ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/265341985

Session-Based Fault-Tolerant Design Patterns

Conference Paper · December 2014

DOI: 10.1109/PADSW.2014.7097875

CITATIONS
8

READS
824

3 authors:

Naghmeh Ivaki
University of Coimbra
37 PUBLICATIONS **145** CITATIONS
SEE PROFILE

Filipe Araujo
University of Coimbra
111 PUBLICATIONS **863** CITATIONS
SEE PROFILE

Fernando Barros
University of Coimbra
92 PUBLICATIONS **880** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

FCT Project Advanced Analytics for Empirical Assessment of Cloud Resilience View project

METRICS: Monitoring and Measuring the Trustworthiness of Critical Cloud Systems View project

All content following this page was uploaded by Naghmeh Ivaki on 11 October 2014.

The user has requested enhancement of the downloaded file.

Session-Based Fault-Tolerant Design Patterns

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

***Abstract*—Despite offering reliability against dropped and reordered packets, the widely adopted Transmission Control Protocol (TCP) provides nearly no recovery options for long-term network outages. When the network fails, developers must rollback the application to some coherent state on their own, using error-prone solutions. Overcoming this limitation is, therefore, a deeply investigated and challenging problem. Existing solutions range from transport-layer to application-layer protocols, including additions to TCP, usually transparent to the application. None of these solutions is perfect, because they all impact TCP's simplicity, performance or ubiquity, if not all.**

To avoid these shortcomings, we contain TCP connection crashes inside a single session layer exposed as a sockets interface. Based on this interface, we create a blocking and a non-blocking fault-tolerant design pattern. We explore the blocking design in an open source File Transfer Protocol (FTP) server and perform a thorough evaluation of performance, complexity and overhead of both designs. Our results show that using one of the patterns to tolerate TCP connection crashes, in new or existing applications, involves a very limited effort and negligible penalties.

***Index Terms*—TCP, Connection Failure, Fault-Tolerance, Session Layer, Design Pattern**

I. Introduction

Most applications depending on long-lived reliable connections resort to the Transmission Control Protocol (TCP) [1] to transport their data. The most common example is, perhaps, web browsing, but video streaming (often in a browser), file transfers [2], and remote shell interactions with the Secure Shell Protocol (SSH) [3] are also common. However, reliable interactions in TCP cannot span across different connections. When a connection crashes, applications tend to abort, occasionally exposing odd behaviors. For general applications, overcoming TCP disconnections is difficult, even when peers are still running. If the TCP layer throws an exception, the application might have no means to determine which data did or did not reach the other endpoint, thus making recovery or roll-back to a coherent state impossible.

The TCP unreliability problem is widely acknowledged in the literature, where we can find many attempts to tolerate TCP connection and endpoint crashes. Some of these try to use multiple alternative paths between the client and the server [4], [5], others chose to replicate components [6], [7], checkpoint the sate of the connections [8] or use a middle layer to intercept TCP system calls [9], [10], [11].

Despite their merits, we can point out limitations in all the available solutions. E.g., MultipathTCP [4] is orthogonal to the problem we are considering and does not overcome disconnections, despite offering more than one path between peers (if available). Other solutions require replication and are therefore quite expensive. The solutions allowing the TCP API to be unchanged seem appealing, but they actually change the semantics of the application with unforeseen consequences. Furthermore, these solutions are often not mature or not available for all the computational platforms. Many other options exist, but their sheer number and diversity demonstrate the importance and the difficulty of providing reliability *across* TCP connections.

In this paper, we argue that ensuring reliability across TCP connections is a recurring need in software design. Despite the availability of newer protocols, including vastly popular ones, such as the HyperText Transfer Protocol (HTTP) [12], TCP is either lying underneath or used as the main player for certain interactions, such as multimedia streaming. On the other hand, TCP owns many advantages as well. Since it works at a lower layer than the alternatives we mentioned, it offers control and performance, not to mention a vast knowledge base from developers. For these reasons, we propose a design pattern [13], as a general reusable solution to solve the common problem of TCP reconnection.

The most difficult part of reconnecting is to resend data that was lost in transit. The naive solution to this problem is to use a double buffering layer with acknowledgments and retransmissions. To avoid this cumbersome approach, we use a circular buffer [11] that keeps all data still possibly in transit in the connection. Should the connection fail, the application can use the buffer to resend data on the next connection. The interesting point here is that even with a buffer of limited size, sender and receiver need not exchange any control messages once they set up the connection.

We also resort to several design patterns that emerged for distributed programming in the last decades, namely the Acceptor-Connector [14], to determine the basic interaction between the client and the server, and the Leader-Followers [15], to provide support for multi-threading. These patterns greatly simplify the creation of large-scale distributed systems, by separating application-specific from general components.

We leverage on existing work and design principles to propose two session-based fault-tolerant design patterns. The idea is to expose the fault-tolerant communication as a session layer and use this session layer in both blocking and non-blocking designs, respectively the *Blocking Session-Based Fault-Tolerant* (`bsft`) and the *Session-Based Fault-Tolerant Multi-Threaded Acceptor-Connector* (`sftmtac`).

We implemented these patterns in Java (see [16], [17]) to demonstrate their validity and we evaluated them under heavy load conditions. Our results show the simplicity and the low overhead of these designs, thus demonstrating the viability of solving an important problem in a general way.

The rest of the paper is organized as follows: in Section II we overview related work. Section III describes the session-based fault-tolerant design, which serves as basis for the blocking and non-blocking solutions. We start with the blocking solution in Section IV, because it is simpler, and continue to the non-blocking approach in Section V. In Section VI we present the experimental evaluations and the results. We conclude the paper in Section VII.

II. Background

Given our attempt to create a design pattern that tolerates TCP disconnections, we are particularly interested in previous work on disconnection tolerance, and design patterns that implement scalable distributed applications.

A. Connection Failure Tolerance

Since writing applications that resist to network outages is very difficult in practice, researchers and developers proposed many different solutions for the problem. We may classify these solutions according to the layer where they operate.

On the transport layer, Multipath TCP [4] or SCTP [5] use one or more redundant paths between client and server, to tolerate network congestion and disconnections. Besides owing other features, like stream separation, their approach to disconnection is redundancy, via multihoming. However, while SCTP is simply not compatible with TCP, multipathTCP cannot recover failed TCP streams.

On the session layer, we can find solutions proposing a change to the Application Programming Interfaces of the sockets or, to avoid this drastic change, some software components that intercept current API calls. This is the case of Robust Socket (RSocket) [10], which changes Java core libraries, leaving the standard Java TCP interface untouched. Zandy and Miller proposed a similar approach in *rocks* [11]. In our work we adopt their idea of using a `Stream Buffer` to keep data in transit. Despite the virtues of switching TCP semantics for a stronger model, both solutions depend on additional layers of software and actually disrupt the default behavior of legacy applications, with unclear consequences. The work of Alvisi *et al.* in [9] is slightly different, as authors enable the server to stop and recover, but they also use the idea of wrapping TCP with additional layers, to transparently tolerate server failures.

Other solutions also modify the server to enable replication. For example, ST-TCP [7] extends TCP to tolerate server failures. HydraNet-FT [6] replicates services across an internetwork, to provide a single view of a fault-tolerant server to the client. This requires a few modifications to TCP on the server side. In MI_TCP [8], servers in a cluster may create checkpoints of the connection state, to resume interaction after failures. All these solutions are more involved than our pattern, because they tolerate server failures.

It is worth mentioning other very popular solutions that run on the application layer, like RPC [18], Java RMI [19], CORBA [20] or even HTTP [12]. All these protocols expose the server as a set of functions or commands working in a request-response type of interaction. Rather than competing with these alternatives, TCP usually runs underneath, playing a part in ensuring their invocation semantics guarantees. Solutions like ZeroMQ [21] provide an approach based on standard distributed interaction patterns, although using an entirely different and less well-known API. All these options require additional layers of software that might not be available in all platforms. Furthermore, some of these protocols could benefit from more robust TCP implementations.

B. Design Patterns for Distributed Applications

The utilization of design patterns for distributed computing has a couple of decades already. Although the work on this field is vast, namely on Enterprise Application Integration [22], we mostly focus on client-server interactions. For example, the Acceptor-Connector pattern [14] tries to simplify the design of connection-oriented applications, by separating event dispatching from connection set up and service handling. However, the original Acceptor-Connector pattern depends on a single thread, and is, therefore, unfit for modern servers. The opposite approach, of creating one additional thread for each new incoming connection might also not scale very well for very busy servers. A common response for this problem is to handle requests using non-blocking technologies, like the C I/O `select()` or the Java NIO API [23], which allow one single thread to control many transport handles (sockets) in a single point of the code.

The Leader-Followers design pattern [15] can help to further refine multi-threading solutions, by dispatching events using a fixed number of threads. This approach keeps one thread, know as "dispatcher", controlling multiple transport handles in the single blocking point. Whenever a new event arrives, the dispatcher assigns the event to a leader thread. This thread starts to process the event, leaving the leadership to another thread. Upon processing the event, the thread returns back into the set and is queued up as a follower.

In our previous work [24], we put forward a design called "Fault-Tolerant Multi-Threaded Acceptor-Connector" (`ftmtac`), which uses the leader-followers pattern and a `Stream Buffer`, to tolerate TCP connection failures in a scalable server. The work we propose in this paper shares many of the positive features of `ftmtac`, namely the possibility of running the pattern in nearly any platform. Nevertheless, we now pushed the fault-tolerance mechanisms of `ftmtac` to work below the service handler. This basically moves fault-tolerance 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 fact, we may now use our fault-tolerant pattern with a simpler blocking approach based on a single thread per connection, but we may also keep a non-blocking Acceptor-Connector solution over the session layer.

In summary, we address a challenging problem using a very straightforward solution, based on existing design patterns and fault-tolerant solutions, like the Acceptor-Connector, the Leader-Follower or the Stream Buffer. This work improves on our previous proposals, by pushing fault-tolerance to the session layer, thus leaving more options to developers.

III. Session-Based Fault-Tolerant Design Pattern

Recovering a transport handle, like a TCP socket, from a connection failure is very difficult, because the operating systems provide no standard means for applications to determine which data did or did not reach peer nodes. This makes recovery much more complex than simply replacing a broken (TCP) transport handle by a new one. In this section, we propose a session-layer design pattern to solve this problem. The fundamental idea is to separate the recovery concern from the main application responsibilities. To promote a simple design, we use a number of components: the `Stream Buffer`, the Client and Server `Connection Handlers`, and a `Connection Set`. In this section we go through each of the components.

As shown in Figure 1, the pattern is divided into three layers: application, session, and transport. This separation exists in all patterns of this paper, blocking or non-blocking. The application layer includes a `Service Handler`, which implements an application service, typically playing the server role, but a client role is possible as well. The application interacts with our session layer using a handler (i.e. a `Connection Handler`) to exchange data with the connected peer. To interact with the transport layer, we resort to a transport handle (e.g. a TCP socket). The most common applications run directly over the transport layer, by means of a TCP socket. In our case, the interaction between the `Service Handler` and the `Transport Handle` is accomplished through a session layer offering transparent recovery from connection failures.

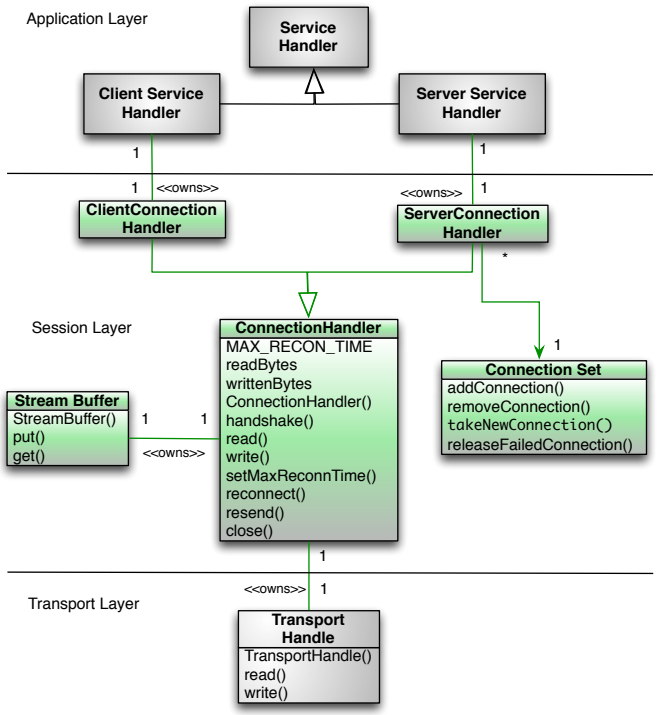

Fig. 1. Fault-Tolerant Session Layer

A. Components

1) The `Stream Buffer`: Since we cannot recover data from the existing TCP buffers, we need to implement our own buffering layer over TCP. To do so, we use a `Stream Buffer`, an idea of Zandy and Miller [11], to implicitly acknowledge messages.

To explain why we do not need such layer, we depict three buffers in Figure 2: sender application, sender TCP and receiver TCP. The receiver application must count the number of bytes received so far. To start, we assume a buffer of infinite size on the sender application. The figure illustrates a broken connection. The receiver received m bytes, whereas the sender has a total of n bytes in the buffer. Since the contents of both TCP buffers disappear on reconnection, once they reconnect, the receiver needs to send the value m, whereas the sender must resend the last $n - m$ buffered bytes.

To limit the size of the send buffer, the sender can release the bytes already received. The trick is to implicitly use the original TCP guarantees. When we know that, say, k bytes were read by the receiver, we can also delete these k bytes from the sender application buffer. Assume that the size of the TCP send buffer is s bytes, whereas the TCP receive buffer of the receiver has r bytes. We use $b = s + r$ to designate the overall send and receive buffer sizes. If we wrote $w > b$ bytes to the TCP socket, we know that the receiver got *at least* $w - b$ bytes. Hence, we need to keep at most b bytes on the sender side. I.e., we solve the problem with a circular buffer that must not be smaller than the TCP send buffer plus the TCP receive buffer of the peer side. When we write to the socket we duplicate the same bytes into the circular buffer. Passing its limit size, we overwrite the oldest bytes. We can even avoid any modulus operation, by using two's complement arithmetic over standard 32 or 64-bit counters that keep the number of sent and received bytes on each side.

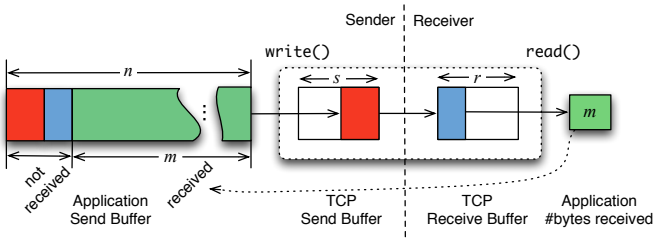

Fig. 2. Sender with infinite buffer

2) The `Connection Handler`: To accomplish recovery, we added a new component called "`Connection Handler`" to our design. Each `Connection Handler` owns a `Transport Handle` to write and read data to and from TCP. The `Connection Handler` implements the actions that should be taken for establishing a connection for the first time and after a failure, including the reconnection and resending procedures. It also provides methods (e.g. `write()` and `read()`) that write and read messages into/from the transport handle.

Setting up a connection is different on the client and server sides. The initiative always belongs to the client, even after a crash. This is mandatory, because NAT schemes and firewalls might preclude servers from initiating connec-

tions. For this reason, we added two concrete and different `Connection Handlers` in the design, the `Client Connection Handler` and the `Server Connection Handler`.

The failure recovery process is done transparently inside the `Connection Handler`, in a predefined and configurable period of time (i.e. `MAX_RECON_TIME`). To resend the lost data, both client and server `Connection Handlers` need to remember the number of `written bytes` and `read bytes`. After reconnection, these data are exchanged during the handshake, to let both sides calculate the number of the bytes they wrote, but the other side did not read.

3) The `Connection Set`: On the server side, we use a new component, named `Connection Set`[1], to synchronize threads upon connection failures and reconnections. Once a thread using the connection associated to a `Connection Handler` tries to use a failed connection, it must wait on the `Connection Set` until some other thread comes in with a new `Connection Handler` from the same client.

To do so, the `Connection Set` needs to keep the identifiers of the connections. Once a new connection is established and a new `Connection Handler` is created, its identifier is inserted into the `Connection Set`. This information is removed from the set when the connection is closed. When a connection fails, the `Connection Handler`, must wait for a new one in the `takeNewConnection()` method of the `Connection Set`. Once this method is invoked, the calling thread blocks until a new handle is inserted, or a timer goes off. Once this thread is released, information of the failed handle is removed from the set.

If the reconnection occurs, the `releaseFailedConnection()` method of the `Connection Set` allows a new `Connection Handler`, created for a failed connection during the recovery process, to deliver the new handle to the older `Connection Handler` still waiting for reconnection. Otherwise, if the timer expires, the server side considers the interaction as lost.

IV. The Blocking Session-Based Fault-Tolerant Design Pattern (`Bsft`)

In this section we present the *Blocking Session-Based Fault-Tolerant* (`bsft`) design pattern, based on a blocking utilization of the session layer of Section III. This pattern follows a simple approach used in distributed interactions, where server and client reads and writes may block the calling thread. Then, using this pattern we adapt an open source FTP server, to make it resistant to connection failures.

A. Interactions Between the Components

Refer to Figure 3. To create a new connection, the client creates a new `Connection Handler`, which internally creates a `Transport Handle`. On the other side, the server uses a passive (unconnected) handle to accept a new connection. Accepting a new connection is the task of this passive handle.

[1]We emphasize the asymmetry of client and server. Although the client could use similar components, it is difficult to come up with realistic examples where that use is necessary.

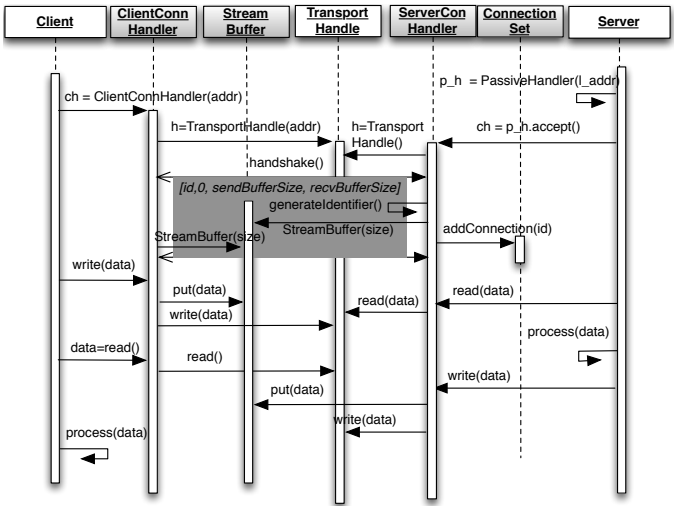

Fig. 3. Component interactions on `bsft` in a failure-free scenario

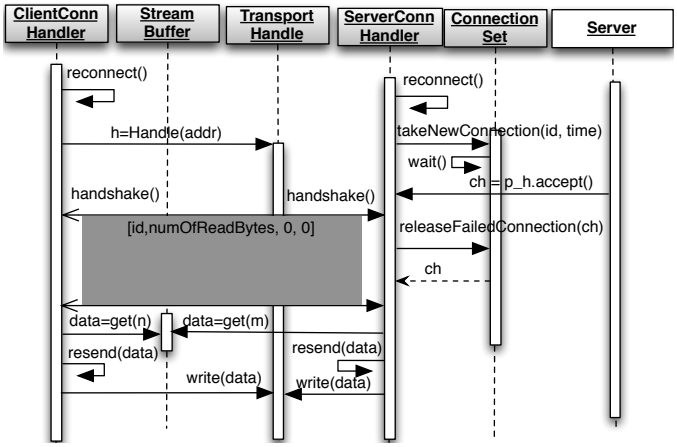

Fig. 4. Component interactions on `bsft` in the presence of a failure

Once it accepts a connection, a new `Transport Handle` and a new `Server Connection Handler` are created.

Upon establishment of a new connection, both, client and server `Connection Handlers` start a handshake protocol, by exchanging the unique identifier of the connection (the client sends 0 and the server generates a unique identifier and sends it to the client) and the size of the send and receive TCP buffers. Then both sides initialize a `Stream Buffer`. Once the handshake finishes, the `Server Connection Handler` insert its identifier into the `Connection Set`. Then, both, client and server start communicating by sending and receiving the messages using the `write()` and `read()` methods.

1) Recovery from Connection Failure: Refer to Figure 4. When a failure occurs, both sides 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` calls the method `takeNewConnection()` by giving the connection identifier and a waiting time.

After acceptance of a connection request and creation of

a new handler, the `Client Connection Handler` starts the handshake protocol. Considering the case of failure, this interaction occurs in three steps: 1) The client sends the identifier of the failed connection, which lets the server distinguish fresh connections from reconnections, and the number of bytes read. 2) The `Server Connection Handler` adds the new handle to the `Connection Set` using the method `releaseFailedConnection()`. This method releases the handler waiting for the new connection. Then, the `Server Connection Handler` completes the handshake and replies with the identifier and the number of bytes it has read, plus any buffered data not previously received by the client. 3) Finally, if the client has data to send, it sends it. After the handshak, both, client and server, continue their communication.

B. Modification of an FTP Server

To explore our approach in practice, we added fault-tolerance to the ANOMIC open source FTP server [25]. We called `ftANOMIC` to the fault-tolerant version of this application, and we made the source code available online [16]. ANOMIC and our fault-tolerant modifications are implemented in Java. We called `FSocket` to the `Connection Handler`, being `CFSocket` and `SFSocket` the client and server implementations, respectively.

To introduce fault-tolerance, we replaced every `Socket` object by an implementation of `FSocket`, either `CFSocket` or `SFSocket`, depending on the application's role. Particularly, in ANOMIC, since the server can play the client's role as well, we needed to use both `CFSocket` and `SFSocket`.

As explained, servers own one passive handle to accept new connections. In Java TCP, this passive handle is called `ServerSocket`. We have an equivalent passive handle in our implementation, called `FServerSocket`. Wherever the server waits for new connections, either in its control or data connection, we used this object instead of the `ServerSocket`. Upon connection, the `FServerSocket` returns an `SFSocket` instead of a `Socket`. Moreover, all the read and write operations done on the TCP socket's `InputStream` and `OutputStream` must be replaced with the read and write operations on the `FSocket` object. These replacements are summarized below:

```
CFSocket fsocket = new CFSocket (server,port)
// instead of
Socket socket = new Socket (server,port)

SFSocket fsocket = fServerSocket.accept()
// instead of
Socket socket = serverSocket.accept()

FServerSocket fServerSocket = new FServerSocket(port)
// instead of
ServerSocket serverSocket = new ServerSocket(port)

int read = fsocket.read(data)
// instead of
int read = inputStream.read(data)

fsocket.write(data)
// instead of
outputStream.write(data)
outputStr.flush()
```

In addition to these changes, we needed to do one more modification for this specific case, as the server may listen on more than one port to accept control and data connections. To make this need clear, we briefly explain the active and passive modes of FTP servers.

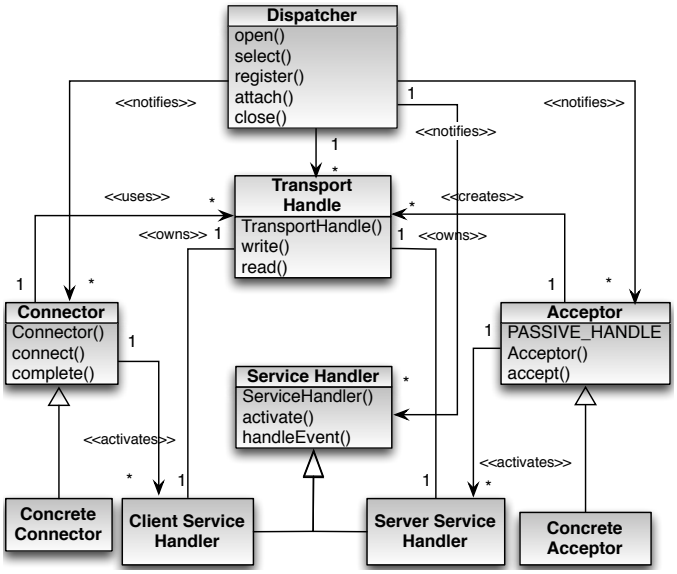

Fig. 5. The Acceptor-Connector design pattern

In the active mode, the client connects from a random port N to the FTP server port (usually 21). Then, the client starts listening on port $N + 1$ and sends a control message to the server, with the number $N + 1$. The server will then connect back to the client's specified data port. In contrast, in the passive mode, the client initiates both connections to the server. After opening an FTP connection, the client sends the $PASV$ command. The server then opens a random port (above 1023) and sends the number back to the client. The client responds by initiating a new data connection to the server on that port.

The modification we needed to do was for the server FTP passive mode. In this mode, although the server continuously checks on the command port (e.g., 21) for new control connections, it checks the port dedicated to the data connection only once. This will cause a problem when the connection crashes, because the client's reconnection to the server fails due to the lack of any listener on the data port. To solve this problem we forced the server to listen on the data port, until the data connection is closed.

Apart from this detail, the implementation of the session layer and the replacement of the standard TCP sockets by the session layer's sockets was straightforward. In Section VI we show that other costs associated to these changes are also negligible.

V. A Non-Blocking Approach: The Session-Based Fault-Tolerant Multi-Threaded Acceptor-Connector (SFTMTAC) Design Pattern

In Section III we propose a generic session-based design to transparently recover from connection failures. In this section we leverage on this design to create a non-blocking pattern, called *Session-Based Fault-Tolerant Multi-Threaded Acceptor-Connector* (`sftmtac`), to efficiently handle a large number of

concurrent connections. We made the code of our pattern available for download [17]. For the sake of self-containment, we first overview related patterns, like the Acceptor-Connector, before presenting our own proposal.

A. The Acceptor-Connector Design Pattern

The Acceptor-Connector pattern [14] (refer to Figure 5) decouples connection setup from service handling in connection-oriented client-server interactions. We consider this to be a valuable principle that we follow on our own fault-tolerant design. Besides the `Acceptor` and the `Connector`, the pattern includes a `Transport Handle`, a `Service Handler`, and a `Dispatcher`. To establish a connection, the client uses the `Connector`. On the server side, the `Acceptor` receives the request and sets up the connection. Both, `Connector` and `Acceptor`, initialize the `Service Handler` to process the exchanged data. The `Dispatcher` plays the role of a central distributor in both sides, disseminating events to the appropriate component, thus enabling peers to manage more than one `Acceptor`, `Connector`, and the resulting connections with only one thread of control[2].

B. Support of Multi-threading in the Acceptor-Connector

Although the Acceptor-Connector design pattern perfectly decouples processes and enables the creation of multiple concurrent connections, it is single-threaded, thus not supporting simultaneous response to events. The opposite extreme of creating one thread per connection might also not be the right choice, whenever the number of connections is high, due to the possibly overwhelming overhead.

An adequate solution to this problem is proposed in the Leader-Followers design pattern [15], which combines the event dispatcher with a thread set. In this design pattern, the service handlers are assigned to the threads only when an event occurs on their `Transport Handle`. This solves the problem with the unbounded growth of threads. The remaining problem is to adjust the size of the `Thread Set`, depending on the available resources.

The `Dispatcher` and the `Thread Set` allow applications to handle multiple concurrent connections. As before, we have a single thread demultiplexing several channels at once in the `Dispatcher` and passing the events to the leader thread of the set. We call *Multi-Threaded Acceptor-Connector* to this pattern (`mtac`), and use it for evaluation purposes.

C. Fault-Tolerance in the Acceptor-Connector

We now propose to use the session layer from Section III to transparently handle connection failures. The interesting point here is to decouple the failure handling processes from the connection establishment and service processing. Moreover, having one `Connection Handler` per connection, and a `Thread Set` to deliver new connections and to replace failed ones, in a non-blocking manner, completely decouples failure handling of one connection, from other operations in other connections.

[2]In fact, in many cases, only the server will resort to a dispatcher.

Refer to Figure 6, where we show a simple single-threaded client owning just one connection, and a multi-threaded, multi-connection server. We show the interactions of these components in four different phases including 1) initialization of connection and `Connection Handler`; 2) initialization of services; 3) execution of services; and 4) failure handling, considering two main scenarios: a failure free scenario (Figure 7) and a scenario with a connection failure (Figure 8). In Figure 7, we fill the boxes of the components that vary with each new application in gray. The remaining components are generic.

D. Interactions Between the Components

In general, each server owns one `Dispatcher` to register and attach `Transport Handle`s. The server also owns one `Thread Set` to control the number of threads created for handling the connections. Thus, when the server starts, it initializes the `Dispatcher` and the `Thread Set`. Figure 7 shows these early steps as well as all interactions between the components.

Phase 1: Initialization of the Connection Handler
To initialize connections, the server can create one or more `Acceptor`s. It needs one `Acceptor` per port. When an `Acceptor` is initialized, a passive-mode handler is created and bound to a network address. The `Acceptor` registers the handler and attaches itself to it in the `Dispatcher`. The `Dispatcher` checks all its passive handlers for new connection requests, calling the `accept()` method of the appropriate `Acceptor`, which was previously attached to the handler, when a new connection shows up.

On the other side of the communication, the client calls the `Connector`'s `connect()` method. This method actively sets up a `Connection Handler` by blocking the calling thread until the connection completes synchronously. Upon reception and acceptance of the connection request, a `Transport Handle` (identified by `h` in the figures) and a `Connection Handler` (identified by `ch` in the figures) are created on the server and client sides.

Upon initialization of the `Connection Handler`s on the client and the server, a handshake, initiated by the client `Connection Handler`, is performed to exchange control data, as we mentioned before: an immutable unique identifier, generated by the server, the sizes of their TCP buffers and, possibly, some data. This lets the `Connection Handler`s recover lost data after recovery from a failure. After the handshake of a fresh (i.e., not recovered) connection, a `Stream Buffer` with a specific size is created in both side's `Connection Handler`s. Then, the identifier of the connection is added to the `Connection Set` by the `Connection Handler`.

Phase 2: Initialization of Service Handler
After the initialization of a `Connection Handler`, the `Connector` completes the process using the method `complete()`, which activates the `Service Handler`, by passing the `Connection Handler` through the method

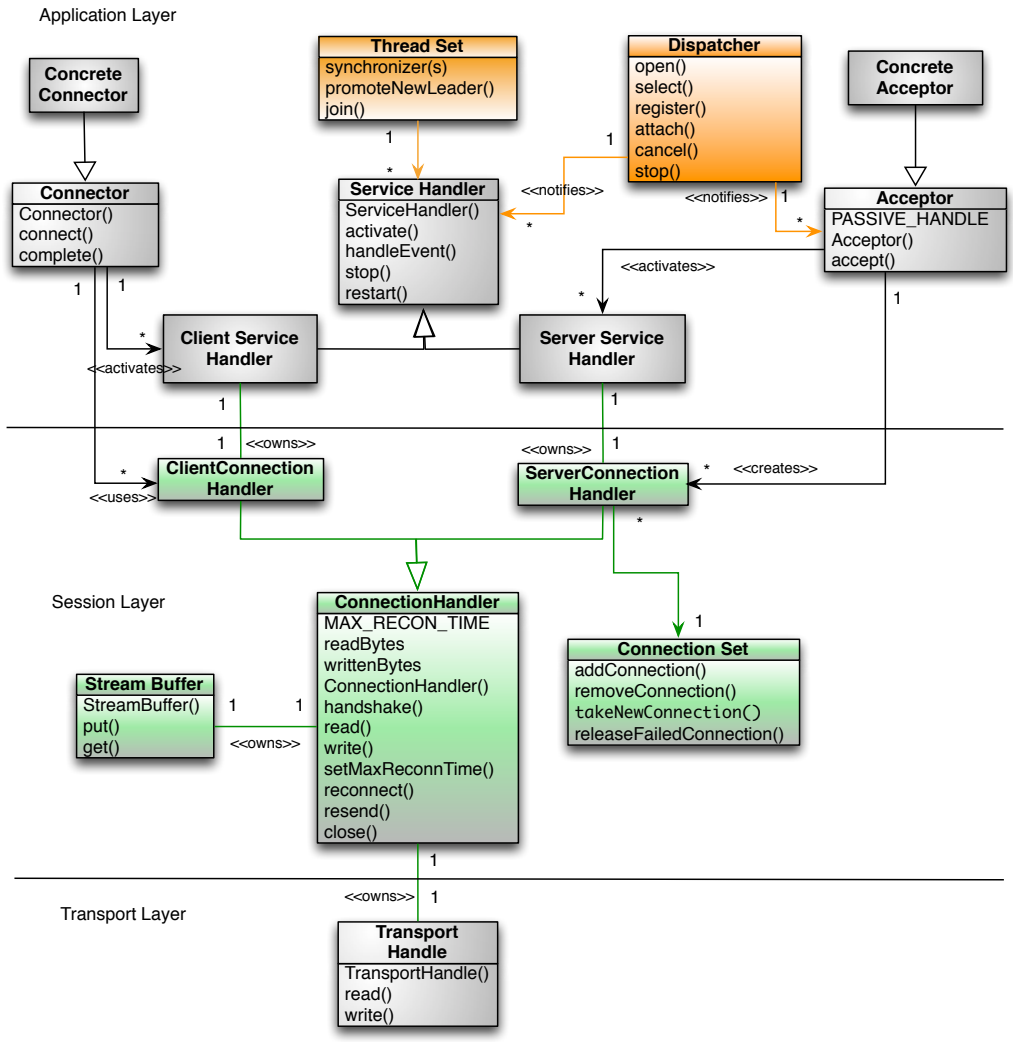

Fig. 6. The Session-Based Fault-Tolerant Multi-Threaded Acceptor-Connector (`sftmtac`) design pattern

`activate()`. On the other side, if the connection is fresh, the `Acceptor` creates a new instance of a `Service Handler` and activates it by passing the `Connection Handler`. Finally, the `Service Handler` registers the handler and attaches itself to the handler in the `Dispatcher`.

Phase 3: Service Processing

Once the connection is established and the `Connection Handler` and `Service Handlers` are initialized successfully, the service-processing phase begins. The `Service Handlers` could use the `write()` and `read()` methods to send and receive data messages through the `Connection Handler`. All the details related to saving the data sent, the number of bytes sent and received are transparently done in the `Connection Handler`. In the case of the server, a `read()` operation usually works differently. The `Dispatcher` delivers new read events to the attached `Service Handlers`, which read data in a non-blocking way (this may involve multiple read attempts, until the entire message is read)[3].

When the `Dispatcher` delivers an event to the `Service Handler` through the method `handleEvent()`, the leader thread from the `Thread Set` is assigned to process the event, and a new thread is promoted to become the leader.

Phase 4: Failure Handling

Figure 8 presents failure handling details. Once a `Transport Handle` fails completing a read or write operation, the `Connection Handler` transparently tries to reconnect. The reconnection is accomplished differently in the `Client Connection Handler` and `Server Connection Handler`. As shown in the figure neither the `Client Service Handler`, nor the `Server Service Handler` are involved on the recovery procedure, to ensure the separation between service and failure handling.

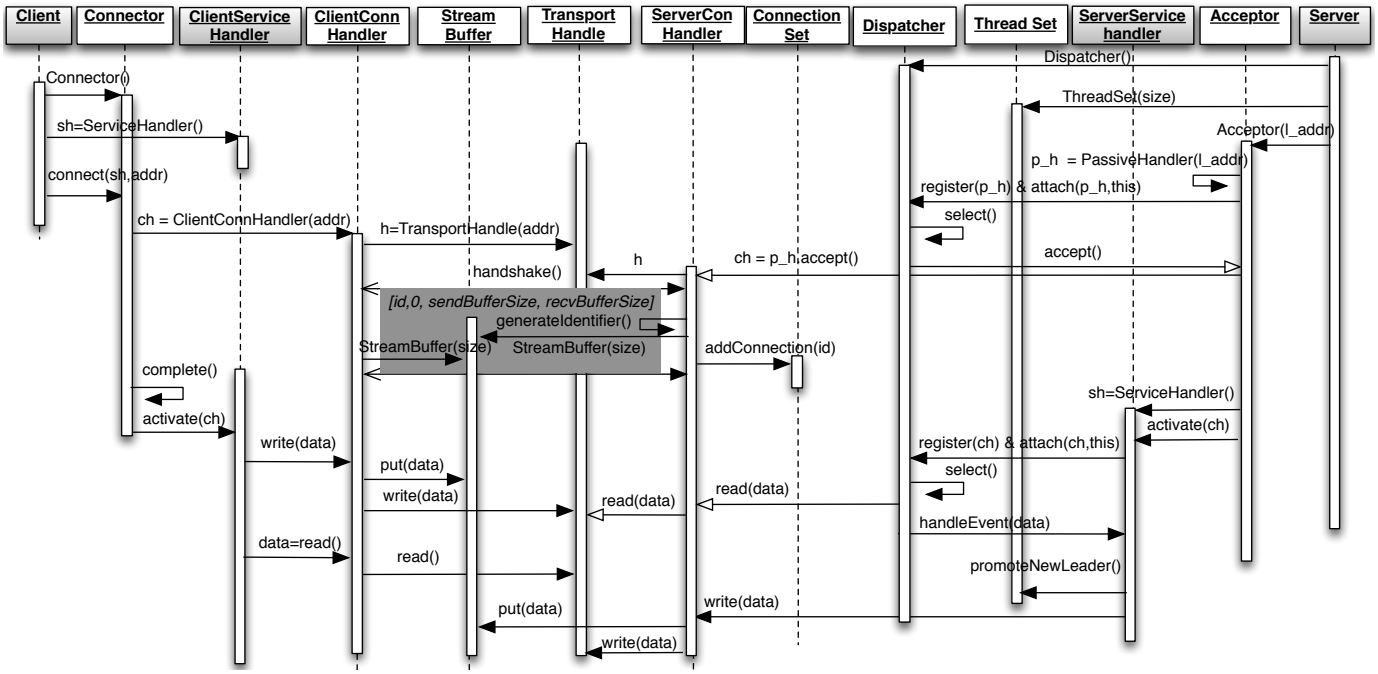

Fig. 7. Component interactions on `sftmtac` in a failure-free scenario

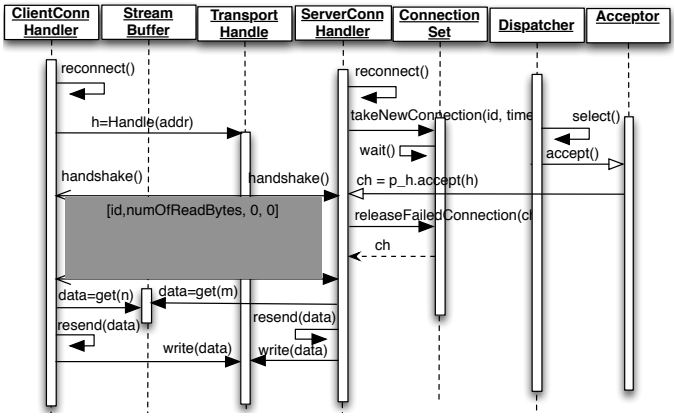

Fig. 8. Component interactions on `sftmtac` in the presence of a failure

TABLE I
SYSTEMS USED IN THE EXPERIMENTS

Endpoint	OS	CPU	Memory
Client	Mac OS X, version 10.6.7	2.4GHz Intel Core 2 Duo processor	4GiB of RAM, 3 MiB of cache
Server	Linux version 2.6.34.8	2.8 GHz Intel processor with 4 cores	12 GiB of RAM, 8 MiB of cache

VI. EXPERIMENTAL EVALUATION

A. Experimental Setup

We evaluated the performance and overhead of our design patterns in this section. We ran client and server peers in computers with the characteristics of Table I, in the same 100 Mbps Local Area Network.

[3]One should notice that besides `accept()`, and `read()`, the `write()` operations could also be non-blocking and controlled by the `Dispatcher`.

To test the blocking fault-tolerant pattern we used the standard (non-fault-tolerant) Anomic FTP server and a fault-tolerant variation of the same server. To test the non-blocking pattern, we compare the performance of `sftmtac` against the Multi-Threaded Acceptor-Connector (`mtac`). In the evaluation we also measure the implementation complexity of the patterns, using our own Java implementations [16], [17].

B. Performance of the Blocking Approach

We considered one FTP server and a growing number of clients requesting files of two sizes: 6 bytes and 1 GiB. We use the former file size to compute the latency of the requests (the time since the client requests the file to the time it gets back the file), while the latter file serves to compute the throughput (in bits per second). The higher complexity of setting up a connection should be noticeable in the latency, whereas memory copies to the Stream Buffer could impact throughput. However, our results show that the effects of these operations are negligible. We downloaded files from 1 to 50 clients, observing only a small degradation of latency, which is common to the non-fault-tolerant version (from 100 ms to 111 ms), whereas throughput held on at 89 Mbps[4], again, with no visible penalty for the fault-tolerant version. These results were the averages of 10 trials.

C. Performance of the Non-Blocking Approach

Since blocking servers have the disadvantage of tying one thread to each client, we also took similar measurements using the non-blocking `sftmtac`. To do such measurements,

[4]A small, but relevant detail here is that we do not write the files to disk on the client. This allows throughput to be closer to the network limit of 100 Mbps.

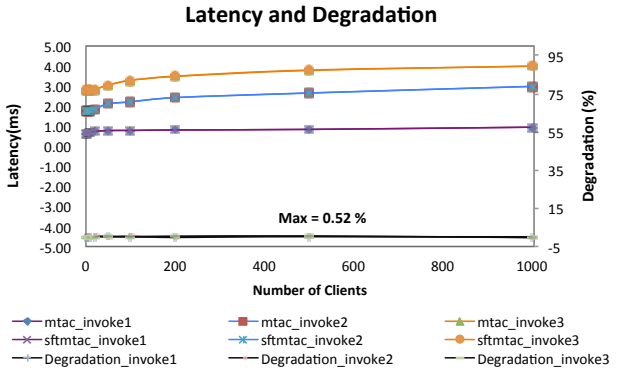

Fig. 9. Latency and Degradation of `sftmtac`

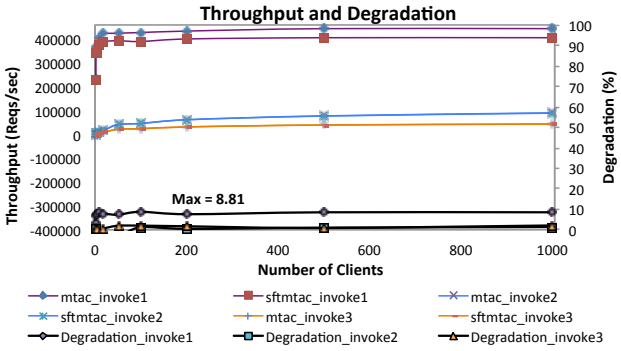

Fig. 10. Server's Throughput and Degradation in `sftmtac`

we created a server offering three operations, `Invoke1`, `Invoke2` and `Invoke3`. They all receive a small string and return another small string, both of 20 bytes. The difference is that `Invoke1` replies immediately, `Invoke2` sleeps 1 millisecond (ms) before replying, whereas `Invoke3` sleeps 2 ms. We put the server threads to sleep to minimize interference with our results.

Figures 9 and 10 show the latency and throughput of `sftmtac` and `mtac` for different numbers of clients (from 1 to 1000). The results shown are the average of $1,000$ trials. Latency is the round-trip-time of a request-response interaction. The plots also show the performance degradation of the `sftmtac`, in comparison to the `mtac` on the right vertical axis. The degradation shown is the difference between the latencies of `sftmtac` and `mtac` relative to the `sftmtac`'s latency ($(Latency_{sftmtac} - Latency_{mtac})/Latency_{sftmtac}$).

To examine the throughput, the client sends a very large number of requests to the server ($100,000$ in our tests) without waiting for any response (a different thread takes care of that). The server sends one reply to the client after receiving and processing all requests. We compute the approximate throughput, based on client-side measurements. The values shown in the figure for degradation are based on the following relation of throughputs: $(Throughput_{mtac} - Throughput_{sftmtac})/Throughput_{mtac}$.

The results show that `sftmtac` is almost on par with `mtac` in performance. As shown in the plots, the maximum degradation we had in our evaluation for latency is less than 1 percent (0.52%). Latency grows slowly and smoothly with

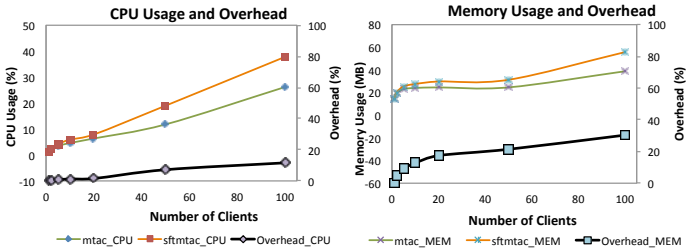

Fig. 11. CPU and Memory Usage

the number of clients in both designs. Throughput for the slower invocations 2 and 3 is pretty much the same in both designs. `Invoke1` shows a higher degradation, but still below 10%. This kind of comparison is indeed the worst case for `sftmtac`, because `Invoke1` does nothing, thus exposing all the communication overheads.

D. Recovery from Network Failures

To evaluate the ability of our session layer to recover from failures, we emulate connection crashes using Tcpkill [26], a network utility program that can terminate established TCP connections. To do the test we made the client and server exchange messages during 5 minutes and emulate the connection failure using Tcpkill. For 10 repetitions of the test, we first observed that all the messages were transferred completely to the server. We further observed that setting up the new connection took 230 ms in average, whereas recovery (handshake plus sending lost messages) took an average of 5.25 ms, which, we believe, is quite fast.

E. Overhead and Complexity

To examine the CPU and memory overheads, we wrote a simple script based on the `ps` command, to periodically monitor memory and CPU usage of the server process. We used the `sftmtac` design and a varying numbers of clients, between 1 and 100 (refer to Figure 11). Each client sends 100 requests to the server per second during 5 minutes. The requests invoke `Invoke1` on the server. The results shows that `sftmtac`'s overhead is reasonable, being a little bit higher in terms of memory, due to the `Stream Buffer`.

We also measured three important complexity metrics, Lines of Code (LOC), Cyclomatic Complexity, and Nested Block Depth, to indicate the complexity of `sftmtac` and compared it with the `mtac`. The measurements show that we used 485 extra lines of code in `sftmtac`. In addition, the average cyclomatic complexity per method in both cases is around 1.87, while the depth of nested blocks of `sftmtac`'s code is 1.4, close to the 1.26 of `mtac`. The effort in providing this session layer to the Acceptor-Connector is, therefore, small.

We took the same measurements for the FTP server without and with fault-tolerance. The increase in these costs is also negligible. The Anomic server grew from 2548 to 2677 lines of code; the cyclomatic complexity did not increase from 4.1; the nested block depth grew slightly from 1.84 to 1.87.

VII. Conclusion

We presented fault-tolerant design patterns based on a session layer providing a simple, generic, and platform-independent solution for recovery from connection crashes. By isolating fault-tolerance in the session layer, we open multiple design alternatives. In this paper we go over two patterns that precisely explore this possibility, by using the session layer in blocking and non-blocking design patterns, respectively, `bsft` and `sftmtac`. These patterns offer a number of benefits, like separating the application core concerns from fault-recovery actions. Our implementation of a real case and the experimental evaluation we performed show the low overhead, the efficiency, and the simplicity of using our patterns.

To further explore this work, we intend to apply the session-based fault-tolerant design patterns in a web server, to support reliable streaming.

Acknowledgments

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

References

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

[2] RFC 959 - File Transfer Protocol (FTP). Internet Engineering Task Force (IETF), October 1985.

[3] RFC 4251 - The Secure Shell (SSH) Protocol Architecture. Internet Engineering Task Force (IETF), January 2006.

[4] Sebastien Barre, Christoph Paasch, and Olivier Bonaventure. MultiPath TCP: from theory to practice. In Jordi Domingo-Pascual, Pietro Manzoni, Sergio Palazzo, Ana Pont, and Caterina Scoglio, editors, *NETWORKING 2011*, number 6640 in Lecture Notes in Computer Science, pages 444–457. Springer Berlin Heidelberg, January 2011.

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

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

[7] Manish Marwah and Shivakant 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.

[8] Hal Jin, Jie Xu, Bin Cheng, Zhiyuan Shao, and Jianhui Yue. A fault-tolerant TCP scheme based on multi-images. In *IEEE Pacific Rim Conference on Communications Computers and Signal Processing (PACRIM 2003)*, pages 968–971, Victoria, Canada, 2003.

[9] Lorenzo Alvisi, Thomas C Bressoud, and Ayman El-Khashab. Wrapping Server-Side TCP to mask connection failures. *In proceeding IEEE INFOCOM*, 2001.

[10] Richard Ekwall, Peter Urban, and Andre Schiper. Robust TCP connections for fault tolerant computing. *In proceeding 9th international confeence on parallel and distributed systems (ICPADS)*, 19:501—508, 2002.

[11] Victor C. Zandy and Barton 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.

[12] Balachander Krishnamurthy and Jennifer Rexford. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley Professional, 1 edition, May 2001.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

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

[15] Douglas Schmidt, Carlos Ryan, Michael Kircher, Irfan Pyarali, and Frank Buschmann. Leader-followers. In *PLoP conference. http://hillside.net/plop/plop2k/proceedings/ORyan/ORyan. pdf*, 1998.

[16] Fault-tolerant anomicftpd: A freeware ftp server in java. https://sourceforge.net/projects/ftanomic/.

[17] Implementation of session-based fault-tolerant multi-threaded acceptor-connector design pattern. https://sourceforge.net/projects/sftmtac/.

[18] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[19] Troy Bryan Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1st edition, 1998.

[20] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.

[21] Distributed computing made simple. http://zeromq.org/.

[22] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns — Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[23] Package java.nio. http://docs.oracle.com/javase/6/docs/api/java/nio/package-summary.html.

[24] Naghmeh Ivaki, Filipe Araujo, and Fernando Barros. Design and development of a fault-tolerant multi-threaded acceptor-connector design pattern. Technical Report TR 2014-003, Centre for Informatics and Systems of the University of Coimbra, June 2014.

[25] Anomicftpd: A freeware ftp server in java. http://anomic.de/AnomicFTPServer/index.html.

[26] Manual Reference Pages - TCPKILL. http://www.irongeek.com/i.php?page=backtrack-3-man/tcpkill.

View publication stats