owulveryck's blog

- This is Home
- Archives
- Tags
- Categories
- About

Simplifying Complexity: The Journey from WebSockets to HTTP Streams

by Olivier Wulveryck
2023-12-02
tools

Introduction

To add a new functionality to my tool, goMarkableStream, I needed to capture gesture positions from the tablet's screen and relay them to the browser to trigger local actions. For example, a swipe left could activate a specific function in the browser.

My approach involved capturing gestures from the device and then communicating them to the browser with a message stating: "this gesture has been made."

In the realm of message exchange between a server and a client in a browser, WebSockets naturally come to mind. WebSockets are inherently designed to support streams of messages on top of TCP, unlike HTTP, which primarily handles streams of bytes without a built-in concept of a message.

Navigating through this journey, I realized the importance of extensive testing and learning to craft an effective solution. The WebSocket protocol, in contrast to HTTP, introduces distinct challenges, especially in debugging and testing, due to its more complex nature.

Acknowledging that gestures are essentially a stream of bytes (I will explain this), I will write about:

- the process of evaluating the trade-off between the added complexity of WebSockets and the functionalities they offer.
- how I streamlined the development by designing my own messaging system over HTTP.

Background

I've previously discussed using my tablet for various live presentations. Through iterative testing, I developed a hybrid solution that combines elements of a whiteboard with static slides. This solution features the screen's drawing as an overlay on existing slides. The challenge now lies in switching slides directly from the tablet to streamline the presentation and minimize interactions with the laptop displaying the slides.

The slides are displayed within an iFrame on the client side of my tool. Consequently, I needed a method to send commands to the iFrame to control slide transitions. The reveal.js presentation framework supports native embedding and allows slide control from the top frame via an API that uses postMessages.

To transmit slide control commands from the tablet to the client, I considered various methods. The optimal solution I identified was to utilize touch gestures on the screen of the reMarkable tablet. By swiping on the tablet, I could send events to the client, which would then respond accordingly to switch the slides.

Capturing the touch events on reMarkable/Linux

The reMarkable operates on a Linux-based system. Input events (both pen and touch) are managed through Event Devices (evdev). The event exposure is as follows:

- `/dev/input/event1` captures the pen events.
- `/dev/input/event2` captures touch events.

In Unix, the philosophy that "*everything is a file*" applies. This means I can easily access these events by opening and reading the file contents in Go. I chose Go for the server-side language due to its self-sufficient packaging, cross-compilation capabilities, and the enjoyment I derive from using it.

> "Everything is a file" is a principle in Unix and its derivatives, where input/output interactions with resources such as documents, hard-drives, modems, keyboards, printers, and some inter-process and network communications are treated as simple byte streams accessible through the filesystem namespace - source Wikipedia.

Reading the events in Go

The "file" event is a character device, offering a binary representation of an event. In Go, an event's set of bytes could be structured like this:

```go
type InputEvent struct {
	Time syscall.Timeval `json:"-"`
	Type uint16
	Code  uint16
	Value int32
}
```

The principle of "*everything is a file*" enables the use of basic operations from the `os` package to open the character device as an `*os.File` and `Read` the binary representation of the event. We create an `ev` object of the `InputEvent` type to receive the information read.

The file functions as an `io.Reader`, and its content is typically loaded into a byte array.

```go
func readEvent(inputDevice *os.File) (InputEvent, error) {
    // Size calculation:
    // Timeval consists of two int64 (16 bytes),
    // followed by uint16, uint16, and int32
    // (2+2+4 bytes)
    const size = 16 + 2 + 2 + 4
    eventBinary := make([]byte, size)

    _, err := inputDevice.Read(eventBinary)
    if err != nil {
        return InputEvent{}, err
    }

    var ev InputEvent
    // Assuming the binary data is in little-endian format
    // which is the most common on Intel and ARM
    ev.Time.Sec = int64(binary.LittleEndian.Uint64(eventBinary[0:8]))
    ev.Time.Usec = int64(binary.LittleEndian.Uint64(eventBinary[8:16]))
    ev.Type = binary.LittleEndian.Uint16(eventBinary[16:18])
    ev.Code = binary.LittleEndian.Uint16(eventBinary[18:20])
    ev.Value = int32(binary.LittleEndian.Uint32(eventBinary[20:24]))

    return ev, nil
}
```

A more efficient approach could involve using an unsafe pointer to directly populate the structure, thereby bypassing Go's safety mechanisms by using the `unsafe` package:

```go
func readEvent(inputDevice *os.File) (events.InputEvent, error) {
    var ev InputEvent
    // by using (*[24]byte), we are explicitly stating that
    // we want to treat the memory location of ev as a byte array of length 24
    // We could have used the less readable form:
    // (*(*[unsafe.Sizeof(ev)]byte)(unsafe.Pointer(&ev)))[:]
    //
    //  Note: the trailing [:] is mandatory to convert the array to a slice
    _, err := inputDevice.Read((*[24]byte)(unsafe.Pointer(&ev))[:])
    return ev, err
}
```

The Problem Statement

Now that I have read the events, I need to send to the client for further processing. The current architecture is based on an HTTP server in Go and a web client in JS. Therefore, I need to find a HTTP-ish way to transfer the events.

It is beyond the scope of this article to delve into the specifics of how I publish events within the Go server. However, for a basic understanding necessary for the rest of the article, here's a brief overview.

Serving Structure in the Go Server

Fundamentally, I have implemented a basic pubsub mechanism to channel the flow of events.

The next step is to make these events accessible to the client. This will be managed by a `http.Handler`. Below is the framework for this handler:

```go
type GestureHandler struct {
    inputEventBus *pubsub.PubSub
}

// ServeHTTP implements http.Handler
func (h *GestureHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // eventC is a channel that receives all the InputEvents
    eventC := h.inputEventBus.Subscribe("eventListener")
    // ....
}
```

The Default Choice: WebSockets

Now that I am within the HTTP handler, I need to devise a method to transfer data "over the wire". In this context, "over the wire" refers to two streams of bytes:

- the `io.Reader` encapsulated in the request's body.
- the `io.Writer` implemented through the ResponseWriter.

The most familiar method I know for exchanging messages between a server and a client is via WebSocket. WebSocket is a Layer 7 protocol that enables bi-directional streams of messages. Its implementation is relatively straightforward, and the client-side in JavaScript provides all necessary primitives for interacting with message flows.

On the server side, the situation differs, as Go's standard library does not include an implementation of WebSockets. This necessitates reliance on third-party libraries. While not inherently problematic, I generally prefer to avoid third-party libraries due to concerns about blackbox elements and dependency management complexities.

Nonetheless, I implemented a basic WebSocket-based message exchange to send events from the server to the client.

Having established the ability to listen to events and serve them over WebSockets, the next step was to accurately detect a gesture before sending the event. I incorporated basic business logic within my handler, using a timer to identify continuous movements. This allowed me to transmit motion in terms of distance moved by the finger, such as 100 pixels left, 130 pixels right, 245 pixels up, and 234 pixels down. While this is a simplistic implementation that does not differentiate between a square and a circle, it suffices for my needs.

However, testing this implementation posed a significant challenge. Being in the exploratory phase of the product's development, the most effective strategy was to adopt a 'test and learn' approach, rather than establishing a comprehensive test suite. This approach is likely to evolve as the product matures, but for the time being, it was necessary to "reverse engineer" the flow to understand the types of events generated by specific interactions with the screen.

Note: Simon Wardley's theory of evolution has significantly influenced my approach to this project. For a deeper understanding of this theory, I recommend consulting relevant literature or reaching out to me for further discussion.

Here lies a limitation of WebSockets: they are distinct from the HTTP protocol, meaning tools like cURL or netcat cannot be used to connect to the endpoint and monitor messages. While there are tools available for this purpose, they often lack certain features, such as trust for a self-signed certificate.

I spent considerable time trying to figure out how to stream messages to the screen while moving my finger on the tablet. I realized that learning the intricacies of WebSocket tooling might not be the most efficient use of my energy, especially when seeking quick results for the gesture functionality.

An Alternative Approach: HTTP Streams

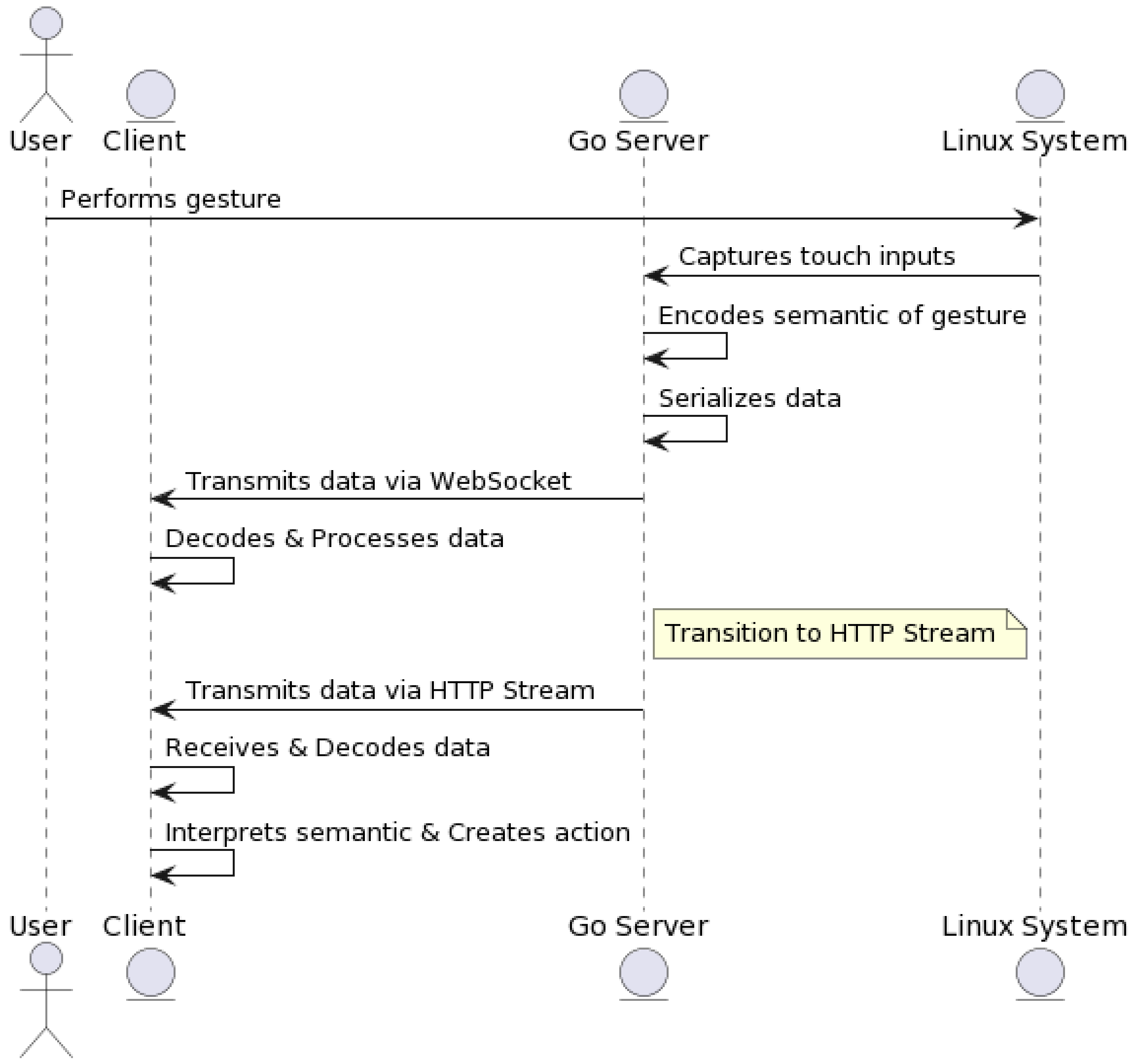
Sticking to a pure HTTP exchange might be a more suitable option. Let's take a step back to analyze the journey so far:

- Touch events are serialized by the Linux Kernel and exposed as a stream of bytes via a file `/dev/input/event`.
- This stream is dissected into a series of discrete events, which are then fed into a channel.
- These events are analyzed to detect a "gesture" – a sequence of events corresponding to the same "touch".
- The aggregated and sanitized events are then transmitted to the client using WebSocket.

Considering that the initial events are presented as a byte stream, and seeing the effectiveness of having the client read and segment these events, aligns well with the Unix philosophy.

Therefore, I decided to explore a low-level stream implementation for communication between the client and the server.

Internet and ChatGPT gave it a name: Server Sent Events


User
Client
Go Server
Linux System
Performs gesture
Captures touch inputs
Encodes semantic of gesture
Serializes data
Transmits data via WebSocket
Decodes & Processes data
Transition to HTTP Stream
Transmits data via HTTP Stream
Receives & Decodes data
Interprets semantic & Creates action

Sequence Diagram

From the server's perspective, the process involves continuously streaming bytes into the communication channel. These bytes are formatted specifically to announce events. A special MIME type (`text/event-stream`) is used to signal to the client that the server will be sending such a stream of bytes, and the client is expected to handle it accordingly.

Initially, I considered implementing Server-Sent Events (SSE), but then I realized I could first explore a simpler approach. This involves streaming bytes without fully adopting the complete logic of SSE, especially since I am managing both the client and server implementations. This approach allows for a more streamlined and controlled development process.

Implementing HTTP Stream in Go

Implementing a stream of bytes in an endpoint is fairly straightforward in Go. The handler is provided with a `ResponseWriter`, which is an `io.Writer`. This means that simply invoking the `Write` method in an endless loop will suffice for the task at hand.

The crucial aspect is to ensure that the stream is fed with the correct payload, namely the appropriate slice of bytes.

Serialization of the message

The concept of serialization is:

> the process of translating a data structure or object state into a format that can be stored (e.g. files in secondary storage devices, data buffers in primary storage devices) or transmitted (e.g. data streams over computer networks) and reconstructed later / source Wikipedia

So there is a need to “serialize” the gesture messages into an array of byte in a way that it can be deserialized in the client. As the client is a Javascript based program, I will use JSON.

So the gesture is implemented as a structure that implenents the JSON Marshaler interface.

```go
type gesture struct {
	leftDistance, rightDistance, upDistance, downDistance int
}

func (g *gesture) MarshalJSON() ([]byte, error) {
	return []byte(fmt.Sprintf(`{ "left": %v, "right": %v, "up": %v, "down": %v}`+"\n", g.leftDistance, g.rightDistance, g.upDistance, g.downDistance)), nil
}
```

What we have now is a collection of events that are aggregated into a `gesture` struct and serialized into binary format for transmission to the client. We have set up a `/gestures` endpoint to continuously serve this flow of gesture data.

Receiving and Decoding the Stream in JavaScript

On the client side, we fetch the data in JavaScript, using a worker thread to retrieve and analyze the gestures.

The worker receives a set of movements (a serialized `gesture` struct) and interprets them into higher-level commands, such as a “swipe left” action.

```javascript
const gestureWorker = new Worker('worker_gesture_processing.js');

gestureWorker.onmessage = (event) => {
    const data = event.data;
    switch (data.type) {
        case 'gesture':
            switch (data.value) {
                case 'left':
                    // Send the order to switch slide to the iFrame
                    document.getElementById('content').contentWindow.postMessage(JSON.stringify({ method: 'left' }), '*');
                    break;
                // ...
```

Within the worker thread, we use the `fetch` method to obtain the data from the `/gestures` endpoint. We then create a `reader` and continuously loop to read the incoming data.

```javascript
const response = await fetch('/gestures');

const reader = response.body.getReader();
const decoder = new TextDecoder('utf-8');
let buffer = '';

while (true) {
    const { value, done } = await reader.read();
    //...
    buffer += decoder.decode(value, { stream: true });

    while (buffer.includes('\n')) {
        const index = buffer.indexOf('\n');
        const jsonStr = buffer.slice(0, index);
        buffer = buffer.slice(index + 1);

        try {
            const json = JSON.parse(jsonStr);
            let swipe = checkSwipeDirection(json);
            //...
        }
//...
```

The `checkSwipeDirection` function analyzes the JSON data, identifying swiping gestures and transmitting them as appropriate actions.

With this setup, we now have a complete mechanism in place to capture events, detect swipe gestures, and initiate corresponding actions.

That’s all, folks!

Conclusion

In conclusion, the development journey of enhancing my tool, goMarkableStream, has been a vivid testament to the adage "simple is complex," underscoring the inherent value in embracing simplicity. While the allure of frameworks and sophisticated protocols is undeniable, this project illustrates that they aren't always the optimal choice for straightforward tasks. By sticking to the basic principles of Unix philosophy, where every interaction is treated as a stream of bytes, I was able to devise a solution that was both effective and elegant in its simplicity.

In this journey I also presented my decision to read and process events directly using out-of-the-box Go tools, without using third-party libraries. In line with Rob Pike's wisdom that "*a little copying is better than a little dependency*", this method not only ensured a more streamlined development process but also granted me a deeper understanding and control over the functionality I was building.

Ultimately, this experience has been a celebration of mastering bytes and the joys of hands-on software craftsmanship. It serves as a reminder that sometimes, the best solutions arise not from the complexity and sophistication of the tools we use, but from our ability to strip a problem down to its bare essentials and tackle it head-on. The old Unix philosophy, often overlooked, still holds a treasure trove of wisdom for modern developers, advocating for simplicity, clarity, and the fun inherent in direct byte manipulation.

stream reMarkable http golang websocket javascript

Data-as-a-Product: the keystone of the data-mesh Next ›

Table of Contents