Robot-Simulator Documenation

https://github.com/revol-xut/robot-emulator

Tassilo Tanneberger

August 6, 2021

Contents

1 Introduction **1**

2 Recorder **2**
2.1 Record Entries . . . 2
2.2 Receiving . . . 2
2.3 Transmitting . . . 2
2.4 Using the template classes . . . 2
2.5 Conducting Recordings . . . 2
2.6 Limitations . . . 3

3 Project Structure **4**
3.1 Network . . . 4
3.2 Recorder . . . 5

4 Building **6**

5 Changelog **7**
5.1 Version: v0.1 . . . 7
5.2 Version: v0.2 . . . 7

1 Introduction

The main objective of this project is to automatically test industry robotics related programs. For this purpose the program is run in this simulated environment and it's actions are recorded. The resulting collected data is then compared to the expected behaviour and the correctness of the program is assessed. The RoboSimulator project also provides tools which can be used to simulated real-world sensors, cameras and any other peripheral devices which supplies data.

2 Recorder

The Recorder is one of the key parts of the environment because it is able to imitate every non responsive system. In code the a Record consist of multiple record entries which save the relative timestamp and the data package. It is important to note that the recorder plays the received packages time accurately back this means that time gaps or delays that are received are conserved and played back the same way.

2.1 Record Entries

The timestamp is the time since start of the recording in microseconds $10^{-6}/\mu$S it's saved as an `uint32_t`. The record entries are written after another in a single file because it is trivial to calculate and the size of an entry based on it's size therefore they are easily separable. Furthermore the ordering of the entries is relevant the record class as in version **v0.2** does not explicitly order the read elements based on the timestamp.

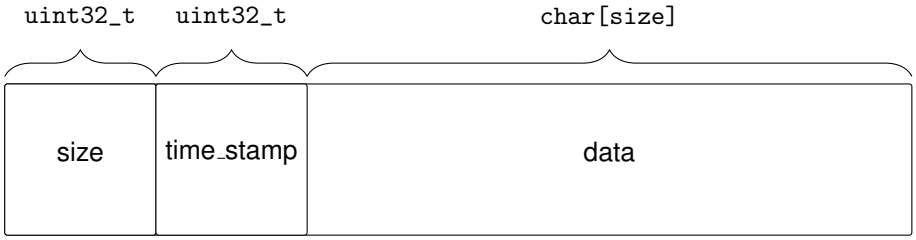


Figure 1: memory-layout of an written record entry

2.2 Receiving

The receiving process of the `Record`-class is split into two threads one receives the packages and appends them to a list and the other writes the packages into record entries. Thus all record entries received via the standard receive function have a data size smaller than the buffer size of the socket. Additionally the multi threaded approach increases throughput. Moreover it is not important for the `Record`-class which `DataSource` it gets see 3.1 for more information.

2.3 Transmitting

Transmitting is done by taking the current object or reading a file. This is accomplished in one `while`-loop where first the entry is read and then the program waits the necessary time this is done by the `std::chrono::sleep_until` method. This function takes the sum of the timestamp from the record entry and the beginning of the playback process. In order to send or playback recorder data the `Recorder`-class needs a writable class which implements the `DataSink` interface.

2.4 Using the template classes

For easy use the Robot-Simulator class comes with a set of prefabricated classes. Those are `RecordServer`, `RecordClient`, `PlaybackServer` and `PlaybackClient` it is obvious which purpose each class follows. Further more it is important to note that as in version **v0.2** there are only TCP implementations.

2.5 Conducting Recordings

There are many ways to record data I will present a collection of possible approaches depending on the circumstances. Given for example that the sensor or system in question is in sort of free firing mode meaning it sends repeatedly data to a given address a possible UDP-Server may look like this:

```
#include<robot-emulator/network/udp_socket.hpp>
#include<robot-emulator/recorder/recorder.hpp>

const std::string this_host = "0.0.0.0";
const unsigned short receive_port = 9008;
const std::string output_file = "my_record.rd";

// bind socket to defined address
UdpSocket socket{this_host, receive_port};

Recorder rec;
rec.receive(socket, output_file, 10s)
Recorder::convertToHumanReadible(output_file, "human_readible.txt")
```

All UDP-Sockets are bound this is because wild sockets (sockets which are unbounded) are occasional trouble makers. The presented piece of code will try to create the UDP-socket and then write down all the received date with the calculated timestamps in the `my_record.rd` file. Another method for recording would be to integrate the necessary classes into the production software and use it as a logger. This approach could yield the best results because it generates a broad band of data which newer version could be tested against.

Moreover the finally way presented is to use the mentioned templates.

```
#include<robot-emulator/network/public_server.hpp>
#include<robot-emulator/recorder/record_server.hpp>

const std::string this_host = "0.0.0.0";
const unsigned short receive_port = 9008;
const std::string output_file = "my_record.rd";

RecordServer record_server{host, port};
record_server.listenAndAccept(output_file);
```

The `RecordServer`-class will create for every new connection a new record file with the current timestamp and the source address.

2.6 Limitations

Currently (**v0.2**) there are no bidirectional or message forwarding recorders meaning the data source has to constantly send data.

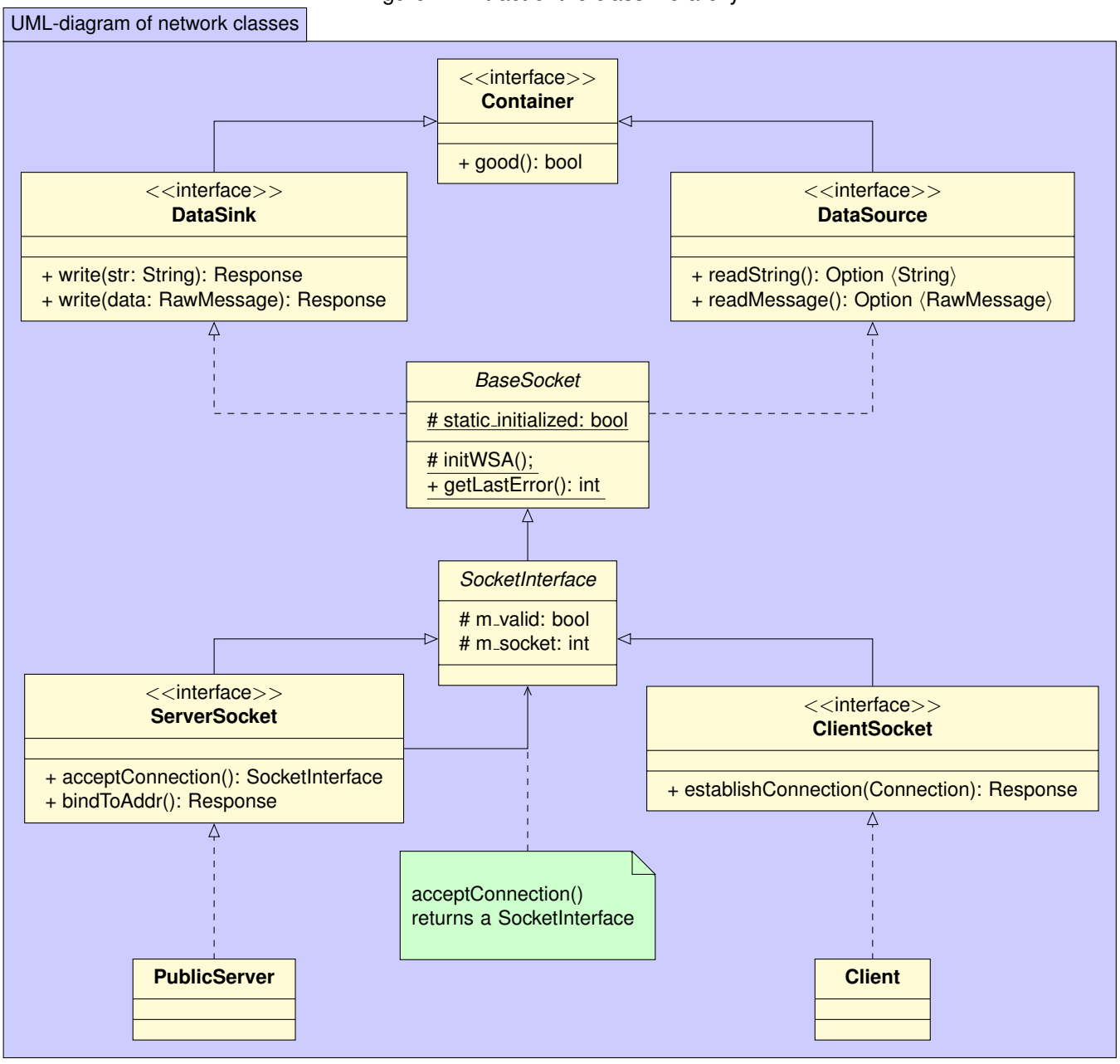
3 Project Structure

The project structure can be easily recognised by the folder hierarchy. The are three main sections `network`, `recorder` and `environment` this document will discuss them in the presented order.

3.1 Network

The network part of supports unix systems as well as the windows operating system. Furthermore it supports udp as well as tcp sockets. The following UML-diagram shows the class hierarchy for the tcp sockets.

Figure 2: Extract of the class hierarchy



The class hierarchy needs to be this steep for creating the necessary abstraction levels. Notably the first three classes (`Container`, `DataSink` and `DataSource`) are contained in the `type_trait.hpp` header and are not network headers. They are needed to clearly defined a SuSi (sources and sinks) software design pattern.

3.2 Recorder

In comparison the `Recorder`-class hierarchy is more simplistic. Thus the `Record`-class only depends on the `RecordEntry`, `DataSource` and `DataSink` classes/interfaces. The template classes mentioned in 2.4 obviously employ the various networking components.

4 Building

The projects works with the msvc or g++ compilers and uses the C++17 standard. Additionally the projects uses cmake for generating the build files and if in the right environment also the installation of dependencies. Under Linux this may look something like this:

```
$ mkdir build
$ cd build
$ cmake .. -DGFAI_BUILD=OFF
$ make
```

Clearly visible is the `GFAI_BUILD` flag which when turned on uses the build server of GFAI and pulls the dependencies from there when they are not installed.

5 Changelog

5.1 Version: v0.1

Finished: 28.6.2021

- TCP Socket implementation with WinSock inside the `SocketInterface`, `PublicServer` and `Client` classes.
- `Recorder`-Class

5.2 Version: v0.2

- Premade record class implementations for many use-cases. `RecordServer`, `RecordClient`, `PlaybackServer`, `PlaybackClient`
- Start-up class `Enviroment` reads the start config file and constructs the enviromnent accordingly
- UDP-Sockets
- BaseSocket for WSA-Init and operating system independent usage.

Finished: 6.8.2021