A programmer friendly communication framework

1st Matteo Reiter
Botball Team FrenchBakery
HTL Anichstraße
Innsbruck, Austria
mareiter@tsn.at

2nd Lilo Zobl
Botball Team FrenchBakery
HTL Anichstraße
Innsbruck, Austria
lzobl@tsn.at

***Abstract*—This document will focus on an important topic in software development: Network communication. It will provide an insight into 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 specific user-level applications as well as introduce concepts and a possible solution for the before-mentioned problems.**

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 is achieved over a computer network. Nowadays this is typically implemented using the internet protocol (IP).

A. 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 data sent is guaranteed to be received in the same order (without missing bytes in between). However, since TCP "streams" bytes, it does guarantee that any segment (group of bytes) sent with one system call is received in one piece.

UDP stands for **User Datagram Protocol**. Unlike TCP, UDP does not require a connection to be established. UDP merely sends a segment of data (user datagrams) to a destination host within a single IP packet. There is no guarantee that the packet reaches the destination.

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

B. 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 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 with the following set of goals:

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, that received data will be in the expected format.
5) Provide additional, commonly used communication schemes and primitives such as metadata exchange or Remote Procedure Calls (RPCs). These further simplify common practices within network communication for the developer

II. State of the Art

There are many different network protocols and accompanying frameworks building on top of TCP, which solve different

problems depending on the application requirements. Not all of them will be described in this paper.

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 is used to implement (among other uses) the touchscreen UI to control FrenchBakery robots for the Botball competition. Since web browsers offer limited to no access to system socket APIs, a large part of the selected protocol stack is already defined by what browser APIs offer.

A. Transmission Control Protocol (TCP)

TCP is selected 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. HTTP works by the client connecting to a server using TCP and then sending an **HTTP Request**. 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 answer with some status code, response headers and finally the requested data. [1]

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 example request
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. [1]

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. [4]

The WebSocket protocol however doesn't expose the raw stream socket to the programmer, instead it introduces 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 are guaranteed to arrive at the other communication party in one piece and in order. [4]

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

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

[5]

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. [2]

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 stack

Although this protocol stack satisfies the most important goals listed, it still has the following problems:

1) While the event system provided by Socket.IO is useful for many applications, sometimes responses to events are required. This is typically named a **Remote Procedure Call** and 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.
2) While Socket.IO provides a very clean and feature-complete API for some programming languages (like Python and JavaScript), it doesn't do 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. [6]
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 negates 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, which is tedious.
   - Even in dynamic languages like JavaScript where objects can have dynamic attributes and variables are dynamically typed it is still helpful to perform the same validation in order 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 (such as rich IntelliSense) even though the underlying language is dynamic.

IV. Solution and Design Overview

To solve the aforementioned problems, and implement the last missing goals of data validation and more, the top-level protocol in the stack (Socket.IO) will be replaced by a new protocol with language support libraries which has been named **msglink**.

msglink will build on top of WebSockets, implementing all features of Socket.IO that are relevant to achieve the listed goals. In addition it will add direct framework support for more commonly used communication schemes/primitives.

It should be noted here, that msglink is a point-to-point communication protocol, like WebSocket and Socket.IO. Socket.IO provides the ability to send events from a server to multiple connected clients, which is possible to implement using msglink but is not a primary goal and will not be covered here.

These are the core principles of the msglink framework:

- Simple API
- Automatic state management
- Platform independence
- Data validation
- Strict type-safety
- Communication party equality
- Provision of common communication primitives
- Bandwidth efficiency

V. Implementation

This section will cover how the the core principles of the msglink framework work and how they solve the problems and implement the goals listed above.

A. Simple API

Similar to the protocols and accompanying libraries listed above (especially Socket.IO), msglink tries to achieve it's core functionality with the least code possible using a declarative API. Rather than writing code explicitly stating how to perform the serialization, validation or communication step-by-step (imperative programming), the developer writes code specifying (declaring) what result is expected (declarative programming). [7]

Like Socket.IO, to connect a msglink client to a server, one only has to specify what server the client should **be connected to** and the library will automatically handle connection and reconnection in case of connection loss.

```
// example msglink client in C++
auto client = new msglink::client<my_link>(
   "ws://10.5.5.5/"
);
```

B. Automatic state management

Like Socket.IO, msglink uses an additional heartbeat (ping-pong) system on top of the one provided by WebSockets to determine when the connection drops. This happens internally and requires no code from the user. It works by sending a small message back and fourth every few seconds. In msglink, the server is responsible for sending this so called "ping" message to the client. If the client doesn't respond with the corresponding "pong" message withing a certain time, the connection is dropped. Similarly, if the client doesn't receive a ping in a certain amount of time, it drops the connection on it's side and attempts to reconnect automatically.

C. Platform independence

msglink should be usable with different programming languages and on any operating system. All implementations should be able to communicate with each other.

D. Data validation and strict type-safety

A key feature of msglink is integrated data validation, along with other type-safety features (as far as the specific programming language allows).

Even though msglink is the top-level protocol in the newly defined protocol stack, the end-user application's communication itself also happens according to some protocol, defined by the application developer. The job of the communication framework (msglink) is to make the implementation and definition of this protocol as simple as possible. In msglink, this user-level protocol is called the **link**.

Unlike Socket.IO, msglink requires this link to be fully defined in a static manner, meaning the programmer has to explicitly state the existence and associated data structure of every communication primitive (e.g. an event, but msglink has others as well) before it can be used for communication.

In case of an event, that would mean explicitly stating that an event with a given name exists, whether it can be emitted or received and providing the schema for the data structure transmitted with the event (using language native format as far as possible such as structures in C++ or Zod schemas in JS/TS).

All of these definitions have to happen statically, so that (for applicable languages) the entire structure of the link (user protocol) is known at compile time. This gives the library the ability to automatically perform validation of the communication and serialization/deserialization of the data from/into language-native formats (even for statically typed languages like C++) without the developer having to manually perform any of those steps.

In msglink, both communication parties have to define the link from their perspective of the communication. This means for example, that an incoming event on one side has to have a matching outgoing event on the other side. When connecting, msglink performs a link-compatibility check, to make sure both parties have defined a compatible protocol. If the two links are not compatible, the connection is aborted before trying to sending any user data.

One of the goals for msglink is to keep the definition of this link entirely native to the respective language, so no extra build step is required to generate code from the definition. This has the slight disadvantage of possibly having to write the same schemas with two different syntaxes.

E. Communication party equality

In msglink, both communication parties (client and server) are treated equally. Apart from a few internal details, after a msglink connection is established, both parties can perform the exact same actions, in the bounds of what is defined by their link.

F. Common communication primitives

msglink defines three commonly used communication primitives that a programmer can use to create the user-protocol.

- **Events** work in the same way as they do with Socket.IO, with the exception of having to define them and their data structure statically.
- **Remote Function Calls (RFCs)** can be viewed as two events with opposite directions. One "call" event and one "result" event. RFCs work as the name implies. The called (receiving) side of the communication defines a function taking some input data (statically defined by schema) and returning some result data (also defined by a schema). Since a traditional function cannot yield multiple results, there can only be one such handler function. The other side of the communication can remotely call the function over the network, using asynchronous abilities of a language if available. This way, the remote function call can be used almost as seamlessly as a local one. The framework automatically handles the matching of results to pending calls so data is not mixed up.
  Other frameworks typically call this scheme "Remote Procedure Call" (RPCs), though "function" was chosen for msglink as it clarifies the fact that the call always yields some result data.
- **Data Sources** can be imagined as functions which can be called with some initial input data, that can follow up with more results over time until canceled. This might seem confusing at first, but it is useful for implementing **dynamic subscription models**. This is best explained using an example: *A web dashboard manages 100 servo actuators and needs to display live updates of their positions. Always transmitting the position of each results is too much traffic. Since only 5 can be displayed on screen at a time, a protocol needs to be implemented to activate or deactivate the updates for each when shown or hidden.* This could be implemented manually using a "sub", "unsub" and "update" event, providing the actuator ID in the "update" event.
  msglink abstracts and standardizes this further. In the UI component requiring the data of a specific actuator, a data source listener could be registered with the actuator ID as an input argument. This can be imagined as the function being called with the actuator ID as a parameter, and the function then following up with a new result event every time this specific actuator's position updates. There can also be multiple of those "functions" (possibly with different input values) active at the same time. Although the function analogy can be used as an illustration, it is important to understand that the initial "function call" when adding a subscriber with specific input parameters is merely a way to "enable" this data source and not an actual remote function call.

G. Bandwidth efficiency

At the time of writing, the msglink protocol uses JSON to encode it's internal messages as well as user data. This is not

very bandwidth efficient, especially when long attribute names are used, which is helpful for development purposes but may be improved in the future.

Nevertheless, msglink tries to perform the communication as efficiently as possible by only transmitting data over the network that is actually required. For example, the msglink framework internally tracks whether an event currently has any listeners and informs the other communication party if there are any. If there are no listeners, the event doesn't need to be transmitted over the network, even if one is emitted. A similar concept applies to data sources.

Data sources can be optimized further. If there are multiple data source "subscribers" with the same request parameters, the request doesn't need to be sent to the server multiple times, and there is no need to send value updates multiple times. The framework can detect when there is already an active data source with the same input parameters and simply use it's update messages until there are no subscribers anymore, at which point the source is disabled.

VI. Results and Applications

msglink is the result of solving very specific problems for a very specific use case. Nevertheless, since it uses widely supported web technologies as it's backbone, it can be easily integrated and expanded with whatever uses these technologies. Examples would be an HTTP reverse proxy to add SSL encryption.

The following is a compressed code example of how one might define a msglink link in C++, which is significantly shorter than implementing a comparable solution manually using WebSocket++. The explanation of the code example is beyond the scope of this paper.

```
// structure name is event name
struct my_event
   : public msglink::incoming_event {
   int a;
   std::string b;
   EL_MSGLINK_DEFINE_INCOMING_EVENT(
      my_event, a, b
   )
};
class my_link : public msglink::link {
   using link::link;
   EL_MSGLINK_LINK_VERSION(1);
   msglink::event_sub_hdl_ptr my_sub;
   void handle_my_event(in_event &_evt) {
      std::cout << _evt.a << _evt.b;
   }
public:
   virtual void define() noexcept override {
      my_sub = define_event<my_event>(
         &my_link::handle_my_event
      );
   }
};
```

Thanks to the rather general API and structure, msglink is useful for potentially many scenarios ranging from server-client communication in UI applications (web and local) to games to data exchange between servers to data exchange between robots in Botball.

VII. Limitations

Since msglink uses WebSockets as an underlying transport layer, a rather inefficient JSON encoding for serialization is an internally rather complex protocol to manage, speed is not one of its strengths. As mentioned in the protocol stackup, directly using a more lower-level protocol like TCP or UDP is required for high-performance applications, trading convenience for speed.

A further limitation is the point-to-point nature of the framework. There always needs to be a msglink client connected to a msglink server, although a server can simultaneously have connections to multiple clients and can therefore distribute messages between them. At this time, it is not planned to directly implement support for broadcast or multicast communication into msglink, even on the server side (which Socket.IO does support), since this was not part of the original goal. For applications where many hosts need to communicate with each other directly in a mesh-like topology, there are other protocols better suited for the purpose that don't have the limitation of the browser. An example is the **Data Distribution Serivce (DDS)** which is used by ROS2 (Robot Operating System 2) and is therefore common in robotics.

VIII. Conclusion

msglink satisfies all the goals defined in this paper for its specific use case and allows building new, innovative applications quicker without worrying about communication.

msglink is an open source project which can currently be viewed and contributed to as part of the **melektron/el-std** library on GitHub (branch "msglink_dev"). At the time of writing, msglink is still in the development phase and has not been fully implemented in multiple languages yet. The C++ library is the most developed at this time, with TypeScript and Python following very soon. [8]

References

[1] Information for HTTP, URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
[2] Information for Socket.io, URL: https://socket.io/
[3] Information for Zod, URL: https://zod.dev/
[4] Websocket standart, URL: https://www.rfc-editor.org/rfc/rfc6455.html
[5] Writing a websocket client, URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_client_applications
[6] Information for WebSocket++, URL: https://github.com/zaphoyd/websocketpp/
[7] Imperative vs. declarative, URL: https://programiz.pro/resources/imperative-vs-declarative-programming/
[8] el-std library for C++, URL: https://github.com/melektron/el-std