A DESIGN PATTERN FOR RECOVERING FROM TCP CONNECTION CRASHES IN HTTP APPLICATIONS

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

Abstract
HTTP is currently being used as the communication protocol for many distributed applications, supporting business and safety-critical services at a world-wide scale. Despite of their increasing importance, HTTP-based applications are still quite exposed to TCP connection crashes, which can result in huge losses for services users and providers, including financial and reputation losses. Typical techniques for achieving reliable HTTP communication rely on buffering and retransmission of complete HTTP messages, and are quite un-adapted to large messages. Stream-based approaches are more efficient as, after a crash, data transmission can be resumed from where it stopped. However, it is very difficult to know how much data is lost after a crash, as TCP provides insufficient support to obtain this information and none to recover from connection crashes. This makes the design of any stream-based reliability mechanism a significant challenge. In this paper we propose a stream-based solution for reliable HTTP communication that is retro-compatible with existing software. The mechanism is presented as a design solution and relieves developers from explicitly designing recovery code for handling connection crashes, providing a standardized way for building reliable applications. Our experimental evaluation shows that the solution is functional and results in acceptable coding and runtime costs.

Keywords: HTTP, Reliable Communication, TCP, Connection Crashes, Fault-tolerance, Stream-Based Solution

1. INTRODUCTION
Distributed applications based on the Hyper-text Transfer Protocol (HTTP) (Fielding et al., 2009) are increasingly being used to support businesses and critical services (e.g., commerce, healthcare, entertainment), having impact on the lives of millions around the world. Thus, HTTP-based connection failures can have a severe impact on the parties involved, including loss of business transactions, customer dissatisfaction, and reputation damages (Jones et al., 2000). Although HTTP applications use the Transmission Control Protocol (TCP) (Postel, 1981), which can provide reliable communication by overcoming packet losses, HTTP communication might still fail due to TCP connection crashes, which ultimately break the end-to-end interaction, if no further reliability mechanism exists.

Technically, a TCP connection crashes when the operating system aborts a connection, for one of the following reasons: i) when data that has been sent is not acknowledged for a given number of retransmissions; ii) when the application waits for reading over a period of time that exceeds the defined timeout; iii) when an underlying network failure is reported by the network layer; iv) and when one of the IP addresses involved changes (Zandy and Miller, 2002). Thus, connection crashes may abort data transmission, causing, for instance, online operations to fail. The problem is that recovering from these crashes is quite challenging, because TCP provides nearly no recovery support for the developers to rollback the peers to some coherent state. As we review in Section 2, recovering from TCP connections and endpoint crashes is a major research topic. However, most solutions involve extra hardware, extra libraries, extensively changing application semantics or all of these problems combined.

To the best of our knowledge, all current solutions for building reliable HTTP-based applications (e.g., HTTPR (Banks et al., 2002), WS-Reliability (Evans et al., 2003)) are message-oriented, requiring, for instance, buffering (or logging) and retransmission to provide reliable delivery of messages. These solutions involve resending whole messages, which is quite inefficient when messages are large. Also, message-based solutions cannot easily offer reliability to long-standing connections. As an example, in AJAX (Garrett et al., 2005) environments, 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, as the HTTP client must be able to repeat requests to obtain the missing parts of responses, whereas the HTTP server must be able to identify and handle repeated requests.

A stream-based solution that buffers and resends unconfirmed data is a much cleaner solution, because it does
not require resending whole messages and does not need changing the application’s semantics. Moreover, from the application perspective, there is no need to explicitly store and resend complete messages; the application can just rely on the channel (and associated middleware). We can find this type of approach in previous work (Zandy and Miller, 2002), including our own (Ivaki et al., 2014b). Our Session-Based Fault-Tolerant (SBFT) design solution (Ivaki et al., 2014b), which we describe in Section 3, can 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. This pattern provides a foundation for reliable communication in TCP-based applications, but does not tackle the specific challenges of the HTTP applications.

A simple reliable stream is unworkable in HTTP, due to the frequent presence of proxies between clients and servers. Proxies are generally unable to deal with 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 the client to the proxy, and from the proxy to the server) and this discards simple recovery mechanisms, which only work after a single connection between client and server (e.g., (Ivaki et al., 2014b) or (Zandy and Miller, 2002)). Finally, legacy non-reliable peers must still be able to communicate with reliable peers (i.e., peers using the reliable communication mechanisms), which can be difficult to achieve when control messages (i.e., messages used by the reliable communication mechanisms) need to be transmitted. This is especially important, given the outsandingly large base of legacy software comprising the web. We go through these problems and tackle them in Section 4.

Recently, in (Ivaki et al., 2015), we proposed an initial stream-based design solution, based on SBFT, for reliable HTTP communication on the web. In this paper, starting in the new Section 5, we extend this latter work, by refining the design solution for clarity and consistency; by adding a developer’s view on the usage and application of our solution (at the server and also at the client-side); and by extending the discussion regarding the solution’s characteristics and observed performance. Overall, we provide a greater level of technical detail and a stronger link to practitioners needs.

Our solution features two key characteristics when compared 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 control channel is used by client and server to exchange acknowledgments. Recovery from connection crashes requires resending lost bytes only, instead of entire application-level messages. We also ensure that all control messages exchanged comply with the HTTP protocol and that reliable and non-reliable HTTP peers can still interoperate. As discussed, this latter aspect is of critical importance in the web environment.

We carried out an experimental evaluation applying our solution in the Apache Tomcat 7.0.13 (Goodwill, 2002) HTTP connector used by JBoss AS 7.1.1 (Fleury and Reverbel, 2003). Results show the applicability, correctness and efficiency of our design solution, but most of all show 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 on reliable communication. Section 3 introduces SBFT. Section 4 introduces challenges and solutions for achieving reliable HTTP communication. Section 4 presents our design solution and Section 6 presents the experimental evaluation carried out with an implementation of our design solution. Finally, Section 7 concludes this paper.

2. RELATED WORK

Developing distributed applications that resist to connection crashes is a quite difficult problem. In the literature, we can find a large number of attempts, which generally fit in the following categories: i) replacement of the transport layer protocol (i.e., replacement of TCP); ii) server-side replication schemes (that allow overcoming server failures); iii) attempts focused on adding reliability mechanisms to HTTP; and iv) solutions based on design patterns for reliable distributed computing. The next paragraphs describe previous work precisely in each of these categories, we also briefly introduce our own previous work to provide further context.

Some of the approaches in the literature opt to replace the transport layer protocol as a means to achieve reliable communication. SCTP (Stewart and Metz, 2001), like TCP, offers a bi-directional, connection-oriented, and reliable transport service to applications communicating over an IP network. It inherits many of TCP’s features, and any application running over TCP can be ported to run over SCTP without loss of function. The main differences revolve around SCTP’s support for multi-homing and partial ordering. Multi-homing enables an SCTP host to establish a session with another SCTP host over multiple interfaces identified by different IP addresses. Concurrent multi-path Stream Control Transmission Protocol (cmpSCTP) (Liao et al., 2008) modifies SCTP to exploit its multi-homing capability by choosing best paths among several available network interfaces in order to improve data transmission rate. Multipath TCP (Scharf and Ford, 2013) uses a similar approach as SCTP to tolerate connection crashes. However, SCTP and its extensions like cmpSCTP, are still not widely used first because the applications must be modified in order to use them, and second because SCTP packets cannot pass through various NATs or firewalls. Moreover, with these solutions, there is no way to recover from connection crashes
if replicated connections fail due to, for instance, an internal network crash.

RSocket (Ekwall et al., 2002) is a session layer solution designed to overcome the limitations of TCP. It uses an extra level of buffering and acknowledgments to ensure delivery of the bytes stream. The RSocket acknowledgments are achieved with the use of an additional UDP control channel. When a TCP connection fails, the client sets up a new connection to resume data exchange. Reliable Sockets (Rocks) and Reliable Packets (Racks) (Zandy and Miller, 2002) are also proposed to address TCP’s limitation to recover from connection crashes, with particular emphasis on mobile users and services, while using only user-level mechanisms. Their goal is to achieve transparent network connection mobility. Both systems can detect connection crashes, keep the state of the endpoint involved in the failure and automatically reconnect (be able to recover the data that was in transit). As with RSocket, they use a control socket to exchange control messages, mainly for the sake of detecting the data connection failure.

FT-TCP (Alvisi et al., 2001) is based on the concept of wrapping, in which a layer of software surrounds the transport layer and intercepts all connections. Data can come from two points, either the IP layer or from the application layer. A logger is used at these points to maintain the current state of the TCP connections. Thus, when the server crashes, the logs are used after restarting the server or moving the server to another host for recovery of TCP connections.

Server-side replication mechanisms try to overcome failures occurring at the server-side. An example is ER-TCP (Shao et al., 2008), which tolerates failures occurring in the TCP connections at the server, by replicating them among the server nodes in a cluster. ER-TCP employs a logging mechanism with active replication to avoid the inconsistency problem that may occur when the replicas do not have the same processing speed as the primary server.

HydraNet-FT (Shenoy et al., 2000) provides an infrastructure to dynamically replicate services across a network and have the replicas provide a single fault-tolerant service access point to clients. HYDRANET-FT uses TCP with a few modifications at the server side to allow: a) one-to-many message delivery from a client to service replicas; and b) many-to-one message delivery from the replicas to the client.

HotSwap (Burton-Krahn, 2002) provides fault-tolerance at the TCP level, by modifying the system call library of the Linux operating system. It creates two identical instances of the same set of programs on two independent machines, a master and a backup. The master and backup systems must start at the same time with identical file systems to ensure they receive the same input from local files. When a TCP related system call (e.g., creating a TCP socket) is made by an application, another replica socket is created at the other host, to tolerate possible crashes of the former one.

ST-TCP (Marwah et al., 2003) is a mechanism that also uses replication to tolerate TCP connection failures occurring when there is a crash of a replicated server instance. It extends TCP to tolerate connection crashes occurring due to server crashes. The approaches used in this set of solutions, that are based on replication, involve a significant cost due not only to the overhead of the replication mechanism itself but also due to the necessary modifications in operating systems.

Several solutions target the HTTP protocol and essentially add reliability mechanisms on top of HTTP. HTTPR (Banks et al., 2002) is one of these solutions. It was put forth by IBM and aims 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.

Web Service Reliability (Evans et al., 2003) (WS-Reliability) is a Web Services specification that is used for exchanging SOAP (Cerami, 2002) messages, which are typically delivered over HTTP, while providing 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-ReliableMessaging (Bilorusesets et al., 2005) is a competing standard that serves similar purposes, including providing the same set of delivery guarantees.

CoRAL (Aghdaie and Tamir, 2009) is a fault-tolerance solution for web-based services, which is based on connection replication and application-level logging mechanisms. In CoRAL, the state of the TCP connection is preserved using active replication, where the TCP stacks of both primary and backup server process incoming packets simultaneously. Message logging is used at the application layer to log HTTP requests and replies, so that the backup server can use them for replaying purposes, if necessary.

iSAGA (Dutta et al., 2001) saves actions, carried out by users in web sites, in stable storage to be able to recover its state after crashes. When the client recovers from crashes, there are no guarantees regarding execution semantics on the server side, since the recovered state might, or might not, be the latest state presented to the user.

EOS2 (Shegalov and Weikum, 2006) 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. All these HTTP mechanisms are message-oriented, which is quite inefficient when a message is long (e.g., a file, or the typically verbose SOAP message). Moreover, ensuring that the browser-server pair can successfully retry an HTTP request is a very challenging task.

The idea of using design patterns for distributed computing started more than two decades ago (Gamma et al., 1994). For example, the Acceptor-Connector pattern (Schmidt, 1996) tries to simplify the design of connection-
oriented applications, by separating event dispatching from connection setup and service handling. However, the Acceptor-Connector pattern does not properly handle multiple connections and is, therefore, unfit for modern servers. To improve the efficiency of the Acceptor-Connector pattern, the Leader-Followers pattern (Schmidt et al., 2000) dispatches events using a fixed number of threads.

Among a huge number of design patterns that we can find in the literature, the ones that address reliability issues in distributed communication are missing. In fact, in the last decade the researchers have assisted to organize distributed interactions into a set of design patterns, first for Enterprise Application Integration (Hohpe and Woolf, 2003), and more recently towards SOAP/WSDL and RESTful web services (Daigneau, 2011). 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 works, our focus is on the lower level details of the interactions, which can provide a more efficient manner to achieve reliable communication (e.g., there is no need to focus on entire application-level messages). There are several works in the literature addressing some aspects of dependability (Avizienis et al., 2004), such as safety (Gawand et al., 2011), security (Laverdiere et al., 2006; Schumacher et al., 2013; Yoshioka et al., 2008), and fault-tolerance (Hammer, 2013). There are also several works related to scheduling algorithms in real-time systems (Douglass, 2003). However, none of these design patterns addresses the reliability issue in a distributed communication.

In previous work, we proposed a design pattern named “Fault-Tolerant Multi-Threaded Acceptor-Connector” (FTMTAC) (Ivaki et al., 2014a), which combines the Acceptor-Connector and Leader-Followers patterns with a custom buffering mechanism that is used to tolerate TCP connection crashes. Later, we moved the fault-tolerance mechanisms of FTMTAC to work below the service handler and proposed a “Session-Based Fault-Tolerant Design Pattern” (SBFT) (Ivaki et al., 2014b). This basically moved the fault-tolerance mechanism from the application layer into a session layer presented as a socket. We adapted the session-based fault-tolerant design pattern, which is a stream-based solution, and created an initial solution to overcome the challenges associated with HTTP communication in the web environment in (Ivaki et al., 2015). In this paper, we extend the work in (Ivaki et al., 2015) by, in short, having matured the design (please refer to Section 5 for further details) and adding a developer’s perspective on the application and usage of the middleware implementing our solution.

3. THE SESSION-BASED FAULT-TOLERANT DESIGN PATTERN

The main goal of the session-based fault-tolerant design pattern (SBFT), proposed in (Ivaki et al., 2014b), 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, enables recovery from connection crashes with minimal overhead. The main components of SBFT involved in the recovery process are named Connection Handler, Stream Buffer, and Connection Set. The Connection handler provides an interface to implement the actions that are required to be taken for establishing a connection, for reconnecting, and for retransmitting lost data. Each Connection Handler owns one Stream Buffer to store the data sent, so that retransmission is possible. The Connection Set keeps references of all open connections and enables their replacement with new connections, if necessary.

In this section, we review the buffering scheme used in SBFT. At the heart of this scheme, there is one crucial component, the Stream Buffer, a circular buffer that discards the need for an extra layer for acknowledgments over TCP. 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 \( n \).

\[
\begin{align*}
\text{Figure 1. Buffers in a Simple Client-Server Scenario} \\
\end{align*}
\]

It is possible to limit the size of the application buffer if we know that \( m \) bytes are read by the receiver, which allows us to delete these \( m \) bytes from the sender buffer. Let us assume that the size of the underlying TCP send buffer is \( s \) bytes, whereas the TCP receive buffer of the receiver has \( r \) bytes. Let \( b = s + r \). If the sender writes \( w > b \) bytes to the TCP socket, we know that the receiver got at least \( w - b \) bytes.

As an example, assume that \( b = 20 \) bytes. If the sender wrote 21 bytes to the TCP socket, at least 1 byte got through to the receiver application. This means that the sender only
needs to keep the last \( b = s + r \) sent bytes in a circular buffer, and may overwrite the data that is older than \( b \) bytes. Note however that, apart from these limits, the buffers can have arbitrary sizes, according to the sender plus the receiver TCP buffer sizes. By using this mechanism, we can simply avoid explicit acknowledgments of the received bytes. In practice, to implement this idea, the peers have to exchange the size of their receive buffer, through a handshake process, right after establishing the connection and before exchanging any data.

The shortcoming of using this 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.

4. ACHIEVING RELIABLE HTTP COMMUNICATION

In this section we 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 that were chosen to integrate our design solution.

4.1 Challenges in the Web Environment

In the web environment, the scenario of direct client-server communication can be rare. Intermediate proxy nodes may stand between the HTTP client and the server and this brings in technical challenges when the goal is to have reliable communication. These nodes may be serving 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 solution 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’s TCP send buffer, the proxy’s TCP receive buffer, the proxy’s internal state, the proxy’s TCP send buffer, and the receiver’s TCP receive buffer. The size of the buffers is now \( b_1 + b_2 + b_3 + b_4 + b_5 \), much more than the \( b_1 + b_5 \) that the SBFT’s Stream Buffer is 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 resent in case of a 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 described in Section 2 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.

Finally, both non-reliable and reliable clients and servers must be able to interoperate. Hence, the design of a solution for reliable communication must ensure that interoperation 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.

4.2 Tackling the Web Challenges

To meet the first challenge (i.e., extra data buffering), we combine the use of explicit and implicit acknowledgment mechanisms. If no proxy exists, client and server can rely on the implicit buffering of SBFT. In this scenario, since the size of the Stream Buffer is greater or equal than the sum of the size of the TCP send buffer of the sender with the size of the TCP receive buffer of the receiver, the data in transit is always guaranteed to be at 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. We designed a handshake procedure that is vital to determine whether there is any proxy in the path between the client and server. In order to exchange acknowledgment messages when there is a proxy, a control channel is created and used by our design solution.
With the goal of meeting the second challenge (i.e., possible filtering of non-HTTP messages) we strictly adhere to the HTTP message format for handshake and acknowledgment messages, which are necessary for the operation of our reliable communication mechanism.

To meet the third challenge (i.e., interoperability with legacy peers), we detect the presence of legacy peers through the handshake procedure. Then, we automatically and transparently switch from the mechanisms that provide reliable communication to the plain transport handle (e.g., the TCP Socket) that is used by any non-reliable peer.

In the following paragraphs, we explain the details of our handshake procedure, which is vital for achieving all the above goals. It serves to detect the presence of a proxy, it allows messages to pass through proxies, and it allows to understand if the other peer involved in a particular interaction is a legacy peer or is able to use our reliable communication mechanism.

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 have several lines that carry necessary information for the handshake. Each line ends with the regular separator (\r\n or CRLF).

![Figure 3. The Handshake Procedure for Reliable HTTP Communication](image)

As we can see in Figure 3, the **client handshake request** includes several headers. The **FT Identifier** header carries the identifier of the connection (0 means that the connection is brand new), which is used at the server side to identify if the connection is new, or if it is for recovery. For setting up a new connection, the client sets the identifier to 0, and the server generates a new immutable identifier for the connection in the response. When a connection failure occurs, a reconnection is attempted and the client sends the identifier (previously generated by the server) to the server.

The **FT Connection** header carries the network address of the client and server, which are used at the server side to identify if there is any proxy. The server detects the presence of a proxy if the address sent in the **FT Connection** header is different from the remote address of the TCP connection it owns. Identifying the existence of proxy is necessary for the peers to adapt their buffering and acknowledgment mechanisms. Thus, the **FT Buffers** (refer to the Figure 3) in the client handshake request, not only carries the size of the TCP send buffer that is used at the server to calculate the size of its Stream Buffer, but also the size of the TCP receive buffer that might be necessary (i.e., when there is a proxy in the path between client and server) to calculate the size of client’s Stream Buffer too.

The Stream Buffer of a given peer is empty when the pointer to the end of the buffer points to one position before the beginning of the buffer; and it is considered to be full when the pointers to the beginning and end of the buffer point to the same place. In the scenarios with proxy, the peers need to update these pointers that mark the beginning and end of their Stream Buffer, respectively after each write operation and receiving each acknowledgment. This allows to understand if the buffer has enough space for new data to be sent.

Whenever a Stream Buffer is becoming full the peer should acknowledge reception of data to allow the sender to release some space in its buffer (which technically means to move the end pointer forward) to be able to send the next data without interrupt. For example, consider that a server is sending a large file, whose size is greater than its buffer size, to its client. If the server does not receive an early acknowledgment from the client, its buffer becomes full and it needs to wait for the acknowledgment to release some space and send the rest of the file. To enable early acknowledgments, once a peer receives a number of bytes equal or greater than half the size of the peer’s Stream Buffer, an acknowledgment must be sent. This allows the sender peer to clean its buffer, thus allowing it to proceed. Implementing this idea is possible because the peers can exchange the size of their send and receive buffers, allowing the other peer to simply calculate the size of the remote Stream Buffer.

The **server handshake reply** also includes three key headers. The **FT Identifier** carries the unique identifier (1 in the Figure 3) of the connection, which is generated by the server and serves to identify a particular connection. The **FT Proxy** informs the client whether a proxy was or was not detected by the server, being set to True or False, respectively. If no proxy was detected, client and server can simply rely on the implicit acknowledgment scheme, otherwise the buffering and acknowledgments scheme must become explicit. Finally, similarly as with the client request, the **FT Buffer** carries the sizes of the server’s TCP send and receive buffers.

Using the above handshaking scheme, a legacy client will obviously simply not receive any handshake message from the server because the reliable server understands that the client is legacy (i.e., it does not receive a handshake request), so both peers just follow the HTTP protocol and use the TCP Socket as usual. In contrast, after a legacy server receives a handshake message from a reliable client, it will ignore the handshake headers, as it is not ready to process them and, in
the likely case the URL does not map to a server resource, the server will reply with a page not found code. The client will easily understand that the server is not reliable and will stop trying to use the reliable communication mechanism. Hence, all the combinations of legacy/reliable client and server work.

In addition to the above mentioned handshake headers serving for establishing a fresh data connection, we may need additional headers to carry information during communication. As previously mentioned, to send explicit acknowledgments, our solution uses one control connection, shared by all client connections to a given server. This control channel is a standard TCP connection to the server’s HTTP port and is created by the client after the above handshake procedure, whenever a proxy is detected. Upon creation of a new control connection, a handshake message including the FT Control header is sent by the client through this control connection, which allows the server to distinguish between a data connection and a control connection. The acknowledgment messages sent through the control connection respect the same format as the handshake messages and include the FT ACK header, which carries the identifier of the connection and the number of bytes that was read so far.

After a connection crashes, when a new connection is created for recovery purposes, the handshake request and response messages include the FT Recovery header, which also carries the identifier of the connection and the number of bytes that was read so far. This allows the peer to identify how many bytes must be retransmitted.

5. DESIGNING RELIABLE HTTP APPLICATIONS

In this section we present the design that has been created to handle the key web challenges discussed in the previous section. We present the design by describing it in three views: a static architecture view of the components; a dynamic representation of the components collaboration; and a developer point-of-view on the application and usage of the middleware.

Figure 4 illustrates our design solution, which we now overview and will describe in detail. As we can see in Figure 4, the components are organized in three key layers: application, session, and transport. The application layer implements business logic and application services. It uses the session layer to contact the transport layer, which is ultimately responsible for exchanging data with the other peer. In this scenario, the session layer, is a middle layer responsible for properly handling possible connection crashes that may break the HTTP communication. Thus, the session layer holds the core of our mechanism for tolerating connection crashes and plays a central role.

The application layer includes an HTTP Client, which implements the actions to start a connection to the server, and then to initialize and activate a service handler. Service Handler implements application services and business logic, and can play two different roles in client and server. For this reason, we have two different components which extend the service handler, namely Service Handler A and Service Handler B. The HTTP Server component may own one or more passive handles (e.g., a Java TCP ServerSocket) to check for the arrival of new connection requests.

In the transport layer, there is a Transport Handle that provides an interface for upper layer mainly to create a connection, write data, and read data. A well known example of a typical transport handle is a TCP Socket. The Passive Transport Handle is a passive-mode (i.e., it passively waits for connection requests and is not used for data exchange) transport handle that is bound to a network address (i.e., an IP address and a port number) and is used by a server to receive and accept connection requests from clients.

In the next subsections, we first detail the components in the session layer, we then explain how the components collaborate, and finally we describe how developers can apply and use our solution.

5.1 Components of The Session Layer

Reliable Transporter is the central component in the session layer. Each Reliable Transporter owns one Stream Buffer and extends the functionalities of the Connection Handler to enable recovery from connection crashes. It implements the actions necessary to establish a connection for the first time and also after a crash, including the handshake, reconnection and retransmission of the lost bytes. The connection establishment process is different on the client and server sides. Even when a connection crashes, the initiative to reconnect always belongs to the client’s Reliable Transporter, due to NAT schemes or firewalls. Thus, the actions of the Reliable Transporter in the methods handshake() and reconnect() need to be done differently for the client and server.

Moreover, each Reliable Transporter owns one Control Connection Handler when the communication involves proxies. Each Control Connection Handler is shared between all connections created from the same client. In the scenarios with proxies, the Reliable Transporter also needs to keep the size of the remote Stream Buffer (remoteBufferSize) and the number of bytes read so far after the last acknowledgment (numOfBytesReadAfterLastAck). Figure 4 presents the Reliable Transporter, its attributes and connected components.

The Stream Buffer implements the actions to save, retrieve, and clean the data. As shown in Figure 4, the stream buffer owns an array of bytes (buffer), pointers to start and end of the buffer, and a boolean property, namely write_constraints, which indicates whether the buffer needs to keep the pointer to the end of the buffer (i.e., used in the communication scenarios with proxy). Methods put() and get() are used respectively to save data (i.e., an array of bytes) to the buffer and to return data that was recently sent.
Methods has_space() and release_space() are used in the scenarios with proxy to check whether the buffer has enough space to write over old data (i.e., acknowledged).

Besides keeping information of connections, the Connection Set serves to synchronize threads upon connection crashes and reconnections. Once a thread associated to a data connection handler tries to replace a failed connection, it must wait on the method take_new_connection() of the connection set until some other thread comes in with a new handler invoking the method deliver_new_connection(). The information of a connection is removed from the set when the connection is closed.

The Passive Reliable Transporter simply owns one passive transport handle (e.g., TCP Server Socket) and provides an interface to the HTTP server to create a passive handle, accept connection requests, return new Reliable Transporter, and close the handle.

5.2 Collaboration Between the Components

Figure 5 presents the collaborations between the components in a failure-free scenario. The figure includes the following three parts, which we explain in the next paragraphs: a) data connection establishment and service handler initialization; b) communication in a scenario without proxy; and c) communication in a scenario with proxy.

To initialize connections, the server creates one (or more, depending on the number of ports defined and assigned to the application server) Passive Reliable Transporter and binds it to the local network address (IP address and port number). Then the server waits for a new connection, by invoking the method accept() of this passive handle. On the other side of the communication, the client initializes a Reliable Transporter by giving the network address of the server in order to establish a new connection. This will internally create a Transport Handle.
Upon reception and acceptance of a connection request in the server, a Reliable Transporter is generated. Right at this point, a handshake procedure is taken place to complete the initialization of the Connection Handler. The client’s Reliable Transporter sends a handshake request including the identifier of the connection (zero in this scenario), the local and remote address of the connection, and the size of its TCP’s send and receive buffers.

The server’s Reliable Transporter, identifies that the connection is new (because the identifier is zero), and

**Figure 5. Components Interactions in a Failure-free Scenario**
registers itself into the Connection Set through the method `register_handler()`, which returns a unique identifier. It can also identify the existence of a proxy using the given information about the local and remote addresses and comparing them with its own information about the connection.

A handshake reply is sent back to the client, which includes the unique identifier of the handler, the size of the buffers on the server side, and information about the existence of a proxy. At this point, both, client and server, can initialize their Stream Buffer with the appropriate configuration, depending on the information exchanged between them.

When no proxy exists, the peers initialize and activate service handlers, by passing the previously created Reliable Transporter (`rt` in the client and `r1` in the server). This means that the client and server’s Service Handler can start writing and reading data. After a successful write operation, the Reliable Transporters put the data into the Stream Buffer and update the value of `written_data`. After a successful read operation they update the value of `data_read` (refer to part (a) of Figure 5).

In contrast, when there is some proxy, the Reliable Transporters require a control connection to exchange acknowledgment messages in both sides (Refer to part (b) of Figure 5). Since the control connection is shared between several connections created by the same client, client and server check the Control Connection for an existing connection, by specifying an identifier that is equal to their peer’s address. If a connection already exists, they simply get it from the list and use it, otherwise the client must create a new one. When a control connection is successfully created, the client sends a handshake request including the `FT Control` header with the local address of the client, which will be used by the server as the identifier of the control connection. The server sends a handshake reply back to the client including the `FT Control` header with the IP address used by the server, which will be used as the identifier of the control connection on the client side. Both client and server store the reference of the control connection in a list (`ctrlConnections`), to be used with other Reliable Transporters if necessary.

The exchange of data is quite different when there is a proxy. The Reliable Transporter checks if there is enough space in the Stream Buffer before writing the data, and checks if the number of bytes read, after the last acknowledgment message, exceeds the half of the remote buffer. If so, it sends an acknowledgment through the control connection. Figure 5, part (c), shows a scenario where an acknowledgment is sent from the client. As shown in the figure, this message carries the identifier of the connection handler and the number of bytes read so far. The Control Connection Handler delivers the read message to the appropriate Reliable Transporter, which is accessed by means of the Connection Set, through the method `notify_ack()`. This lets the Reliable Transporter release some space from the Stream Buffer.

Figure 6 presents failure handling details. Once a Reliable Transporter fails completing a read or write operation, it transparently tries to reconnect. The reconnection is accomplished differently in the client and server’s Reliable Transporter. As shown in the figure, neither the client’s Service Handler, nor the server’s are involved on the recovery procedure, to ensure the separation between service and failure handling.

When a failure occurs, both sides will eventually start the reconnection phase, by calling the method `reconnect()`. Upon invoking this method, the client’s Reliable Transporter tries to create a new connection to the server during a predefined period of time. On the other side, the server’s Reliable Transporter waits for a new connection, by giving the connection identifier and a waiting time to the Connection Set through the `get_event()` method.

Figure 6. Components Interactions in the Presence of Connection Crashes

After acceptance of a connection request and creation of a new handler, the client’s Reliable Transporter starts the handshake protocol. It uses a predefined message format with an `FT Recovery` header, consisting of the identifier of the failed connection, and the number of bytes received on the client side. This lets the server distinguish a fresh connection from reconnection. The server side accepts the new connection and initializes a new Reliable Transporter. This component is then responsible for notifying the failed handler through the method `put_event()` of the Connection Set. Then, the server’s Reliable Transporter completes the handshake procedure by sending a message back to the client. Then, both sides start retransmission of data lost due to connection crashes.

5.3 Using The Design Solution for Developing Reliable Applications

Our design solution can be implemented using any programming language in any platform. It defines a very simple API that the applications can use for achieving reliable communication. As a proof of concept, we have already implemented the design solution in Java and named this
middleware for reliable communication as FSocket (current version is 1.2). FSocket can be used in any application implemented in Java just by adding the library package (available at https://sourceforge.net/projects/fsocket_v12/) to the application’s classpath and doing very limited and straightforward changes to the source code of the client and server. Developers that are creating a new client application can simply resort to the use of the FSocket API methods, which are quite easy to use, as we illustrate in this section. If the client application already exists, the developers need to replace every Socket object by an FSocket object which implements the actions that are necessary for performing the handshake, temporarily storing the sent data, reconnecting and retransmitting data. In this context, the FSocket class can be regarded as an extension of the Java Socket. In addition to replacing the object that represents the connection, all 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 simple replacements are summarized below:

```java
FSocket fssocket = new FSocket (server,port) // instead of
Socket socket = new Socket (server,port)
int read = fssocket.read(data) // instead of
int read = inputStream.read(data)
fsocket.write(data) // instead of
outputStream.write(data)
```

For server developers, who need to apply reliable communication to their already developed servers, changes are as simple as in the client case. As previously discussed, servers own one passive handle to accept new connections. In Java TCP, this passive handle is named ServerSocket. We have an equivalent passive handle in our implementation, called ServerFSocket. Wherever the server needs to wait for a new connection, it will need to use this object instead of the ServerSocket. When a connection is received, the ServerFSocket returns an FSocket instead of a Socket. This FSocket must then be used for all the read and write operations. The replacements necessary at the server are summarized below:

```java
ServerFSocket serverFSocket = new ServerFSocket(port) // instead of
ServerSocket serverSocket = new ServerSocket(port)
FSocket fsocket = serverFSocket.accept() // instead of
Socket socket = serverSocket.accept()
int read = fsocket.read(data) // instead of
int read = inputStream.read(data)
fsocket.write(data) // instead of
outputStream.write(data)
```

6. EXPERIMENTAL EVALUATION

In this section we present the experimental evaluation carried out to evaluate our solution. The experiments mainly focus on several key aspects: correctness, performance, overhead, and complexity of the solution. To evaluate the correctness, we verified all the key functionality provided by the solution, in particular, tolerating connection crashes, interoperating with legacy peers, interoperating and connection failure tolerance in the presence of proxies.

With the goal of evaluating performance, we measured latency, throughput, overhead, and implementation complexity. Latency refers to the round-trip-time of a request-response interaction, throughput is the number of operations per time unit; to evaluate the overhead of the solution, we measured the CPU and memory usage. Finally, to evaluate the complexity, we measured the following three typical code complexity metrics: Lines of Code (LOC); Cyclomatic Complexity; and Nested Block Depth (Jorgensen, 2008).

In order to execute the experiments, we first applied FSocket (the Java implementation of our solution) to the Apache Tomcat 7.0.13 HTTP connector (Goodwill, 2002) that is included in JBoss AS 7.1.1 (Fleury and Reverbel, 2003). This involved a few modifications in the Apache Tomcat 7.0.13 HTTP connector, as explained in Section 5, which were quite easy to achieve. In practice, we simply replaced every Socket object by an FSocket object and the ServerSocket object by a ServerFSocket. Also, all the read and write operations done on the original TCP socket’s InputStream and OutputStream were replaced with the read and write operations on the FSocket object. Concerning the tests that require the presence of a proxy, we selected a quite popular one, Squid 3.1 (Saini, 2011).

Regarding the infrastructure used to support the experiments, we configured two computers to share the same isolated Local Area Network (LAN) to run the two endpoints (client and server). It is expected that the results can extend to a Wide Area Network (WAN), as the server’s throughput, overhead, and implementation complexity are independent from the network environment. Using an isolated LAN allows us to better control the experimental conditions, as uncontrolled environments can influence the results in unexpected ways.

To simulate the presence of multiple clients, we chose to multithread a single process, using different threads on the client machine. The server obviously did not require changes to support multiple clients, as the codebase is already prepared for this kind of scenario. Table 1 describes the infrastructure used for the experiments and in the next

**Table 1. Systems Used in the Experiments.**

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>OS</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Mac OS X</td>
<td>2.4GHz Intel</td>
<td>4GiB RAM</td>
</tr>
<tr>
<td></td>
<td>version 10.10.1</td>
<td>Core 2 Due</td>
<td>3MiB cache</td>
</tr>
<tr>
<td>Server</td>
<td>Linux 2.6.34.8</td>
<td>2.8GHz Intel(R)</td>
<td>12GiB RAM,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 Cores(TM) i7</td>
<td>8MiB cache</td>
</tr>
</tbody>
</table>

49
sections we describe each of the three sets of experiments in detail.

6.1 Verification of Correctness

To evaluate the correctness of our solution, we considered different HTTP client-server communication scenarios. In each scenario, we refer to reliable and non-reliable peers (i.e., client or server), respectively, as using or not using our reliable communication solution. The scenarios are as follows: 1) a reliable HTTP client communicating with a non-reliable (legacy) 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. Scenarios 1) and 2) are used to show that our solution is compatible with legacy and unreliable software; and scenarios 3) and 4) are used to show that our design pattern is able to tolerate connection crashes with and without 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 JBoss AS during the tests. For each of the four scenarios, we let client and server exchange messages during 5 minutes (each test was repeated 10 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 crashes (scenarios 3 and 4) without and with proxy, we used TCPkill to cause connection crashes at random instants during each test (three crashes per test), and observed that all interactions worked correctly even in the presence of the crashes and all expected messages were correctly received.

6.2 Evaluation of Performance

We used a set of scenarios to evaluate performance. The goal is to be able to compare the communication performance of peers using non-reliable communication against peers using reliable communication. We also need to consider the cases where the communication is not direct, but involves an intermediate node (i.e., a proxy.). Thus, this involves the creation of the following 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.

Scenarios 1) and 2) (non-reliable scenarios) are used as baseline. We compare the behavior measured in scenario 1) with the one observed in scenario 3) to understand the overhead introduced by the reliability mechanisms in a direct client-server link. We use scenarios 2) and 4) to understand the impact of the reliability mechanisms in a situation where there is a proxy involved.

The metrics selected to describe performance were latency (round-trip-time of a request-response interaction) and throughput (number of operations per time unit). These metrics are quite useful to understand how well a given system performs its functions and have been extensively used in contexts similar to ours (Antonopoulos and Gillam, 2010; Weil et al., 2006).

In each scenario we exponentially vary the number of clients from 1 to 1000 (i.e., running concurrently), where each client sends 1000 requests. To calculate the latency of the proposed solution, we send a request to the server and calculate the time taken from sending the request to receive the respective reply from the server. Thus, each client waits for the response after sending each request. In our case, we force the server to send an acknowledgment message to the client after the application layer confirmed that the message is processed, and then we divide the time calculated for latency by two. All the results for latency are the average of 1, 000 trials. The latency degradation ($L_d$) is computed by Equation 2, where $r$ and $u$ refer to reliable and unreliable scenarios, respectively:

$$L_d = (\text{Latency}_r - \text{Latency}_u) \times 100 / \text{Latency}_r$$

To calculate throughput, the clients send a large number of requests to the server (1000 in our tests), without waiting for any response (a different thread waits for the response). The server calculates the time taken from receiving the first request to sending the last reply. Similarly, as with latency, we calculate the throughput degradation ($T_d$) using Equation 2 (again, $r$ and $u$ refer to reliable and unreliable scenarios, respectively):

$$L_d = (\text{Throughput}_u - \text{Throughput}_r) \times 100 / \text{Throughput}_u$$

Figure 7 and Figure 8 show the results obtained for latency and throughput. As we can see, latency increases progressively in all cases; the same happens with throughput.

![Figure 7. Latency Observed during the Experiments](image-url)
In the scenarios with proxy, the latency is higher in comparison to the scenarios without proxy. The throughput in all scenarios increases rapidly in the beginning and then stays at the same level, as expected. The main observation is that the throughput of unreliable applications in both scenarios, with and without proxy, reaches the same level, although in the beginning it is slightly higher when there is no proxy. This does not happen for the reliable application. This difference is caused by the extra control connection and extra actions taken in FSocket when a proxy exists. However, the important aspect for both latency and throughput is that, when we compare the scenarios that use reliable peers with those that use the non-reliable peers, even with proxy, performance degradation shows low values (about 3 percent). In fact, although we have all necessary mechanisms for reliable communication in place and in operation, performance degradation is quite small.

We also performed some tests to understand the performance of the reliable stream-based application in comparison to a reliable message-based application. In these tests, we used two versions of the same application: a messaged-based one, using a message-based middleware implementing our recovery mechanism but retransmitting complete messages after reconnection, and a stream-based application, using FSocket. These applications contain three main operations, Invoke1, Invoke2 and Invoke3. These operations receive a small request and return a small response. The major difference between them is that Invoke1 replies immediately, Invoke2 sleeps 1 millisecond (ms) before replying, whereas Invoke3 sleeps 2 ms. The reason why we used different sleep times essentially is to emulate scenarios where processing time is negligible or where there is some processing.

Figure 9 shows the results obtained for latency of both stream-based and message-based applications. In both applications and with all invocations, the latency increases smoothly with the increasing number of clients. We see three different levels of latency for different invocations, due to the different processing time of the invocations.

Figure 10 shows the results obtained for throughput in these applications. Unlike latency, the throughput increases rapidly in the beginning, with the increasing number of clients, and then pretty much levels out. However, our main observation is that we obtained better performance for the stream-based application than the message-based application for both latency and throughput. We found out that the main source of overhead in Messengers is associated with the transformation of messages into arrays of bytes (e.g., serialization) at the sending point and to the transformation of arrays of bytes into messages (e.g., deserialization) at the receiving point. It is worth mentioning that the maximum difference in performance belongs to the Invoke1 and when the number of the clients is very low (76.13% for latency and 95.61% for throughput), which is the worst case for measuring and comparing the performance, and the minimum difference belong to the Invoke3 (9.47% for latency and 1.94% for throughput).

Regarding the recovery time in these two applications, we performed a simple analysis. The recovery process in both applications is the same and it includes the reconnection time plus retransmission time of the lost data. The reconnection phase is done independently from the data type, being...
irrelevant in this analysis. In contrast, the retransmission time directly depends on the size of date to be resent. Assuming \( t \) as retransmission time of a lost message in the message-based application, the retransmission time in the stream-based application would, in average, be \( t/2 \) because it varies from 0 to \( t \) depending on the size of the part of the message that is lost.

6.3 Evaluation of Implementation Complexity and Overhead

With the goal of analyzing the complexity of our solution, we measured three important code complexity metrics: Lines of Code (LOC), Cyclomatic Complexity, and Nested Block Depth (Jorgensen, 2008). To accomplish our evaluation, we implemented the following three versions of a simple HTTP client-server application:

- A plain HTTP application, without no reliability mechanisms;
- A reliable HTTP application using SBFT;
- A reliable HTTP application using our design solution for reliable HTTP communication.

Table 2 compares these three versions in terms of code complexity. The metrics 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 solution for HTTP. If we consider the, the average Cyclomatic Complexity per method, we can see that the first two cases fit in the 1.7-1.8 range, while it increases by a small amount to 1.9 for our design solution for reliable HTTP communication. 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 this purpose.

To evaluate overhead, we set each HTTP client to send 100 requests per second to the server during 5 minutes, which we experimentally observed to be enough to show the usage of resources. Then, we ran the \( ps \) command to periodically read memory and CPU occupation on the server. Figure 11 and Figure 12 show that the overhead is kept under acceptable limits. The memory used by our reliable server is, as expected, higher than the non-reliable one, with a maximum overhead of 60%, due to the extra buffering placed on top of TCP. The CPU overhead is again quite low (maximum of 15%), which is an excellent indication, as this resource can be many times of critical importance. Moreover, we can see that both CPU and Memory Usages are higher in the scenarios with proxy. This overhead is caused by the extra control channel and extra messaging (e.g., acknowledgment messages) of FSocket when proxies exist.

<table>
<thead>
<tr>
<th>Applications</th>
<th>LOC</th>
<th>Cyclomatic Complexity</th>
<th>Nested Block Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain HTTP App</td>
<td>572</td>
<td>1.74</td>
<td>1.28</td>
</tr>
<tr>
<td>Reliable HTTP App with SBFT</td>
<td>1057</td>
<td>1.77</td>
<td>1.40</td>
</tr>
<tr>
<td>Reliable HTTP App using HTTP-based Pattern</td>
<td>1373</td>
<td>1.95</td>
<td>1.40</td>
</tr>
</tbody>
</table>

7. CONCLUSION

Achieving reliable communication in the web environment can become an extremely difficult task. The lack of effective and practical solutions leads developers to create custom, error-prone, solutions, often depending on special hardware or software libraries and compatible peers. In this paper, we presented a stream-based design solution for HTTP that is able to overcome TCP connection crashes. Our solution is able to provide reliable communication over HTTP proxies and enables interaction with legacy peers. All the reliability mechanisms we use adhere strictly to the HTTP protocol, this meaning that our control traffic will not be stopped by any content-based filtering proxies residing between client and server. We carried out an experimental evaluation using an implementation of our design pattern in the popular JBoss Application Server. Results show that this approach introduces a small overhead, while ensuring that
network glitches do not prevent service delivery, even when intermediate nodes are present in the communication. Thus, we believe, it can help developers to create more reliable distributed applications. As future work, we plan to generalize the design pattern to other communication protocols and to enable recovery from endpoint crashes.

8. REFERENCES


Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.


Naghmeh Ivaki is a PhD student at the Department of Informatics Engineering, University of Coimbra. She obtained her Master of Science in Information Technology Engineering at the Faculty of Engineering in the Tarbiat Modares University (TMU), Tehran, Iran, December 2008. Her PhD work is on the dependable distributed systems domain. She has authored more than 10 papers in international conferences and workshops.

Nuno Laranjeiro received the PhD degree in 2012 from the University of Coimbra, Portugal, where he currently is an assistant professor. His research focuses on robust software services as well as experimental dependability evaluation, web services interoperability, services security, and enterprise application integration. He has authored more than 40 papers in refereed conferences and journals in the dependability and services computing areas.

Filipe Araujo is an Assistant Professor at the University of Coimbra, Portugal. He received his graduation in Electrical Engineering in 1996 and his Master of Science in Informatics Engineering in 1999, both from the University of Coimbra. He received his PhD in 2006 from the University of Lisbon. His current research interests are focused on parallel, grid and cloud computing. He participated in several national and international projects.