A Design Pattern for Reliable HTTP-Based Applications

Naghmeh Ivaki, Nuno Laranjeiro, Filipe Araujo
CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
naghmeh@dei.uc.pt, cnl@dei.uc.pt, filipius@uc.pt

***Abstract*—HTTP is currently being used as the communication protocol for many applications on the web, supporting business and safety-critical services throughout the world. Despite the growing importance, HTTP-based applications are quite exposed to network failures, which can bring in huge losses to the service users and providers, including financial and reputation losses. Several approaches try to achieve reliable communication by using logging and retransmission of whole HTTP messages, which is especially ill-adapted to large messages. Stream-based approaches are more efficient as, upon failure, they transparently resume data transmission from where it stopped. Despite this, designing a stream-based mechanism involves significant challenges, as it is very difficult to know how much data is lost due to failure. In this paper we propose a stream-based mechanism for reliable HTTP communication that is entirely based on HTTP messages and is compatible with existing software. The mechanism is presented as a design pattern and relieves developers from explicitly handling connection failures, providing a standard way for building more reliable applications. Results show that the mechanism is functional, compatible with legacy applications, and that the coding and runtime costs of this design pattern are quite low.**

***Index Terms*—Reliable Communication, Design Pattern, HTTP**

I. Introduction

Distributed applications based on the HyperText Transfer Protocol (HTTP) are overwhelmingly popular and are being used to support not only businesses and critical services (e.g., e-commerce, financial services, e-health), but also entertainment, having impact on the lives of millions around the world. Despite using the Transmission Control Protocol [15] (TCP), which can overcome network packet losses, HTTP connection might fail due to long network outages that interrupt the underlying TCP connection. These failures may abort media transmission or cause online operations to fail, thus possibly bringing in serious personal, financial or reputation losses. Recovery from these common failures is challenging, because TCP provides nearly no recovery options for long-term network outages.

To the best of our knowledge, all current reliable solutions for HTTP-based applications (e.g., HTTPR [1], WS-Reliability [6]) are message-oriented, requiring, for instance, logging and retransmission to provide reliable delivery of messages. These solutions involve resending whole messages, which is quite inefficient when messages are large. Furthermore, message-based solutions cannot easily offer reliability to long-standing connections. For instance, with AJAX [7], the server often needs to keep the connection open for a long time, to push updates to the browser. If this connection fails, orchestrating a workable solution can be very difficult. The browser must repeat requests to obtain the missing parts of incomplete responses, whereas the server must be able to handle repeated requests.

A stream-based solution that buffers and resends unconfirmed data is a much cleaner solution, because it does not require changing the application semantics. The application does not need to log and resend messages, it can just rely on the channel. The main problem of using this approach in HTTP-based applications is the presence of proxies, a common element on the web, because they are, in general, unable to deal with the non-HTTP contents, which are necessary to set up a durable stream or to send partial content upon reconnection. In addition, proxies enforce at least two TCP connections (from client to proxy, and from proxy to server) which excludes simple recovery mechanisms that only work within a single direct connection between client and server (e.g., [14] or [22]). Finally, legacy non-reliable peers must still be able to communicate with reliable peers which can be difficult to achieve when control messages need to be transmitted. This is especially important, given the outstandingly large legacy software comprising the web.

We recently proposed the Session-Based Fault-Tolerant (`SBFT`) design pattern [14] to overcome TCP connection crashes in direct client-server communication. `SBFT` enables developers to reconnect their application after TCP connection crashes, without losing data and without requiring an additional layer for acknowledgments and retransmissions. Although this pattern provides a foundation for implementing reliable communication in TCP-based applications, there are still no practical and efficient design solutions that can tackle the specific challenges of the web.

In this paper we propose a stream-based solution for reliable HTTP communication in the web environment. Our solution features two key characteristics in comparison to `SBFT`: i) a handshake procedure that has been tailored to handle the specificities of HTTP applications; and ii) a control channel per client (shared by all the connections to the server) which is used in communication scenarios that involve proxies. This channel is used by client and server to exchange acknowledgments of large messages. Recovery from failure thus requires re-sending lost bytes instead of entire application-level messages. In our solution, all messages exchanged comply with

the HTTP protocol and we ensure that reliable and non-reliable HTTP peers can still inter-operate, which is, as discussed, a key aspect in the web.

We carried out an experimental evaluation using our design pattern in the Apache Tomcat 7.0.13 [8] HTTP connector within JBoss AS 7.1.1 [12]. Results show the correctness and efficiency of the approach, but most of all its overall usefulness, providing developers with an easy and standard means for developing more reliable HTTP-based applications.

This paper is organized as follows. The next section presents the state of the art and Section III reviews SBFT. Section IV presents our design pattern for reliable HTTP communication. Section V presents an experimental evaluation carried out with an implementation of our design pattern. Finally, Section VI concludes this paper.

II. Background

Developing distributed applications that resist to network outages is a quite difficult problem. In the literature, we can find a large number of attempts, of which we chose to review *a*) server-side replication schemes (to overcome server failures), *b*) replacement of the transport layer (i.e., of TCP), and *c*) attempts focused on HTTP. We also review *d*) design patterns for distributed computing and *e*) our previous work.

a) Server-Side Replication: Solutions as ST-TCP [11], ER-TCP [19], and HydraNet-FT [21] use replication on the server side, to tolerate TCP connection failures occurring when there is a crash of a replicated instance. Although most of these solutions can be used for HTTP applications, they do not tolerate connection failures caused by network crashes.

b) Alternatives to TCP: Multipath TCP [2] or SCTP [16] are transport layer protocols that use one or more redundant paths between client and server, to tolerate network congestion and disconnections. Their approach to disconnection is redundancy, via multi-homing. With these protocols there is no way to recover a connection failure if replicated connections fail due to, for instance, an internal network crash.

c) Reliable HTTP: HTTPR [1] was put forth by IBM, but later abandoned by lack of interest. It aimed at ensuring that each message is either delivered to its destination exactly once or is reliably reported as undelivered, even in the presence of network and peer failures.

EOS2 [20] is a solution for web-based services that uses a logging mechanism on the client and server sides, which ensures exactly-once execution of requests in the presence of endpoint failures. This solution does not deal with connection crashes and it may deadlock when both parties are alive but the TCP connection crashed.

Web Service Reliability [6] (WS-Reliability) is a Web Services specification that is used for exchanging SOAP [3] messages, typically over HTTP, with several reliability guarantees. The WS-reliability specification defines the following reliability semantics: guaranteed message delivery, guaranteed message duplicate elimination, guaranteed message delivery and duplicate elimination, and guaranteed message ordering. WS-Reliability was eventually superseded by the concurrent WS-ReliableMessaging [5], which is another specification that serves similar purposes, including the same set of delivery guarantees.

All these HTTP mechanisms ([1], [20], [6], [5]) are message-oriented and use logging to guarantee message delivery, which means that if a message does not reach the destination due to a connection crash, it has to be sent again. This is quite inefficient when a message is long (e.g., a file, or the typically verbose SOAP message). Moreover, ensuring that the pair browser-server can successfully retry an HTTP request is a very difficult task.

d) Design Patterns: The idea of using design patterns for distributed computing started more than two decades ago. For example, the Acceptor-Connector pattern [18] tries to simplify the design of connection-oriented applications, by separating event dispatching from connection setup and service handling. However, the Acceptor-Connector pattern cannot support multiple threads, and is, therefore, unfit for modern servers. To improve the efficiency of the Acceptor-Connector pattern, the Leader-Followers [17] dispatches events using a fixed number of threads.

In fact, in the last decade we have assisted to organize distributed interactions into a set of design patterns, first for Enterprise Application Integration [9], and more recently towards SOAP/WSDL and RESTful web services [4]. This latter book collects multiple known types of high-level interaction between client and server, e.g., request/acknowledge/polling or request/acknowledge/callback, respectively for client polling or server callbacks. Unlike these books, we focus on lower level details of the interactions.

e) Our Previous Work: In [13], we proposed a design pattern named "Fault-Tolerant Multi-Threaded Acceptor-Connector" (FTMTAC), which uses the combination of the Acceptor-Connector and Leader-Followers patterns with a custom buffering mechanism to tolerate TCP connection failures. Recently, we moved the fault-tolerance mechanisms of FTMTAC to work below the service handler and proposed a "Session-Based Fault-Tolerant Design Pattern" (SBFT) [14]. This basically moves the fault-tolerance mechanism from the application layer into a session layer presented as a socket, thus releasing the developer from using the Acceptor-Connector pattern, and paving the way to the use of this or other patterns, as s/he likes.

In this paper we modify and extend the session-based fault-tolerant design pattern, which is a stream-based solution, to overcome the challenges associated with HTTP communication in the web environment. A distinctive feature of our work is that we present the solution as a design pattern, which we intend to generalize for other protocols besides HTTP in future work.

III. The Session-Based Fault-Tolerant Design Pattern

The main goal of SBFT, which we proposed in [14], is to allow TCP-based applications to transparently recover from connection crashes in client-server communication. SBFT allows to decouple the recovery concerns from the application logic and, at the same time, enable recovery from failures with

minimal overhead. In this section we review a crucial piece in this design, the `Stream Buffer`, a circular buffer that discards the need for an extra layer for acknowledgments over TCP [22]. Each peer involved in the communication uses one `Stream Buffer`, to keep all sent data that might not have been received by the other peer.

To understand how this `Stream Buffer` works, let us first consider the simple scenario shown in Figure 1 (without `Stream Buffer`), which displays a sender and the corresponding receiver application, at the precise moment when their TCP connection fails. The figure shows three buffers: the sender application buffer, the TCP send buffer (in the sender), and the TCP receive buffer (in the receiver). Up to the connection failure, the receiver had received m bytes, whereas the sender had written a total of n bytes to the TCP socket. It is easy to see that, upon reconnection, the sender only needs to send the data marked in red and blue, which still was in the TCP buffers and that was lost due to the connection failure. This means that the sender needs to send the last $n - m$ buffered bytes and for this to work, the receiver and sender must respectively keep the values m and n.

Assume now that the (maximum) size of the TCP send buffer is s bytes, whereas the receive buffer of the receiver has r bytes. The number of bytes we may lose when a connection crashes is at most $b = s + r$. The properties of TCP ensure that it must have delivered the remainder bytes [15]. Hence, a `Stream Buffer` of size equal to or greater than $b = s + r$ can guarantee that we do not lose any data in the presence of a failure. Apart from reconnections, this buffer discards the need for any acknowledgment, because it is able to store all data potentially in transit between the peers.

The shortcoming of using such type of solution for reliable communication in the Web environment is that this type of buffering mechanism cannot withstand proxies, which are a frequent element in this kind of setting. In fact, these intermediate nodes can keep an arbitrary amount of data outside their own buffers, causing the data in transit to exceed the $b = s + r$ bytes available on the `Stream Buffer`. This means that data can be lost if the connections that have the proxy as endpoint crash (or if the proxy itself crashes). Moreover, we must not expect proxies to adhere to specific reliable communication mechanisms, which means that any mechanism for reliable communication must consider that legacy proxies might stand between client and server and thus the information exchanged must conform to the HTTP protocol to pass through the proxy. This has clear implications on the design of any solution for reliable messaging, which we describe in detail in the following section.

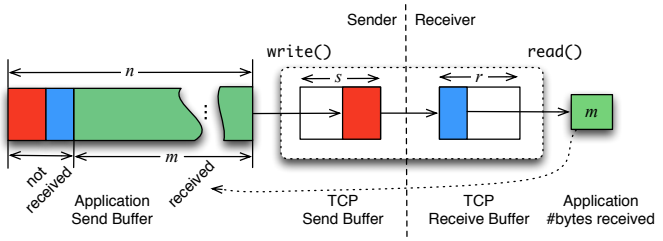

Fig. 1. Buffers in a simple client-server scenario. [14]

IV. Designing Reliable Communication in HTTP-based Applications

In this section we first describe the main challenges for reliable communication in HTTP-based applications. Then, we explain the technical solutions that we selected to overcome such challenges and describe the design of our session-based pattern for reliable HTTP-based applications.

A. Challenges for Reliable Communication in HTTP-based Applications

In the Web environment, the scenario of direct client-server communication can be rare. In fact, if the HTTP client and server communicated to each other through a direct TCP connection, we could easily adapt our `SBFT` solution [14] to overcome the TCP connection crashes. However, intermediate proxy nodes may stand in front of HTTP servers for different purposes, including security (e.g., a filtering firewall), translation (e.g., to route traffic to an appropriate site), and performance (e.g., for load balancing or caching content). When a proxy exists, the client TCP connection is not established directly to the server, but is instead established to the proxy. Consequently, the connection accepted by the server is not established directly by the client, but by the proxy on behalf of the client.

Figure 2 shows a simple client-server scenario, which involves a proxy, and depicts the internal data buffers involved. As we can see, there is **extra buffering of data at the proxy**. While our `SBFT` pattern depends on having a `Stream Buffer` as large as the TCP send and TCP receive buffers combined, now we have a total of five points in the traffic that can serve as buffers: the sender TCP send buffer, the proxy TCP receive buffer, the proxy internal state, the proxy TCP send buffer, and the receiver TCP receive buffer. The size of the buffers is now $b1 + b2 + b3 + b4 + b5$, much more than the $b1 + b5$ that the `SBFT`'s `Stream Buffer` was prepared to take. The problem becomes quite serious as we cannot know the sizes of most of these buffers and thus do not know how much data should be kept to be re-sent in case of failure.

Considering the case where the proxy performs a security function, in particular content-based filtering, it will very likely **filter out non-HTTP messages**. As discussed, the solutions discussed in Section II and `SBFT` exchange handshake messages that do not comply with the HTTP messages format and, as such, the critical handshake step will fail in the presence of content-based filtering proxies.

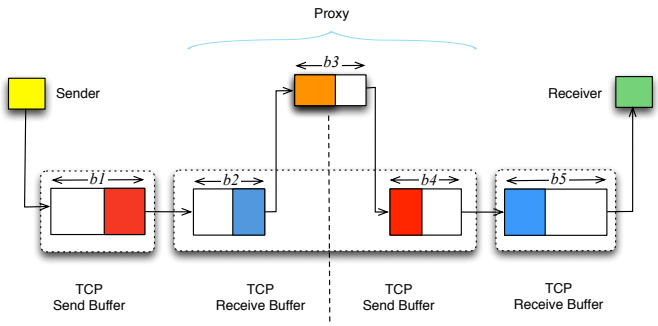

Fig. 2. Buffers in a client-server model with proxies

Finally, **both non-reliable and reliable clients and servers must be able to inter-operate**. Hence, the design of a solution for reliable communication must ensure that inter-operation with legacy software is possible. This is especially important in the Web environment, which comprises a very large base of legacy software that must not be prevented from communicating with reliable peers.

B. Achieving Reliable HTTP-based Communication

We designed a handshake procedure to not only transparently pass through proxies, but also to determine whether there is actually any proxy in the path. When the client establishes a connection, we send a message to initiate the handshake. We must do a handshake *prior* to sending the client's request, otherwise, if the response failed to come, the client would not know whether it would be safe to repeat the request.

Figure 3 shows the handshake for a new connection, which consists of a normal HTTP request and response, with a few modifications. First, the request points to a specific non-existing URL common to all reliable servers (the actual URL used should be long, so that it is unlikely that it collides with a real name in the system). Also, the HTTP messages carry a few extra headers. Considering the request, the `FSocket Connection` header holds the IP address and port of the client and server. The `FSocket Handshake` carries the identifier of the connection (0 means that no connection previously existed), the number of bytes it has read so far (used for recovery), the size of the TCP send buffer size, and the size of the TCP receive buffer size. Regarding the response, corresponding headers are included, where the identifier of the connection is 1. The `FSocket Proxy` header, informs the client whether a proxy was detected by the server. The server detects the presence of a proxy if the address sent in the header is different from the address of the TCP connection it receives. Once the handshake finishes, the client can start sending the actual request. It is worth noting that a legacy client will simply not use these headers, whereas a legacy server will ignore the extra headers. Hence, all the combinations of legacy/reliable client and server work.

The exchange of messages is slightly different if the client is reconnecting (i.e., a connection failure has occurred). In this case, the client informs the server (during the in the `FSocket Handshake`) with the identifier of the connection and the number of bytes it has read so far. This lets the server determine if the client is missing any bytes from previous responses. The server can then send back the number of bytes *it* has read so far in the `FSocket Handshake` header, plus the (possibly) missing bytes, as part of a normal HTTP response. If necessary, the handshake finishes with another message of the client (not shown), which sends lost bytes from previous unfinished requests.

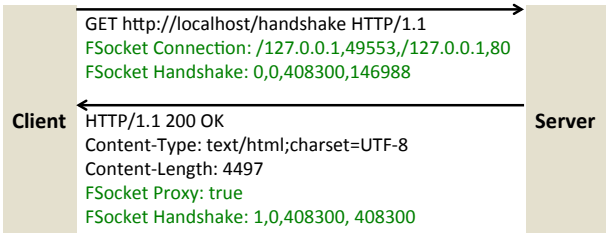

Fig. 3. The handshake procedure for reliable HTTP communication.

If no proxy exists, client and server can rely on the implicit buffering of `SBFT`. In this case, since the size of the `Stream Buffer` is not smaller than the TCP send buffer plus the peer TCP receive buffer, the data in transit is always guaranteed to be in the sender. However, if a proxy exists, the buffering and acknowledgments scheme must become explicit, because the sender side must never allow the amount of data in transit to exceed the size of its `Stream Buffer`. To send the acknowledgments, we use one control connection, shared by all the client connections to a given server. This control channel is a standard connection to the server HTTP port.

Whenever a `Stream Buffer` is becoming full, the peer should acknowledge reception of data. For example, if the server is sending a large file, the client application should acknowledge reception of the data, to let the server release space from its `Stream Buffer`. To do this, once an application receives a number of bytes equal or greater than half the size of the peer's `Stream Buffer`, we send an acknowledgment via the control channel. This allows the peer application to clean its buffer, thus allowing it to proceed. These acknowledgments are HTTP messages that include the number of bytes read so far by the application.

C. Design Pattern for Reliable HTTP-based Applications

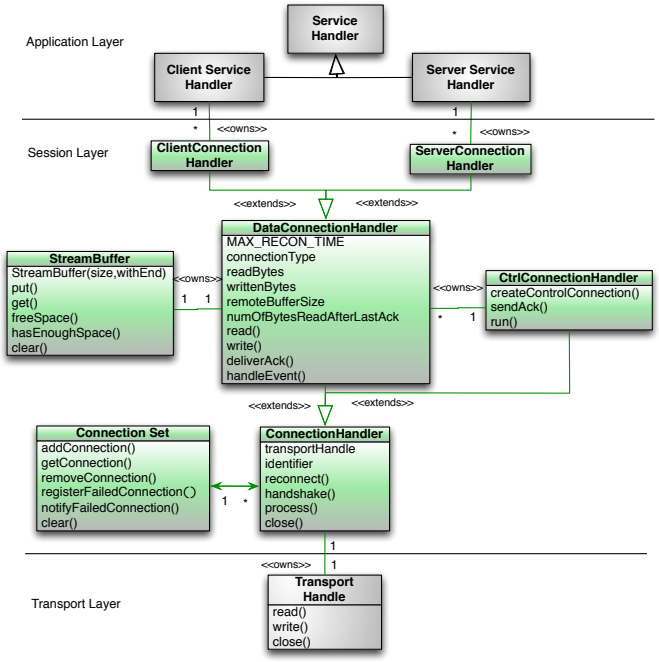

Fig. 4. Session-Based Fault-Tolerant Design Pattern for HTTP-Based Applications

Figure 4 presents the class diagram of our new pattern that has been designed specifically to handle the key web challenges. The components are set in three key layers, application, session, and transport. There are three main components in the session layer, including `Connection`

Handler, Connection Set, and a modified Stream Buffer. The Connection Handler takes care of setting up connections and exchanging the handshake messages. The Connection Set is used to keep the information (i.e. an identifier and a reference) about all connections, including data and control connections. The Stream Buffer keeps unacknowledged data that is still in transit. We now explain how these components interact.

1) Key Components: The **Stream Buffer** can be initialized as an endless circular buffer (i.e. with no limitation to write, if no proxy exists) or as a limited circular buffer (when a proxy exists). This component implements the actions to save, retrieve, and clean the data.

Each **Connection Handler** owns one Transport Handle (e.g., a TCP socket) to read and write data. It provides some abstract methods that should be implemented appropriately to take actions for accomplishing the handshake procedure with the remote peer, processing the handshake headers, and reconnecting after failure. The Connection Handler is extended as a **Data Connection Handler** or **Control Connection Handler**, depending on the type of connection. The Data Connection Handler provides a simple interface to the Service Handler (i.e., browser or server) to read and write messages. It implements all the actions required to save data into the Stream Buffer, and for recovery of failed connections and retransmission of lost data. The Control Connection Handler provides an interface for the Data Connection Handler to create a control channel, if necessary, to send acknowledgment messages. A Control Connection Handler, which might be shared among several data connections, checks for the arrival of new acknowledgments and delivers them to the appropriate Data Connection Handler.

Besides keeping information of connections, the **Connection Set** serves to synchronize threads upon connection failures and reconnections. Once a thread associated to a Data Connection Handler tries to replace a failed connection, it either must wait on the Connection Set until some other thread comes in with a new Data Connection Handler (i.e., when the design of the service handler is blocking), or it must register its data in the Connection Set to be notified later on, when a new Data Connection Handler arrives from the same client for the reconnection purpose (i.e., when design of the service handler is non-blocking).The information of the connections is removed from the set when the connection is closed.

2) Interactions Between the Components: Figure 5 presents the collaborations between the different components of the pattern in a failure free scenario. To create a new connection, the client creates a new Client Connection Handler, which internally creates a Transport Handle (e.g., a TCP socket). On the other side, the server uses a passive (unconnected) handle to accept a new connection through the method *accept()*.

Once the connection is set up, the Client Service Handler starts the handshake procedure, by inserting two extra lines in the header of a valid HTTP request. The first line includes the information of the connection created by the client, which consist of the local IP address, local port number, destination IP address, and destination port number. As we mentioned before, this information is required, to let the other peer find out whether a proxy between client and server exists. The second line includes the unique identifier of the connection (i.e. which is equal to zero if the connection is new) and the size of the TCP send and receive buffers.

When the server receives the handshake request it generates a unique identifier for the connection, identifies the existence of the proxy, by finding any mismatches between the client connection information and its real peer connection information, and accordingly initializes an appropriate Stream Buffer. The Server Connection Handler completes the handshake protocol (The handshake() invocation on the server side) by generating a valid HTTP response including two extra lines in the header. The first line indicates the existence or absence of a proxy, whereas the second line carries the unique identifier of the connection, and the TCP buffers sizes. Upon receiving the message, the Client Connection Handler initializes its own appropriate Stream Buffer.

If there is no proxy, the Data Connection Handlers can simply write the data into the Stream Buffer after any successful write operation. If there is some proxy, then a Control Connection Handler should be created to exchange acknowledgment messages on both sides. To distinguish between a Data Connection Handler and a Control Connection Handler in the server side, a handshake message is sent by the client control connection. The new control connection is inserted into the Connection Set through the method *addConnection()*, with the identifier of the remote peer. After initialization, both Control Connection Handlers start reading acknowledgment messages, which they deliver them to the appropriate Data Connection Handler, so that it can release space from the corresponding buffer.

Figure 6 describes the collaborations between components in presence of a failure. When a failure occurs, both sides will eventually start the reconnection phase, by calling the method *reconnect()*. Upon invoking this method, the Client Connection Handler tries to create a new connection to the server during a predefined period of time. On the other side, the Server Connection Handler gives the connection identifier and a waiting time to the *registerFailedConnection()* method.

After acceptance of a connection request and creation of a new handler, the Client Connection Handler starts the handshake protocol. It uses a predefined HTTP request (which does not correspond to any available resource on the server) with an extra handshake header, consisting of the identifier of the failed connection, and the number of bytes received in the client side. This lets the server distinguish fresh connections from reconnections. The server side accepts the new connection and initializes a new Server Connection Handler. This component is then responsible for notifying the failed handler through the method *notifyFailedConnection()* of the Connection Set. If the implementation of the service handler is non-blocking, the new Data Connection Handler is delivered to the failed

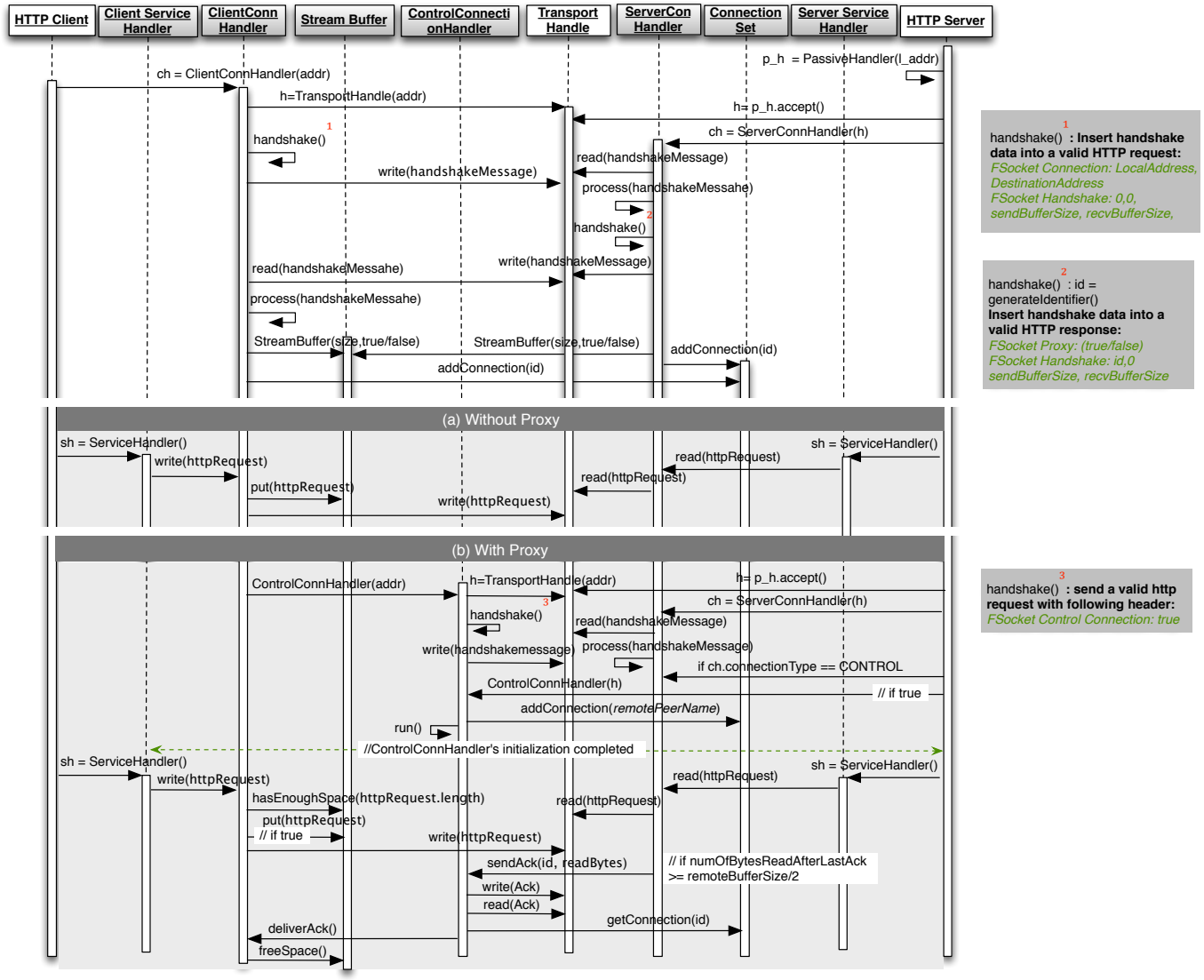

Fig. 5. Component interactions in a failure-free scenario

one through the method *handleEvent()*. Then, the `Server Connection Handler` completes the handshake by inserting the number of bytes received. Then both sides start retransmission of data lost due to connection failure. In the case there was a control connection, the `Client Connection Handler` invoke the method *reconnect()* of the `Control Connection Handler` to establish a new connection if it fails too.

V. Experimental Evaluation

In this section we present the experimental evaluation carried out to illustrate the applicability of our proposal, and we discuss the results. The experiments focus on three key aspects: correctness, performance, and complexity of the solution.

We implemented our design pattern in Java and used it in an HTTP client and in the Apache Tomcat 7.0.13 HTTP connector [8] included in JBoss AS 7.1.1 [12]. Table I presents

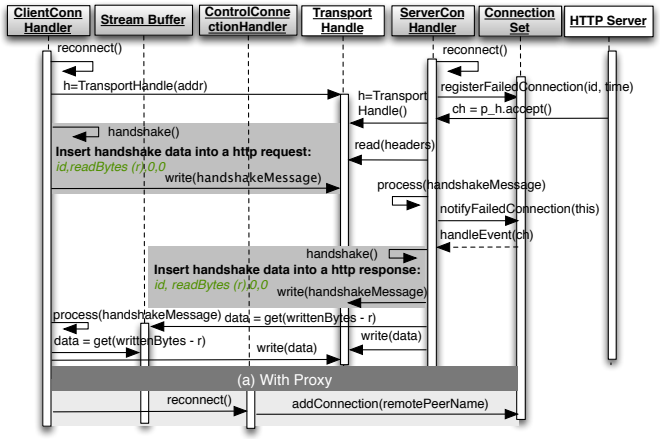

Fig. 6. Component interactions in the presence of a failure

the client and server machines, which were placed on an isolated 100 Mbps Local Area Network to execute the tests.

A. Verifying Correctness

To evaluate the correctness of the design pattern implementation, we considered the following four different client-server communication scenarios: 1) a reliable HTTP client communicating with a non-reliable JBoss AS; 2) a non-reliable HTTP client communicating with a reliable JBoss AS; 3) a reliable HTTP client communicating with a reliable JBoss AS, without any proxy in the middle; 4) a reliable HTTP client communicating with a reliable JBoss AS via a proxy. The scenarios 1) and 2) are used to show that our design pattern is compatible with legacy and unreliable software; and the scenarios 3) and 4) are used to show that the design pattern is able to tolerate connection failures with and proxies.

We first used a browser to generate HTTP requests for a set of typical web resources deployed in the non-reliable JBoss AS. We used those requests within our custom HTTP client and also used the responses as oracle for comparison with the responses obtained from the reliable version of JBoss AS during the tests.

For each of the scenarios we let client and server exchange messages during 5 minutes (each test was repeated ten times). We observed that reliable and non-reliable peers were able to communicate perfectly in scenarios 1) and 2). To evaluate the ability to recover from failures (scenarios 3 and 4)), we used `tcpkill` to cause connection crashes at random instants during each test (three crashes per test). We observed that all interactions worked correctly even in presence of the failure and all expected messages were correctly received.

B. Evaluating Performance and Resource Usage

To evaluate **performance** we measured latency (round-trip-time of a request-response interaction) and throughput (number of operations per time unit) in the following four scenarios: 1) Non-reliable client and server interacting without proxy; 2) Non-reliable client and server with proxy; 3) Reliable client and server without proxy; 4) Reliable client and server with proxy. The Scenarios 1) and 2) (non-reliable scenarios) are used to evaluate overhead and performance degradation in the reliable scenarios.

The proxy server used in our tests was Squid 3.1 (squid-cache.org). In each scenario we exponentially vary the number of clients from 1 to 512, and each client sends 1000 requests. To calculate throughput, the clients send each request and do not wait for the response (a different thread receives the responses). Throughput is then calculated when the last response arrives. To calculate latency we set each client to wait for the response after sending each request.

Figure 7 shows the results obtained for latency (a) and throughput (b). As we can see, latency progressively increases

TABLE I
SYSTEMS USED IN THE EXPERIMENTS

Endpoint	OS	CPU	Memory
Client	Mac OS X, version 10.10.1	2.4 GHz Intel Core 2 Duo	4GiB RAM, 3 MiB cache
Server	Linux 2.6.34.8	2.8 GHz Intel quad core	12 GiB RAM, 8 MiB cache

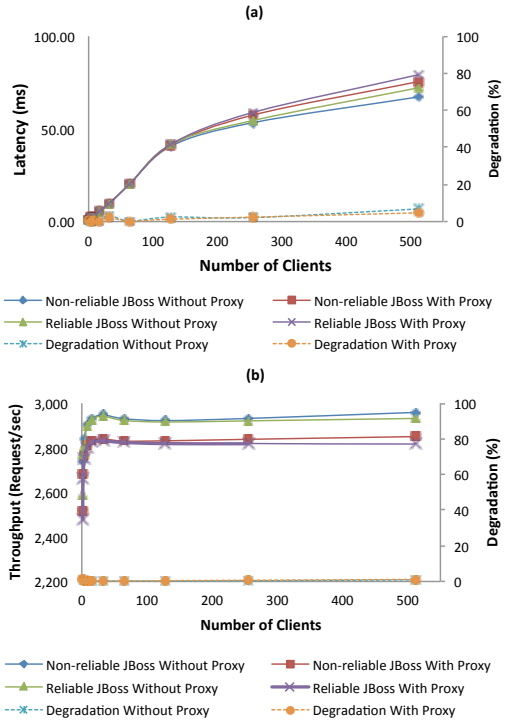

Fig. 7. Latency (a) and throughput (b) observed during the experiments.

with the number of clients, the same happens with throughput. The important aspect is that, when we compare the scenarios that use reliable peers with those that use the non-reliable peers, throughput and latency degradation show low values. In fact, although we have all necessary mechanisms for reliable communication in place and in operation, performance degradation is quite small.

Regarding **resource usage**, we used JConsole 1.6.065-b14-466.1 to examine the CPU and memory usage at the server. We again exponentially varied the number of clients from 1 to 512, and used each HTTP client to send 100 requests per second during 3 minutes, which we experimentally observed to be enough to show the usage of resources. Figure 8 shows that the overhead in terms of memory (a) and CPU (b) is kept under acceptable limits. The memory used by our reliable server is, as expected, obviously higher than the non-reliable one, due to the extra buffering placed on top of TCP. The CPU overhead shows to be again quite low, which is an excellent indication as this resource can be many times of critical importance. Moreover, the key goal is, at this point, to provide a reliable communication mechanism and resource usage can be object of further optimization in future work.

C. Complexity of the Design and Implementation

We measured three important complexity metrics, Lines of Code (LOC), Cyclomatic Complexity, and Nested Block Depth [10], to analyze the complexity of our design pattern implementation. To accomplish our evaluation we implemented three versions of a simple HTTP client-server application: plain HTTP application, fault-tolerant HTTP application using `SBFT`, and fault-tolerant HTTP application using HTTP-based design pattern. Table II presents the comparison between these

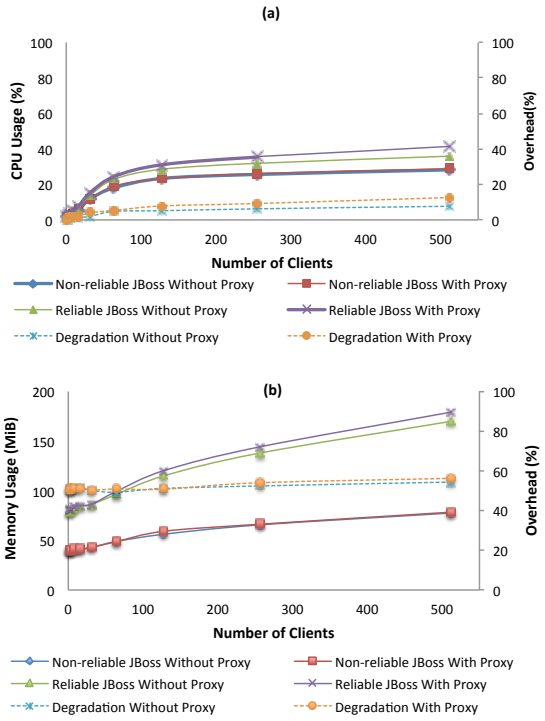

Fig. 8. CPU (a) and memory (b) usage observed during the experiments.

three versions. The measurements show that we used 485 extra lines of code in SBFT, to turn a non-reliable into a reliable application, and we used an extra 316 lines of code to adapt the pattern for HTTP. In addition, the average cyclomatic complexity per method in the first two cases is in the 1.7-1.8 range, while it increases by a small amount to 1.9 for our HTTP-based design pattern pattern. Finally, the depth of nested blocks of the non-reliable application is 1.28, close to the 1.4 of the reliable versions. These results show that providing reliable communication for HTTP applications is quite inexpensive, especially when considering the huge gains that our solution brings for developers, by eliminating the effort that would be needed to create a custom solution for reliable HTTP communication.

TABLE II
CODE COMPLEXITY

	LOC	Cyclom. Complexity	Nested Block Depth
Plain HTTP App	572	1.74	1.28
FT HTTP App using SBFT	1057	1.77	1.40
FT HTTP App using HTTP-based design pattern	1373	1.95	1.40

VI. CONCLUSION

In this paper we presented a stream-based solution for HTTP peers that can transparently overcome connection failures. The proposed design pattern works with legacy clients, servers, and proxies. The experimental evaluation carried out with an implementation of the pattern in a widely used server shows that this design imposes a small overhead, while ensuring that network glitches do not prevent service delivery, even when intermediate nodes are present in the communication. As future work, we plan to modify the pattern to allow recovery from endpoint crashes and generalize it to other communication protocols.

ACKNOWLEDGMENTS

This work was supported 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

[1] A. Banks, J. Challenger, P. Clarke, D. Davis, R. P. King, K. Witting, A. Donoho, T. Holloway, J. Ibbotson, and S. Todd. HTTPR specification. *IBM Software Group*, 10, 2002.

[2] S. Barre, C. Paasch, and O. Bonaventure. MultiPath TCP: from theory to practice. In J. Domingo-Pascual, P. Manzoni, S. Palazzo, A. Pont, and C. Scoglio, editors, *NETWORKING 2011*, Lecture Notes in Computer Science, pages 444–457. Springer Berlin Heidelberg, 2011.

[3] E. Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Inc., Feb. 2002.

[4] R. Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional, 1 edition, 2011.

[5] D. Davis et al. Web services reliable messaging (WS-ReliableMessaging). Technical report, Technical report, OASIS, http://docs. oasis-open. org/ws-rx/wsrm/200608/wsrm-1. 1-spec-cd-04. html, retrieved 12.11, 2006.

[6] C. Evans, D. Chappell, D. Bunting, G. Tharakan, H. Shimamura, J. Durand, J. Mischkinsky, K. Nihei, K. Iwasa, M. Chapman, et al. Web services reliability (WS-Reliability), ver. 1.0. *joint specification by Fujitsu, NEC, Oracle, Sonic Software, and Sun Microsystems*, 2003.

[7] J. J. Garrett et al. Ajax: A new approach to web applications. 2005.

[8] J. Goodwill. *Apache jakarta tomcat*, volume 1. Springer, 2002.

[9] G. Hohpe and B. Woolf. *Enterprise Integration Patterns — Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[10] P. C. Jorgensen. *Software Testing: A Craftsman's Approach*. Auerbach Publications, Boston, MA, USA, 3rd edition, 2008.

[11] M. Marwah and S. Mishra. TCP server fault tolerance using connection migration to a backup server. *In proceeding international conference on dependable systems and networks (DSN)*, pages 373—382, 2003.

[12] R. H. Middleware. JBoss Application Server, available: http://www.jboss.org/jbossas/. 2008.

[13] F. B. Naghmeh Ivaki, Filipe Araujo. Design of multi-threaded fault-tolerant connection-oriented communication. *The 20th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2014.

[14] F. B. Naghmeh Ivaki, Filipe Araujo. Session-based fault-tolerant design pattern. *In proceeding 20th international conference on parallel and distributed systems (ICPADS)*, 2014.

[15] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.

[16] C. M. R. Stewart. Sctp: new transport protocol for tcp/ip. *IEEE Internet Computing*, Vol. 5, No. 6:pp. 64–69, 2001.

[17] D. Schmidt, C. Ryan, M. Kircher, I. Pyarali, and F. Buschmann. Leader-followers. In *PLoP conference. http://hillside. net/plop/-plop2k/proceedings/ORyan/ORyan. pdf*, 1998.

[18] D. C. Schmidt. Acceptor-connector: an object creational pattern for connecting and initializing communication services. *Pattern Languages of Program Design*, 3:191–229, 1996.

[19] Z. Shao, H. Jin, B. Cheng, and W. Jiang. ER-TCP: an efficient fault-tolerance scheme for cluster computing. *The Journal of Supercomputing*, 2007.

[20] G. Shegalov and G. Weikum. EOS2: unstoppable stateful PHP. *Proceedings of the 32nd international conference on Very large data bases*, page 1223–1226, 2006. ACM ID: 1164249.

[21] G. Shenoy and S. K. Satapati. HYDRANET-FT: network support for dependable services. *In international conference on distributed computing systems*, 2000.

[22] V. C. Zandy and B. P. Miller. Reliable network connections. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, MobiCom '02, pages 95–106, New York, NY, USA, 2002. ACM.