Fault-Tolerant Bi-Directional Communications in Web-Based Applications

Naghmeh Ivaki, Filipe Araujo
CISUC, Dept. of Informatics Engineering, University of Coimbra, Portugal
naghmeh@dei.uc.pt, filipius@uc.pt

Abstract—The Hypertext Transfer Protocol (HTTP) and the Transmission Control Protocol (TCP) are the most popular protocols used in the development of web-based applications. Despite their popularity, the use of these protocols brings two limitations to applications and systems that require reliable interactive real-time communications: 1) HTTP forces applications to work in a request-response paradigm, even if a reply is not necessary, not allowing the server to send anything to a client without the client explicitly requesting it; 2) TCP provides no recovery options for network outages, thus forcing developers to write their own error-prone, complex, and ad hoc solutions. In this paper we introduce a solution that offers both bi-directional and reliable communication to web-based applications, even in presence of connection failures. To make this possible, we combine the idea behind WebSockets and a Session-Based Fault-Tolerant design pattern.

Index Terms—Fault-Tolerance, Bi-Directional Communication, Connection Failure, WebSocket, Web-Based Applications

I. INTRODUCTION

As the number of businesses, services and applications running on the internet grows, their interactions become more complex, thus requiring even faster and more reliable communication. Today, the Hypertext Transfer Protocol (HTTP) [11] is unquestionably the glue that binds most running services together. Although HTTP is not constrained to use a specific transport protocol, the Transmission Control Protocol (TCP) [14] is the usual choice to run underneath, as it provides reliable communication between web peers. Despite being so popular among web developers, the HTTP+TCP combination brings two limitations to applications and systems that require real-time reliable interactions, such as event-driven applications.

One of these limitations comes from HTTP itself, as it follows a strict request-response paradigm, between clients and servers. This type of paradigm limits the interactive real-time communication between web parties, because HTTP servers can only send messages back to clients in response to client requests. Hence, on one hand, distributed applications and systems are forced to work in this manner, even if a reply is not actually needed for some requests; on the other hand, an HTTP-based server cannot send anything to a client without the client explicitly requesting it.

The second limitation is related to the TCP protocol. With TCP, when a connection crashes, applications have no means to determine which data did or did not reach the other endpoint, thus making recovery or roll-back to a coherent state very difficult. In other words, HTTP applications cannot be entirely reliable, even if they use the TCP protocol.

Most attempts to provide interactive web applications use polling or server-side streaming [7], [5]. With polling, the browser sends requests at regular intervals and immediately receives a response, while with streaming, the browser sends one request, but the server maintains an open response that is continuously updated and sent. The most notable example of these techniques is Comet [5]. These methods impose unnecessary overhead and introduce latency, because their request and response headers contain a large amount of additional data. Many other efforts rely on two connections to emulate full-duplex communication, one for downstream and one for upstream. This method imposes a significant overhead and adds lots of complexity, due to the maintenance and coordination of two connections.

So far, the best solution to this problem is the WebSocket protocol [13]. WebSockets allow full-duplex, bi-directional communication through a single socket over the web, thus eliminating many of the problems that Comet-based solutions are prone to. It also removes the overhead and dramatically reduces complexity. There are several implementations of the WebSocket protocol in the literature, such as [1], [2], and [3]. Despite their merits, WebSockets are still vulnerable to TCP connection failures, as these interrupt communication. In fact, the problem of overcoming TCP connection failures is widely addressed, but still open, as all solutions have important drawbacks. For example, [4] and [15] try to use multiple alternative paths between the client and server to replace a failed path with another one, but these only work if at least one path survives. Other solutions replicate components to survive application crashes [12], but, these are resource demanding and may go beyond the needs of many online services. Some solutions checkpoint the state of the connections [10]; and others use a middle layer to intercept TCP system calls, such as [6] and [16]. These usually involve some intermediate layer that might not be readily available and that changes application semantics.

In this paper, we explain our session-based solution for recovery from TCP connection crashes and then propose the Fault-Tolerant WebSocket (FtWebSocket), to offer a reliable full-duplex communication between peers. Our solution uses a fault-tolerant session layer to ensure reliability. Web peers that may use FtWebSocket can reliably communicate together in a full-duplex space, either in a one-way or request-response manner. The core of our solution lies on a component called
Stream Buffer [16], which serves to keep unacknowledged data. If the connection providing the HTTP communication crashes, peers can use this Stream Buffer to resume communication without losing any data. To access the actual implementation of the session layer, we suggest the use of an Application Programming Interface (API) called FSocket, with only minor differences to a standard TCP socket implementation. This makes any code code that uses FtWebSocket, pretty much similar to code that uses WebSockets.

The solution we propose is efficient, because it includes nearly no overhead over standard WebSocket implementations (apart from the connection setup). Additionally, since we propose minimal changes to the standard TCP socket API, we do not impose a great amount of changes to the current practices of web programmers, in exchange for reliable communication. The approach we propose might improve interactive applications, like games, shared boards or stock market systems.

II. SESSION-BASED FAULT-TOLERANCE

Recovering a TCP socket from a connection failure is very difficult, because TCP does not provide any information regarding lost data and there is no other standard means for applications to determine which data reached the destination. This makes recovery much more complex than simply replacing a broken TCP socket by a new one. In this section, we present our session-based fault-tolerant solution, which allows the applications to transparently recover from connection failures. This solution is based on a session-based fault-tolerant design pattern we proposed in [9].

A. Components of the Session Layer

As shown in Figure 1, the solution is divided into three layers: application, session, and transport layers. The application layer includes a Client and a Server, which interact with the session layer to exchange data. The most common applications run directly over the transport layer by means of a TCP socket, while in our solution, the interaction between the application layer and the TCP Socket is accomplished through a session layer.

To recover data lost due to connection failures, we need to implement our own buffering over TCP. To do so, we use a Stream Buffer [16], to implicitly acknowledge messages. This Stream Buffer, which is the core of the session layer, is a circular buffer with a carefully selected size, \( b \), which must be at least equal to the TCP send buffer size, \( s \) plus the peer’s TCP receive buffer size, \( r \), i.e. \( b \geq s + r \). The point of using the Stream Buffer is to take advantage of the TCP acknowledgments, as no more than \( s + r \) bytes might remain unacknowledged. Hence, we only need to keep the last \( s + r \) bytes in the Stream Buffer and overwrite older bytes as new data keeps flowing into the TCP channel. Once the connection crashes, and after recovery, the sender can simply calculate the number of bytes sent and not received at the destination, by exchanging the number of written and read bytes. More details about this buffer are presented in [8] and [9].

To establish a connection and accomplish recovery from connection failures, we have a component called “FSocket”. Each FSocket owns a TCP Socket, to write and read data over TCP. FSocket implements the actions that should be taken for establishing a connection for the first time and after a failure, including the reconnection and retransmission procedures. It also provides read() and write() methods to read/write messages from/into the TCP Socket. One should notice that, as usual, setting up a connection is an asymmetric procedure that works differently on the client and server sides. The initiative always belongs to the client, even after a crash, due to NAT schemes and firewalls. For this reason, we added two concrete FSocket in our design, the Client FSocket and the Server FSocket.

Upon creation of a connection, both client and server start a handshake, to exchange the necessary information about a unique identifier of the connection, which is generated by the server; the size of the send and receive buffers, which are necessary to calculate the size of the Stream Buffer; and also the number of the bytes read, which is needed to calculate the number of bytes for retransmission if the connection is actually a reconnection.

On the server side, we use a new component, named Connection Set, to synchronize threads upon connection failures and reconnections. Once a thread using the connection associated to an FSocket tries to use a failed connection, it must wait on the Connection Set until some other thread comes in with a new FSocket from the same client. The failure recovery process is done transparently inside the FSocket, for a predefined and configurable period of time (the MAX_RECON_TIME).

B. API of Session Layer

The Session Layer we are using provides a very simple API, which is similar to the TCP Socket’s API, as any TCP Socket can be simply replaced with an implementation of FSocket, either CFSocket or SFSocket, depending on the
application’s role. Just as in standard TCP implementations, TCP servers own one passive handle to accept new connections. In Java TCP, this passive handle is called ServerSocket. We have an equivalent passive handle in our implementation, called FServerSocket. Moreover, all the read and write operations done on the TCP socket’s InputStream and OutputStream must be replaced with the read and write operations on the FSocket object. These replacements are summarized below:

```
Socket socket = new Socket (server,port)
// replaced with
CFSocket fsocket = new CFSocket (server,port)

Socket socket = serverSocket.accept()
// replaced with
SFSocket fsocket = fServerSocket.accept()

int read = inputStream.read(data)
// replaced with
int read = fsocket.read(data)

outputStream.write(data)
outputStr.flush()
// replaced with
fsocket.write(data)
```

III. FAULT-TOLERANT WEB SOCKET

Fault-Tolerant WebSocket (FtWebSocket) aims to provide reliable full-duplex communication in the application layer, for Web-based applications. It combines the idea behind WebSockets to offer full-duplex communication and the idea of session-based fault-tolerance, explained in the previous Section, to offer reliable interaction.

A. The FtWebSocket Protocol

The FtWebSocket follows an adaptation of the WebSockets protocol [13]. However, instead of TCP, it uses an FSocket underneath, to exchange messages and transparently handle connection failures. The FtWebSockets protocol has three main phases during an interaction: opening handshake, data transfer, and closing handshake (refer to Figure 2).

The opening handshake is intended to be compatible with HTTP-based servers and intermediaries, so that a single port can be used by both HTTP clients and FtWebSocket clients. Thus the FtWebSocket client’s handshake, the same as a WebSocket’s standard handshake, is an HTTP Upgrade request. The HTTP Upgrade is used to indicate the requirement to switch to the WebSocket protocol, if possible. If the server supports the protocol, it replies with the same “Upgrade: websocket” and “Connection: Upgrade” headers.

The main difference between the WebSocket and FtWebSocket handshake is that an FSocket’s handshake is combined into the HTTP request, by adding an extra field FSocket_params, which is inserted into the request, by the session layer. The data exchanged through this field includes the unique identifier of the connection, the size of the send and receive buffers, and the number of bytes received (this last data is sent if the connection is created for recovery purposes). After finishing the handshake, both client and server own one long-term full-duplex reliable channel to exchange their messages. This connection can be closed by any of the peers, by sending a close control message, just like in a WebSocket.

One should notice that the FSocket_params exchanged in the handshake are necessary for the Stream Buffer mechanism to work. On a reconnection, each peer’s Stream Buffer must know to which (previous) connection does this reconnection refer to and whatever data was missing, must be sent. The point of using HTTP for the handshake, instead of running a session-layer protocol for that, as we did in [8], [9], is to let the server inform the client if it supports the extra features offered by Stream Buffer. If it does, then both can engage in a reliable connection; if not, the client reverts to the basic WebSockets.

B. FtWebSocket Architecture

As shown in Figure 3, FtWebSocket is just a layer on top of FSocket. Both FtWebSocket client and server use FSocket for communication. To put it in simple terms, the idea of FtWebSocket is to replace a standard TCP Socket in WebSocket, by an FSocket (i.e. as explained in the Section II), to allow recovery from connection failures. Thus, the functionalities provided by FtWebSocket in the absence of connection failures are the same as those we find in a WebSocket. They just differ when a connection crashes. In this case, the FSocket enables the creation of a new connection to resend lost data and resume the interaction. Unlike this, to keep the connection alive, the WebSocket reconnects when an abnormal closure occurs, but it does not recover the lost data.

---

**Figure 2.** Three Important Interaction Phases with FtWebSocket

**Figure 3.** Architecture of Client and Server with FtWebSocket
In future implementations, we intent to provide the following general guarantees for FtWebSocket to be on a par with other implementations of the WebSocket protocol:

- Compatibility: It should have a simple structure to be rapidly and easily implemented in current languages in many different browsers;
- Platform Independence: It has to run with numerous browsers and supports communication between different platforms;
- Easy of Use: It should be used easily for different communication purposes with a quick set-up and a minimum effort of installation and configuration.

C. FtWebSocket Target Applications

We believe that FtWebSocket can help developing many web-based applications that require reliability in their interactive real-time communication. Here we enumerate three target applications for FtWebSocket.

- Collaborative online editing could be one of FtWebSocket target applications. A distributed group of people working directly on the single copy of a document or board may benefit from improved communication resilience, as this may reduce editing conflicts;
- Online multi-players games. These games allow several players to interact simultaneously via the Internet. The fundamental requirements of these applications are, to have a real-time interactive communication among the players, and to show exactly the same view of the game to the players. Reliable protocols must not impair performance, and all the players must reliably receive all events. FtWebSockets could be the right solution to the developers of this kind of applications. In fact, by imposing no overhead to WebSockets/TCP (apart from the connection moment), FtWebSockets do not compromise performance in exchange for reliability;
- Stock market systems. For instance, companies that use dashboard tracking systems to monitor stock indexes should get up to date and reliable information, to take accurate decisions. FtWebSockets may help to stream this kind of data rapidly and reliably.

IV. CONCLUSION AND FUTURE DIRECTIONS

In this paper we propose an initial step towards a solution that offers a fault-tolerant bi-directional communication to web-based applications and systems. For this, we use the main ideas behind WebSockets and combine them with our own session-based fault-tolerant socket, in a scheme that is simple and similar to the widely accepted practice. We believe that FtWebSocket can help developing many web-based applications that require reliable interactive real-time communication, including event-driven applications like multi-player games.

We believe that one of the most interesting challenges to improve this work is to enable FtWebSockets to work with HTTP proxies. In particular, proxies raise a complicated challenge for the Stream Buffer, as the Stream Buffer can only manage the TCP send and receive buffers, thus being unable to cope with intermediary agents holding data. Hence, finding a solution that simultaneously supports proxies and is backward compatible with WebSockets and HTTP seems like a very promising achievement.

ACKNOWLEDGMENTS

This work was partially supported by the Portuguese Foundation for Science and Technology contract SFRH/BD/67131/2009 and by the project iCIS - Intelligent Computing in the Internet of Services (CENTRO-07-ST24 FEDER-002003), co-financed by QREN, in the scope of the Mais Centro Program and European Union’s FEDER.

REFERENCES