Programmer friendly communication framework

1st Lilo Zobl
Botball Team FrenchBakery
HTL Anichstraße
Innsbruck, Austria
lzobl@tsn.at

2nd Matteo Reiter
Botball Team FrenchBakery
HTL Anichstraße
Innsbruck, Austria
mareiter@tsn.at

***Abstract*—This document will focus on an important topic in software development: Network communication. It will provide an insight into why what network communication is needed for and how it is typically realized. Additionally it will cover problems and annoyances with currently available communication frameworks for a specific user-level applications as well as introduce concepts and a possible solution for the before-mentioned.**

I. Introduction

When it comes to computer science and robotics, sooner or later, multiple computers need to communicate with each other. In many cases this communication is achieved over a computer network. Nowadays this is typically implemented using the internet protocol (IP).

A. IP Networks

Without going into detail of how the Internet Protocol or any of it's underlying technologies work (out of the scope of this paper), this section provides a quick overview of IP.

In an IP network, every participating communication party (so called "host",such as a computer) is assigned an IP Address (either manually by the Administrator or using automated systems such as DHCP which will not be explained further). The IP address is the lowest level of addressing most user-level applications have to deal with. It is a 32-Bit number typically expressed by four octets (bytes) in dot notation: **192.168.1.1**

The IP address is split into two parts: the network address bits (some number of bits on the left of the address) and the host address bits (the rest of the bits on the right side). How many of the bits are part of the network and host area is variable and defined by a so-called netmask, which will not be discussed in more detail. All host with the same network-address-bits are considered to be on the same network and can therefore communicate with each other. Each host is uniquely identified in the network by the host-address-bits.

The Internet Protocol provides the foundations for transmitting data between hosts on this network (or possibly multiple networks): IP network packets. These contain Among other details, the sender and receiver addressing information and a block of data to be transmitted between the host. The operating system's network stack and other networking hardware then switch and route the packet from the sender to the desired destination host.

When developing applications however, the programmer almost never has to worry about the working of these underlying systems. Instead, they use higher level protocols build on top of IP, which abstract abstract IP packets and allow programmers to more conveniently achieve the desired functionality.

B. TCP and UDP protocols

There are two main protocols on top of IP for user applications - TCP and UDP.

TCP stands for **Transmission Control Protocol** and is a connection based protocol, meaning there will be a standing connection between two hosts. Before any data can be sent over TCP, a connection has to be established during an initial handshake between both communication parties. This is done automatically in the background. Once established, a TCP connection provides a pipe-like environment for transmitting a continuous stream of data bytes. The stream is of undefined length and data sent by one host is guaranteed to be received in the same order as sent by the other (without missing bytes in between). However, since TCP "streams" bytes, it does guarantee that any block of bytes send is received in one piece.

UDP stands for **User Datagram Protocol**. Unlike TCP, UDP does not require a connection to be established. The UDP merely sends a block of data (called the user datagram) all at once to a destination host in a single IP packet. There is nothing set in place to ensure that the packet reaches the destination. Without further protocols, UDP packets do not have any relation to each other, therefore there is no insurance of the order packets are received. Since the entire datagram is sent in one IP packet, it's maximum is restricted to the limits imposed by the latter.(65.535 bytes) (Achtung!!!!!!!!!!!!!!!!!!!! wenn man mehrere geräte hat verwendet man oft eigentlich UDP da das schneller geht)

UDP is less reliable than TCP. Before choosing a respective protocol one has to decide if they need speed or reliability more. A common uses for UDP are videos whereas a common use for TCP are websites.

From here on, this paper will focus on point-to-point communication between two hosts, although many of the concepts described later are also applicable to multicast communication. For this reason, it will focus on higher-level protocols and frameworks based on TCP from here-on out.

C. Communication frameworks

While TCP communication is the basis of many point-to-point networking applications requiring data integrity, it is still rather tedious for a programmer to implement in high-level applications. The programmer still has to know how to use the OS APIs to create and manage TCP sockets and since only bytes can be sent over the socket, the programmer needs to manually serialize and deserialize internal program structures and variables to transmit them. Besides this, raw TCP is only really suited for continuous data streams, while many applications require event-based bidirectional communication where messages (blocks of data with clear boundaries) are sent back and fourth.

Providing solutions for these annoyances/problems is the job of communication frameworks. A communication framework does not only implement a higher level protocol to support the additional functionality, but also provides libraries of programming language functions to integrate the functionality as seamlessly as possible with language features.

The work described in this paper aims to create a communication framework to solve problems for a rather specific set of requirements.:

1) Abstraction of low-level socket APIs, exposing the most important functionality with little code and automatically handle the rest in the background (according to common use-cases) including but not limited to:
   - Automatic management of connection state, handling disconnects and reconnects
2) Provide a way to exchange messages, which are blocks of data of a known (but still possibly dynamic) length, guaranteeing arrival in same grouping and order as sent
3) Provide facilities to serialize and deserialize language-native data structures to easily send them over the network with the least code possible
4) Provide data validation for sent and received messages according to programmer defined schemas (in language native format as far as possible), so the user code can trust received data to be in valid format.
5) Provide additional, commonly used communication schemes such as metadata exchange or Remote Procedure Calls (RPCs) that further simplify common use cases of network communication for the developer

II. State of the Art

There are many different network protocols and accompanying frameworks building on top of the above described, which solve different problems depending on the application requirements. Not all of them will be described in this paper.

(possible section "Motivation") The following section explains a typical protocol stack used for web apps, since one of the original primary use-cases of the work described in this paper is the communication between a single-page web-app and a high performance, low level server written in C++. This used to implement (among other uses) to implement the touchscreen UI to control FrenchBakery robots for the Botball competition. Since web browsers offer only limited to no access to system socket APIs, a large part of the protocol stack used is already defined.

That being said, the following is the protocol stack used as a starting point for the improvements described later.

A. Transmission Control Protocol (TCP)

TCP as the underlying transportation layer. In an end-user application, this would typically only be used directly if a very high performance custom application protocol is key.

B. Hyper Text Transfer Protocol (HTTP)

HTTP builds on top of TCP. This protocol is used to transmit files and other chunks of data (such as JSON documents) between servers or between servers and a browser. One limitation is that HTTP is (mostly) unidirectional communication. HTTP works by the client connecting to a server using TCP and then sending an **HTTP Request**. The request consists of at least one line of human readable text denoting the requested resource (like an HTML file) and some optional parameters, called headers. The end of the request is identified by a CRLF sequence (an empty line of text). Depending on the request details, the server may then answers with some status code, response headers and finally the requested data.

There are libraries for most programming languages, allowing the programmer to send and listen for HTTP request with very little code, varying depending on language.

```
// JavaScript
let resp = await fetch("http://10.5.5.5/");
```

By automatically handling TCP connection establishment and implementing a way to transfer connection metadata and status codes, HTTP already implements parts of point 1) and 5) of the before mentioned goals for a communication framework.

C. WebSockets

WebSockets are a bidirectional communication protocol building on top of and somewhat besides HTTP. A WebSocket works by first establishing a TCP connection and sending an HTTP request to a server. However instead of requesting or transmitting a resource directly, the client requests the server to change the protocol on top of the already open TCP socket to the WebSocket protocol using HTTP protocol upgrade headers. Since browsers don't allow web pages and web applications to create TCP sockets directly, this is the only way to create a long-term bidirectional communication channel between the browser and a server application.

The WebSocket protocol however doesn't expose the raw stream socket to the programmer, instead it introduced the concept of messages, hereby implementing point 2) of the goals for a communication framework. A message is a self contained chunk of data (typically a string, raw binary is

also possible) the transmission of which over the socket is transparent to the user. When the programmer sends messages, they guaranteed to arrive at the other communication party in one piece and in order in a message handler.

Like with HTTP, many languages provide simple APIs to create and use WebSocket connections.

```
// JavaScript
// connect
let socket = new WebSocket("ws://10.5.5.5/");
// send message
socket.send("hello");
// receive message
socket.onmessage = (e) => {
   console.log(e.data);
}
```

D. Socket.IO

Socket.IO again builds on top of WebSockets and is the last protocol in the here-described stack. Socket.IO works on the principle of sending events over the network. An event can be imagined like a message consisting of an **event name** and some **event data**. Events are identified by their name and there can be multiple different event names in an active connection. As soon as the connection between the server and client is established, both sides can **emit** and **listen** for events identified by their name. Listening works by registering callback (handler) functions, to be called when a specific event is received (i.e. emitted by the other communication party). When emitting an event, one can also send some additional data with it, which will be passed to the listener functions.

Thanks to support libraries for multiple programming languages, Socket.IO also provides the functionality to serialize and deserialize the data to language-native structures (at least for some languages) and therefore partially satisfies goal 3). All this functionality can be achieved with very little code.

```
// JavaScript client connect
const socket = io("ws://10.5.5.5");
// emit event
socket.emit("myevent", "some", [47, "data"]);
// listen for event
socket.on("event2", (arg1, arg2, ...) => {
   console.log("event2:", arg1, arg2);
})
```

Socket.IO also further implements the abstractions outlined by goal 1) of a communication framework, by implementing automatic connection state management and reconnects. Where with WebSockets, the programmer explicitly needs to write code to **connect to a server**, **detect disconnects**, and repeat the procedure, with Socket.IO, the programmer is merely required to tell the computer to **"be connected to a specified server"**. The library will then connect, detect failures and reconnect automatically to satisfy this request.

III. Problems with current offer

Although this protocol stack satisfies the most important goals listed, it still has some problems to be solved, namely the following:

1) The Events system provided by Socket.IO is useful for many applications, sometimes responses to events are required, which is often times named a **Remote Procedure Call**. This needs to be implemented manually for every event:
   - define and keep track of a separate request and response event
   - add an identifying element to request and response data to identify which response belongs to which request and prevent mixing up responses if multiple requests are sent out in short succession
   - keep a list of open requests to handle responses once received.

   Since the implementation is the same every time and is potentially needed many times in an application, it should be abstracted by the framework.
2) While Socket.IO provides a very clean and feature-complete API for some programming languages (like Python and JavaScript), it doesn't to so for others, like C++ which is only supported for clients. While there are libraries to create C++ WebSocket servers and clients (like websocket++ which is used in this paper), they require a lot of code (to some part due to the nature of the language) and tedious state management when used directly. Additionally, the event system would need to be implemented manually for every event every time which is tedious.
3) The data serialization and deserialization of Socket.IO is fully dynamic, meaning that before an event is actually processed, the code cannot possibly know the structure of the event data. This presents multiple problems:
   - Deserialization of event data to language-native data structures is impossible for statically typed languages like C++, which nullifies (bad wording?) the benefits of static typing. It requires the use of dynamic containers like maps to represent objects with strings as attribute names. Since the containers need to be fully dynamic, the programmer first needs to write code to check if an expected attribute exists, assert its's type and finally convert it to the representative language-native type.
   - Even in dynamic languages like JavaScript where objects can have dynamic attributes and variables are dynamically typed, it is still necessary or at least helpful to perform the same validation to provide adequate error handling and avoid unexpected behaviors. There are also additional systems (TypeScript for JS and type hints for Python) that provide the benefits of static typing even though the underlying language is dynamic.

An example is rich IntelliSense editor support to avoid typos in attribute names.

Since validation should always be performed (expect when explicitly specifying a type to be unknown), it should be abstracted by the communication framework. Instead of the programmer having to write the validation code, the programmer will define the structure of the expected value in a language native format as far as possible (structures in C++, Zod schemas in JS/TS). The framework is then responsible for performing the validation, so data reaching the user's code is guaranteed to have the expected structure.

[1] Proof for NordPass study, URL:https://tech.co/password-managers/how-many-passwords-average-person