Design and Development of a Fault-Tolerant Multi-Threaded Acceptor-Connector Design Pattern

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

Centre for Informatics and Services of the University of Coimbra

Technical Report TR 2014-003

June, 2014

```
@TechReport{FTMTAC-14,
   author = {Naghmeh Ivaki and Filipe Araujo and Fernando Barros},
   title = {Design and Development of a Fault-Tolerant Multi-Threaded Acceptor-
Connector Design Pattern },
   institution = {Centre for Informatics and Systems of the University of Coimbra},
   year = {2014},
   month={June},
   number = {TR 2014-003},
}
```

Design and Development of a Fault-Tolerant Multi-Threaded Acceptor-Connector Design Pattern

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. Fault-tolerance is vital for dependable distributed applications that can deliver service, even in the presence of faults. Over the last few decades, above all protocols proposed to offer reliability and fault-tolerance, TCP grew to become one of the cornerstones of the Internet. However, despite emulating reliable communication in distributed environments, TCP does not handle connection failures when the connectivity is lost for some time, even if both endpoints are still running. When this occurs, developers must rollback the peers to some coherent state, many times with error-prone, ad hoc, or custom application-level solutions.
In this report, we refine the Acceptor-Connector design pattern to tackle the TCP unreliability problem. The pattern decouples the failure-related processing from the connection and service processing, efficiently handling different connections and their possible crashes concurrently, thereby yielding more reusable, extensible, and efficient distributed communication. The solution we propose incorporates proven multi-threaded solutions and a buffering scheme that discards the need for an application-layer acknowledgment scheme. This simplifies the development of reliable connection-oriented applications using the ubiquitous TCP protocol.

1 Introduction

The growing importance of the Internet in people's life and businesses, including e-commerce, financial services, health care, government, and entertainment, increases the need for large-scale dependable distributed applications. At the heart of most distributed applications, especially of those requiring reliability, we find the Transmission Control Protocol (TCP) [1]. The popularity of TCP is unquestionable: any major operating system provides a TCP/IP communication stack with Application Programming Interfaces (APIs) for a large number of programming languages. At first glance, TCP looks as a simple and powerful solution to overcome network unreliability, which is true to a certain point. However, if connectivity is lost for a period of time, the TCP connection breaks, making any kind of recovery very difficult for the endpoints. In many programs and protocols, such as FTP [2], SSH [3], TLS [4] or the X Windows System [5], it would be worthwhile to keep the interaction alive.

Although many solutions for reliable communication over faulty channels exist in the literature, most of them focus on replication schemes resorting to additional configuration and hardware. The problem with all these solutions is that they either try to replace TCP or require special software or hardware that may not be readily available or mature for deployment in all platforms and languages. In fact, the number of solutions available demonstrates the difficulty of ensuring reliable communication using TCP. The asynchrony and unreliability of the network concur to complicate a timely detection of message losses. For example, one peer application may send many messages unwitting that they are not reaching the peer endpoint. If the API ever returns an error notification, the application will be unable to tell which write operations did or did not get through the channel.

One possible approach would be to use a session layer to buffer TCP messages and retransmit them as necessary [6]. This however, incurs in traffic and delay overheads, besides forcing the programmer to resort to a non-TCP API. Other middleware approaches may also serve a similar purpose, by providing extra layers over TCP, but they share the same overhead and non-TCP

API shortcomings. Some solutions try to keep the TCP API, by using special hardware or software layers that somehow intercept the TCP communication. But, besides requiring additional components and/or configurations, these have the questionable side-effect of changing the TCP semantics for oblivious applications.

In this report we specifically aim to tolerate TCP connection crashes using the standard TCP API. We argue that TCP might be the protocol of choice for many developers, because it is deeply well-known, it exists in nearly all platforms, and it can ensure high performance, when compared to higher-level solutions.

To deal with the complexity of this task, we propose a design pattern called "Fault-Tolerant Multi-Threaded Acceptor-Connector" (`ftmtac`) that tolerates TCP connection crashes in large-scale connection-based applications. Over the years, several reusable design patterns, including the Acceptor-Connector, the Publisher-Subscriber, the Leader-Follower, the Reactor, the Proactor, and so on [7] emerged to simplify distributed programming. For example, to provide multi-threading to the "Acceptor-Connector" design pattern (`ac`), we may combine it with the Leader-Follower, to create the "Multi-Threaded Acceptor-Connector" (`mtac`) design pattern. However, neither this, nor any other pattern we are aware of, manages to provide the fault-tolerance we seek on top of the TCP transport layer, despite the important merits of such option.

A crucial piece of our solution is the `Stream Buffer`, a circular buffer that is simple to implement and replaces a layer of acknowledgments and retransmissions over TCP [8]. Apart from reconnections, this buffer discards retransmissions, because it manages to store all the data potentially in transit between the peers. With the help of `Stream Buffer`, the `ftmtac` pattern lets developers write their own fault-tolerant large-scale client-server applications from scratch, using standard approaches like multi-threading and Java NIO or C `select`s. The `ftmtac` extends the Acceptor-Connector [9] with a few additional components, including the `Connection Handler`, the `Stream Buffer`, the `Connection Set`, and the `Thread Set`. A `Connection Handler` implements the actions to take once the connection crashes. Each `Connection Handler` owns one `Stream Buffer`. The `Connection Set` keeps the existing connections and enables their replacement if necessary. The `Thread Set` provides a pool of threads to the application to be used whenever a new event occurs (e.g., a new request).

We experimentally demonstrate the validity of our approach. Adding fault-tolerance to the client-server communication requires twice the buffering space, but has negligible overhead in performance, requiring little more than memory copies at send time. This, we believe, is a low price to pay for fault-tolerance. To lower the barrier for developers, we wrote the multi-threaded acceptor-connector design pattern in Java and made it publicly available for download [10].

We organize the report as follows: in Section 2 we review the related literature. In Section 3, we describe the Multi-Threaded Acceptor-Connector design pattern, then we proceed to present our own fault-tolerant solution. Section 4 demonstrate the implementation of the pattern. In the Section 5 we demonstrate the performance of the pattern. Finally we conclude the report.

2 Background

Tolerating network faults is of paramount importance for large-scale highly available distributed systems. If it is true that TCP overcomes problems like packet losses, corruption or reordering, many other problems lack general solutions to this day, partially because they are outside the scope of the transport layer. Given the importance of the problem, many researchers have came up with different types of fault-tolerant solutions. Here, we review a number of these approaches and divide them into transport-layer, session-layer and application-layer. Our own proposal fits in the last category.

Starting from the transport layer, one of the many protocols that emerged in the last decades was the Stream Control Transmission Protocol (SCTP) [11]. SCTP explores multiple network interfaces and alternative paths between peers to provide redundancy, thus tolerating network faults and increasing throughput, namely in wireless environments. However, since SCTP is not compatible with TCP, Barre *et al.* proposed Multipath TCP [12], a protocol that works with the

standard TCP sockets API. However, since transport level connections may still fail in both SCTP and Multipath TCP, the problem of resuming connections after a failure persists. This makes us believe that these approaches are orthogonal to the solution we propose in this report, as we may use a similar principle for SCTP or Multipath TCP.

Other researchers propose session-layer solutions, with an extra layer of buffering and retransmissions invisible to the application. Usually authors strongly subscribe to the point of view that TCP cannot be changed and try to insert this layer between the sockets API and TCP itself, for the sake of compatibility. This is the case of Robust Socket (RSocket) [13], which leaves the standard Java TCP interface untouched and uses a UDP connection for acknowledgement. A slightly different case is the work [14], but again, no changes are required on the client side or server's source code, as all additional layers are hidden from the application. *Rocks* [8] also leaves the API untouched, but intercepts calls to keep data sent in a buffer. This buffer is only necessary when connections fail. We borrow this idea for our *Stream Buffer*. This buffer lets us achieve reliability, without using any additional layer of acknowledgment, nor hidden in some session layer, neither explicit in the application layer. Senders just need to keep a copy of their data in the buffer, to enable connection recovery after crashes. The difference between these solutions and our pattern is that we do not hide the buffers, but make them explicit to the application layer. Hence, our solution works with any TCP/IP stack.

Some extremely popular solutions work on the application layer, like our own. This is the case of Sun's Remote Procedure Calls [15], Java Remote Method Invocation (RMI) [16], CORBA [17], or even HTTP [18]. However, they fundamentally differ from our proposal in two aspects: they expose the server as a set of remote functions and they create a layer over TCP (or UDP), thus narrowing their scope to specific languages or platforms. Hence, despite being widely used, they cannot reach the same level of availability and adoption that standard TCP sockets still enjoy to this day. The settings where developers can apply our solution are different from the ones we find for the former approaches, not only in the technology, as we do not add another layer, but also because these serve for applications with strict request-response interactions. Our design pattern also applies to cases where interaction is absent for long periods: e.g., to transfer large contents, such as a file, screen contents, or other multimedia streams. In fact, our design pattern leaves the flexibility and ubiquity of sockets untouched. Other less well-known solutions like ZeroMQ [19] provide an approach based on standard distributed interaction patterns that go beyond the request-response, but these solutions are still far less popular than plain TCP.

One should notice that our work also differs in a fundamental way from application-level solutions that aim to tolerate endpoint crashes. This is not our goal, as we only tolerate network crashes. Such solutions are more complicated, often requiring replication of components, typically on the server. For example, HydraNet-FT [20] replicates services across an internetwork and provides the view of a single, fault-tolerant service. It uses TCP with a few modifications on the server side. We can also find related solutions in ST-TCP [21], an extension of TCP to tolerate server faults. In MI_TCP [22] servers of a cluster write a checkpoint of the TCP connection state, to enable the TCP connection to migrate to another server in the cluster.

We are not aware of any other design pattern that recovers from TCP crashes, because previous patterns simply try to provide client-server communication in fault-less scenarios. Nevertheless, we base our work on some of these patterns, namely on the Acceptor-Connector [9], which tries to decrease the design complexity of connection-oriented distributed applications, by decoupling event dispatching from connection set up and service handling. Technologies like the C I/O multiplexing (as in the `select()` function call) or the Java NIO API [23] directly support this design option. Multi-threaded designs, such as [24] emerged to respond to the needs of servers with many clients. In our solution we adopt the Leader-Follower design pattern [25] that further refines Acceptor-Connector by dispatching events to a fixed number of threads.

3 Multi-Threaded Fault-Tolerant Acceptor-Connector Pattern

The Acceptor-Connector design pattern (refer to Figure 1) includes an `Acceptor`, a `Connector`, a `Transport Handle`, a `Service Handler`, and a `Dispatcher`. In this pattern, the client asks the

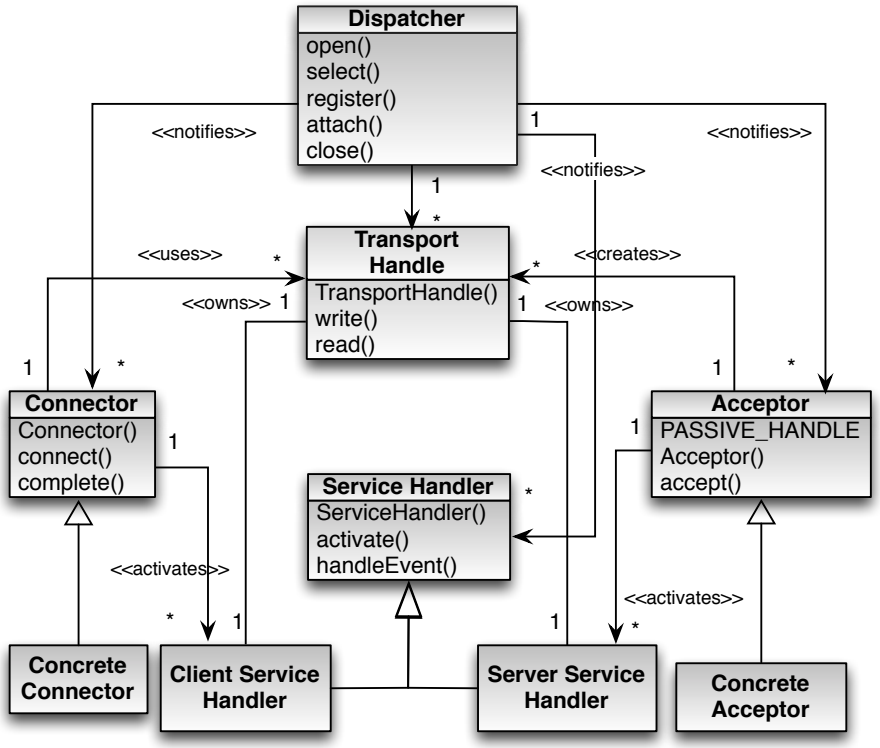

Fig. 1. Acceptor-Connector design pattern

`Connector` to set up a connection to the remote server, which accepts it using the `Acceptor`. To process data exchanged, both, `Connector` and `Acceptor`, initialize `Service Handler`s. The actual operations of reading and writing data are in charge of the `Transport Handle`. Since writing to and reading from the `Transport Handle` might block a process with more than one connection, the `Dispatcher` multiplexes several connections, thus providing a single blocking point for the client or server application, while waiting for operations on several `Transport Handle`s, `Acceptor`s, or `Connector`s to complete. When the resource is ready, the `Dispatcher` notifies the appropriate component.

Despite being extremely common, this design pattern has a number of drawbacks from the scalability and reliability points of view. However, many applications nowadays are not low-scale, and need to handle a large number of concurrent connections. For example, in a multi-tier system, the front-end communication servers receive requests (maybe arriving simultaneously from hundreds or thousands of remote clients) and forward them for processing to back-end application servers. These, in turn, may forward some requests to the back-end database (or file) servers. To take advantage of this decoupled multi-tier design, receiving requests could be done simultaneously while processing other requests. Moreover, the requests received from different connections could also be handled concurrently. In addition, in the case of TCP/IP, the `Transport Handle`, which is a TCP socket, may fail, leaving both endpoints in the middle of their interaction.

We added several components to the original Acceptor-Connector design, which allow: 1) efficient multi-threading for large-scale applications, and 2) recovery from connection crashes. In the proposed design, the failure handling is decoupled from the main functionalities of connection establishment and service handling. Our design should include the following features:

– it should keep the possibility of adding new types of services, new service implementations, and new communication protocols, without affecting the existing connection establishment and service initialization;

– it should efficiently handle a large number of connections;
– it should decouple failure handling from service handling;
– it should enable to decouple the failure handling processes of different connections.

To achieve these objectives, we resort in part to existing multi-threading solutions, but propose our own approach for fault-tolerance. In this section we start by refining the Acceptor-Connector design pattern, to address the scalability issues.

3.1 Support of Multi-threading in Acceptor-Connector Design Pattern

A common strategy to support concurrency is to use multiple threads, assigning one thread for each incoming connection. The shortcoming of this option emerges when the number of connections is very high, thus requiring a large number of threads and a considerable consumption of resources. To limit the number of threads, one may use a thread set. The idea is to assign threads from the set to new connections, instead of creating new ones. When the connection terminates, the thread returns to the set and waits for another connection assignment. The association between threads and connections in this design solution is bounded. With this form of association, new connections may have to stay on hold until older connections finish. Furthermore, some of the threads may not have events to process for some periods of time, thus contributing to degrade the performance of the server.

The Leader-Follower design pattern [25] does not present these disadvantages. In this pattern, the service handlers are assigned to the threads only when an event occurs on their transport handle. When the event processing finishes, the thread rejoins the set. We have refined the Acceptor-Connector design pattern by adding this design solution (refer to Figure 2). In this design, only one thread from the `Thread Set` (the leader), is allowed to wait for an event. Meanwhile, other threads (the followers) can queue up waiting their turn to become the leader. As soon as the `Dispatcher` assigns a leader thread to an event, it promotes a follower thread to become the new leader. At this point, the former leader and the new leader thread can execute concurrently. In this design, association of threads to services is unbounded, which means that any thread can process any event that occurs on any Transport Handle.

Component Interaction Figure 3 shows the interactions between the different Multi-Threaded Acceptor-Connector components. To simplify the description of this interaction, we divide it into three steps: from the initialization to the connection, from the connection to the data exchange, and, finally, the data exchange.

Phase 1: Initialization of the Connection
The server owns one `Dispatcher` and initiates it by calling the method `open()`. It also owns one `Thread Set` with a limited number of threads. To receive connections, the server can create one (or more) `Acceptor`(s) that will listen to one (or more) port(s). When an `Acceptor` is initialized, a passive-mode `Transport Handle` is created and bound to a network address. The `Acceptor` registers this handle in the `Dispatcher` and attaches itself to the passive handle using the methods `register()` and `attach()` respectively. Then, the `Dispatcher` waits for new events on the previously registered passive handles, by calling the method `select()`. On the other end of the channel, a client starts a connection by calling the `Connector`'s `connect()` method. This method blocks the thread of control, until the connection completes synchronously. When a new connection event arrives, the `Dispatcher` does a non-blocking invocation of the `accept()` method on the appropriate `Acceptor` (non-blocking invocations are shown in the figures by arrows with white head).

Phase 2: Initialization of Connection and Service Handler
On the client side, the `Connector` completes the initialization phase, by calling the `complete()` method. This method activates the `Service Handler` by passing the `Transport Handle` as a parameter of the method `activate()`. On the server side, after a connection request is accepted, the `Acceptor` initializes and activates the `Service Handler` which registers the `Transport Handle`

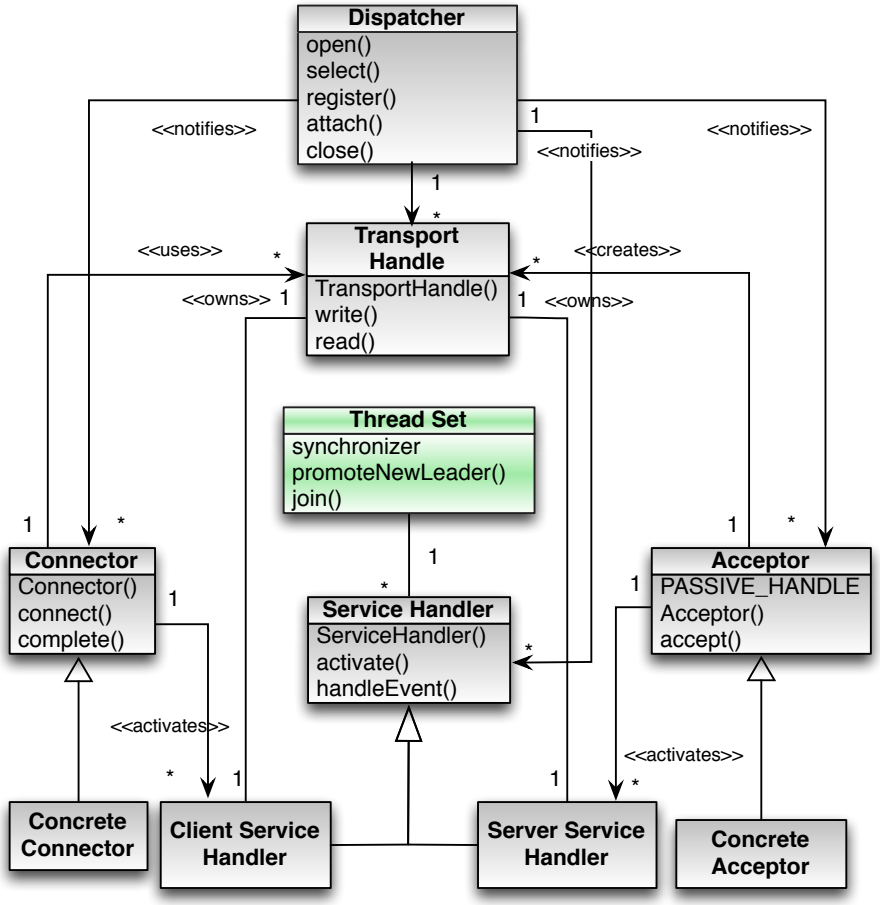

Fig. 2. Multi-Threaded Acceptor-Connector design pattern

in the `Dispatcher` and attaches itself to the handle, again, through the method `register()` and `attach()` respectively.

At this point, a new handle (identified by `h` in the figure) exists on both sides.

Phase 3: Service Processing

Then the client and the server start to exchange data. To perform the `write()` and `read()` operations, the client's `Service Handler` will typically use the `Transport Handle` directly, instead of using the `Dispatcher`[1]. On the other hand, the server's `Service Handler` waits for the `Dispatcher` to call it back for new events. To receive these events, the `Dispatcher` invokes the `read()` method of the selected `Transport Handle` in a non-blocking manner. Then, the event is given to the appropriate `Service Handler`, and the leader thread is assigned to handle the event and process the request. A new thread in the thread set is promoted as the leader, to wait for the next event. The `write()` operation is accomplished directly through the `Transport Handle`.

3.2 Fault-Tolerant Multi-Threaded Acceptor-Connector Design Pattern

To create a fault-tolerant multi-threaded design, we refined the Multi-Threaded Acceptor-Connector pattern explained in Section 3.1, by adding the following extra components, which enable recovery from connection failures: the `Stream Buffer`, the `Connection Handler`, and the `Connection`

[1] Although it could work as the server, the client may allow itself to block in `read()` and `write()` operations, because it is usually much simpler.

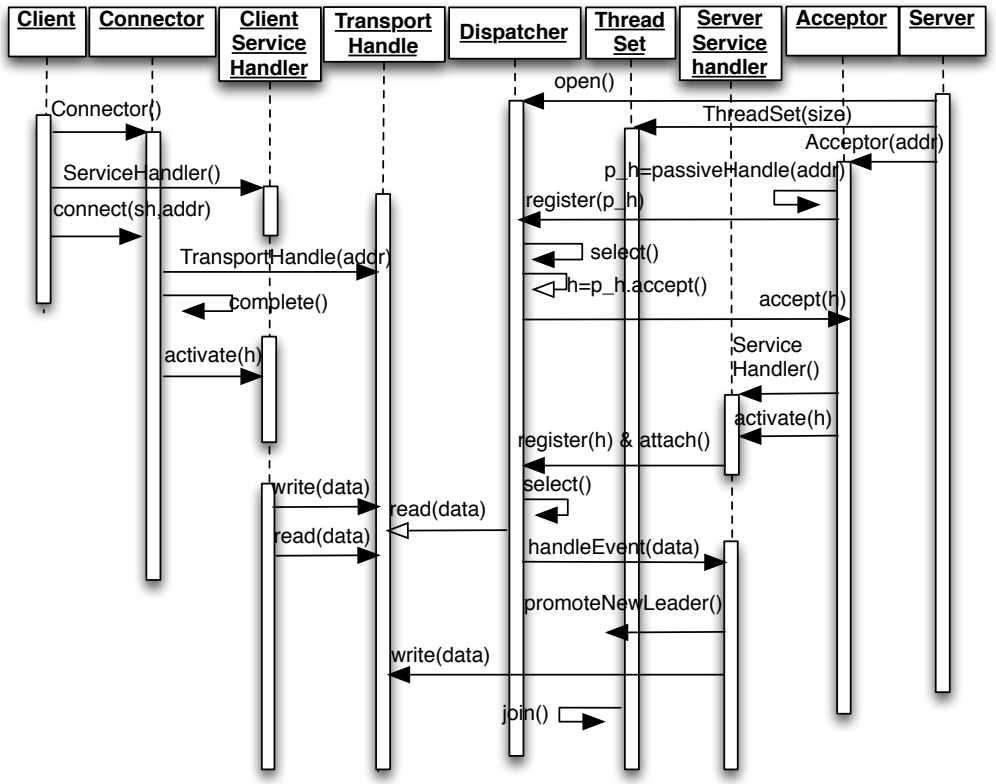

Fig. 3. Collaborations between the components in the Multi-Threaded Acceptor-Connector design pattern

`Set` (refer to Figure 4). In this section, we explain these components, as well as the collaborations between them.

Stream Buffer When a Transport Handle, like a TCP socket, fails, the connection state, including the sequence number and the number of bytes sent or received, is lost, because operating systems usually lack standard means to provide the contents or number of bytes available in internal TCP buffers. Therefore, to recover this information, we need to implement our own layer of buffering over TCP. To avoid duplicate retransmissions and acknowledgments, we resort to the `Stream Buffer`, which is based on an idea of Zandy and Miller [8]. To explain how this works, we depict three buffers in Figure 5: sender application, sender TCP and receiver TCP. The receiver got m bytes so far, whereas the sender has a total of n bytes in the buffer. Since the contents of both TCP buffers disappear on reconnection, the receiver needs to send the value m, whereas the sender must resend the last $n - m$ bytes it has in the buffer (the blue and red parts in the figure, the green part is already on the receiver side).

We could limit the size of the application send buffer if we knew that, say, k bytes were read by the receiver, to delete these k bytes from the sender buffer. 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[2]. This means that the sender only needs to keep the last $b = s + r$ sent bytes in a circular buffer, and may overwrite its data older than b bytes. Interestingly, we can avoid any modulus

[2] For 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.

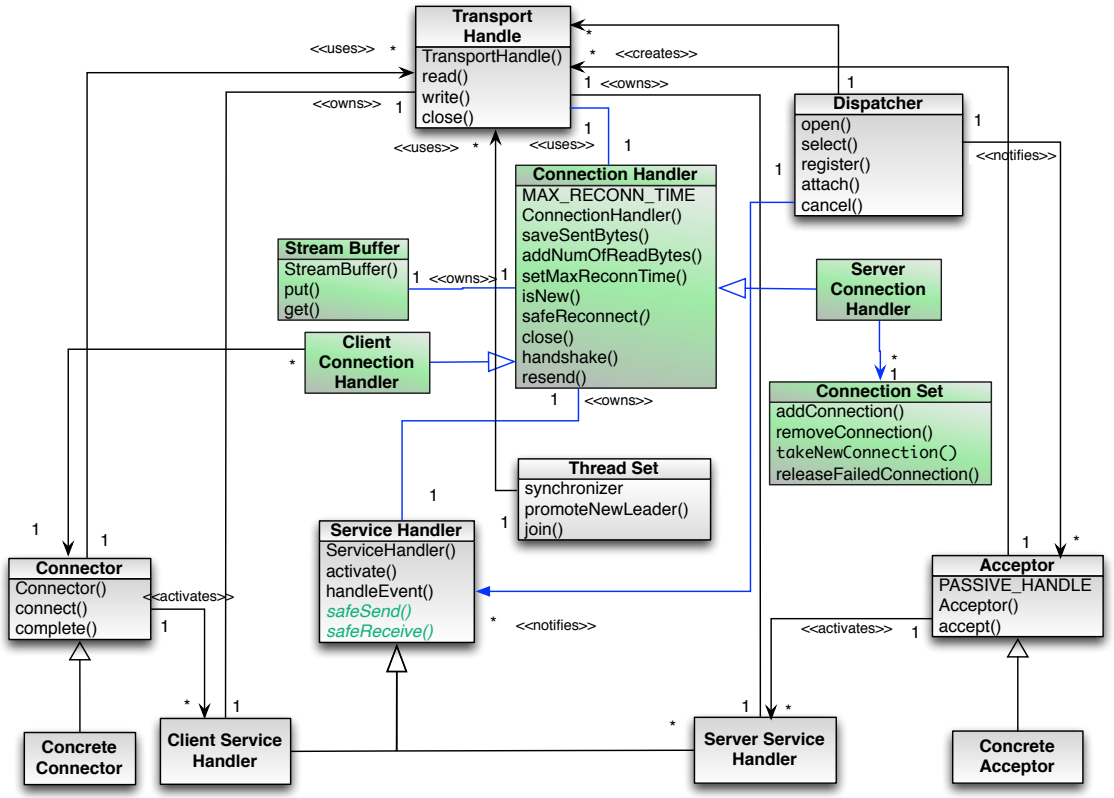

Fig. 4. Fault-Tolerant Multi-Threaded Acceptor-Connector design pattern

operation, by using two's complement arithmetic over standard 32 or 64-bit counters that keep the sent and received bytes on each side, for buffer sizes strictly smaller than 2^{32} and 2^{64} respectively[3].

Connection Handler To accomplish the recovery-purpose operations transparently from the `Service Handler`, we added the `Connection Handler` component to our design. Each `Connection Handler` owns one `Stream Buffer` and implements the actions necessary to establish a connection for the first time and also after a failure, including reconnecting and resending the lost data. 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 `Connection Handler`, due to NAT schemes or firewalls. Thus, the operations of the `Connection Handler` need to be done differently on the Connector (or client) and Acceptor sides (or server). For this reason, we added two concrete `Connection Handlers` in the design: a `Client Connection Handler` and a `Server Connection Handler`.

To simplify the reconnection, each fault-tolerant connection has a unique identifier. Refer to Figure 6. Once a reconnection occurs, the client sends this identifier and the number of bytes it received up to the disconnection. The server replies with a similar message. Finally, the client and server send the buffered data that the other peer did not receive due to the connection failure. Setting up a new connection is slightly simpler, because peers do not exchange buffered data, only of the sizes of the send and receive buffers of the `Transport Handle`. In this case, the client sets the identifier to 0, and the server generates a new immutable identifier for the fault-tolerant connection in the response.

[3] Note that apart from these limits, the buffers can have arbitrary sizes, according to the sender plus receiver TCP buffer sizes.

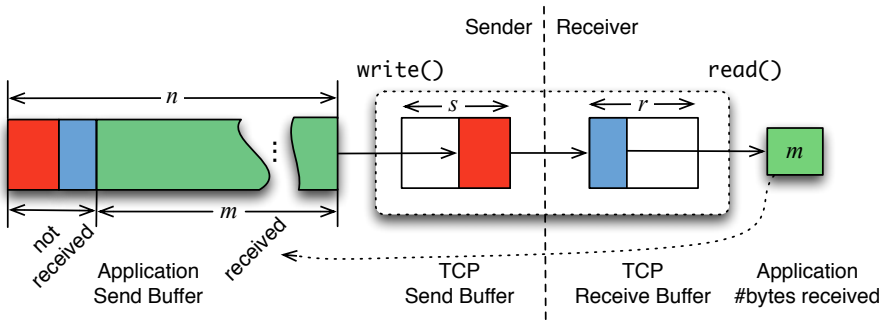

Fig. 5. Sender and receiver buffers

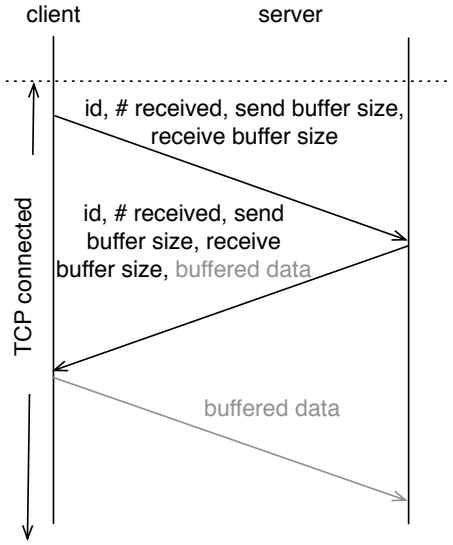

Fig. 6. The handshake procedure

Connection Set On the server side, we use a new component, named `Connection Set`, to replace a failed connection with a new one. To do so, we need to keep the information (i.e., the identifiers) of the connections. Thus, 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.

This `Connection Set` allows a `Connection Handler`, whose connection is failed, to wait for a new one by invoking the method `takeNewConnection()`. 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.

The method `releaseFailedConnection()` of the `Connection Set` allows new `Connection Handlers`, which are created for a failed connection during the recovery process, to deliver the new handle to the older `Connection Handlers` still waiting for reconnection. Hence, the main purpose of the `Connection Set` is to synchronize threads upon connection failures and reconnections.

Interactions between the New Fault-Tolerant Components As we did in the previous section, we divide the interactions between the new components into different phases, considering two main scenarios: a failure-free scenario (Figure 7) and a faulty scenario (Figure 8).

Phase 1: Initialization of the Connection
In general, each server owns one `Dispatcher`, one `Thread Set` and one `Connection Set`. When-

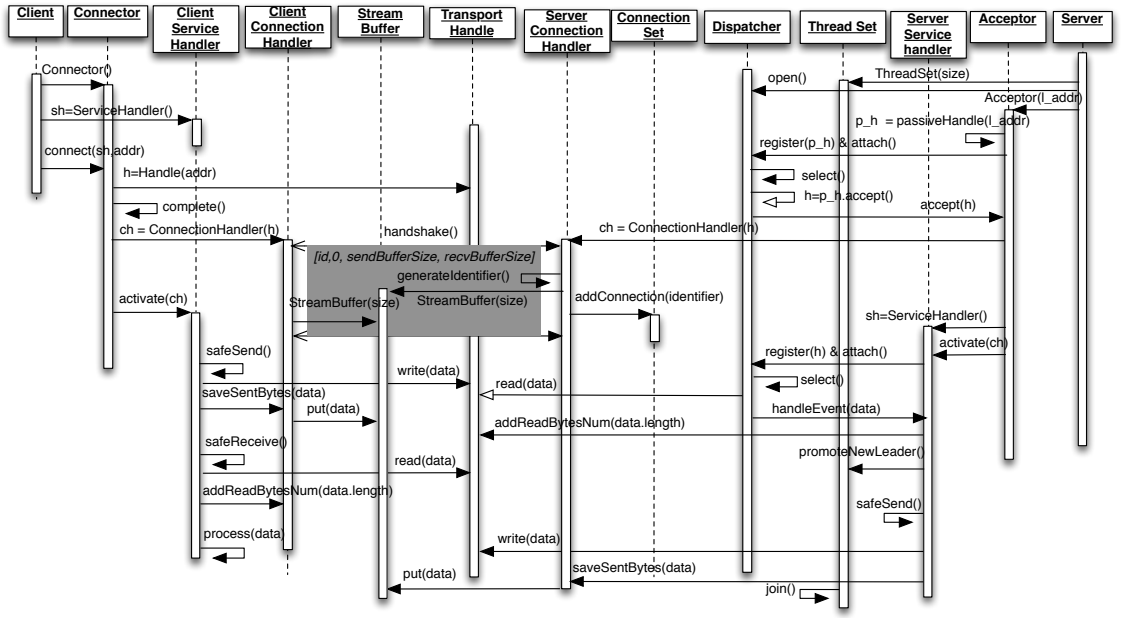

Fig. 7. Interactions between the components of Fault-Tolerant Multi-Threaded Acceptor-Connector design pattern in a failure-free scenario

ever a server starts running, it creates a `Thread Set`, with a given size. Next, it initializes the `Dispatcher`, by calling its method `open()`. Figure 7 shows these early steps as well as all interactions between the components. We omit the description of the first phase, because it does not change.

Phase 2: Connection and Service Handler Initialization
Since FTMTAC uses a few more components, like the `Connection Handler`, to tolerate connection failures, this phase must change accordingly. The client `Connector` completes the connection process, by creating a `Connection Handler` and passing it to the `Service Handler`, by calling the method `activate()`. On the server side, when the connection is ready, the `Acceptor` creates a new `Connection Handler`, which inserts its own unique identifier into the `Connection Set`. This identifier is only removed from the `Connection Set` when the `Service Handler` is stopped and the `Connection Handler` is closed, i.e., when the server considers the interaction to be definitely finished. Once the `Connection Handler`s are initialized, they engage in the handshake as we described before (the gray part of the figure). During the handshake, a `Stream Buffer` with the necessary size is created in both `Connection Handler`s.

Phase 3: Service Processing
A major difference in this phase comes from the need to keep track of the data exchanged so far. Both sides save the sent data in the `Stream Buffer` through the `Connection Handler`s, by calling the method `safeSend()`. They also keep the number of bytes read in the `Connection Handler`, by calling the method `addNumOfReadBytes()`. The methods `safeSend()` and `safeReceive()` take care of most operations involved here: together they send and receive the data, they save the data sent, and keep the number of bytes sent and received. Thus the `Service Handlers` use `safeSend()` and `safeReceive()` methods to safely write and read data through the `Transport Handle`s.

As before, client and server usually work asymmetrically. While the client's `Service Handler` reads the data messages, by invoking the method `read()` of the `Transport Handle`, the server's `Service Handler` waits to receive `Dispatcher` events.

Phase 4: Failure Handling
Figure 8 presents failure handling details (dashed arrows represent the return of the invocations). Upon failure, the client's `Service Handler` invokes the `Connection Handler` for recon-

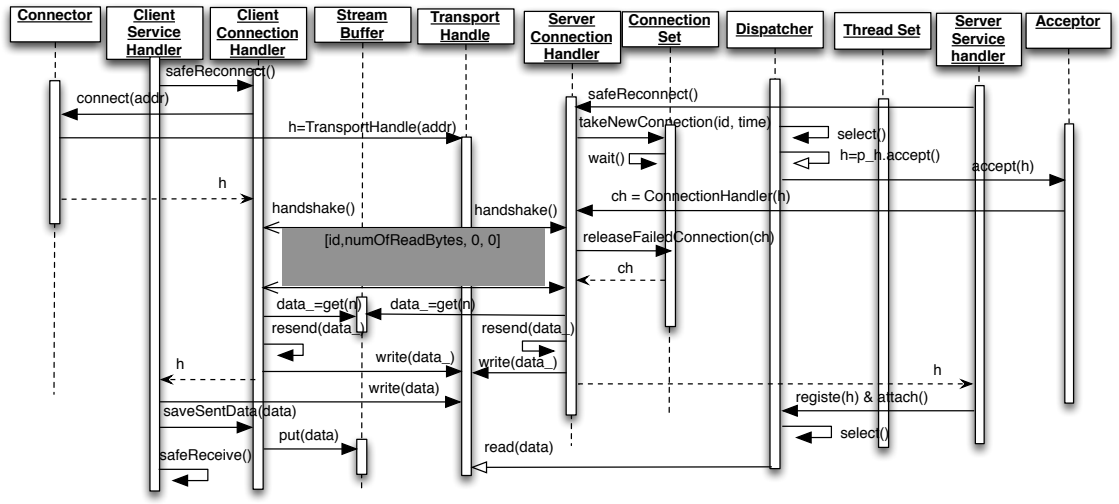

Fig. 8. Interactions between the components of Fault-Tolerant Multi-Threaded Acceptor-Connector design pattern after a failure occurs

nection. The client repeatedly tries to reconnect during a limited, but configurable time (i.e `MAX_RECONN_TIME`). The server checks the `Connection Set` for a new handle, and sets a timeout to wait if there is no new handle in the set[4]. If the timeout expires, an exception (i.e. `ReconnectionException`) is returned, allowing the server to delete all connection resources, turning reconnection impossible past this point.

In case the client's request for reconnection comes after the server's reconnection timeout, the handshake is never completed, and, after the client's own timeout a `ReconnectionException` is raised on the client. Both client and server timeouts might be determined by developers using the method `setMaxReconnTime()`.

If the reconnection is successful, the `Connector` returns the new handle to the `Connection Handler`. Then, the `Connection Handler` performs a new handshake with the server. On the server side, when the connection is accepted, a new `Connection Handler` is created. When the `Connection Handler` receives the control messages from the client, it identifies the connection as a reconnection, and invokes the `releaseFailedConnection()` of the `Connection Set` giving it the new handler. The `Connection Set` returns the new handle to the `Connection Handler` (that was waiting for this handle after the failure), which updates it and finishes the handshake. Then the new handle is returned to the `Service Handler`, who registers it in the `Dispatcher`. From this point on, the connection is set and peers can resume their interaction.

4 Implementation Of the Design Pattern

In this section we illustrate the implementation and utilization of our design pattern. We have implemented the Fault-Tolerant Multi-Threaded Acceptor-Connector (`ftmtac`) design pattern for Java TCP. We made the source code of this design pattern available for download [10]. We also implemented the related Multi-Threaded Acceptor-Connector (`mtac`) and the plain Acceptor-Connector (`ac`) patterns. We considered a scenario with multiple clients, each setting up one connection to a multi-threaded server application, which uses a `Dispatcher` and a `Thread Set` to handle these multiple connections.

[4] One should notice that the server may detect the crash later than the client, which may have reconnected already.

4.1 Connection Establishment

To identify the server side of the connection, we need an IP address and a port number or a range of ports. In the constructor of our example, the server creates a `Thread Set` with 100 threads and a `Dispatcher`. The `Dispatcher` takes care of existing data streams and new connections, redirecting them to the appropriate `Connection Handler` or `Acceptor`, respectively. Depending on the number of the ports, one or more `Acceptor`s could be initialized. In this example we use only one port. The `Server` must pass the name of the class implementing the server handler to the `Acceptor`, to let it know how to initialize the handler. One should notice that passing a string with the name, easily allows the Acceptor to later initialize as many objects as it needs. Passing a new instance of an object by doing `new ServerServiceHandler()` would not be an option here, because the `Acceptor` needs one `new ServerServiceHandler()` per connection.

```
public class Server {
  int port = 9000;
  // Server Constructor
  public Server() throws IOException {
    new ThreadSet(100);
    new Dispatcher();
    new Acceptor(port, "ServerServiceHandler");
  }
}
```

On the other endpoint, clients typically do not need a `Dispatcher`, because they only manage one or two connections. For this reason, in our example, we simply invoke the static method `connect()` of the `Connector`, by passing the network address of the server and an instance of a `Client Service Handler` for further activation.

```
public class Client {
  // Client Constructor
  public Client() throws UnknownHostException, IOException {
    Connector.connect(server, port, new ClientServiceHandler());
  }
}
```

The `Connector` creates a new connection, by initializing the `Transport Handle` with a connection to the server. Then, the `Connector` calls the `complete()` method to activate the `Service Handler`, by passing the `Client Connection Handler` created before.

```
public class Connector {
  public static void connect(String server, int port, ServiceHandler sh) throws
       UnknownHostException, IOException {
    TransportHandle handle = new TransportHandle(new Socket(server, port));
    complete(sh,handle);
  }
  // This method is used to activate the service handler.
  private static void complete(ServiceHandler sh, TransportHandle handle) throws IOException{
    ClientConnectionHandler ch = new ClientConnectionHandler(handle);
    sh.activate(ch);
  }
}
```

Compared to the `Connector`, the `Acceptor` is more complicated. It owns a passive handle and a path to the service handler that is going to be initialized and activated for each connection accepted. Upon initialization of an `Acceptor`, the path given is assigned to the variable dedicated for this purpose, and a passive transport handle, bound to a network address, is created. The handle is configured as non-blocking, and the `Acceptor` attaches itself to the passive handle, to be called back by the `Dispatcher`. When a new connection arrives, the `Dispatcher` invokes the `accept()` method of the `Acceptor`, which first creates a new `Server Connection Handler`, and then checks if the incoming connection is a brand new connection or if it is replacing a failed one. In the former case, the `Acceptor` initiates a `Service Handler` and activates it, by passing the

Connection Handler.

```
public class Acceptor{
  // The passive handle is to accept the requests for new connections.
  private ServerSocketChannel passive_handle = null;
  // This keeps the path to the Server Service Handler that implement the Service Handler abstract
      class.
  private String serviceHandlerImpPath = "ServiceHandlerServerImp";
  // Acceptor Constructor
  public Acceptor(int port, String imp) throws IOException {
    this.serviceHandlerImpPath = imp;
    passive_handle = ServerSocketChannel.open();
    passive_handle.socket().bind(new InetSocketAddress(port));
    passive_handle.configureBlocking(false);
    Dispatcher.attach(passive_handle, this);
  }
  // This method handles a new connection.
  public void accept(TransportHandle handle) throws IOException {
    ServerConnectionHandler ch = new ServerConnectionHandler(handle);
    if (ch.isNew()) {
      try {
        Class theClass = Class.forName(this.serviceHandlerImp);
        ServiceHandler sh = (ServiceHandler) theClass.newInstance();
        sh.activate(ch);
      } catch (IOException e) {
      } catch (ClassNotFoundException e) {
      } catch (InstantiationException e) {
      } catch (IllegalAccessException e) {
      }
    }
  }
}
```

In our Java NIO (Non-blocking I/O) example, a `Dispatcher` owns a `Selector`, which allows registering, selecting and canceling the handles, including the transport and the passive, still unconnected, handles. The `Selector` is initialized by invoking the `open()` method. A thread from the `Thread Set` is permanently assigned to the `Dispatcher`, to check for the occurrence of new events on the handles that have been already registered.

The `Dispatcher` provides two `attach()` methods, which allow the `Acceptor` and `Service Handler` to register their passive and transport handles respectively. After the registration, they attach their own objects to the keys returned by the `Selector`. The type of the key is "acceptable" for a passive handle, and "readable" for a transport handle.

When the `Dispatcher` runs, it first gets the list of keys belonging to the registered handles, by calling the `select()` method from the `Selector`. Then it checks all the keys (i.e. handles) for new events. If the selected key is "readable" (e.g. Java NIO `SocketChannel`), the `Dispatcher` reads the bytes and then calls the attached `Service Handler`, to handle the read data, if any. Otherwise, if a selected key is "acceptable" (e.g. Java NIO `ServerSocketChannel`), the `Dispatcher` checks the passive handle for incoming connections, by calling the `accept()` method. Then, if there is a new connection (i.e. in this case a new transport handle is returned), it is given to the appropriate `Acceptor`, to take care of it. Any exception occuring while the `Dispatcher` is reading data, will be notified to the appropriate `Service Handler` through a different `handleEvent()` method. The `Dispatcher` follows this procedure until the `stop()` method is invoked. All the details are shown below.

```
public class Dispatcher implements Runnable {
  private static Selector selector = null;
  private boolean stopped = false;
  // Dispatcher constructor
  public Dispatcher() throws IOException {
    selector = Selector.open();
    ThreadSet.execute(this);
  }
  // This method is used to register a transport handle (e.g. SocketChannel) and attach a Service
      Handler to its key
  public static void attach(SocketChannel socket,
      ServiceHandler serviceHandler) throws ClosedChannelException {
    selector.wakeup();
```

```
    SelectionKey key = socket.register(selector, SelectionKey.OP_READ);
    key.attach(serviceHandler);
  }
  // This method is used to register a passive handle (e.g. ServerSocketChannel) and attach a
       Acceptor to its key
  public static void attach(ServerSocketChannel serverSocket,
      Acceptor acceptor) throws ClosedChannelException {
    selector.wakeup();
    SelectionKey key = serverSocket.register(selector,
        SelectionKey.OP_ACCEPT);
    key.attach(acceptor);
  }
  // Dispatcher Runs.
  @Override
  public void run() {
    while (!stopped) {
      try {
        selector.select();
        if (selector.selectedKeys().size() == 0)
          continue;
      } catch (IOException e1) {
        break;
      }

      for (Iterator<SelectionKey> i = selector.selectedKeys().iterator(); i
          .hasNext();) {
        SelectionKey key = i.next();
        // acceptors
        try {
          if (key.isAcceptable()) {
            try {
              SocketChannel sh = ((ServerSocketChannel) key
                  .channel()).accept();
              if (sh != null)
                ((Acceptor) key.attachment())
                    .accept(new TransportHandle(sh));
            } catch (IOException e) {
              key.cancel();
            }
          } else if (key.isReadable()) {
            ByteBuffer inputdata = ByteBuffer.allocate(1024);
            try {
              int read = ((ReadableByteChannel) key.channel())
                  .read(inputdata);
              if (read == -1) {
                key.cancel();
                ((ServiceHandler) key.attachment()).stop();
              } else if (read > 0) {
                byte[] event = ByteUtils.subByte(
                    inputdata.array(), 0, read);
                ((ServiceHandler) key.attachment()).handleEvent(event);
              }
            } catch (IOException e) {
              // exception on read
              ((ServiceHandler) key.attachment()).handleEvent(e);
              key.cancel();
            }
          }
        } catch (CancelledKeyException e) {
        }
      }
    }
  }
  // Dispatcher Stops.
  public void stop() throws IOException {
    stopped = true;
    selector.close();
    selector = null;
  }
}
```

The `Thread Set` has a limited number of threads, which might be configured according to the application and resources available. We used the Java class `Executors` to create a fixed thread pool with the exact specifications defined for a `Thread Set`. To control the pool, the Java `ThreadSet` class provides static methods that access the underlying `Exectuors`' methods.

```
public class ThreadSet {
  private static ExecutorService executor = null;

  public ThreadSet(int size) throws Exception{
    if(executor == null)
      executor= Executors.newFixedThreadPool(size);
    else throw new Exception("The ThreadSet has been already initialized and some threads might be
        running!");
  }
  public static void execute(Runnable th){
    executor.execute(th);
  }
  public static void shutdown(){
    executor.shutdown();
    try {
      executor.awaitTermination(30, TimeUnit.MINUTES);
    } catch (InterruptedException e) {
    }
    executor=null;
  }
  public static void shutdownNow(){
    executor.shutdownNow();
    executor=null;
  }

  public static boolean isInitialized(){
    if (executor == null)
      return false;

    return true;

  }
}
```

4.2 Connection Handler and Service Initialization

A `Connection Handler` is identified by a unique identifier. It owns a `Transport Handle` and a `Stream Buffer`. The `Stream Buffer` stands in the path of all data. The size of any data that is read must be added to `readBytes`, which may wrap around as explained in Section 3.2. We keep the number of bytes written into the `Stream Buffer` in `sentBytes`. The `Connection Handler` provides the methods `saveSentBytes()` and `addNumOfReadBytes()` to the `Service Handler` for these purposes.

To complete the initialization of the Connection Handler, client and server must do a handshake. Since the handshake is done differently in the client and the server, the method `handshake()` is defined as an abstract method that needs to be overridden in the `Client Connection Handler` and the `Server Connection Handler`. To recover from the connection crash, the `Service Handler` calls the method `safeReconnect()` of the `Connection Handler`. The reconnection and recovery procedure is also different in both peers. Thus, we defined an abstract method that needs to be implemented. Developers can set a maximum waiting time needed for reconnection through the method `setMaxReconnTime()`. The method `resend()` is used to resend the lost data after recovering from the crash.

```
public abstract class ConnectionHandler {
  protected int identifier = 0;
  protected TransportHandle handle = null;
  protected StreamBuffer buffer = null;
  protected int sentBytes = 0;
  protected int readBytes = 0;
  protected long MAX_RECONNECTION_TIME = 300;

  public ConnectionHandler(TransportHandle h) throws IOException {
    handle = h;
  }
```

```
  public int getIdentifier() {
    return identifier;
  }
  public TransportHandle getHandle() {
    return handle;
  }
  public void saveSentBytes(byte[] data) {
    sentBytes += data.length;
    buffer.add(data);
  }
  public void addNumOfReadBytes(int read) {
    readBytes += read;
  }

  public void close() throws IOException {
    handle.close();
    buffer.deleteAll();
  }
  public void setMaxReconnTime(long t){
    MAX_RECONNECTION_TIME = t;
  }
  protected abstract void handshake() throws IOException;
  public abstract TransportHandle safeReconnect() throws ReconnectionException;
  protected void resend(int peerReceived) throws IOException {
    handle.write(buffer.get(this.sentBytes - peerReceived));
  }
}
```

The ability to resend data that was lost in the channel is crucial to our design pattern. We use a `Stream Buffer`, whose size is equal to the size of the local socket send buffer plus the remote socket's receive buffer. Send and receive buffers in the TCP sockets are usually inaccessible, thus providing little help when a connection crashes. Therefore, for recovery, each sender needs to keep all bytes that are possibly in transit. The `Connection Handler` keeps the number of bytes sent and received on both ends of the connection, so, when a connection crashes, the difference between the number of bytes sent and the number of bytes received on the other side, tells the total number of bytes to resend. The source code for the `Stream Buffer` is presented below. It has an array of bytes and an index referring to the position to insert the next byte. It provides the method `add()` to insert an array of bytes to the buffer, a method `get()` to retrieve a given number of bytes from the buffer, and a method `deleteAll()` to clear the buffer.

```
public class StreamBuffer {
  byte[] bytes = null;
  int index = 0;
  public StreamBuffer(int s) {
    bytes = new byte[s];
  }
  public int getSize() {
    return bytes.length;
  }
  public void add(byte[] bs) {
    add(bs, 0, bs.length);
  }
  public void add(byte[] bs, int off, int len) {
    for (int i = off; i < off + len; i++){
      this.bytes[index] = bs[i];
      index = (this.index+1) % this.bytes.length;
    }
  }
  public byte[] get(int bytesToSend) {
    if (bytesToSend == 0)
      return null;
    int start = (index - bytesToSend) % bytes.length;
    byte[] remained = new byte[bytesToSend];
    for (int i = 0; i < bytesToSend; i++)
      remained[i] = bytes[(start+i) % bytes.length];
    return remained;
  }
  public void deleteAll() {
    this.bytes = null;
    index=0;
```

```
  }
}
```

Upon creation of the `Connection Handler` in the client and server, a handshake, initiated by the `Client Connection Handler`, is performed to exchange some information as we show in Figure 6. The exchange starts with an immutable unique identifier, generated by the server when the connection starts for the first time (hence the clients sends a 0 on the first connection). In addition, the client sends the size of the send and receive buffers of the transport handles (e.g. TCP Socket) and the number of bytes received so far (0 on the first time). The server does the same, taking the initiative of setting the identifier when it comes as 0. Exchange of buffered data might occur at this point, if there is any. This allows the `Connection Handler` to resend the lost data after recovery from the failure.

After exchanging the above information, if the connection is new, a `Stream Buffer` with the necessary size is created in both `Connection Handlers` (i.e., the size of the buffer depends on the size of the buffers in the TCP sockets as explained in section 3.2). Otherwise both sides start to resend the lost data. We first show the client side:

```
// Connection Handler initialization in Client
public class ClientConnectionHandler extends ConnectionHandler {

  public ClientConnectionHandler(TransportHandle h) throws IOException {
    super(h);
    handshake();
  }
  @Override
  public void handshake() throws IOException {

    CtrlMessage outputControlMessage = new CtrlMessage(identifier, readBytes,
         handle.getSocket().getSendBufferSize(), handle.getSocket().getReceiveBufferSize());

    handle.write(ByteUtils.serialize(outputControlMessage));
    byte[] inputData = new byte[1024];
    int read = handle.read(inputData);

    if (read == -1)
      throw (new EOFException());

    CtrlMessage inputControlMessage = (CtrlMessage) ByteUtils.deserialize(inputData, read);

    if (identifier == 0) {
      identifier = inputControlMessage.getIdentifier();
      super.buffer = new StreamBuffer(handle.getSocket().getSendBufferSize()
          + inputControlMessage.getRecvBufferSize());

    } else{
      resend(inputControlMessage.getReceivedBytes());
    }
  }
  ...
}
```

And next, the server side:

```
// Connection Handler initialization in Server
public class ServerConnectionHandler extends ConnectionHandler {

  boolean isNew = true;
  int numOfBytesReceived = 0;

  public ServerConnectionHandler(TransportHandle h) throws IOException {
    super(h);
    handshake();
  }
  @Override
  protected void handshake() throws IOException {
    byte[] inputData = new byte[1024];
```

```
    int read = handle.read(inputData);
    CtrlMessage inputControlMessage = (CtrlMessage) ByteUtils.deserialize(
        inputData, read);

    if (inputControlMessage.getIdentifier() == 0){
      this.identifier=Identifier.next();
      CtrlMessage outputControlMessage = new CtrlMessage(identifier,
          readBytes, handle.getSocket().getSendBufferSize(),
              handle.getSocket().getReceiveBufferSize());
      handle.write(ByteUtils.serialize(outputControlMessage));
      buffer = new StreamBuffer(handle.getSocket().getSendBufferSize()
          + inputControlMessage.getRecvBufferSize());
      ConnSet.addConnection(this.identifier);
    }else {
      identifier = inputControlMessage.getIdentifier();
      numOfBytesReceived = inputControlMessage.getReceivedBytes();
      isNew = false;
      ConnSet.releaseFailedConnection(this);
    }
  }

  public boolean isNew() {
    return isNew;
  }
  ...
}
```

The `Service Handler` implements an application service, typically playing the client role, server role or both roles. It owns a `Transport Handle` and a `Connection Handler`, to handle possible connection crashes. It provides a hook method (`activate()`) that is called by an `Acceptor` or `Connector` to activate the application service once the connection is established.

If the `Dispatcher` is used in the application (typically on the server), the `Service Handler` has to configure the `Transport Handle` as a non-blocking handle, then it registers the handle and attaches itself through the `Dispatcher`. Alternatively, the programmer may assign a thread to run the `Service Handler`.

We also provide two methods to read and write the data streams in the `Service Handler`: `safeSend()` and `safeReceive()`. These methods allow the application to write and read harmlessly, as they go through the `Connection Handler`, thus saving data in the `Stream Buffer`, correctly managing the counters, and appropriately handling the failures. We already implemented these methods, to enable their simple use by developers.

We use two separate methods for handling events that may happen in the `Dispatcher`: arrival of new data, and occurrence of an `IOException`. To handle an exception, a new thread must be allocated to take care of the reconnection. This method is implemented in the `Service Handler` abstract class.

The abstract methods `run()` (i.e. from the Runnable abstract class), `restart()`, `stop()`, and `handleEvent()`, to take care of the new data, should be implemented by the developers based on the application logic.

```
public abstract class ServiceHandler implements Runnable {
  protected ConnectionHandler connectionHandler = null;
  protected TransportHandle handle = null;
  protected AtomicBoolean stopped = new AtomicBoolean(false);

  public void activate(ConnectionHandler ch) throws IOException {
    connectionHandler = ch;
    handle = connectionHandler.getHandle();
    handle.getSocket().setTcpNoDelay(true);

    if (Dispatcher.isOpen()) {
      handle.getSocketChannel().configureBlocking(false);
      Dispatcher.attach(handle.getSocketChannel(), this);
    } else {
      if (ThreadSet.isInitialized())
        ThreadSet.execute(this);
      else
        (new Thread(this)).start();
    }
```

```
  }
  protected void safeSend(byte[] data) throws ReconnectionException {
    boolean done = false;
    while (!done) {
      try {
        handle.write(data);
        connectionHandler.saveSentBytes(data);
        done = true;
      } catch (IOException e) {
        if (handle.getSocket().isClosed())
          throw new ReconnectionException(
              "The connection has been already closed!");
        else{
          handle = connectionHandler.safeReconnect();
          try {
            handle.getSocket().setTcpNoDelay(true);
            if (Dispatcher.isOpen()) {
              handle.getSocketChannel().configureBlocking(false);
              Dispatcher.attach(handle.getSocketChannel(), this);
            }
          } catch (SocketException e1) {
            e1.printStackTrace();

          } catch (IOException e1) {
            e1.printStackTrace();
          }

        }
      }
    }
  }

  protected byte[] safeReceive() throws ReconnectionException {
    byte[] inputdata = new byte[1024];
    boolean done = false;
    int read = 0;
    while (!done) {
      try {
        read = handle.read(inputdata);
        if (read > 0) {
          done = true;
          connectionHandler.addNumOfReadBytes(read);
        }
      } catch (IOException ioe) {

        if (handle.getSocket().isClosed()) {
          throw new ReconnectionException(
              "The connection has been already closed!");
        } else{
          handle = connectionHandler.safeReconnect();
          try {
            handle.getSocket().setTcpNoDelay(true);
          } catch (SocketException e) {
            e.printStackTrace();
          }
        }

      }
    }
    return ByteUtils.subByte(inputdata, 0, read);
  }

  protected void handleEvent(IOException event) {
    ThreadSet.execute(new ExceptionHandler(this.handle,
        this.connectionHandler, this));
  }

  protected abstract void restart();
  protected abstract void stop();
  protected abstract void handleEvent(byte[] event);

}

class ExceptionHandler implements Runnable {

  ServiceHandler serviceHandler;
  TransportHandle handle;
  ConnectionHandler connectionHandler;

  public ExceptionHandler(TransportHandle h, ConnectionHandler ch,
```

```
      ServiceHandler sh) {
    serviceHandler = sh;
    handle = h;
    connectionHandler = ch;
  }

  @Override
  public void run() {
    try {
      handle = connectionHandler.safeReconnect();
    } catch (ReconnectionException e1) {
      serviceHandler.stop();
    }
    try {
      handle.getSocket().setTcpNoDelay(true);
      handle.getSocketChannel().configureBlocking(false);
      Dispatcher.attach(handle.getSocketChannel(), serviceHandler);
    } catch (Exception e) {
    }
  }
};
```

4.3 Application Processing

Here, we give a very simple example of implementation of the `Service Handler`'s abstract methods. The `Client Service Handler` sends an increasing number, starting at 0 and ending at 1000, to the server, before stopping (in the `run()` method). After sending each number, the `Client Service Handler` waits to receive the reply from the peer's `Service Handler`, and then processes it (method `handleEvent()`).

In this example, we need to allocate 0 to the number that is being sent for restarting the service (i.e. in method `restart()`), and to close the `Transport Handle` and the `Connection Handler` for stopping the service (i.e. in method `stop()`). The example (refer to the source code shown below) shows how the developers can simply use the methods `safeSend()` and `safeReceive()` to exchange data with the other peer.

```
public class ClientServiceHandler extends ServiceHandler {
  int outputData=0;
  @Override
  public void run() {
    while (outputData<1000) {
      try {
        safeSend((String.valueOf(outputData++)).getBytes());
        byte[] inputdata = safeReceive();
        handleEvent(inputdata);
      } catch (ReconnectionException e) {
        stop();
      }
    }
    stop();
  }

  @Override
  protected void handleEvent(byte[] event) {
    System.out.println(new String((byte[])event));
  }
  @Override
  protected void restart() {
    outputData=0;
  }
  @Override
  protected void stop(){
    stopped.set(true);
    try {
      connectionHandler.close();
    } catch (IOException e) {

    }
  }
}
```

On the other side, the `Service Handler` is initialized once its method `activate()` is invoked by the `Acceptor`. We use a queue to order the requests and use locking to ensure First-Come-First-Served execution by a single thread (see methods `run()` and `handleEvent()`). The method `handleEvent()` delivers the data read by the `Dispatcher` to the `Service Handler`. It adds the event to the data queue and assigns a thread to this object, if no thread is assigned yet. In the `run()` method, the assigned thread successively polls and processes events from the data queue, until the queue is empty. When no more events are available, the thread joins back the `Thread Set`.

```
public class ServerServiceHandler extends ServiceHandler {
  protected Queue<byte[]> events;
  Boolean activeThread = false;

  @Override
  public void activate(ConnectionHandler ch) throws IOException {
    super.activate(ch);
    events = new LinkedList<>();
  }

  @Override
  protected synchronized void handleEvent(byte[] event) {
    events.add(event);
    if (activeThread == false) {
      ThreadSet.execute(this);
      activeThread = true;
    }
  }

  @Override
  public void run() {
    byte[] event = null;
    do{
      synchronized (this) {
        event = events.poll();
        if (event == null)
          activeThread = false;
      }
      if (event!=null)
        process(event);
    }while((!stopped.get() && event != null));
  }

  public void process(byte[] inputdata) {
    try {
      safeSend(inputdata);
    } catch (ReconnectionException e) {
      stop();
    }
  }

  @Override
  protected void restart() {

  }

  @Override
  protected void stop() {
    stopped.set(true);
  }
}
```

4.4 Connection Failure Handling

Due to NAT schemes or firewalls, when a connection crashes, the initiative to reconnect always belongs to the client. Thus, the server stays in a passive mode until either a new connection arrives or a timeout occurs.

Passive Reconnection As soon as the `Service Handler` detects a failure in a connection, it calls the `safeReconnect()` method. The server `Connection Handler` calls the method `takeNewConnection()` of the `Connection Set`, by passing the connection identifier and `MAX_RECONNECTION_TIME`. This call is blocking, thus making the `Connection Handler` wait until a new handle is returned or the timer goes off. If a new handle returns, then it completes the handshake with the client, by entering the handshake scheme we mentioned before. Otherwise, the `ReconnectionException` is returned to the `Service Handler`.

```
// Server Connection Handler: reconnection
  @Override
  public TransportHandle safeReconnect() throws ReconnectionException {

    ServerConnectionHandler sch = (ServerConnectionHandler) ConnSet
        .takeNewConnection(identifier, MAX_RECONNECTION_TIME);

    if (sch == null)
      throw new ReconnectionException("Server reconnection timeout.");

    handle = sch.getHandle();
    numOfBytesReceived = sch.numOfBytesReceived;

    try {
      CtrlMessage outputControlMessage = new CtrlMessage(identifier, readBytes,
          handle.getSocket().getSendBufferSize(), handle.getSocket()
              .getReceiveBufferSize());

      handle.write(ByteUtils.serialize(outputControlMessage));
      resend(numOfBytesReceived);
    } catch (IOException e) {
      safeReconnect();
    }
    return handle;
  }
```

The `Connection Set` keeps the list of identifiers of alive and failed connections. It serves as a synchronizing point, letting the `Server Connection Handler` and the `Transport Handle` synchronize for a new connection after a failure.

```
public class ConnectionSet {

  private static ConcurrentHashMap<Integer, Server Connection Handler> recoveredConnections = new
       ConcurrentHashMap<>();
  private static Set<Integer> allConnections = new HashSet<>();

  public static  Server Connection Handler takeNewConnection(int identifier,int t) {
    long timeout = t * 60 * 1000;
      Server Connection Handler fsocket = null;
    synchronized (recoveredConnections) {
      long start= System.currentTimeMillis();
      fsocket = recoveredConnections.remove(identifier);
      while (fsocket == null && timeout > 0) {
        try {
          recoveredConnections.wait(timeout);
          fsocket = recoveredConnections.remove(identifier);
          timeout = timeout - (System.currentTimeMillis()-start);
        } catch (InterruptedException e) {}
      }
    }
    return fsocket;
  }

  public static void releaseFailedConnection( Server Connection Handler ch)
      throws IOException {
    if (allConnections.contains(ch.getIdentifier()))
      synchronized (recoveredConnections) {
        recoveredConnections.put(ch.getIdentifier(), ch);
        recoveredConnections.notifyAll();
      }
  }

  public static void removeConnection(int identifier) {
    recoveredConnections.remove(identifier);
    allConnections.remove(identifier);
```

```
  }

  public static void addConnection(int identifier) {
    allConnections.add(identifier);

  }
  public static void clear(){
    recoveredConnections.clear();
    allConnections.clear();
  }
}
```

Active Reconnection On the client side, the reconnection scenario is quite different. The `Connection Handler` periodically tries to reconnect to the server. After successfully reconnecting, the client starts a handshake to exchange the data needed for recovery. Then, it resends the data that was lost due to the failure. If the client could not establish a new connection during the given maximum reconnection time, it returns an exception to the `Service Handler`.

```
// Client Connection Handler: reconnction
  @Override
  public synchronized TransportHandle safeReconnect() throws ReconnectionException {
    if (handle.getSocket().isConnected() && !handle.getSocket().isClosed()
        && handle.getSocket().isBound())
      return handle;

    boolean reconnected = false;

    long start_reconnection= System.currentTimeMillis();
    while (!reconnected) {

      try {
        handle = Connector.connect(handle.getSocket().getInetAddress(),
            handle.getSocket().getPort());
        handshake();
        reconnected = true;
      } catch (IOException e) {
        if ((System.currentTimeMillis() - start_reconnection) > (this.MAX_RECONNECTION_TIME *
            60000))
          break;
        try {
          Thread.sleep(10000);
        } catch (InterruptedException e1) {

        }
      }
    }
    if(!reconnected)
      throw new ReconnectionException("FTClient reconnection unsucceed after several tries!");
    return handle;
  }
```

5 Experimental Evaluation

In this section we measure the performance of the Fault-Tolerant Multi-Threaded Acceptor-Connector (`ftmtac`) and compare its overhead to the Multi-Threaded Acceptor-Connector (`mtac`) and the plain Acceptor-Connector (`ac`) design patterns. In addition, we also measure the implementation complexity of the patterns, using our own Java implementation of each design pattern.

To carry out the performance experiments, we used two machines (client and server) in the same Local Area Network. The client machine ran the Mac OS X v10.6.7 operating system, with a 2.4GHz Intel Core 2 Duo processor, 4GiB of RAM and 3 MiB of cache. The server ran on a virtualized infrastructure, with Linux kernel version 2.6.34.8, a quad-core 2.8 GHz Intel processor, 12 GiB of RAM and 8 MiB of cache. The client application ran on a single process, using different threads to emulate multiple clients.

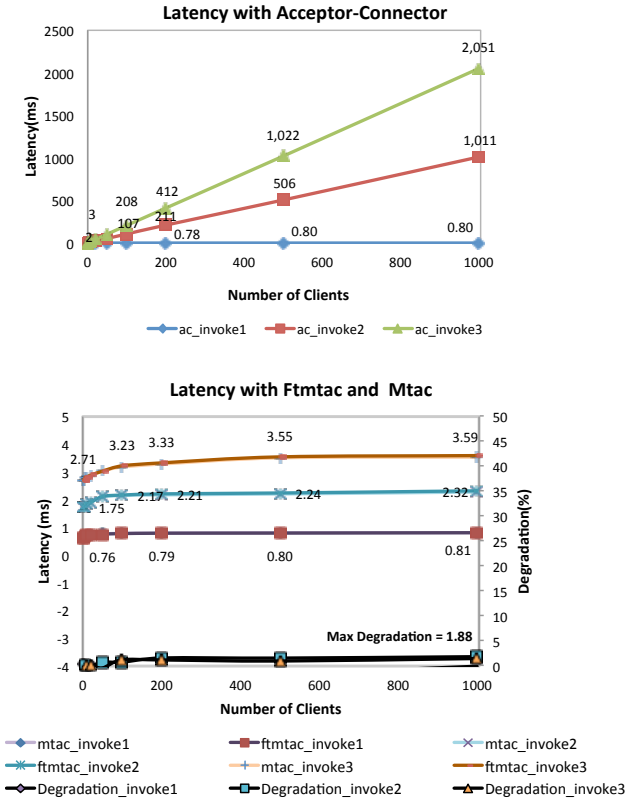

Fig. 9. Latency of the Fault-tolerant Multi-Threaded Acceptor-Connector and its comparison to the `ac` and `mtac` versions

We used the following three server operations in the tests: `Invoke1`, `Invoke2` and `Invoke3`. All of these operations 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 because only the communication is important to us. This way, we precisely control the server execution time.

5.1 Performance

Figure 9 and Figure 10 show the latency and throughput of the three patterns for an increasing number of clients. The plots also show the latency and throughput degradation of the `ftmtac` in comparison to the `mtac`. The latency is defined as the round-trip-time of a request-response interaction, and the latency degradation is computed as $(Latency_{ftmtac} - Latency_{mtac})/Latency_{ftmtac}$. To compute the average latency, $1,000$ requests are sent and the round-trip-time is calculated for each single request. To compute the overall throughput of the server, we set each client to send $100,000$ requests to the server, leaving the reception of responses to another thread. We then divide the total number of requests by the time the server takes to reply to these requests. We compute degradation of throughput as $(Throughput_{mtac} - Throughput_{ftmtac})/Throughput_{mtac}$. For both cases, the results are presented as the average of 30 tests.

In Figure 9 (top), we show the latency for the Acceptor-Connector pattern. One should notice that, in this pattern, the server is single-threaded. Latency increases linearly with the number of clients, for the `Invoke2` and `Invoke3` operations. As an example, for `Invoke3`, the latency increases from 2.8 to $2,051$ ms, by increasing the number of clients from 1 to $1,000$. We get two plots with different slopes for these two invocations due to the differences in their sleeping time. The slope

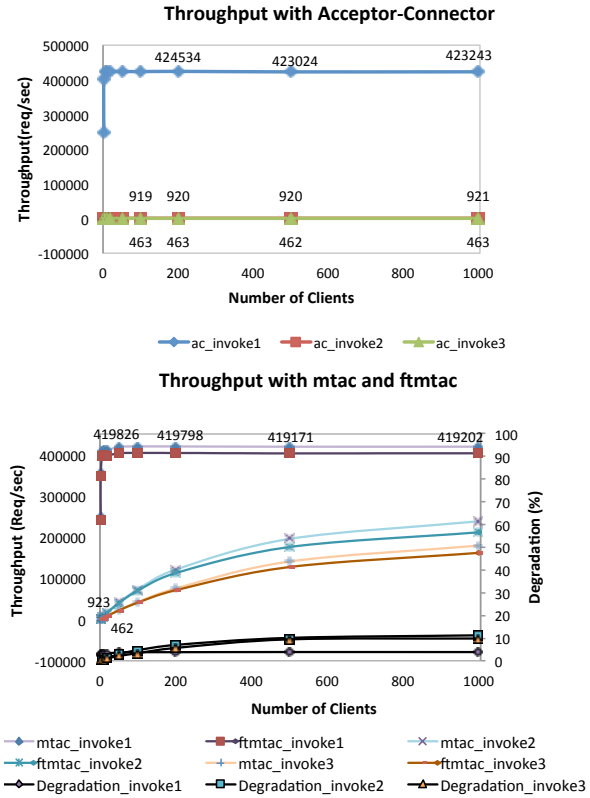

Fig. 10. Throughput of the Fault-tolerant Multi-Threaded Acceptor-Connector and its comparison to the `ac` and `mtac` versions

for the first invocation is almost equal to 0, because the single server thread manages to respond very quickly to a very large number of clients, as it does no work.

With the `mtac` and `ftmtac` (see Figure 9 on the bottom) we get a very similar latency, as shown in the plots. Latency degradation is negligible (below 1.88 percent). Some lines are not visible because they almost completely overlap for the same invocations (e.g., `mtac_invoke3` and `ftmtac_invoke3`). The latency increases slowly and smoothly by increasing the number of the clients in both designs. For `Invoke1` the latencies of these two designs are almost equal to the `ac`'s latency. This happens because the processing time (or sleeping time) of this invocation is 0 and, as such, having several threads to process the request does not increase the performance, but may decrease it due to thread management overheads.

The throughput tests, of Figure 10 confirm the aforementioned observations. With the Acceptor-Connector design pattern the throughput is different for the distinct invocations, but it always stays the same, even when the number of the clients grows. Again, this is due to the fact that the server is single-threaded. For `mtac` and `ftmtac`, the plot lines generally increase with the number of clients. In the `Invoke1` operation, throughput increases rapidly and then does not change (until about $419,200$ requests per second). As shown in Figure 10, the throughput of `mtac` and `ftmtac` for the `Invoke1` is slightly lower than the Acceptor-Connector's throughput, due to more overhead caused by thread synchronization. In contrast, the throughputs of `mtac` and `ftmtac` are much better for the other invocations (i.e., `Invoke2` and `Invoke1`).

The throughput tests also show a small degradation for the fault-tolerant design in comparison to the `mtac`. This degradation increases very slowly with the number of clients, and it results from the overhead of copying extra bytes to the `Stream Buffer`. To better understand this overhead, we measured the cost of copying bytes in the `Stream Buffer`, to observe the overhead for different sizes of requests (ranging from 20 bytes to 64 KiB). Thus, by increasing the size of the strings

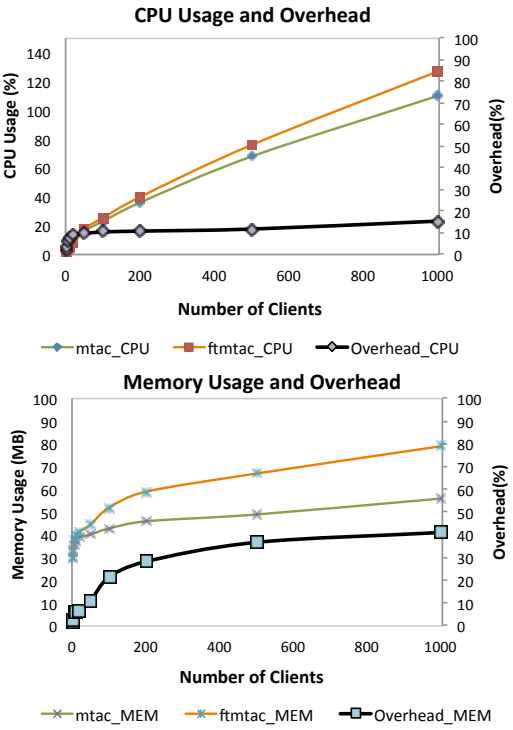

Fig. 11. CPU and Memory Usage of Multi-Threaded Acceptor-Connector and Fault-Tolerant Multi-Threaded Acceptor-Connector, and overhead of fault-Tolerance

from 20 bytes to 1 KiB, 2 KiB, 3 KiB, 4 KiB, 8 KiB, 16 KiB, 32 KiB, and 64 KiB, the cost linearly increases from 0.001 ms to 0.58 ms.

5.2 Overhead and Complexity

In our experiments, the extra `Stream Buffer` required 293,976 bytes, being the TCP send and receive buffer sizes half of this value. One must not forget that the original TCP already consumes this value per connection on each peer. To examine the CPU and memory overhead, we periodically ran the `ps` command to read memory and CPU usage on the server. Each of the 1 to 1,000 clients sends 100 requests for `Invoke3` during 5 minutes. Inevitably, as we show in Figure 11, the memory and CPU usage increase by increasing the number of the clients. The overhead of `ftmtac` increases smoothly with the number of clients. When we look at numerical values, the memory overhead of `ftmac` is larger than the CPU overhead. This suggests that the impact of the double buffering is more important than the impact of copying memory for the CPU.

We also measured three important complexity metrics, Lines of Code (LOC), Cyclomatic Complexity, and Nested Block Depth, to compare the complexity of `ac`, `mtac`, and `ftmtac`. The measurements show that we used 40 extra lines of code in `mtac` in comparison to `ac`, and 297 extra lines of code in `ftmtac`, in comparison to the `mtac`. These extra lines of code, respectively, add an average of 0.018 and 0.025 independent paths per method to the program's source code, which indicates that the cyclomatic complexity imposed by our design and implementation is very low. The depth of nested blocks of code is 1.397 in `ftmtac`, which is very close to `mtac`'s 1.258 and `ac`'s 1.192. These results show that our design is simple and does not compromise scalability.

5.3 Network Failure and Recovery

To evaluate the recovery from failures we uploaded a file to a server through a proxy. To simulate network crashes, we stopped this proxy for one minute and then restarted it. For 10 repetitions

Table 1. Code Complexity

	LOC	Cyclomatic Complexity	Nested Block Depth
ac	220	1.846	1.192
mtac	260	1.871	1.258
ftmtac	557	1.889	1.397

of the test, we first observed that the file was always transferred completely in the presence of a network failure. Reconnection took only 26 ms, while the first connection establishment and service activation took 300 ms, on average. This reconnection time includes a delay in the client to reconnect and also the time of exchanging the bytes that were pending in the buffers. The former is entirely configurable, the latter is inevitable and is not really a loss of time.

6 Conclusion

In this paper, we presented a Fault-Tolerant design pattern for connection-oriented applications based on the Multi-Threaded Acceptor-Connector pattern. This pattern provides the following benefits: 1) it allows the developers to handle connection failures without losing any data; 2) it decouples the failure-related processing from the connection and service processing; 3) it efficiently recovers the state of a failed connection by applying a circular buffer (the `Stream Buffer`); 4) it provides flexible behavior, by allowing developers to configure the reconnection procedure, and 5) it efficiently supports multi-threading, by applying the Leader-Follower pattern. We implemented the pattern in Java and demonstrate that it is efficient. The pattern we proposed in this paper may benefit from future improvements, namely concerning the co-existence between fault-tolerant and non-fault-tolerant clients and the security of the reconnection procedure. We are currently working on a session-based fault-tolerant design-pattern to hide the details of failure handling completely from the service handlers.

References

1. "RFC 793 - Transmission Control Protocol (TCP)." Internet Engineering Task Force (IETF), September 1981.
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. "RFC 4251 - The TLS Protocol - Version 1.0." Internet Engineering Task Force (IETF), January 1999.
5. R. W. Scheifler and J. Gettys, "The x window system," *ACM Trans. Graph.*, vol. 5, pp. 79–109, Apr. 1986.
6. N. Ivaki, S. Boychenko, and A. Serhiy, "A fault-tolerant session layer with reliable one-way messaging and server migration facility," pp. 75–82, The Third IEEE Symposium on Network Cloud Computing and Applications (NCCA14), 2013.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. No. 707, Springer Berlin Heidelberg, Jan. 1993.
8. V. C. Zandy and B. P. Miller, "Reliable network connections," in *Proceedings of the 8th annual international conference on Mobile computing and networking*, MobiCom '02, (New York, NY, USA), pp. 95–106, ACM, 2002.
9. D. C. Schmidt, "Acceptor-connector: an object creational pattern for connecting and initializing communication services," *Pattern Languages of Program Design*, vol. 3, pp. 191–229, 1996.
10. "Multi-threaded fault-tolerant acceptor-connector design pattern implementation." `http://sourceforge.net/projects/ftmtac/`.
11. C. M. R. Stewart, "Sctp: new transport protocol for tcp/ip," *IEEE Internet Computing*, vol. Vol. 5, No. 6, pp. pp. 64–69, 2001.

12. S. Barre, C. Paasch, and O. Bonaventure, "MultiPath TCP: from theory to practice," in *NETWORKING 2011* (J. Domingo-Pascual, P. Manzoni, S. Palazzo, A. Pont, and C. Scoglio, eds.), no. 6640 in Lecture Notes in Computer Science, pp. 444–457, Springer Berlin Heidelberg, Jan. 2011.
13. R. Ekwall, P. Urban, and A. Schiper, "Robust TCP connections for fault tolerant computing," *In proceeding 9th international confeence on parallel and distributed systems (ICPADS)*, vol. 19, pp. 501—508, 2002.
14. L. Alvisi, T. C. Bressoud, and A. El-Khashab, "Wrapping Server-Side TCP to mask connection failures," *In proceeding IEEE INFOCOM*, 2001.
15. A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, pp. 39–59, Feb. 1984.
16. T. B. Downing, *Java RMI: Remote Method Invocation*. Foster City, CA, USA: IDG Books Worldwide, Inc., 1st ed., 1998.
17. S. Vinoski, "CORBA: integrating diverse applications within distributed heterogeneous environments," *IEEE Communications Magazine*, vol. 35, no. 2, pp. 46–55, 1997.
18. B. Krishnamurthy and J. Rexford, *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley Professional, 1 ed., May 2001.
19. "Distributed computing made simple." `http://zeromq.org/`.
20. G. Shenoy and S. K. Satapati, "HYDRANET-FT: network support for dependable services," *In international conference on distributed computing systems*, 2000.
21. M. Marwah and S. Mishra, "TCP server fault tolerance using connection migration to a backup server," *In proceeding international conference on dependable systems and networks (DSN)*, pp. 373—382, 2003.
22. H. Jin, J. Xu, B. Cheng, Z. Shao, and J. Yue, "A fault-tolerant TCP scheme based on multi-images," in *IEEE Pacific Rim Conference on Communications Computers and Signal Processing (PACRIM 2003)*, (Victoria, Canada), pp. 968–971, 2003.
23. "Package java.nio." `http://docs.oracle.com/javase/6/docs/api/java/nio/package-summary.html`.
24. D. C. Schmidt and T. Suda, "An object-oriented framework for dynamically configuring extensible distributed systems," *Distributed Systems Engineering*, vol. 1, p. 280, Sept. 1994.
25. D. Schmidt, C. Ryan, M. Kircher, I. Pyarali, and F. Buschmann, "Leader-followers," in *PLoP conference. http://hillside. net/plop/plop2k/proceedings/ORyan/ORyan. pdf*, 1998.