image path: Image is specified by 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.

6 Integration

To integrate the protocol into an existing tracker one has to identify the tracking loop in the algorithm and place the protocol handles to the appropriate locations. A sketch of integration in a pseudo-code tracker is shown in Figure 4.

A far better solution than implementing the protocol yourself is to use an open-source reference implementation library, available at `https://github.com/lukacu/trax`. The core library is written in C programming language

```
Setup tracker
TraX: Initialize protocol, report introduction message
loop
  TraX: Wait for message from client
  if initialize message then
    Initialize tracker with provided region and image
    TraX: Report state message
  else if frame message then
    Update tracker with provided image
    TraX: Report state message
  else if quit message then
    Break the tracking loop
  end if
end loop
Cleanup tracker
TraX: Cleanup protocol (terminate if necessary).
```

Figure 4: Pseudocode sketch of server protocol integration.

with bindings available for other languages. Using the library, the integration can be in most cases done very quickly.

7 Conclusion

In this report we have presented the first iteration of a visual tracking exchange protocol that attempts to standardize the communication between evaluation toolkits and tracker implementations. The idea of the tracker is to make the development of tracking algorithms easier by separating the core algorithm implementation from the auxiliary functionality like visualization and performance evaluation. The first publicly available evaluation solution that supports the TraX protocol is the Visual Object Tracking toolkit, which uses the protocol as the default integration technique with the VOT2014 challenge[1]. We hope that this will help promoting the interoperability and common integration platform which will benefit the research community in general.

[1]For more details check the VOT challenge website at `http://votchallenge.net`.