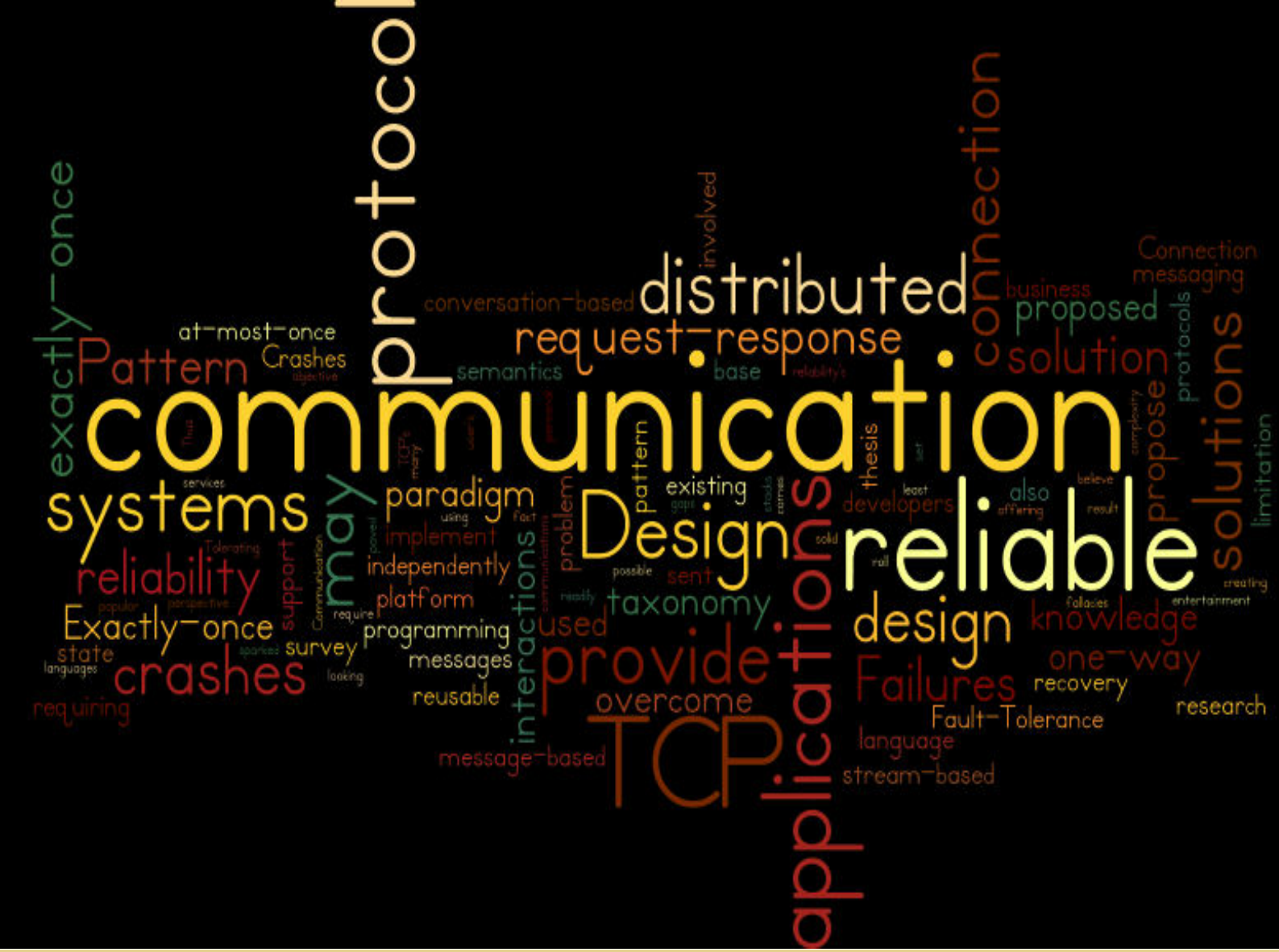

Naghmeh Ramezani Ivaki

Reliable Distributed Communication: Design Solutions and Protocols

Tese de doutoramento
do Programa de Doutoramento em Ciências e Tecnologias da Informação
orientada pelo Professor Doutor Filipe João Boavida de Mendonça Machado de Araújo
e apresentada ao Departamento de Engenharia Informática
da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Abril de 2016

· U C ·

Universidade de Coimbra

Reliable Distributed Communication: Design Solutions and Protocols

Naghmeh Ramezani Ivaki

Dissertation submitted to the University of Coimbra
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

April 2016

Department of Informatics Engineering
Faculty of Science and technology
University of Coimbra

This research has been developed as part of the requirements of the Doctoral Program in Information Science and Technology of the Faculty of Sciences and Technology of the University of Coimbra. This work is within the Dependable Distributed Systems specialization domain and was carried out in the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra (CISUC). Funding for this work was partially provided by the Portuguese Foundation for Science and Technology through the 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.

This work has been supervised by Professor **Filipe João Boavida de Mendonça Machado de Araújo**, Assistant Professor of the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

No problem can be solved from the same level of consciousness that created it.

Albert Einstein

Abstract

From entertainment to personal communication, and from business to safety-critical applications, the world relies on distributed systems more than ever. Despite looking simple on the surface, distributed systems hide many subtleties, specially when they must provide reliable communication. A major source of complexity comes from the fact that any component involved in a distributed communication may fail. Tolerating crashes and recovering to a consistent state is a very difficult task, if possible at all, mainly due to the incomplete and inconsistent knowledge of the peers involved.

The need to overcome this problem and provide reliable communication, proved to be a huge research effort, which outputted a vast body of protocols, communication stacks, middleware, etc. Despite all the best efforts of the last few decades, TCP and HTTP stand firmly as the cornerstones of reliable communication on the Internet, in spite of their shortcomings. For instance, TCP does neither handle connection crashes, nor provide any information to facilitate the recovery. Moreover, neither TCP, nor HTTP provide any support to process non-idempotent requests exactly-once. On the other hand, alternative solutions often try to modify or replace TCP, or require special software or hardware that may not be readily available or mature for deployment in all platforms and languages. This indeed paved the way for continued research in this area.

In this thesis, we argue that the best approach for the recurrent reliability problem of distributed point-to-point applications is precisely to leverage on TCP and HTTP to build reusable design patterns that are completely detached from operating systems, libraries, programming languages or other sorts of middleware, thus having the property of being available for all platforms.

To support this idea, we first survey and classify a wide set of popular distributed applications, requiring reliable communication, and a large number of reliable communication solutions that might be used to implement such applications. This is done in order to build a knowledge base, by identifying matches and gaps that may exist between applications requirements and solutions.

We then propose a reusable solution, named Connection Handler design pattern, to enable the existing connection-oriented protocols, in particular TCP, to recover from

connection crashes. This solution can be used, independently of the platform and programming language, and provides support not only for TCP, but for other technologies, like WebSockets. We then use the Connection Handler design pattern and propose a reusable, extensible, and efficient design solution to stream-based applications, requiring reliable transmission of byte streams (e.g., multimedia streaming) even in the presence of connection crashes. We also propose design solutions to message-based applications, following the one-way messaging paradigm, to tolerate connection crashes and track the status of sent messages. Furthermore, we propose an exactly-once protocol and design solution for conversation-based applications with request-response interaction patterns, tolerating both connection and endpoint crashes. Finally, we create a comprehensive taxonomy of reliable request-response protocols offering exactly-once and at-most-once semantics.

We believe that the positive outcome of our experimental evaluation demonstrates that this thesis advances the state of the art in reliable point-to-point distributed communication, by providing a set of designs and protocols for different forms of interactions from one-way to reliable request-response, including non-idempotent interactions with exactly-once or at-most-once semantics. The design patterns we propose help developers to implement more reliable distributed communication simply, correctly, and independently of the platform, programming language, and application's business logic.

Keywords:
Reliability, Fault-tolerance, Point-to-Point Communication, One-way Interaction, Request-Response Interaction, TCP, Connection Crashes, Reliability Semantics, At-least-once, At-most-once, Exactly-once, Reliability Targets, Stream-Based Communication, Message-Based Communication, Conversation-Based Communication, Design Pattern, Protocol, Taxonomy

Resumo

Do entretenimento à comunicação pessoal, passando por aplicações críticas para negócio e segurança, o mundo depende cada vez mais dos sistemas distribuídos. Apesar de parecerem simples, os sistemas distribuídos escondem muitas subtilezas, especialmente quando a comunicação tem de ser fiável. A origem da complexidade está no facto de que qualquer componente envolvido na comunicação distribuída poder falhar. Tolerar falhas e regressar a um estado coerente é uma tarefa bastante difícil, por vezes impossível, principalmente devido ao conhecimento incompleto e inconsistente dos pares envolvidos na comunicação.

A necessidade de disponibilizar comunicação fiável, mostrou ser uma tarefa de investigação imensa, que resultou num largo conjunto de protocolos, pilhas de comunicação, *middleware*, etc. No entanto, mesmo com todo este esforço, TCP e HTTP permanecem como as pedras angulares da comunicação fiável na Internet, isto apesar das suas evidentes limitações. Por exemplo, o TCP não consegue lidar com falhas nas ligações, nem disponibiliza informação que possibilite a recuperação. Adicionalmente, nem TCP, nem HTTP disponibilizam suporte para processar pedidos não-idempotentes *uma e uma só vez*.

Por outro lado, muitas das soluções alternativas tentam modificar ou substituir o TCP, ou requerem *software* ou *hardware* especial que pode não estar imediatamente disponível ou que nunca atingiu um grau de maturidade que permitisse a utilização em todas as plataformas e linguagens. As limitações das soluções dominantes por um lado, e as evidentes limitações das alternativas, por outro, ditaram que a investigação nesta área se mantivesse extremamente ativa.

Nesta tese, defendemos que a melhor abordagem para o problema recorrente da fiabilidade em aplicações distribuídas ponto-a-ponto é precisamente partir de TCP e HTTP para criar padrões de desenho completamente desligados de sistemas operativos, bibliotecas, linguagens de programação ou outros tipos de *middleware*, podendo, dessa forma, ser implementadas em todas as plataformas.

Para suportar esta ideia, primeiro analisamos e classificamos um grande conjunto de aplicações distribuídas, que necessitam de comunicação fiável, e um grande número

de soluções de comunicação fiável, que podem ser usadas para implementar essas aplicações. Isto é feito com o objetivo de construir uma base de conhecimento, identificando correspondências e lacunas entre requisitos de aplicações e soluções.

Propomos então uma solução reutilizável, denominada de padrão de desenho *Connection Handler*, para permitir que os protocolos existentes orientados a ligações, nomeadamente TCP, possam recuperar de falhas nas ligações. Esta solução pode ser usada, independentemente da plataforma e linguagem de programação, e disponibiliza suporte não apenas para TCP, mas para outras tecnologias como *WebSockets*. Usamos então o padrão de desenho *Connection Handler* e propomos uma solução de desenho reutilizável, extensível, e eficiente para aplicações baseadas em fluxos de dados, que necessitem de transmissão fiável de fluxos de *bytes* (e.g., *streaming* multimédia), mesmo na presença de falhas nas ligações. Também propomos soluções de desenho para aplicações baseadas em mensagens, que seguem um paradigma de comunicação unidirecional, para tolerar falhas de ligação e seguir o estado de mensagens enviadas. Adicionalmente, propomos um protocolo *uma e uma só vez* e solução de desenho para aplicações baseadas em conversação, com padrões de interação pedido-resposta, que tolera falhas na ligação e nos participantes. Finalmente, criamos uma taxonomia exaustiva de protocolos fiáveis pedido-resposta, oferecendo semânticas *uma e uma só vez* e *no máximo uma vez*.

Acreditamos que o resultado positivo da nossa avaliação experimental demonstra que esta tese representa um progresso no estado da arte em comunicação fiável ponto-a-ponto, ao disponibilizar um conjunto de desenhos e protocolos para diferentes formas de interações desde unidirecional a pedido-resposta fiável, incluindo interações não-idempotentes, com semânticas *uma e uma só vez* e *no máximo uma vez*. Os padrões de desenho que propomos ajudam os programadores a implementar comunicação distribuída mais fiável de forma simples, correta e independente da plataforma, linguagem de programação e lógica de negócio da aplicação.

Keywords:
Fiabilidade, Tolerância a falhas, Comunicação ponto-a-ponto, Interação unidirecional, Interação Pedido-Resposta, TCP, Falha de Ligação, Semânticas de Fiabilidade, Pelo menos uma vez, No máximo uma vez, Uma e uma só vez, Alvos de Fiabilidade, Comunicação Baseada em Fluxo, Comunicação Baseada em Mensagens, Comunicação Baseada em Conversação, Padrão de Desenho, Protocolo, Taxonomia

Acknowledgements

I would like to thank all those who gave me comments and support during my work.

Foremost, I would like to express my special appreciations to my advisor, Professor Filipe Araújo, for his continuous support of this research, motivation, immense knowledge, and patience. Without his orientation, this work would never be possible.

Besides my advisor, I would like to thank Professor Nuno Laranjeiro, Professor Raul Barbosa, and Professor Fernando Barros for their partial participation in this research and their insightful comments.

I also thank to all the anonymous reviewers for their comments that helped to improve the quality of the work developed.

Special thanks go to Nuno, for his endless encouragement during this research and my life far from my family.

Last but not the least, I would like to express my gratitude to my parents Sayareh Esmaeili and Ali Mohammad Ramezani Ivaki, for supporting me throughout my life.

List of Publications

This thesis relies on the published scientific research present in the following peer reviewed papers:

- Naghmeh Ivaki, Nuno Laranjeiro, Filipe Araújo, "A Design Pattern for Recovering from TCP Connection Crashes in HTTP Applications", The International Journal of Services Computing (IJSC), 2016.
- Naghmeh Ivaki, Nuno Laranjeiro, Filipe Araújo, "Towards Designing Reliable Messaging Patterns", The 15th IEEE International Symposium on Network Computing and Applications (NCA 2016), Cambridge, MA USA, October 31 - November 2, 2016.
- Naghmeh Ivaki, Nuno Laranjeiro, Filipe Araújo, "A Design Pattern for Reliable HTTP-Based Applications", The 12th IEEE International Conference on Services Computing (SCC 2015), New York City, USA, June 27 - July 2, 2015.
- Naghmeh Ivaki, Nuno Laranjeiro, Filipe Araújo, "A Taxonomy of Reliable Request-Response Protocols", The 30th ACM/SIGAPP Symposium On Applied Computing (SAC 2015), Salamanca, Spain, April 13-17, 2015.
- Naghmeh Ivaki, Filipe Araújo, Fernando Barros, "Session-Based Fault-Tolerant Design Patterns", The 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2014), Hsinchu, Taiwan, December 16-19, 2014.
- Naghmeh Ivaki, Filipe Araújo, "Fault-Tolerant Bi-Directional Communications in Web-Based Applications", The 2014 International Symposium on Ubiquitous and Cloud Computing Frontiers (UCCF 2014), In the 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2014), Hsinchu, Taiwan, December 16-19, 2014.
- Naghmeh Ivaki, Filipe Araújo, Fernando Barros, "Design of Multi-Threaded Fault-Tolerant Connection-Oriented Communication", The 20th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2014), Singapore, November 18-21, 2014.

- Naghmeh Ivaki, Serhiy Boychenko, Filipe Araújo, "A Fault-Tolerant Session Layer with Reliable One-Way Messaging and Server Migration Facility", The Third IEEE Symposium on Network Cloud Computing and Applications (NCCA 2014), Rome, Italy, February 5-7, 2014.

- Naghmeh Ivaki, Filipe Araújo, Raul Barbosa, "A Middleware for Exactly-Once Semantics in Request-Response Interactions", The 18th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2012), Niigata, Japan, November 18-19, 2012.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is not an overstatement to say that modern society stands on distributed systems. From basic grid services, such as electricity, water and telecommunications, to business, health, and leisure, it is difficult to come up with a human activity that does not rely on distributed systems in some form. Their growing importance in people's life and businesses, including e-commerce, financial services, healthcare, government, and entertainment, increases the need for more "dependable" distributed applications.

Dependability can be defined as the ability to deliver a service that can be justifiably trusted (Avizienis et al., 2004). It includes the following attributes: reliability (continuity of correct service), availability (readiness for correct service), safety (absence of catastrophic consequences on the user(s) and the environment), confidentiality (absence of unauthorized disclosure of information), integrity (absence of improper system state alterations), and maintainability (ability to undergo repairs and modifications) (Avizienis et al., 2004). In this thesis, we focus on the reliability of distributed applications, which should continually deliver service, even in the presence of faults and partial failures. Fault-tolerance is, therefore, vital to such applications.

Despite looking straightforward at surface, correctly programming a reliable distributed application is anything but simple. Crashes, in pretty much any component involved in the communication (Birman, 1997), turn distributed programming into a complex and subtle task. According to Leslie Lamport, "A distributed system is one in which the failure of a computer you did not even know existed can render your own computer unusable." (Lamport, 1987). This simple definition contains two important facts about

distributed systems. First, it clearly mentions that a distributed system contains faulty components. Second, it points out the consequences of transparency in distributed systems (Linington et al., 2011). Hiding the components and their communication is a fundamental aspect of distributed systems, but makes fault detection and tolerance more complex.

Within distributed systems, there are several forms of communication, each having its own concerns regarding reliability. In this thesis, we focus on "point-to-point communication", which is perhaps the most used communication model in reliable distributed systems, including mission-critical, safety-critical, business-centric, service-oriented, and transactional applications, where reliability is a primary concern (Garro and Tundis, 2015; Rushby, 1994). The number of protocols, middleware, libraries, and in general terms, communication stacks, offering "reliability" in point-to-point communication is quite large (Bilorusets et al., 2005; Ekwall et al., 2002; Hintjens, 2013; Marwah et al., 2005; Postel, 1981; Reis and Miranda, 2012; Richards et al., 2009; Scharf and Ford, 2013). This is mainly due to the increasing need for highly reliable applications, the diversity in reliability requirements (e.g., exactly-once semantics), the diversity in interaction patterns (e.g., one-way versus request-response), and also because implementing this communication model reliably is a difficult task (Halpern, 1987).

Over the last few decades, among all protocols, middleware, and libraries proposed for building reliable communication, TCP and to some extent HTTP are, surprisingly, the most widely used protocols, despite their lack of reliability features. This has, in fact, triggered our research effort to look for solutions that incorporate this reality, instead of searching for yet another alternative.

1.1 Problem Statement and Motivation

A distributed application may fail due to an unexpected crash of some component. Since distributed applications comprise so many components, ensuring their reliability in a distributed environment is a very difficult task, if possible at all, especially considering the unreliable nature of the Internet (Fekete et al., 1993; Gray, 1979; Halpern, 1987). Furthermore, applications need widely different interaction patterns (e.g., one-way or request-response, synchronous or asynchronous) and require different reliability

semantics (e.g., at-most-once, at-least-once, or exactly-once), depending on their characteristics and goals (Tanenbaum and Steen, 2006).

At the heart of most distributed applications, especially of those requiring reliable communication, we find the Transmission Control Protocol (TCP) (Postel, 1981). The popularity of TCP is unquestionable, as 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 the unreliability of the network, which is true up to a certain point. Despite providing reliable connections in distributed environments, TCP does not handle connection crashes, when the connectivity is lost for some time, even if both endpoints are still running. In fact, TCP does not provide any information to applications regarding data that was already written or read. Thus, to recover from connection crashes, developers must rollback the application peers to some coherent state, many times with error-prone, ad hoc, or custom application-level solutions. It is worth mentioning that this problem is not only associated to the applications that directly use TCP Sockets for exchanging data, but also extends to any middleware or session-based communication stacks that use TCP or any TCP-like transport protocol (i.e., a protocol that establishes a connection and buffers the data) for communication.

TCP exposes more limitations in message-based applications, particularly when application peers do not follow a request-response interaction (e.g., event-driven applications). As a stream-based protocol, TCP has no means to place application data into an envelope, in order to be sent and received as a "Message". Furthermore, with TCP, there are no means for a sender to track the status of sent messages. This leads many developers to either use heavy-weight middleware (e.g., JMS (Richards et al., 2009)), or implement request-response protocols (e.g., HTTP (Fielding et al., 2009)) for applications that only require one-way interactions.

TCP's limitations also extend to request-response interactions with strong reliability requirements, in which the server should execute all requests but must not execute a given request more than once. This type of interaction is important in business and safety-critical sectors, such as banking, e-commerce, or healthcare. Guaranteeing exactly-once semantics in these applications requires a fault-tolerant mechanism, ensuring that each request is processed once (and not more than once) and its response is reliably delivered to the client, even in the presence of endpoint and connection crashes.

The aforementioned problems are widely acknowledged in the literature, where we can find many attempts to tolerate TCP connection crashes. Some of these try to use multiple alternative paths between the client and the server (Liao et al., 2008; Scharf and Ford, 2013; Stewart and Metz, 2001), others chose to replicate components (Marwah et al., 2003; Shenoy et al., 2000), checkpoint the sate of the connections (Jin et al., 2003), or use a middle layer to intercept TCP system calls (Alvisi et al., 2001; Bicakci and Kunz, 2012; Ekwall et al., 2002; So-In et al., 2009; Zandy and Miller, 2002). Despite their merits, we can point out limitations in all these approaches. For example, the solutions that use multiple connections (e.g., cmpSCTP (Liao et al., 2008), or MPTCP (Scharf and Ford, 2013)), are orthogonal to the problem we are considering. They do not overcome connection crashes, despite offering more than one path between peers, if available (i.e., basically to increase the data transfer rate). The solutions that require server replication or checkpointing are quite expensive, respectively in terms of infrastructure cost and performance. The solutions allowing the TCP API to be unchanged (e.g., RSocket (Ekwall et al., 2002)) seem appealing, but they are often not mature, or not available for all the computational platforms.

For one-way messaging, we can find solutions in the literature that offer reliability, but they usually are complex and enable loosely-coupled (offline) asynchronous communication (e.g., JMS (Richards et al., 2009), MSMQ (Horrell, 1999)), typically using a persistent broker between the peers. Regarding request-response interactions requiring strong reliability semantics, there are only a few solutions (e.g., EOS2 (Shegalov and Weikum, 2006)), mainly due to their complexity. They usually require special hardware or software that is not readily available or mature for deployment.

The sheer number of options for reliable communication and their diversity demonstrate the importance and the difficulty of providing reliable distributed communication, considering the unreliable nature of the Internet, the differences in the interaction patterns and reliability semantics. Building reliable applications can, therefore, become an extremely complicated task for developers, which have to make the right design and development choices, to meet the reliability requirements of applications.

Developers must either build their own solution from scratch, or use an existing solution that involves selecting and configuring the right reliability guarantees. To make informed decisions, developers must be supported by an appropriate knowledge base, or otherwise they will design and develop applications that implement the wrong reliable communication mechanisms. In fact, the lack of synthesized information and of an

appropriate knowledge base about the existing solutions and applications requirements often leads developers to create their own ad hoc and custom solutions for reliable communication, which is an error-prone task.

Given the above explanation, the problem we tackle in this thesis can be summarized as follows: reliable communication is a fundamental requirement for many distributed applications; among the many protocols, libraries, and middleware offering reliable communication, almost none of them, except for TCP and HTTP, gained strong acceptance among developers; development of reliable communication in the faulty Internet is difficult and fallible; and finally there is no straightforward solution helping developers to correctly implement reliable distributed communication and meet the application requirements in the presence of faults. We believe that the need for reliable communication is shared among most distributed applications, regardless of the platform and programming language. This is a recurrent problem, which should, therefore, have a general reusable solution.

1.2 Main Objectives and Approach

There are several solutions for building reliable communication over faulty networks, but none of them is widely used, apart from TCP and HTTP, which do not provide enough reliability guarantees for a large number of applications. Thus, as explained in the previous section, there are still several key issues regarding reliable communication that need to be properly addressed. These issues map to the objectives of this thesis, as follows:

1. To build a knowledge base for researchers and developers involving the main concepts in reliable distributed communication, reliability requirements of distributed applications, reliable communication solutions and protocols, and reliability semantics offered by existing solutions, to identify the gaps between applications requirements and solutions features.

2. To propose a straightforward and reusable solution for tolerating connection crashes that may occur in any kind of distributed application, using a TCP-like connection, for communication.

3. To propose a reliable, efficient, and reusable solution to stream-based applications, such as file and multimedia streaming systems, necessary to complete streaming of data, even in the presence of connection crashes.

4. To propose a reusable solution to message-based applications with one-way interactions that allows them to track messages sent without losses in the presence of connection crashes.

5. To propose a reliable protocol and reusable design solution for developing conversation-based applications with request-response interactions, requiring exactly-once execution of requests, even in the presence of connection and endpoint crashes.

In order to achieve the first objective, we collect and try to clarify the main concepts and aspects involved in reliable distributed communication. We survey and synthesize a wide set of popular distributed applications, and a large number of reliable communication solutions that might be used to implement such applications. We then select or define some key reliable communication aspects, to characterize and classify the applications and the solutions. Finally, we perform an analysis to find out the main gaps between application requirements and features provided by the solutions, and accordingly, provide insights into research possibilities.

We then particularly focus on the reliable request-response interaction pattern, due to their importance in business and safety-critical applications, and also due to the complexity involved in providing exactly-once semantics to this type of interactions. We build a knowledge base about the protocols offering exactly-once, their characteristics, and implementation complexity. One possible approach would be to collect the existing protocols from real applications, by analyzing their implementation or by studying their specifications. We believe that this is an exhaustive and impractical task, first because many critical applications do not provide the necessary details about their systems, and second because we would be left with no coverage guarantees regarding all possible protocols. For these reasons, we chose a different approach, which is based on the formal definition of all reliable request-response protocols. This approach contains several important stages. We first need to define a valid set of client and server actions. We then generate all possible protocols by interleaving the client and server actions. In the next step, we need to eliminate the invalid or unreliable protocols and categorize

the remaining valid protocols. At the end, we perform an analysis over the protocols based on the reliability semantics and memory requirements.

To achieve the remaining objectives of this thesis, we argue that using design patterns is the best way to present a correct solution to a commonly occurring problem in distributed communications, such as connection crashes. Since implementing reliable communication in a faulty distributed environment is a very common, but difficult and error-prone task, this thesis proposes several design patterns for that purpose.

The second objective targets the problem of connection crashes. For this, we first build a simplified model of connection-based applications apart from their business logic, interaction pattern (e.g., one-way or request-response), and platform. We then propose a solution, based on buffering and acknowledgment mechanisms, that can be applied to this model, enabling automatic reconnection and retransmission of data when a connection crash occurs.

The third objective focuses on stream-based applications, like file transfers or video broadcast. To handle connection crashes in these applications, we apply the previously proposed solution to connection-based applications, and use an efficient buffering mechanism that eliminates the needs for explicit acknowledgments. We also address several challenges involved in supporting proxies and legacy endpoints, when considering HTTP.

The fourth objective of this thesis has to do with the message-based one-way interaction pattern. We argue that the right solution for achieving a messaging service that allows the sender to track their messages is somewhere in-between the extremes of no-feedback (i.e., never sending any feedback from receiver to sender) and request-response messaging paradigm (i.e., sending one feedback for each message). On one hand, closing the loop and letting the sender know the result of its invocations enables the creation of more reliable applications. On the other hand, we must not do it on a synchronous single-message basis (i.e., one response for each request), because this is too costly. This approach offers end-to-end reliability to one-way operations, by using an asynchronous acknowledgment mechanism. This decouples the sender from the receiver and lets them progress independently. Moreover, to resolve the problem of connection crashes, we resort to our previously proposed solution to connection-based applications.

The fifth objective is about the conversation-based request-response interaction pattern, typically containing three different roles, including client, server, and channel, which must engage in a very rigid manner, to ensure reliability. Some common reliability semantics used in this interaction pattern are at-least-once, at-most-once, and exactly-once, which respectively refer to the server executing the request once or more than once; once but not more than once; and once and only once. Among these semantics, the exactly-once semantics is the most challenging one to be ensured. It needs that all components involved in an interaction work correctly, even in the presence of failures, which is difficult to guarantee and error-prone to implement. This is because the client, server, or communication channel may fail, potentially requiring diverse and complex recovery procedures. Furthermore, there is no global knowledge that could be used to facilitate recovery procedures. For this reason, we propose an exactly-once protocol, then transform it to a session-based protocol, to factor out the main complexities involved in the implementation of conversation-based applications requiring an exactly-once request-response interaction pattern. We further propose a design solution facilitating the implementation of the session-based exactly-once protocol. Technically, in this design, we resort to reliability mechanisms like buffering, logging and retransmission, rather than complex and heavy approaches involving distributed transactions, to ensure the reliability semantics required by these applications. To efficiently recover from connection crashes, we use the solution previously proposed to connection-based applications.

1.3 Results and Contributions

In our survey of Chapter 2, we classify the solutions for reliable distributed systems into four groups, based on their reliability target (e.g., stream, message, object, and conversation), and characterize them by considering the interaction pattern they support, the reliability mechanisms they use, the failures they may tolerate (e.g., connection crashes, endpoint crashes), and finally, the reliability semantics they offer. The classification of the applications is done by considering several key features, including the objective (e.g., user-centric, business-centric), the interaction pattern (e.g., one-way vs. request-response), the timeliness (e.g., soft real-time, hard real-time), the reliability target, the criticalness (business-critical, safety-critical), and the reliability requirements (e.g., at-most-once).

We observe in our survey that there is a large variety of applications requiring reliable communication, but at the same time have quite distinct objectives, different interaction patterns and different reliability semantics. We also observe that there is a clear mismatch between the features offered by communication solutions and the features needed by applications, thus noting that standards and implementations are lagging behind real application requirements. In fact, in most cases, the more elaborate communication solutions offering a larger number of guarantees are purely academic efforts that can, by no means, compete with the popularity, maturity and importance of older, more established, albeit poorer solutions in terms of reliability, like TCP. Thus, the developers must either use some existing solution that does not perfectly match their needs, or implement their custom reliability mechanism on top of these more mature and popular solutions, which is a difficult and fallible task.

Our efforts in making a general solution to connection crashes for connection-based applications, in Chapter 3, resulted in a simple reusable design pattern, named "Connection Handler Design Pattern", that enables recovery from connection crashes and facilitates the implementation to the developer. This design pattern includes several components to implement the actions required to store the data sent, establish a new connection after crashes in the client, replace a failed connection with a new one in the server, and retransmit the data lost after a successful reconnection phase. Since this solution is built independently of the platform, programming language, application's business logic, and data type, it can be used for any reliable distributed application requiring TCP connection crash-tolerance.

We then built a design pattern, called "Reliable Transporter Design Pattern" in Chapter 3, to efficiently develop large-scale reliable stream-based applications that are able to recover from connection crashes without losing any byte in transit, by using the Connection Handler design pattern. The cornerstone of this solution is a stream buffer that eliminates the need for explicit acknowledgments, although it also tackles the challenges regarding proxies and legacy endpoints.

Regarding message-based applications, we proposed three design solutions, named "Messenger", "Trackable Messenger" and "Reliable Messenger" in Chapter 4. The Messenger builds a message-based session layer on top of a stream-based transport layer, and provides a simple interface to the applications, enabling them to easily send and receive messages independently of the application layer protocol. The Trackable Messenger provides a strong support to applications following the one-way messaging pattern.

It allows a sender to simply track its messages, by exchanging multi-level acknowledgments at the send, receive, and processing points. The Reliable Messenger, uses the Connection Handler design pattern, to, besides keeping track of messages, enable recovery from connection crashes.

We also proposed an exactly-once protocol to conversation-based applications with request-response interaction patterns. This protocol, which we present in Chapter 5, ensures that each request is executed only once and that responses arrive to the clients, even in the presence of failures. Based on this, we propose a session-based protocol and design for an exactly-once middleware.

The taxonomy we created in Chapter 6 reveals three families of protocols matching common real-world implementations that try to deliver exactly-once or at-most-once. We accomplished a comprehensive analysis over the taxonomy of the protocols, mainly based on two important aspects: reliability semantics and memory requirements. We propose some solutions to all groups of protocols allowing the server to safely delete the state of the interactions from memory and stable storage. The strict organization of the protocols provides a solid foundation for creating correct services, and we show that it also serves to easily identify fallacies and pitfalls of existing implementations. We believe that the results of our analysis will provide a deeper insight on reliable conversation-based applications.

As an overall contribution, this thesis proposes design solutions and protocols that allow developers to implement more reliable stream-based, message-based and conversation-based applications. In detail, the main research contributions are:

1. A Survey on the main concepts involved in reliable point-to-point communication, characterization and classification of well-known applications requiring reliable communication, characterization and classification of existing reliable solutions, and identification of matches and gaps between applications requirements and features of reliable communication solutions.

2. An efficient and reusable solution, named Connection Handler Design Pattern, for recovering from connection crashes without losing data in transit. This solution can be used for developing reliable distributed applications, independently of the platform and programming language.

3. An efficient and reusable solution, named Reliable Transporter Design Pattern, to stream-based applications that require reliable transmission of byte streams even in the presence of connection crashes. This solution uses a buffering mechanism, thus eliminating the need for explicit acknowledgment and addressing several challenges regarding legacy software and proxies.

4. Reusable solutions, named Messenger, Trackable Messenger and Reliable Messenger Design Pattern, to the message-based applications with one-way interactions. The Reliable Messenger, which is built based on the Messenger, Trackable Messenger, and Connection Handler design patterns, allows the applications to track the state of the messages sent and provides reliable transmission of messages, even in the presence of connection crashes.

5. An application-level exactly-once protocol, and accordingly a session-level exactly-once protocol for reliable request-response applications. A design solution, named Exactly-Once Middleware, is proposed to implement the session-based protocol, facilitating implementation of such applications.

6. A comprehensive taxonomy of reliable request-response protocols, including three different families of protocols that clearly match common real-world implementations that might be used by developers, to select the right solution for their applications. An analysis is done on the protocols, assuming unreliable and non-FIFO (First In, First Out) channels and taking memory requirements into consideration.

This thesis also has several technical contributions, which are:

1. An open-source Java implementation of the stream-based Reliable Transporter design pattern that is available online as FSocket (Ivaki and Araujo, d)

2. An open-source Java implementation of the Reliable Messenger that is available online as FTSL (Ivaki and Araujo, c).

3. An open-source Java implementation of the Exactly-once Middleware that is available online as EoMidd (Ivaki and Araujo, b).

In addition to the above direct contributions, we believe that this thesis helps to open a new window toward development of distributed systems, by projecting the light of

software design patterns over their natural complexity. This is part of the effort to develop more dependable distributed applications, and consequently, a more trustable world.

1.4 Thesis Structure

This first chapter introduced the problem addressed and the main contributions of the thesis. Chapter 2 presents background knowledge and the state of the art in distributed point-to-point interactions, with particular emphasis on reliable communication. A classification and analysis of a wide set of popular distributed applications and reliable communication solutions is presented too. This chapter aims to identify the gaps that exist between the reliability requirements of reliable applications and the reliability semantics offered by reliable solutions.

Chapter 3 presents a solution, as a design pattern, to TCP's limitations regarding recovery from connection crashes. Then, based on this and some other well-known design patterns, a solution is given to the large-scale stream-based applications (e.g., multi-media streaming) that require reliable data transmission for streams of bytes.

Chapter 4 presents a solution for overcoming the limitations of TCP for reliable one-way messaging. This solution aims to provide full-duplex reliable communication that allows tracking the state of messages sent and recovery from connection crashes.

Chapter 5 presents protocols for implementing exactly-once semantics in application layer or in session layer. It then presents a design pattern that helps to build an exactly-once middleware for applications with reliable request-response interactions.

Chapter 6 presents a taxonomy of reliable request-response protocols. The taxonomy is built by defining a list of client and server actions, interleaving these actions, in order to generate all possible protocols, removing invalid protocols by applying some constraints, and organizing the valid protocols into a prefix tree. This chapter also presents and analyses the protocols based on two important aspects: reliability semantics and memory requirements.

Chapter 7 presents the experimental evaluation accomplished to measure the correctness, performance, complexity and resource utilization of the solutions proposed in this thesis, as well as their results and related discussion. It also presents the results of the

analysis done over the taxonomy of reliable protocols, to verify the applicability of the protocols and their implementation cost.

Finally, Chapter 8 concludes the thesis and proposes topics for future research.

Chapter 2

State of the Art on Reliable Distributed Communication

For many distributed applications supporting businesses and services, reliable communication, i.e., communication that can justifiably be trusted (Avizienis et al., 2004), is of vital importance. In general, two different communication models are used to accomplish communication between distributed peers: point-to-point (also known as unicast), and multicast (including broadcast communication) (Tanenbaum and Steen, 2006). In the point-to-point communication model, a message is sent from one peer to another peer, whereas in the multicast communication model, a message is sent from one peer to several other peers. In most applications, including critical ones, where reliable communication is a primary concern (Rushby, 1994), e.g., in healthcare, e-commerce, or banking, the point-to-point model is, by far, the most popular means of interaction. Even when several peers are involved in a distributed communication, e.g., for sharing a file, or exchanging emails, communication is still predominantly point-to-point, usually through some intermediate server, which is responsible for properly handling data for the peers involved.

Depending on the application's specific objectives, very different concerns may apply, if the goal is to achieve reliable communication. Clearly, no application can deliver service that can justifiably be trusted, if it is unreliable, or supported by unreliable communication mechanisms. Disruption in services caused by unreliable components can, not only, result in huge direct losses, in the form of human lives, financial costs, or others, but also bring in severe indirect costs, for example, in terms of reputation (Jones et al.,

2000). However, ensuring the reliability of communication is very difficult, especially considering the unreliable nature of the Internet and applications (Fekete et al., 1993; Gray, 1979; Halpern, 1987). These can exhibit a large spectrum of failures, resulting from pretty much any component. When the network, or one of the peers crashes and restarts, client and server need to engage in a complex process of rolling back to some consistent state (Chandy and Lamport, 1985). This is a complex distributed process, almost always lacking any support from the communication stack. For instance, when using the Transmission Control Protocol (TCP) (Postel, 1981) — usually considered as reliable —, peers have no mechanism to know which data to resend.

Indeed, our overview of reliability in distributed interactions, in Section 2.1, makes it clear that the acknowledgments of a transport layer protocol, such as TCP, cannot solve all reliability problems, because applications display a large range of different interaction patterns. For instance, messages may or may not need a response; senders may need to wait for an acknowledgment of the application itself, or they may accept such acknowledgment at a later time; peers may need to be running at the same time, or they might be decoupled by persistent storage. Moreover, applications have different reliability semantics, depending on their characteristics and goals (Tanenbaum and Steen, 2006). For example, file sharing needs ordered and guaranteed delivery of messages — no gaps or byte swaps would be acceptable in a file; bank transfer orders need these properties and more, because payments should be retried in case they fail, but must not occur more than once. Furthermore, TCP only takes care of byte streams. However, byte streams are only one of the targets to care for: different applications, such as publish-subscribe, may also require reliability for a message or an object, while banking applications may require an entire conversation to be reliable.

Each one of these targets, alongside with the reliability semantics, or interaction pattern, requires its own specific solution, such as logging, retransmission, message filtering, etc. In theory, no developer needs to implement such mechanisms from scratch: he or she should rely on available middleware to provide (at least some of) the desired goals. As we see in Section 2.2, where we review a large number of protocols, libraries and Application Programming Interfaces (APIs) for stream, message, object and conversation-based applications, this middleware exists in vast amounts. In practice, some of these solutions are similar to each other, but target different operating systems and languages; some never gained traction; others are purely academic efforts.

Therefore, in Section 2.3, we provide evidence supporting the point of view that much of that undertaking on middleware was, to some extent, in vain. We categorize distributed applications that require reliable communication and identify their reliability requirements. From this effort, it becomes very clear that only a few solutions have actually thrived. We can narrow down the successful options to TCP, HyperText Transfer Protocol (Fielding et al., 2009), and a few more, including message-oriented middleware. The limited number of choices involves a clear penalty for developers. Depending on the application, they must manage most communication issues: keeping track of all the peers involved in the interaction; setting TCP connections on and off; detecting faulty TCP connections and handling subsequent reconnections; or detecting and avoiding duplicate HTTP requests. This is far from ideal, because it is complex, error-prone, and requires a very high level of expertise.

In summary, in this chapter we survey and synthesize the state of the art in distributed systems, with particular emphasis on reliable communication. We aim to: 1) outline the body of knowledge on reliable communication, by collecting the main related concepts (Section 2.1); 2) review the most important reliable communication solutions and identify their characteristics according to key reliable communication aspects, such as reliability semantics (e.g., at-most-once, exactly-once), or interaction patterns (e.g., request-response, one-way)(Section 2.2); 3) categorize well-known applications requiring reliable distributed communication, according to key reliable communication aspects, and identify their requirements in terms of reliability (Section 2.3); and 4) discover the gaps between the applications requirements and the existing solutions, and accordingly, provide insights into future research possibilities (Section 2.4).

The analysis carried out in this chapter ended up being especially complex, not only given the huge amount of combinations of concepts, configurations, and solutions, but also considering the multiple definitions, sometimes overlapping or contradictory, present in the literature (Avizienis et al., 2004; Birman, 1997; Coulouris et al., 2005; Elnozahy et al., 2002; Popescu et al., 2007; Tay and Ananda, 1990). As a result of this effort, we observed that, in many cases, elaborate communication solutions offering a larger number of guarantees are purely academic efforts that can, by no means, compete with the popularity, maturity and importance of older, more established, albeit poorer solutions. This suggests that research and development work in libraries, APIs, and protocols is still necessary, to build the reliable distributed systems of the future.

2.1 Reliability in Distributed Interactions

In this section, we review the main concepts concerning reliable distributed communication. In addition to the definition of "reliable communication", we present the different interaction patterns of distributed applications and then discuss the following key aspects involving reliable communication: the types of failures that can threaten reliable communication; the reliability semantics, i.e., the conceptual levels of reliability (e.g., best-effort) that are expected to be offered by some solution (e.g., middleware, protocol); the reliability targets of solutions (e.g., a message or a byte stream); and finally the specific reliability mechanisms that can be used to build reliable communication (e.g., acknowledgments, buffering).

Reliability is a very broad concept as each application in a distributed environment can have its own specific requirements (Birman, 1997). Sometimes reliability refers to fault-tolerance (recoverability and continuity of correct service) and availability (readiness for correct service) (Lee and Anderson, 2012) and some other times it may refer to security (protection of data, services or resources against misuse by unauthorized users) (Azaiez and Bier, 2007), privacy (protection of identity and locations of users from unauthorized disclosure), correct specification (guarantee that a system solves its intended problems), correct implementation (guarantee that a system correctly implements its specification), predictable performance (guarantee that a system achieves desired levels of performance) (Roman et al., 2013), or timeliness (guarantee that actions are taken within the specified time bounds) (Dantas et al., 2009). In this chapter, reliability is discussed in the context of the communication used by distributed applications, and it mostly refers to fault-tolerance and continuity of correct service (Avizienis et al., 2004). Thus, we define it as follows:

Reliability in distributed communication is the ability of a distributed application to correctly accomplish an interaction, initiated by one of the peers, even in presence of faults, which may lead to a failure in the correct delivery or processing of the data exchanged. A reliable distributed application should either "predict and prevent" or "detect and handle" (recover from) any failure, without performing any incorrect behavior. In other words, a reliable distributed application should be able to achieve its goals (e.g., delivering correct service) even during the periods when some of the components involved in the communication have failed.

Given the above definition, implementing and ensuring reliable communication in distributed systems seems a very complex task. The major difficulties arise from the innate uncertainties in these systems with faulty distributed components, and unexpected behavior in presence of failures. The next sections introduce the basic interaction patterns in distributed communication and discuss precisely the key issues involved in ensuring reliable communication in this context.

2.1.1 Distributed Interaction Patterns

In the point-to-point communication model, a message is sent from one point (a sender peer) to another point (a receiver peer). The focus of this chapter is set on this model, also known as unicast, which is the predominant form of communication in distributed applications and systems and provides a flexible framework for implementing business requirements. Most web and service-based applications that follow a client-server paradigm are examples of this communication model.

Reliable Communication in distributed systems requires the involvement of an application, a communication channel, and in many cases a middleware. The application implements business logic and usually plays two important roles as client and server or sender and receiver. The application peers communicate with each other using the communication channel, sometimes using the middleware, when it is present in the system. The middleware encapsulates a set of services underneath the application layer and on top of the network, and it usually facilitates reliable communication and coordination of the distributed application peers (e.g., RSocket (Ekwall et al., 2002)). Middleware typically provides application developers with high-level programming abstractions (e.g., the use of remote objects instead of sockets (Downing, 1998)) and it may also provide an intermediate broker to decouple the connectivity of sender and receiver (e.g., ZeroMQ (Hintjens, 2013)), among other possibilities.

Communication based on the point-to-point model can also vary according to the type of interactions being used, which fit into distinct patterns, depending on the applications' requirements. We describe these *interaction patterns* from a reliability point-of-view in the next sections and according to three perspectives: messaging pattern between the peers (one-way or request-response), synchronization between the peers (synchronous or asynchronous), and persistency of the exchanged data between peers in a communication (transient or persistent). When referring to these perspectives we

aim to explain the kind of interaction perceived by the user or application layer, regardless of the concrete technology that supports the interaction. As an example, most event-driven applications, which have one-way interactions at the application layer, use TCP for transportation that certainly uses some form of request-response (i.e., bytes stream and its acknowledgment).

One-Way versus Request-Response Interactions

Sending data to a peer without expecting any reply, known as a *one-way* interaction, is the simplest possible interaction pattern. A one-way interaction pattern is extremely useful in event-driven systems and multimedia streaming, where data flows in a single direction. Publish-subscribe (Eugster et al., 2003) and Complex Event Processing (CEP) (Buchmann and Koldehofe, 2009) systems are some of the scenarios where this interaction pattern fits rather well. One-way messaging is very simple and fast to use, but holds one large limitation concerning reliability. When using this pattern, there is no way for the sender to ensure that the data reached its destination (i.e., the receiver) or that it is processed correctly.

The *request-response* interaction pattern seems to overcome the above-mentioned limitation of one-way messaging by forcing the receiver to send a reply. Many interactions in distributed systems are based on this pattern, where a client sends a request to a server that, in turn, sends back a response. In contrast to one-way messaging, this pattern allows the sender to know about the delivery and processing of requests at the receiver, but only when a response is received. If the sender does not obtain a response, it may re-send the request, risking duplicate execution of a non-idempotent operation. "Idempotence" means that performing an operation multiple times will have the same effect as performing it exactly once (Helland, 2012). On the other hand, if multiple executions can produce a different result, the operation is "non-idempotent". A bank transfer order, which must not be executed more than once, is a typical example of a non-idempotent operation.

Asynchronous versus Synchronous Interactions

In *asynchronous* communication, a sender continues to execute immediately after sending data and does not wait to ensure the delivery of the data to the destination. In

the case of the one-way interaction pattern, communication is naturally asynchronous because once data is sent, the sender does not expect any reply. It is also possible to have asynchronous request-response interactions, where the response can be delivered to the sender application later (probably using a callback method). Obviously, in many cases it is unsuitable to use asynchronous interactions with request-response patterns because the sender may not know about the delivery and processing of its requests. From a reliability perspective, this is due to the difficulty or impossibility of distinguishing a crash or a loss from slow transmission or slow processing of messages (Fischer et al., 1985).

In a scenario involving *synchronous* communication, the sender sends data and waits, for a given period of time, until it is sent, delivered, processed, or the corresponding response arrives. This time-based coordination between the sending process and the remaining components involved in an interaction is called synchronization. There are essentially three points where synchronization can take place (Tanenbaum and Steen, 2006): 1) the sender might be blocking in the send operation until the middleware returns the control to the application, which means that the middleware will take care of transmitting the data; 2) the sender may wait until its data has been received by the receiver (usually by a confirmation message); 3) the sender may wait until its data (or request) has been fully processed, which in practice corresponds to the receiver returning a response back to the sender. These three points of synchronization can be considered as key reliability points, since they allow the sender to know about the state of interaction, which in turn allows to carry out further actions in case of crashes.

Transient versus Persistent Interactions

In *persistent* interactions, a message that has been sent is also stored (usually in an intermediate broker) in a persistent storage until it is delivered to the receiver. As such, it is not necessary for the sender to continue running after sending the message. In the same manner, the receiver does not need to be running when the message is sent by the sender. Thus, sender and receiver are decoupled in time. Examples of this type of interactions include E-Mail or messaging systems, such as Java Message Service (JMS) (Richards et al., 2009). Although a broker facilitates reliable communication between decoupled peers, it can also become a single point of failure.

When communication is *transient*, a message is kept by the communication system only as long as the sender and receiver peers are simultaneously running and connected. More precisely, the message is simply discarded by the communication channel (or middleware) if any of the endpoints crashes or interrupts communication (Tanenbaum and Steen, 2006). It is worth mentioning that, in the case of the above mentioned persistent messaging systems, communication can also be set to transient, which means that it is supported by messages that are stored in volatile storage. In this case, if the broker crashes the volatile messages are lost.

2.1.2 Failure Types

The very basic elements involved in a distributed interaction include the sender (or client), the receiver (or server), and the communication channel (in some cases, middleware is also involved as a way of moving some of the concerns outside of the application's code). Reliable end-to-end interaction usually requires several key actions involving each of these parts. These key actions include: 1) the sender needs to initiate the communication process by sending a message (in the presence of network or endpoint crashes, this might involve multiple attempts); 2) the communication channel must eventually deliver these messages (uncorrupted); and 3) the receiver needs to process the message (this might involve filtering duplicate requests for non-idempotent operations (Spector, 1982)).

The faulty nature of distributed systems may result in different types of failures that prevent the overall system from communicating reliably. In a reliable communication, the main objective is to deliver service correctly, thus in this context, the term **failure** can be defined as a transition from delivering correct service to delivering incorrect service. This may occur due to some deviation of one or more components involved in a communication, from correct function. This deviation from correct function is called **crash** (or error), which many times occurs due to **faults** in software or hardware (Avizienis et al., 2004).

Here, we describe the main failure types that may occur in a distributed interaction, allowing to better understand causes and their effects, but also allowing to understand the necessary mechanisms to tolerate such types of failures. Several classifications of failures can be found in the literature (Avizienis et al., 2004; Gartner, 1999; Tanenbaum and Steen, 2006). The following paragraphs describe three failure types (also known to

some authors as failure models), according to Coulouris *et al.* (Coulouris et al., 2005). We consider this classification to be particularly clear, as it emphasizes the separation between causes (e.g., network crashes) and effects (e.g., an omission failure):

- **Omission Failures** occur when a process fails to send or receive data that is expected to be sent or received. Omission failures can refer to send-omission, receive-omission, and channel-omission failures. Send-omission failures are usually caused by lack of buffering space in the network interface or operating system, which can cause data to be lost after the sender sends the data, but before that data leaves the sender's machine. Receive-omission failures are similar to send-omission failures, but they occur when data is lost at the receiver side, often due to lack of buffering space in the network interface or operating system. The omission failures, in general, can also be caused by endpoint crashes (or process crashes) (Coulouris et al., 2005), resulting from complete or partial crash of the communicating peers.

  Channel-omission failures occur when data is lost in the communication channel, while in transit between the end peers. These failures are usually caused by the lack of buffering space at intermediate gateways or proxies (Cristian, 1991). Channel-omission failures can also be caused by network crashes, which refer to the complete or partial crash or malfunction of network components. Such failures may occur due to hardware or software faults, bad configuration, external or internal attacks, or simply lack of power. Besides having the potential to cause data losses, network crashes may also result in network partitioning, when the network breaks into disconnected sub-networks, thus preventing communication between some of the peers (Turner et al., 2011).

- **Timing Failures** occur when a temporal property of a system is violated, for example, when service is delivered too late or too early. These failures generally apply to real-time distributed systems, where the correctness of the service being delivered depends not only on the correctness of the results but also on the time they are actually delivered (Mok, 1983). In this context, we can have early timing failures, in which the response arrives (or the service is completed) before the expected deadline; or late timing failures, in which the response arrives after the expected period (Avizienis et al., 2004). Although early timing failures can be masked or prevented by artificially delaying the response, there is no simple

solution to prevent or recover from late timing failures that might be caused by endpoint and network crashes.

- **Byzantine Failures** (also known as arbitrary failures) capture a wide range of failures, in which components or processes of a system may crash in an arbitrary ways, not just by stopping, but also by processing the requests incorrectly, corrupting their local state, and producing incorrect outputs. When a Byzantine failure occurs during the execution of a service, the system may respond in any unpredictable way. For this reason, and considering the given definition of reliable communication, building distributed applications that are able to tolerate Byzantine failures is a non-trivial task (Driscoll et al., 2003).

Usually, researchers and developers assume the following two crash modes, the manner in which crashes occur. Processes, components, and network may either crash and stop (**crash-stop**, also known as fail-stop) or crash and recover (**crash-recovery**), but never performing any incorrect actions, thus causing only omission or timing failures (and not Byzantine failures). In order to allow applications to survive faulty conditions and still ensure their specific requirements, each of the above types of failures requires the use of the right reliability mechanism or set of mechanisms, which the following sections further contextualize and present.

2.1.3 Reliability Semantics

Reliability semantics refers, in general, to the conceptual level of reliability that is offered or expected to be provided by a network, protocol, mechanism, or application. In a distributed interaction, the conceptual level of reliability usually refers to delivery of data to a destination or to processing of data, and it may range from best-effort (lowest reliability guarantee, also considered as unreliable in some scenarios) to exactly-once (highest reliability guarantee).

Best-effort, which is usually (Chakradhar and Raghunathan, 2010) related to delivery of data over the network and not to data processing, describes a semantics in which a solution (e.g., network, protocol, system) in general does not provide any guarantees that data is delivered but tries anyway to achieve it (Feng et al., 1998). As an example, UDP offers a best-effort service to applications, which is a fast and simple way of

data transmission (for instance, when compared to TCP). Usually this type of data transmission is considered as unreliable transportation (Protocol, 1980).

Correct data delivery describes a semantics, in which a given reliable communication solution (e.g., network, protocol, system) ensures that data is delivered without any error concerning content. Different techniques can be used to detect, or to detect and correct the errors in data (e.g., checksums (Braden et al., 1989), error correcting codes (Sloane and MacWilliams, 1981)). **On-time delivery** semantics refers to the delivery of data within a given time frame, defined by the application (Kopetz et al., 1989). **Ordered data delivery** semantics, ensures that data is received in the same order in which they were sent. This guarantee, which is also known as First-In First-Out (FIFO), is important and necessary for many distributed applications (Lamport, 1978).

The most popular reliability semantics that refer to the delivery and processing of data are at-most-once, at-least-once, and exactly-once. With **at-most-once** semantics, data must not be delivered and processed more than once. This can be simple to achieve: the sender sends some data and it may or may not reach the destination; or, considering a request-response paradigm, a client sends a request and its response may or may not arrive. This semantics becomes more difficult to achieve when the client resends the requests for which a reply was not received. In this case, the server needs to be able to detect duplicate requests and prevent their re-execution. In contrast, in the **at-least-once** semantics, data must be delivered and processed one or more times, even in presence of failures. For instance, to achieve this guarantee in a request-response application, a client must re-send the same request until it gets some response, although it may cause multiple executions of the request.

Unfortunately, neither the at-most-once, nor the at-least-once semantics can be used in several real-world operations, such as performing bank transfers, or buying a flight ticket, as these represent non-idempotent operations. These are the typical cases that require a "reliable" interaction, to ensure that each request is executed once and only once. The **exactly-once** semantics is entirely appropriate for such cases.

Despite the importance of exactly-once semantics, the presence of faults makes it very difficult, or even impossible to ensure (Fekete et al., 1993; Halpern, 1987). As a simple example, to achieve exactly-once in a client-server application with a request-response interaction pattern, both client and server should store their state into stable storage in order to be able to recover from crashes, but this is not enough. While it is reasonable

to assume that servers eventually recover, clients may not recover (e.g., a browser with a user that gives up using a service). For this reason, what is actually provided by many protocols and systems, claiming to provide exactly-once, is at-most-once semantics (Ivaki et al., 2015).

2.1.4 Reliability Targets

We used the term "data" in previous sections in a generic way to refer to the information that needs to be reliably exchanged between peers in each interaction. In practice, "data" can refer to a stream of bytes, a message, an object, or to a set of messages within a conversation. Thus, a solution for reliable communication may focus on these four different aspects. In the next paragraphs, we discuss not only these different reliability targets, but also the corresponding different middleware (which has focus on each of the distinct targets).

In a **stream**, there is no concept of discrete messages, there is a flow of bytes instead. File and multimedia systems are examples of applications that transmit data without an envelope (i.e., without message boundaries). In such applications, if reliable communication is needed, the goal is to ensure that this flow of bytes is delivered and processed in a reliable way. As an example, TCP is a reliable transmission protocol, whose reliability target is the stream of bytes and not individual messages.

A **message** refers to a discrete message that can however be sent in several chunks. A message is placed into an envelope, including a header and a body, when sent (i.e., which determines the message boundaries), and it should be exactly the same when it is read (Hohpe and Woolf, 2003). Most distributed applications exchange messages for communication, in various types of envelopes (e.g., HTTP message, SMTP message). Thus, in this context, the reliability objective is to deliver and process each message reliably.

A **Conversation** refers to a series of messages that must be exchanged between two peers to complete a set of related interactions. For instance, a bank transfer usually involves exchanging several messages, first the operation details, then confirmation, and, at the end, a security code. Thus, the reliability objective in this context is to ensure the delivery of all of these messages, so that if one is lost or corrupted the conversation must be repeated from the beginning.

The term **Object** refers to an application-level object, consisting of state and behavior, as in the object-oriented programming definition (Rentsch, 1982). Applications might either transmit complete objects or remotely invoke their methods. In the former case, applications usually need to serialize the object before sending and deserialize them before reading, and in the latter case, a client is able to request the execution of an operation on the server by invoking a method over a reference to the remote object. In either case, information will be transmitted through the network using the specific mechanisms required by the client and server-side platforms involved (e.g., Java Serialization, Java Remote Method Invocation).

In object-based communication, the reliability target is the object. From the reliability perspective and depending on the interaction context, this object target can be considered as a subclass of message or conversation. For instance, when an object is used as data to be transmitted in a communication, the reliability concern is similar to message-based communication because the target is the reliable delivery and processing of the object as a whole, just like a message. When the remote invocation of the object's methods (or functions) is used for communication, the reliability concern is similar to conversation-based communication, because the whole interaction from sending the request to invoke the remote method until receiving the result of the invocation must occur reliably, otherwise the client needs to invoke the remote method again.

There are currently different types of *middleware* that target one of the four different targets mentioned. As previously referred, middleware is a software layer that stands between the application and the lower layers (network and operating system), thus aiming to separate the reliability concerns from the application logic. It provides developers with a higher level of abstraction based on primitives that are provided by the lower layers. Many types of middleware are currently being used for several distinct purposes. However, in this chapter, we classify them in the following four groups, based on their distinct reliability targets:

Stream-Oriented Middleware provides a continuous data streaming abstraction usually used by multimedia applications. In many multimedia applications (e.g., video conferencing or Internet telephoning), reliability may not be the most important aspect. But there are applications that allow access to multimedia products, such as music and movies, for download (or upload), in which reliability is important. We usually do not find the term "stream-based middleware" as a type of middleware when it comes to reliable stream-based communication (i.e., which is offered by TCP), because middleware

usually is supposed to offer a higher level of abstraction than what is provided by the transport layer. There are, however, stream-based middleware solutions (although not very popular) that provide a higher level of reliability on top of TCP (note that TCP is not completely reliable and can not handle connection crashes). RSocket (Ekwall et al., 2002) is an example of this type of middleware.

Message-Oriented Middleware is a very popular type of middleware that facilitates message exchange between distributed peers. The most well-known message-oriented middleware solutions use a broker between sender and receiver to decouple the communication parties. Message queues at the broker are used to store messages, thus providing asynchronous communication to the peers. Microsoft MSMQ (Horrell, 1999) and ZeroMQ (Hintjens, 2013) are two examples that fit this category. Other solutions support synchronous message-oriented communications (without the presence of a broker). The WS-ReliableMessaging standard (Bilorusets et al., 2005) implemented by the Windows Communication Foundation (WCF) (Smith, 2007) is an example of this type of middleware.

Conversation-Oriented Middleware provides support to reliably complete a given distributed conversation, which includes a set of message exchanges, such as online bank transfer or online ticket reservation. This type of middleware may use different mechanisms to ensure that a conversation is completed reliably. For instance, if the conversation actually refers to a distributed transaction, the middleware may support the two-phase commit (2PC), which is a popular protocol to ensure that a distributed transaction either concludes successfully or nothing occurs. Message Logging is another technique used to support the reliable completion of a conversation. This approach is based on logging the state of the interactions and enabling the retransmission of the messages until it is completed. Phoenix/APP (Barga et al., 2003), iSAGA (Dutta et al., 2001), and EOS (Shegalov and Weikum, 2006) are examples of this type of middleware.

Object-Oriented Middleware is essentially built based on both the object-oriented programming paradigm and the Remote Procedural Call (RPC) architecture (Birrell and Nelson, 1984). We consider all middleware that is based on the RPC architecture to be object-oriented middleware (also known as Procedural Middleware) (Sadjadi and McKinley, 2003). Such middleware provides the abstraction of a remote object, whose methods (or functions) can be transparently invoked as if the object was in the same address space as its client. Java RMI (Downing, 1998), CORBA (Vaysburd and Yajnik, 1999), and DCOM (Brown and Kindel, 1998) are well-known examples that fit this type.

2.1.5 Reliability Mechanisms

Conceptually, there are several well-known mechanisms that can be implemented and used to build reliable communication in distributed applications. In the next paragraphs, we present and discuss these mechanisms, in general, from the more simple ones to those that involve more complexity. Note that this type of mechanisms, or combinations of these mechanisms are many times part of specific reliability solutions, which we will in turn describe in Section 2.2.

Buffering-Acknowledgment-Retransmission

Buffering data, acknowledgment of reception, and retransmission of missing or damaged data, are well-known mechanisms to tolerate omission failures and implement reliable communication. Although these are individual mechanisms, they are frequently used together to achieve reliable communication. The term buffering refers to storing of data in the volatile memory. Acknowledgment is used to obtain feedback from the receiver about the delivery or processing of the data sent. The acknowledgments can be used by the sender to remove the data stored in the buffer. If some data is not acknowledged after a given period of time, the sender retransmits (i.e., it sends the data again) until the delivery of the data is confirmed.

Transmission Control Protocol (TCP) (Postel, 1981) is a concrete example that relies on these techniques. It buffers data and retransmits if it is not acknowledged. Regarding the acknowledgment, several methods have been proposed in the literature, including positive acknowledgment (ACK), negative acknowledgment (NAK), cumulative acknowledgment (CACK), and selective acknowledgment(SACK) (Waldby et al., 1998). Regardless of the the above types, acknowledgments can be performed implicitly where the sender uses a different method, instead of sending explicit messages, to identify the reception of data. For instance, the reception of a response, in an application with request-response interaction, is an implicit acknowledgment for reception of a request.

Logging

Logging is another technique used to ensure reliability. This approach is based on storing the data exchanged or the state of interactions into stable storage, thus enabling the recovery of the state and retransmission of the data, if necessary. The difference between this mechanism and the previous one (buffering-acknowledgment-retransmission) is that with logging, a system is able to ensure reliable communication, even in presence of endpoint crashes, albeit at a higher cost, due to the operations with stable storage (Chakravorty and Kale, 2007; Wang et al., 2009).

Filtering

Filtering is a mechanism used to detect repeated and orphan invocations and prevent duplicate delivery or execution of data (Pleisch et al., 2003). It is mainly associated with retransmission (and mechanisms that use retransmission). In the simplest form, filtering is accomplished by using a unique identifier, which is associated to each request, invocation, or transaction. Filtering is usually used by applications that require at-most-once or exactly-once semantics (Koloniari et al., 2011).

Checkpointing

Checkpointing is a technique for building reliable applications and for supporting reliable communication. Using this technique, a snapshot of a system, component, or process state is periodically stored in stable storage, so that it can be used later on, for restarting the execution upon a crash. To recover from crashes, the system, component, or process restarts its execution from one of the previous correct states stored in stable storage (i.e., it restarts from one of the checkpoints) (Johnson, 1989). Although this mechanism guarantees correct recovery from crashes, in comparison to logging, it suffers from high overhead associated with the checkpointing process (Chakravorty and Kale, 2007).

Broker

The use of a Broker appears usually associated with asynchronous distributed communications. In short, a broker is a node (located between the sender and receiver) that

intermediates the delivery of data to a destination, that may be off-line when the data is being sent to the broker (i.e., the broker decouples the peers in time). In practice, it allows to tolerate temporary peer crashes in the sense that they will be able to get their messages later (after recovery). However, the broker may also become a single point of failure. Microsoft's MSMQ (Horrell, 1999), Websphere MQ (Hart, 2003) and HornetQ (Lui et al., 2011), which implement the Java Message Service (JMS) (Richards et al., 2009) specification are examples of specific implementations of a broker.

Transactions

A transaction usually represents a unit of work (e.g., database operation) that must be done as a whole and respecting ACID properties. In short, it either finishes correctly and successfully or produces no effect at all (Haerder and Reuter, 1983). A distributed ACID transaction (Thomson et al., 2012) ensures that multiple participants in a transaction actually agree on the outcome of an interaction. It usually offers at-most-once semantics, because the operation either successfully finishes for all participants involved, or nothing occurs and everything reverts back to the initial condition. Protocols like the Two-Phase Commit (Boutros and Desai, 1996) (2PC) can reliably implement distributed ACID transactions. However, distributed transactions have several drawbacks: 1) they are difficult to use, as they involve a fairly complex configuration and Application Programming Interface (API); 2) they are heavy, because they involve a coordinator process; and 3) they are slow, due to the several steps involved in protocols like the two-phase commit.

Queued Transaction Processing is also a technique to deal with transactional operations (Gray and Reuter, 1993). Using this mechanism, a client starts a transaction and enqueues the request at the server's queue. Then, the server starts another transaction, dequeues and processes the request, and enqueues the reply at the client's queue. A third transaction is started and the reply is dequeued and processed by the client. Briefly, this technique involves two queues in front of the server and client, and three distributed commits. Transactions, in general, can resist to a large spectrum of crashes.

Replication

Replication is a quite powerful mechanism to improve availability and reliability of distributed applications. Replication could be applied for processes, components, connections, or to whole systems. Generally there are two strategies for replication: primary-backup and active replication. In the former strategy, one of the replicas (i.e., the primary), plays the main role in the system. Considering a client-server scenario, the primary replica receives the requests from the client, processes them, and sends the response back to the client. The state of the system should be shared between all replicas so that the primary replica can be replaced with a backup replica, in the event of a failure. In contrast, with the active replication strategy, all replicas play the same role in the system. All the requests go to all active replicas and all process the requests and prepare a response (just one of the responses is sent back to client, possibly the fastest one) (Guerraoui and Schiper, 1996).

In alternative to the above techniques, which consider that replicas are identical to each other, lazy replication (also known as optimistic replication) allows replicas to diverge. Replicas only converge when the system goes down (Ladin et al., 1992). Replication, in general, can improve reliability, as its implementation allows tolerating endpoint and network crashes.

Migration

Migration refers to the process of moving running application processes from one machine (or a more complex environment, such as a cluster or a cloud) to another and is usually used for proactive fault-tolerance. Proactive fault-tolerance is an important recent concept in high-performance computing, which aims to prevent the impact of computing node (process) crashes on running applications, processes, or connections (Egwutuoha et al., 2012; Ji et al., 2015). One of the very well-known proactive fault-tolerant mechanisms is to preemptively migrate application from nodes that are about to fail to another one (Chakravorty et al., 2006). The combination of migration with checkpointing or logging mechanisms, for storing the state of alive connections, can ensure a full migration of an application with all its interactions (Ivaki et al., 2014).

2.2 Solutions for Reliable Communication

In this section, we overview several existing solutions for building reliable distributed applications. We divide these solutions depending on their reliability targets (i.e., stream, message, object, or conversation) and describe them according to the other reliability features, such as reliability mechanisms and reliability semantics.

2.2.1 Stream-Based Solutions

We initiate the discussion with the stream-oriented solutions for reliable communication. The list of the solutions covered, along with their particular features, is presented in Table 2.1. In this table, the "Reliability Mechanism" column refers to the mechanisms we presented in Section 2.1.5. The following column, "Fault tolerance", refers to the kind of failures that might be tolerated by each solution (refer to Section 2.1.2). Since, all the solutions we review in this table only address omission failures in presence of crashes in some component along a distributed communication, we try to be more specific regarding the causes for the omission failures and separate connection crashes from server crashes. In the last column, we present the reliability semantics, presented in Section 2.1.3, offered by the solutions.

We do not refer to the interaction patterns of Section 2.1.1 in the table, because, for instance, TCP is usually the base for many different communication protocols and stacks, thus supporting nearly any interaction pattern needed by the application layer (one-way, request-response, synchronous, asynchronous, etc.). The same might be said, in principle, regarding the other solutions in this category. However, none of these solutions is so deeply understood, used or carefully implemented as TCP. Hence, in practice, some of the other solutions might not support all interaction patterns. For example, RSocket trying to extend the TCP Socket's functionalities, does not provide non-blocking sockets, thus making asynchronous interactions slightly more complex to implement.

TABLE 2.1: Stream-based reliable solutions and their characteristics

Solutions	Reliability Mechanisms	Fault Tolerance	Reliability Semantics
TCP	Buffering, Acknowledgemnt and Retransmission, None (regarding connection or endpoint crashes)	None (regarding crashes)	Best-effort (regarding crashes)
RTP	None (regarding connection or endpoint crashes)	None (regarding crashes)	Best-effort (regarding crashes)
RSocket	Buffering and Explicit Acknowledgment	Connection Crashes	At-most-once
FT-TCP	Logging	Server Crashes	At-most-once
Rocks & Racks	Buffering and Implicit Acknowledgment, Checkpointing, and Migration	Connection Crashes and supports for server crashes	At-most-once
SCTP, cmpSCTP, MPTCP	Multiple connections	Connection Crashes	At-most-once
ST-TCP, HotSway, HydraNet-FT	Active Replication	Server Crashes	At-most-once
ER-TCP	Active Replication and Logging	Server Crashes	At-most-once

TCP

The Transmission Control Protocol (Postel, 1981) provides reliable bi-directional byte stream communication between two processes running over a network. TCP is perhaps the most well-known protocol providing some form of reliable communication. It is connection-oriented (i.e., a logical connection is established between two peers before transferring data) and uses acknowledgments combined with buffering and retransmission of data segments. When an application writes some data to the socket, the operating system stores the data into the socket's send buffer, before transmission. Once an acknowledgment is received, the data might be deleted. On the receiver side, when the data is received by the operating system, it is kept in the socket's receive buffer and an acknowledgment is sent back. The data remains in the receive buffer until it is read by the application. To prevent data losses, applications must wait for the send buffer to have space available, before sending new data. This mechanism, known as the "sliding window mechanism", limits the maximum amount of in-transit data.

Most reliable applications resort to TCP in one way or another, to transport their data. Web browsing, emailing, video streaming (often in a browser), file transfers, and remote shell interactions with the Secure Shell Protocol (SSH) are but a few examples. Despite being very common, TCP cannot handle data losses that occur when there is a connection crash. Technically a TCP connection fails when the operating system aborts a connection, for one of the following reasons: 1) when data in the send buffer is not acknowledged after a given number of retransmissions; 2) when the application waits for reading from the receive buffer for a period of time that exceeds the timeout defined for read; 3) when an underlying network failure is reported by the network layer; 4)

and when the IP address changes (Zandy and Miller, 2002). When a TCP connection fails, resuming communication between peers is quite challenging, even when both endpoints are still running. In fact, the application might have no means (provided by the transport layer) to determine which data did or did not reach the other endpoint, thus making recovery or roll-back to a coherent state very difficult. Therefore, TCP can only offer best-effort delivery to applications, under connection and endpoint crashes.

RTP

The Real-time Transport Protocol (RTP) (Schulzrinne et al., 2003) is designed for real-time transfer of audio and video over the Internet. For this sort of task, packet losses might be much less important than user-perceived delays. RTP offers timeliness guarantees to media streaming applications, usually over UDP instead of TCP, because TCP favors reliable delivery over timeliness. RTP uses timestamps for synchronization purposes and sequence numbers to ensure ordered delivery.

RSocket

Robust Socket (Ekwall et al., 2002) is a session layer solution to overcome the limitations of TCP. It uses an extra level of buffering and acknowledgments to ensure delivery of the bytes stream. The RSocket acknowledgment is done using an additional UDP control channel. When a TCP connection fails, the client sets up a new connection, to resume data exchange. Although RSocket takes care of connection crashes, by transparently reconnecting and resending lost data, endpoint crashes are not handled. Thus, the reliability semantics that can be guaranteed in presence of endpoint crashes is at-most-once delivery of data. The interaction patterns offered by RSocket are similar to the plain TCP protocol, with this difference that the RSocket implementation does not support non-blocking operations (e.g., non-blocking read operation).

FT-TCP

Fault-Tolerant TCP (Alvisi et al., 2001) is based on the concept of wrapping, in which a layer of software surrounds the transport layer and intercepts all connections. Data can come from two points, either the IP layer or from the application layer. A logger is used

at these points as well, to maintain the current state of the TCP connections. Thus, when the server crashes, logs are used after restarting the server or moving the server to another host for recovery of TCP connections. This solution can not recover TCP connections in the case of client and network failures, thus it only can offer at-most-once delivery of data.

Rocks and Racks

Reliable Sockets and Reliable Packets (Zandy and Miller, 2002) are solutions that allow recovery from connection crashes, thus providing transparent network connectivity to applications. They automatically detect connection crashes, including those caused by link failures, extended periods of disconnection, change of IP address, and process migration. They use a control socket to exchange control messages, mainly for detecting the data connection crashes. These systems automatically recover broken connections without loss of in-transit data, by buffering sent data. Rocks is a library that transparently changes the behavior of the application by replacing the default TCP sockets, whereas Racks filters application packets to avoid intercepting the application libraries. These solutions also use process checkpointing in order to support server migration. Justification about reliability semantics offered by this solutions is the same as the previous ones, because exactly-once can not be ensured when all crashes are not properly handled (e.g., endpoint crashes).

SCTP

Stream Control Transmission Protocol (Stewart and Metz, 2001), like TCP, offers a bi-directional, connection-oriented, and reliable transport service to applications. It inherits many of the TCP features, as any application running over TCP can be ported to run over SCTP without loss of function. The main differences revolve around SCTP's support for multi-homing and partial ordering. Multi-homing enables an SCTP host to establish a session with another SCTP host over multiple interfaces identified by different IP addresses. With partial ordering, SCTP maintains ordering only within some sub-flows of the related data streams. Thus, SCTP can benefit applications that require reliable and fast delivery and processing of multiple, unrelated data streams. Regarding the reliability, SCTP is more reliable than TCP, due to its special support for connection replication, and can guarantee at-most-once semantics.

cmpSCTP

Concurrent multi-path Stream Control Transmission Protocol (Liao et al., 2008) modifies SCTP to exploit its multi-homing capability by choosing best paths among several available network interfaces in order to improve data transmission rate. The cmpSCTP tries to reduce latency by selecting the best paths based on updated information about the paths, thus is more suitable than SCTP for real-time applications. However SCTP and its extensions like cmpSCTP, are still not widely used first because the applications must be modified in order to use them, and second because SCTP packets can not pass through various NATs or firewalls.

MPTCP

Multipath TCP (Scharf and Ford, 2013) is an ongoing project of IETF, which has the same objective as cmpSCTP, but tries to address its aforementioned drawbacks. It allows to efficiently exploit several connections between two communicating peers, while presenting a single connection to the application. This is enabled by extending the TCP protocol and choosing several efficient paths between the peers.

ST-TCP

Server fault-Tolerant TCP (Marwah et al., 2005) tolerates TCP server crashes using replication. It uses an active backup that keeps track of the TCP connection state, to take over whenever the primary fails. The migration of the TCP connection to the backup server is transparent to the client. ST-TCP assumes that the backup server is not slower than the primary server, which is an unrealistic assumption. Using this solution, connection crashes caused by network outage or client crashes are not handled.

HotSwap

HotSwap provides fault-tolerance at the TCP level, by modifying the system call library of Linux. It creates two identical instances of the same set of programs on two machines, a master and a backup. The master and backup systems must start at the same time with identical file systems, to ensure they receive the same input from local files.

HotSwap ensures that both copies are synchronized. When a TCP related system call (e.g., creating a TCP socket) is made by an application, another replica socket is created at the other host, to tolerate possible crashes of the former one (Burton-Krahn, 2002).

HydraNet-FT

HydraNet-FT provides an infrastructure to dynamically replicate services across an internetwork and have the replicas provide a single fault-tolerant service to clients (Shenoy et al., 2000). HYDRANET-FT uses TCP with a few modifications on the server side to allow: a) one-to-many message delivery from a client to service replicas; and b) many-to-one message delivery from the replicas to the client. A communication channel between the replicas provides atomicity and message ordering. HydraNet-FT, same as the other later solutions that use active replication, only handles server crashes and does not provide any support for network and client crashes. Thus it only can ensure exactly-once data delivery in presence of these crashes.

ER-TCP

ER-TCP tolerates crashes occurring on the server-side TCP connections, by replicating them among multiple nodes in a cluster. ER-TCP employs a logging mechanism with active replication, to avoid the inconsistency problem that may occur when the replicas do not have the same processing speed as the primary server (Shao et al., 2008). ER-TCP's reliability mechanism is similar to ST-TCP, with this difference that its unrealistic assumption (i.e., backup server is not slower than the primary server) is removed from this solution.

2.2.2 Message-Based Solutions

In this section, we analyze the message-oriented solutions, which are summarized in Table 2.2. The table now includes the interaction patterns described in Section 2.1.1, as the solutions show different communication characteristics.

TABLE 2.2: Message-based reliable solutions and their characteristics

Solutions	Interaction Patterns	Reliability Mechanisms	Fault Tolerance	Reliability Semantics
HTTP	Request-Response, Synchronous, Transient	None (regarding crashes)	None (regarding crashes)	None (regarding crashes)
XMPP	One-way, Request-Response, Synchronous, Transient	None (regarding crashes)	None (regarding crashes)	None (regarding crashes)
HTTPR	Request-Response, Synchronous, Transient	Buffering, Logging & retransmission	Connection and endpoint crashes	Exactly-once delivery and at-most-once processing
CoRAL	Request-Response, Synchronous, Transient	Active replication and logging	Server crashes	At-most-once delivery
WS-Reliability, WS-ReliableMessaging	Request-Response, Synchronous, Asynchronous, Transient	Buffering and retransmission	Connection crashes	At-most-once delivery and processing
ZeroMQ	One-way, Request-Response, Synchronous, Asynchronous, Transient , Persistent	-	None	At-most-once delivery
JMS, AMQP, MSMQ, WebSphere MQ, Oracle AQ	One-Way, Asynchronous, Persistent	Broker, acknowledgment, and support for transactions	Connection crashes and endpoint crashes	At-most-once delivery

HTTP

HTTP (Fielding et al., 2009) is a request-response protocol. It is the cornerstone of the world wide web, and is thus serving as the application communication protocol in many distributed applications, including business and safety-critical services at a world-wide scale. HTTP offers a request-response interaction pattern on top of TCP, thus being a first step towards reliability. Developers can explore the explicit responses of HTTP as a basis (i.e., because they can be considered as acknowledgment for the requests) to implement proper reliability mechanisms for their own applications (e.g., by means of logging and retransmission).

XMPP

Extensible Messaging and Presence Protocol (XMPP) (Saint-Andre, 2011) is a message-oriented protocol, providing a synchronous messaging service based on Extensible Markup Language (XML). It enables the real-time exchange of structured data between any two or more peers. The original transport protocol for XMPP is TCP, but the XMPP community has also developed an HTTP transport for web clients (Paterson et al., 2010). XMPP uses polling method with HTTP to regularly fetch messages stored on a server-side database by an XMPP client using HTTP GET and POST requests. XMPP also uses WebSocket(Fette and Melnikov, 2011) to provide a more efficient transport for real-time messaging (Stout et al., 2014).

HTTPR

HTTPR, as HTTP, is a request-response protocol, which is built on top of HTTP and provides reliable transport of messages between application peers, even in the presence of network or endpoint crashes. It uses logging and retransmission to ensure that each message is delivered to the application exactly-once, but there is no guarantee that the message will be processed exactly-once too (Banks et al., 2002). This is because the server does not accept the requests that are previously delivered to the server, even if the requests are not successfully processed due to the server crashes.

CoRAL

CoRAL (Aghdaie and Tamir, 2009) is a solution for handling the server crashes in web-based services. This solution is built based on connection replication and application-level logging mechanisms. In CoRAL, the state of the TCP connection is preserved using active replication. The TCP stacks of the primary and backup servers process incoming packets simultaneously. Message logging at the application layer is used to log HTTP requests and replies into the backup for replaying purposes, if necessary.

WS-Reliability

WS-Reliability (Evans et al., 2003) is a SOAP-based specification designed to support the exchange of SOAP messages with guaranteed delivery, message ordering, and no duplicates. This specification considers a set of abstract operations (e.g., submit, deliver, response, and notify), to model reliability contracts between the messaging middleware and its users. The specification defines the following reliability semantics: Guaranteed message delivery (At-Least-Once delivery), Guaranteed message duplicate elimination (At-Most-Once Delivery), Guaranteed message delivery and duplicate elimination (Exactly-Once Delivery), and Guaranteed message ordering (ordered delivery). Note that WS-Reliability only protects against network crashes, not against endpoint crashes. Thus, considering this fact, it can only guarantee at-most-once delivery of messages.

WS-ReliableMessaging

WS-ReliableMessaging (Bilorusets et al., 2005) is a web services specification for reliable delivery of SOAP messages. It allows the same set of delivery assurances as WS-Reliability, supported by a modular way of guaranteeing reliable message delivery, and defines a messaging protocol to identify, track, and manage the reliable delivery of messages between two parties. Despite the similarities shared by WS-Reliability (WS-R) and WS-ReliableMessaging (WS-RM), there are significant differences between them, namely: 1) In WS-R, acknowledgments are sent only after the messages have been successfully delivered to the application layer. In contrast, with WS-RM, acknowledgments are sent right after the middleware passes the message to the application, without waiting for the completion of processing; 2) WS-R offers three message reply patterns including *response*, *callback* and *polling* (for asynchronous communication), while WR-RM does not explicitly define message reply patterns (although it also supports asynchronous communication).

JMS

Java Message Service (Richards et al., 2009) is a messaging standard that provides an API for Java developers to create, send, receive and consume messages asynchronously. JMS uses a broker between producers and consumers of messages, to enable loosely-coupled communication. Thus, producer and consumer do not need to be online at the same time. JMS provides the following mechanisms, to ensure reliable delivery of messages: 1) producers and consumers of messages are able to use acknowledgments, to confirm successful production and consumption of the messages. The broker can also send an acknowledgment to the message producer, confirming that it received the message; 2) the broker is also able to store messages in persistent storage, to ensure that messages are not lost if the broker fails before messages are consumed; 3) to ensure exactly-once delivery of messages, producers and consumers can use distributed transactions in their communication with the broker, to group the production and/or consumption of one or more messages into an atomic unit. However, with this asynchronous interaction between the producers and consumers, it is very difficult to have an end-to-end transaction encompassing both the production and consumption of the same message, thus guaranteeing the exactly-once delivery is almost impossible. This fact can be applied to any other asynchronous broker-based solution too. Nevertheless,

multiple implementations of JMS, including Apache ActiveMQ, HornetQ, Sonic MQ, OpenJMS, FioranoMQ, Oracle Message Broker, SAP PI, TIBCO Enterprise Message Service, and JORAM, prove the applicability of this type of solutions.

AMQP

Advanced Message Queueing Protocol (Vinoski, 2006) is also designed to support loosely-coupled and asynchronous communication patterns. It provides flow control, message-delivery and security guarantees. While JMS provides a standard API for the Java platform, AMQP provides a standard messaging protocol across all platforms. This protocol assumes an underlying reliable transport layer protocol (e.g., TCP). AMQP, in a similar manner to JMS, uses acknowledgments to ensure reliable delivery of messages both from the producer to the broker and from the broker to the consumer. AMQP can be found in several popular implementations, including RabbitMQ, WSO2, and OpenAMQ.

ZeroMQ

ZeroMQ (Hintjens, 2013) is a high-performance messaging library aimed for scalable distributed applications. It provides a message queue, but unlike message-oriented middleware, a ZeroMQ system can run without a standalone message broker. The library is designed to have a familiar socket-style API. It supports several messaging patterns including: 1) request-response (similar to remote procedure calls); 2) publish-subscribe, which is one-to-many loosely-coupled asynchronous queue-based communication; 3) push-pull, which is similar to publish-subscribe with the difference that the sender sends the messages using several sockets arranged in a pipeline; and 4) pair, which provides a communication similar to what normal sockets do, with the difference that the messages are received completely at once.

MSMQ

Microsoft MSMQ (Horrell, 1999) is another message queuing protocol for the Windows operating systems. MSMQ enables applications running at different times to communicate across heterogeneous networks and using systems that may be temporarily

off-line. Message Queuing provides guaranteed message delivery, efficient routing, security, and priority-based messaging. It can be used to implement solutions for both asynchronous and synchronous scenarios. MSMQ ensures reliable delivery, by placing messages that fail to reach their intended destination in a queue and then resending such messages once the destination becomes reachable. Similarly to JMS, MSMQ also supports transactions that can encompass multiple operations over multiple queues.

WebSphere MQ

IBM WebSphere MQ (Hart, 2003) is also a queue-based message oriented middleware that simplifies the integration of applications across multiple platforms. WebSphere MQ provides one-time delivery of messages across several operating systems. The vendor emphasizes reliability and robustness of message traffic, and ensures that a message should never be lost if MQ is appropriately configured. Messages can be sent from one application to another, regardless of whether the applications are running at the same time (a queue manager will hold the message until a receiver requests it). Ordering of all messages is preserved (FIFO by default, but priority ordering also exists). WebSphere MQ enables reliable messaging, by using acknowledgments, negative acknowledgments, and sequence numbers.

Oracle AQ

Oracle Advanced Queuing (Gawlick, 1998) is developed by the Oracle Corporation and integrated into its Oracle database as a repository to provide message queuing for asynchronous communications. Since Oracle Advanced Queuing is developed in database objects, all operational benefits of high availability, scalability, and reliability are also applicable to queue data. Typical database features such as recovery, restart, and security are also supported by Oracle AQ. Although all the above broker-based (or queue-based) solutions prevent duplicate delivery of messages but there is no guarantee that the messages will be delivered exactly-once.

2.2.3 Object-Based Solutions

In this section, we review the object-oriented solutions summarized in Table 2.3. We do not include the interaction pattern, as this is Request-Response, Synchronous, and Transient for all solutions.

TABLE 2.3: Object-based reliable solutions and their characteristics

Solutions	Reliability Mechanisms	Fault Tolerance	Reliability Semantics
RPC	-	none	At-most-once
RMI, CORBA, DCOM	Retransmission & Filtering	Connection crashes	At-most-once
.Net Remoting	-	none	At-most-once
FT-CORBA, FTRMI	Active replication	Server crashes	At-most-once

RPC

Remote Procedure Calls (RPC) (Birrell and Nelson, 1984) is a well-known concept designed by Sun Microsystems for distributed communication. Instead of accessing remote services by sending and receiving messages, a client invokes services by making a procedure call that looks similar to a local invocation, thus hiding the details of network communication. Performing a remote procedure call works as follows: 1) the server process registers the service and waits for a call; 2) the client process sends an RPC message including procedure parameters to the server process. The client then usually blocks while waiting for a response; 3) the server process checks authentication, runs the procedure, and returns results to the client. RPC can be used in a synchronous or asynchronous manner, blocking or non-blocking. The reliability mechanisms of competing RPC solutions vary from the at-least-once of the original RPC implementation, to the at-most-once semantics capable of filtering duplicate requests occurring after timeouts. This latter semantics should not be confused with the best-effort semantics (also known as "maybe"), where requests are not re-invoked.

RMI

Java RMI (Remote Method Invocation) (Downing, 1998) is the name of the Java SE RPC implementation. The expression "Remote Method Invocation" is also used for RPC implementations in object-oriented languages. Regarding reliability, Java RMI provides at-most-once semantics: if a method returns normally, the client can be sure

that it was executed exactly once, otherwise when some network exception occurs, the caller cannot determine whether the remote method executed, therefore can not risk second invocation of the same request.

CORBA

CORBA (Vaysburd and Yajnik, 1999) follows the same paradigm as Remote Procedure Calls (RPC). As in Java RMI, CORBA also provides the at-most-once semantics. However, unlike Java RMI, CORBA also provides the maybe option for one-way interaction, with evident performance gains.

DCOM

DCOM (Distributed Component Object Model) (Brown and Kindel, 1998) is the distributed extension of COM (Component Object Model, a component based development model for Windows) that builds an object remote procedure call (ORPC) layer on top of DCE RPC to support remote objects. Reliability semantics are similar to RMI and CORBA.

.NET Remoting

.NET Remoting (McLean et al., 2002) is a technology to create distributed objects in Microsoft .NET. It provides a flexible and customizable architecture as it allows, for example, to replace one communication protocol with another, or one serialization format with another without recompiling the client or the server. .NET Remoting architecture seems to be much easier to use and extend than DCOM, but its reliability related features are weaker.

FT-CORBA

Fault-Tolerant CORBA uses transparent entity redundancy to provide a higher level of reliability than CORBA. This is achieved via replication of CORBA objects, where each replicated object is implemented by a set of distinct CORBA objects called an object group. The members of an object group are referenced using an Interoperable

Object Group Reference (Natarajan et al., 2000). Although FT-CORBA decreases the likelihood of failure, the best reliability semantics it can offer is at-most-once (as CORBA).

FTRMI

Fault-tolerant Transparent Remote Method Invocation (Reis and Miranda, 2012) is a middleware that enhances the Java implementation of the Remote Method Invocation (JRMI) with strong replica consistency to increase reliability. FTRMI extends JRMI with an additional communication layer that multicasts every request to all server's replicas, simplifying the development of fault-tolerant services. As in the previous cases, the reliability semantics provided by FTRMI is at-most-once.

2.2.4 Conversation-Based Solutions

In this section, we review solutions that provide reliability guarantees for distributed interactions that go beyond a simple request and response. Table 2.4 summarizes the features of the conversation-oriented solutions.

Table 2.4: Conversation-based reliable solutions and their characteristics

Solutions	Reliability Mechanisms	Fault Tolerance	Reliability Semantics
EOS2	Logging	Endpoint crashes	Exactly-once
Phoenix	Logging and checkpointing	Endpoint crashes	Exactly-once
iSAGA	Logging	Client crashes	At-least-once

EOS2

Exactly-Once e-Service middleware (Shegalov and Weikum, 2006) uses a logging mechanism on both client and server sides (for web-based applications), to ensure exactly-once execution of requests, even in the presence of crashes. EOS2 masks transient crashes, such as OS crashes, component crashes, and message losses. It is not clear how this solution deals with connection crashes. It may deadlock when both parties are alive, but the TCP connection has crashed (Shegalov and Weikum, 2006).

Phoenix

Phoenix deals with system crashes, by logging component interactions and checkpointing the state. The argument is that Phoenix assumes a window of opportunity for the client failure. Exactly-once delivery is achieved only if the client does not fail during this small window. Phoenix does not deal with connection crashes due to network crashes (Barga et al., 2003).

iSAGA

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

2.2.5 Design Solutions

The idea of using design patterns for software development started more than two decades ago. As said: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander et al., 1977). Although this phrase is talking about the patterns in buildings, it is totally valid in the context of software engineering and object-oriented programming.

In general, a pattern includes four essential elements (Gamma et al., 1994): 1) A name, which describes a design problem and its solution in a word or two; 2) A problem, which describes when to apply the pattern; 3) A Solution, which describes the elements that make up the design, their relationships, responsibilities, and collaborations; and 4) Consequences, which describe the results and trade-offs of applying the pattern.

Among a huge number of design patterns that we can find in the literature, the ones that address reliability issues in distributed communication are missing. In fact, in the last decade the researchers have assisted to organize distributed interactions into a set of design patterns, first for Enterprise Application Integration (Hohpe and Woolf,

2003), and more recently towards SOAP/WSDL and RESTful web services (Daigneau, 2011). This latter book collects multiple known types of high-level interaction between client and server, e.g., request-acknowledge-polling or request-acknowledge-callback, respectively for client polling or server callbacks.

There are several works in the literature addressing some aspects of dependability such as safety (Gawand et al., 2011), security (Laverdiere et al., 2006; Schumacher et al., 2013; Yoshioka et al., 2008), and fault-tolerance (Hanmer, 2013). There are also several works related to scheduling algorithms in real-time systems (Douglass, 2003). However, none of these design patterns addresses the reliability issue in a distributed communication. In this section, we list and explain several design patterns that can be used in design and development of distributed applications, although reliability is not their main concern. We will use some of these design patterns in building and presenting our own solutions for reliability.

Reactor Design Pattern

Reactor Design Pattern (Schmidt, 1995) is used for handling concurrent events that may arrive on several inputs. This design pattern, as shown in Figure 2.1, includes two components: `Reactor` and `Event Handler`. The `Event Handler` implements actions to process and handle an incoming event. It provides an interface to be used as callback method (*handle_ event()*) for delivery of events. The `Reactor` defines an interface for registration, and deregistration of the `Event Handler`s. The `Reactor` continuously checks the arrival of new events and deliver them to an appropriate `Event Handler`.

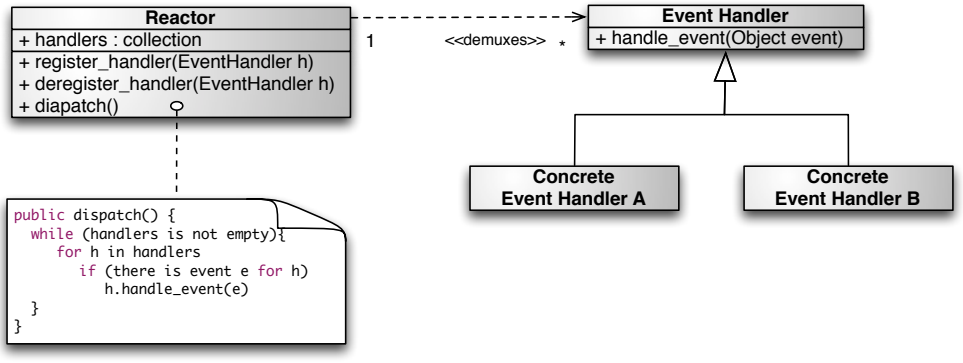

FIGURE 2.1: Reactor design pattern

Regarding distributed communication, Reactor design pattern can be used to implement concurrent connections on both client and server side, using a dispatcher that delivers incoming data to appropriate service handlers.

Observer Design Pattern

Observer Design Pattern (Hohpe and Woolf, 2004) allows an object, named `Subject`, to maintain a list of its dependents, named `Observer`s, to notify them of any state changes, by calling their method *notify()*. Figure 2.2 presents this design pattern. The main difference between the Reactor and Observer design patterns is that the Reactor uses a dispatcher to demultiplex events to a correct event handler, while in the Observer design pattern, notifications are pushed to all registered observers when an event occurs.

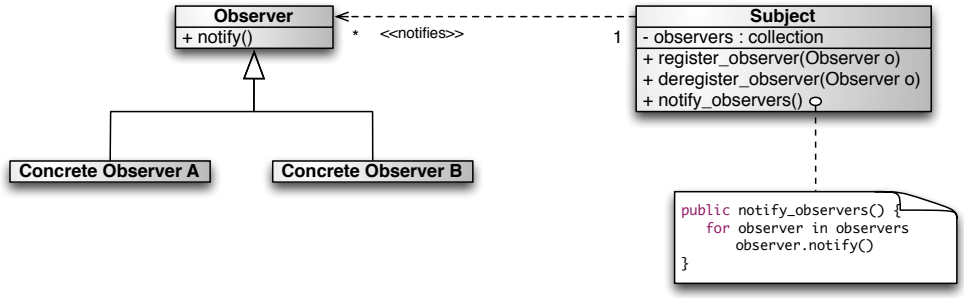

FIGURE 2.2: Observer design pattern

Acceptor-Connector Design Pattern

Acceptor-Connector Design Pattern (Schmidt, 1996) tries to simplify the design and implementation of connection-based applications, by decoupling event dispatching process from connection setup and service handling. It is worth mentioning that the Reactor design pattern is used to implement the event dispatching process.

This design pattern (refer to Figure 2.3) includes an `Acceptor`, a `Connector`, a `Transport Handle`, a `Passive Transport Handle`, a `Service Handler`, and a `Dispatcher` (i.e., these two later components belong to the Reactor design pattern). In the Acceptor-Connector Design Pattern, the client asks the `Connector` to send a connection request to the server by initialization of a `Transport handle`. In contrast, the server uses a `Passive Transport Handle` to receive the connection requests. When

a connection request is accepted by the server, a new `Transport Handle` (or a new connection) is created in both client and server. In the server, the new connection is given to the `Acceptor` to be handled properly. Both `Connector` and `Acceptor` initialize `Service Handler`s by passing the transport handle created previously, which in turn register the handle and their references into the `Dispatcher`.

The actual operations of reading and writing data are in charge of the `Transport Handle`, as the actual operation of receiving the connection requests is in charge of the `Passive Transport Handler`. But, since reading from the `Transport Handle` and receiving connection request by the `Passive Transport Handle` might block the application process, the `Dispatcher` takes the responsibility of receiving new events (either application data or connection request) and multiplexes them to the appropriate handles (`Service Handler`, `Acceptor`, and `Connector`). In fact, it provides a single blocking point for the client or server application.

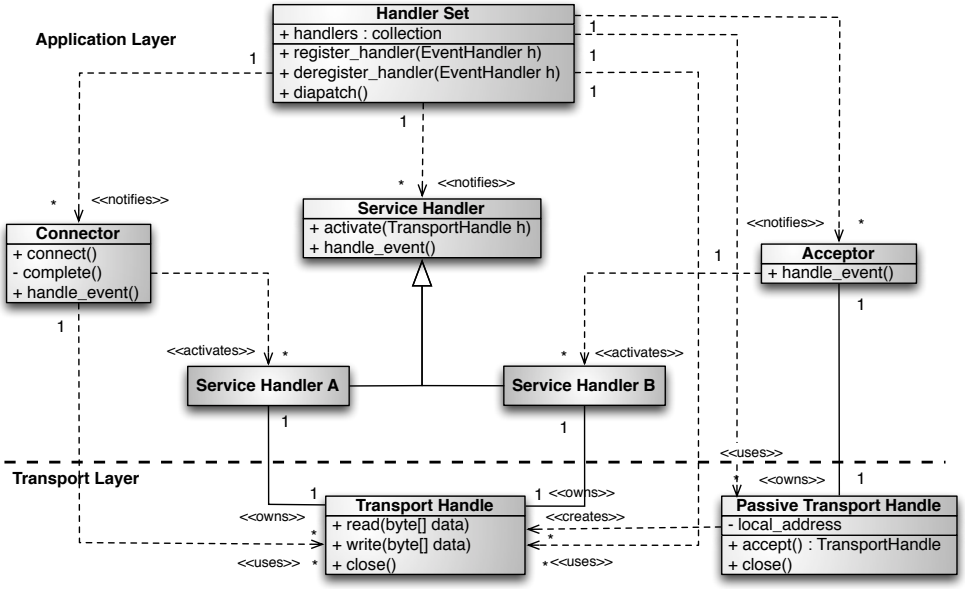

FIGURE 2.3: Acceptor-Connector design pattern

Despite being extremely common, this design pattern has a number of drawbacks from the scalability 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.

Leader-Followers Design Pattern

A common strategy to support concurrency is to use multiple threads, assigning one thread to 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 limited set to new connections. When the connection is closed, the thread returns to the set and waits for another connection assignment. The association between threads and connections in this solution is bounded. With this form of association, new connections may have to stay on hold until older connections finish, because there is no free thread in the set to be assigned. Furthermore, some of the threads that are already dedicated, may not have events to process for some periods of time, thus contributing to degrade the performance of the server.

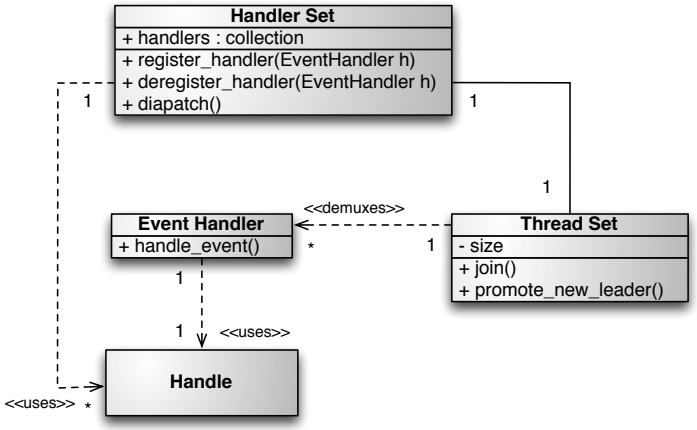

FIGURE 2.4: Leader-Followers design pattern

The Leader-Followers Design Pattern (Schmidt et al., 2000) does not present these disadvantages. It is used to support high concurrency in handling independent events. Refer to Figure 2.4. In this pattern, the threads of the `Thread Set` are assigned to

the `Event handlers` only when an event occurs on their `Handle`. When the event is processed, the thread rejoins the set.

2.3 Applications and Reliability Requirements

The number and diversity of distributed applications requiring reliable communication is quite large. In this section, we present a set of nine well-known groups of applications with distinctive communication features. We organize applications according to the following features (please refer to Table 2.5): objectives (a), criticality (b), timeliness requirements (c), interaction patterns (d), reliability target (e), reliability requirements (f), and solutions (g). We selected criteria (a), (b), and (c) because they are related to the reliability requirements of the applications used in criterion (f), whereas criteria (d), (e), and (f) were selected due to their outstanding importance for selecting the right protocol or middleware solution for the applications. Finally, (g) was selected to show, under the light of the previous criteria, real protocols or middleware supporting the reliable communication needs of these applications. In the next paragraphs, we go through these criteria and present them in detail.

Table 2.5: Applications and their objectives, characteristics and reliability requirements

ID	Applications	Description			(d) Interaction Patterns	(e) Reliability Targets	(f) Reliability Requirements	(g) Protocols & Middleware
		(a) Objectives	(b) Criticality	(c) Timeliness				
1	Media streaming (e.g., Youtube, Netflix, Amazon Instant Video)	Resource-Centric	Non-critical	Soft	One-way	Stream	Best-effort (ordered and correct delivery in real implementations), on-time delivery	HTTP over TCP, UDP, or RTP
2	Online multi-player games, Online shared documents (e.g., Google Documents), Chat (e.g., Gtalk, WhatsApp)	User-Centric	Non-critical	Soft	One-way	Message	Ordered, correct, and on-time delivery	HTTP over TCP
3	File Sharing (e.g., Dropbox, Google Drive)	User-Centric	Non-critical	None	One-way	Stream	Ordered and correct delivery	HTTP, FTP, NFS over TCP
4	Email (e.g., Yahoo Mail, Gmail), Usenet Newsgroups	User-Centric	Non-critical	None	One-way	Message	Ordered, correct and exactly-once delivery	HTTP, SMTP, IMAP, POP3 over TCP, NNTP over TCP
5	Self-Adaptive Systems (e.g., rainbow), System of Systems (e.g., space exploration SOS), Air traffic control system, Aircraft control systems, Industrial control systems (e.g., DCSs)	System-Centric	Business-Critical, Safety-Critical	Hard	One-way	Message	Ordered, correct, on-time delivery, exactly-once execution (or at-least-once)	TCP
6	Publish/Subscribe systems (e.g., News feeds)	Resource-Centric	Business-critical	None or Soft	One-way	Message or Object	Ordered, correct, on-time, and at-least-once delivery	JMS, AMQP, MSMQ, WebSphere MQ, Oracle AQ over TCP
7	Static information systems (e.g., dictionary, encyclopedia) Dynamic information systems (e.g., stock market website), File Server, Remote services (e.g., Google, Wolfram search), News Server	Resource-Centric	Non-critical	None	Request-Response	Message	Ordered and coret delivery	HTTP, NFS, FTP
8	Transactional services (e.g., online banking, online shopping, online auctions)	Service-Centric	Business-critical	None	Request-Response	Conversation	Ordered and correct delivery, exactly-once or at-most-once execution	HTTP over TCP
9	Peer-to-peer systems (e.g., bittorrent,	Resource-Centric	Non-critical	None	Request-Response	Stream	Ordered and correct delivery	TCP

(a) Objectives: This criterion distinguishes the applications according to their main goals. Inspired by the classification in (Tanenbaum and Steen, 2006), we characterize distributed applications as *user-centric*; *resource-centric*; *system-centric*; and *service-centric*:

- *User-centric:* The application's goals are set on the user. In general, this type of application serves the purpose of connecting users by allowing them to collaborate and exchange different types of information (e.g., text, audio, or video). Examples include social networks and messaging applications (e.g., Facebook and Skype), and on-line multilayer games. Applications of groups 2, 3, and 4 belong to this category.

- *Resource-centric:* In this case, the main goal of the application is set on resources (Tanenbaum and Steen, 2006). Resources can be tangible, such as printers, processors, storage facilities, or intangible, such as data, news, musics, movies, and pictures. Applications used for media streaming, such as Youtube and publish/-subscribe applications, e.g., in the form of news feeds are resource-centric examples. In Table 2.5, applications in groups 1,6,7, and 9 belong to this category.

- *System-centric:* Refers to distributed applications whose focus is set on one or more systems. Group 5 belongs to this category and examples include software to control air flight navigation and distributed control systems (Galdun et al., 2008; Tindell and Burns, 1994); self-adaptive systems, which self-manage to adapt to changing runtime conditions (Garlan et al., 2004; Huebscher and McCann, 2008); systems that use other systems as in systems-of-systems, whose goal is to use the composition of simpler systems, to create a more complex one that delivers more functionality (Maier, 1996).

- *Service-centric:* In this category, we can find applications whose goals are centered on delivering service. Usually these applications enable companies and organizations to bring their services on-line (Papazoglou, 2003) and allow users to pay bills, transfer money, and buy or sell all kinds of goods remotely. Although they might use resources or other systems in the process, they are highly focused on the service. Thus, typical examples include on-line financial services and e-banking. Applications in group 8 belong to this category.

We can easily see that some applications can have more than a single goal. Nevertheless, it is usually fairly easy to understand which goal better expresses the purpose of the application. As an example, although Dropbox clearly touches the resource-centric goal, as it can store files for public access, its main orientation is towards sharing user files.

(b) Criticality: This criterion essentially describes how important the applications and their operations are. Some applications are non-critical and a weak reliability semantics should suffice. On the other hand, business or safety-critical applications require stronger reliability guarantees (Rushby, 1994):

- *Non-critical:* Some applications do not require strong reliability semantics as they do not support critical processes (e.g., a music streaming service). Thus, complex reliability mechanisms are not required, as the reliability guarantees are minimal or none.
- *Business-critical:* These applications support enterprises and their failure could result in loss of revenue and reputation. In these applications, message delivery and processing must be guaranteed (Abie et al., 2009; Jones et al., 2000). Groups 5, 6 and 8 refer to business-critical applications.
- *Safety-critical:* In this type of application, a failure could result in loss of lives, significant property damage, or damage to the environment (Knight, 2002). Group 5 refers to safety-critical applications.

(c) Timeliness: This criterion focuses on the non-functional attribute of timeliness. In the context of our thesis, it indicates how important time is in a distributed application. In an application with timeliness requirements, the correctness of the service depends not only on the correct computation results (e.g., the correct content of a response), but also on the time at which the results are delivered (Lann, 1997; Stankovic and Ramamritham, 1989). Regarding this objective, we can find the following three cases of timeliness requirements:

- *None:* Applications with no timeliness requirements essentially refer to the case where messages can arrive at any time. This does not influence the correctness

of the service being delivered (Ferrari, 1990). Mail delivery in group 4 imposes no time bounds, or at best, very relaxed time bounds.

- *Soft:* This reflects the case where a message must arrive before a given time limit. In this case, this time limit might be violated up to a given maximum extent, without impacting the correctness of the service being delivered. Thus, in a soft real-time system, a failure does not necessarily occur if the system is unable to meet the given time limit (provided that the message arrives within some additional time constraint, often bounded by then needs of human interaction). Multimedia streaming and some on-line interactions between users (groups 1 and 2) have these "soft" timeliness requirements. Group 6 of publish/subscribe systems may also expect messages to arrive within some time frame (e.g., to feed sports results) (Carvalho et al., 2005).

- *Hard:* In this case, messages must arrive within a time limit that is strict and cannot be violated in any way. The violation of this time limit impacts the correctness of the service being delivered. In hard real-time applications, when a message does not arrive in time, the interaction fails (or the component or even the whole system) (Northcutt and Kuerner, 1992). Group 5 is the only one requiring hard timeliness constraints (Kopetz et al., 1989).

(d) Interaction Patterns: This criterion corresponds to the messaging pattern described in Section 2.1.1 (i.e., one-way or request-response). The purpose of this criterion is to capture the kind of interaction perceived by the user, regardless of the concrete technology that supports the interaction and the fact that such technology will most certainly use some form of request-response. As shown in the table, groups 1 to 6 (e.g., multimedia streaming, multi-player games) inherently require a one-way messaging paradigm to achieve their objectives, despite of their use of request-response protocols, such as HTTP. The remaining applications (groups 7 to 9) necessarily require the request-response messaging pattern (e.g., online shopping).

(e) Reliability Targets: As previously discussed in Section 2.1.4, the reliability target of applications can be *stream*, *message*, *object*, or *conversation*. Applications of groups 1, 3, and 9, which usually deal with multimedia data, target *streams* of bytes for reliability. Group 8, which deals with online transactions and business operations,

targets whole *conversations* for reliability. The remaining applications are message-oriented, thus targeting *messages* for reliability. We may also find some applications that use object-based communication, e.g., in publish/subscribe systems (Pietzuch and Bacon, 2002), but as explained in Section 2.1, from a reliable communication perspective, they can usually be considered as a specific type of message-based or conversation-based applications.

(f) Reliability Requirements: This criterion refers to the requirements of the applications in what concerns data delivery and execution (please refer to Section 2.1.3 for details). The best-effort requirement, which means that it is acceptable for data to be lost, delayed, or delivered out of order, fits the needs of applications in group 1. Although these applications can be implemented with an unreliable lightweight protocol as UDP, in practice they tend to use HTTP and TCP, thus ensuring ordered and correct delivery.

Ordered and correct delivery, which respectively refer to the delivery of data in the same order and with the same content as it was sent, are a need of the reliable applications presented in Table 2.5. TCP sockets could ensure these properties very easily, but once we consider connection and endpoint crashes, implementations that deliver these semantics become much more involved. On-time delivery of data is required for the applications that have timeliness constraints (either soft or hard), which is the case of groups 1, 2, 5, and 6.

Finally, some applications and services require stronger guarantees. We can find essentially two distinct cases. The first case includes applications belonging to groups 5, 6, and 8, which simply need to ensure the execution of invocations. The second case includes applications in groups 5 and 8 involving non-idempotent operations, which need to ensure that invocations are not duplicated and that every peer knows the outcome (e.g., the success or failure of some transaction). The former case requires exactly-once or at-least-once semantics and the later case requires exactly-once or at-most-once semantics.

(g) Protocols and Middleware: This identifies which protocols or middleware are being used, or at least are typically used, by the applications identified. The media

streaming applications (group 1) usually resort to HTTP as the application layer protocol and use TCP, RTP, or UDP underneath that respectively offer guaranteed and ordered delivery; not guaranteed but ordered and having timing information delivery; and best-effort delivery. The applications that belong to group 2 (e.g., multi-players games) typically use HTTP over TCP, although their timeliness requirement cannot be met using these protocols.

Applications of group 3 (e.g., file sharing), in addition to HTTP, use File Transfer Protocol (FTP) (Postel and Reynolds, 1985) or Network File System (NFS) (Nowicki, 1989) in the application layer. FTP was widely popular in a recent past and is still used today for file transfer over Internet. It uses TCP underneath, to guarantee reliable delivery of messages and data through its message-based control channel and stream-based data channel. Indeed, there is no provision in FTP for detecting lost data (it relies on TCP for this purpose), but a restart procedure is provided, to help developers handling network and endpoint crashes. To use the restart procedure, it requires the data sender to insert a special marker in the data stream with some information (e.g., bit-count, or a record-count). The receiver of data then marks the corresponding position of this marker, and returns this information back to the sender. Should a failure occur, the sender can restart the data transfer by identifying the marker point (Postel and Reynolds, 1985). NFS is another file transfer protocol that provides transparent remote access to shared files. Regarding reliability, the initial versions of NFS used stateless servers, thus having a very clear advantage when recovering from server crashes. It also tried to make operations idempotent (in the current version of the protocol, some operations are not idempotent). Thus when a client did not receive a response, it could safely resend its request until it got a server response. The client does not even needed to know whether the server had crashed, or the network went down (Nowicki, 1989).

Applications that belong to group 4 (e.g., email) use the Simple Mail Transfer Protocol (SMTP) (Postel, 1982), Internet Message Access Protocol (IMAP) (Crispin, 2003), Post Office Protocol version 3 (POP3) (Myers and Rose, 1996), and Network News Transfer Protocol (NNTP) (Feather, 2006), in addition to HTTP. These protocols assign identifiers to each message, thus enabling message confirmation and retransmission. NNTP, which is mainly used in the usenet newsgroups, is an application layer protocol that serves to exchange news articles between news servers and for users to read and post news articles. NNTP does not guarantee any reliability semantics by itself, but

provides facilities to implement some reliability mechanisms above it (e.g., assigning unique message-id to each article). NNTP goes into details on how to implement servers that ensure exactly-once delivery, in the presence of endpoint and network crashes.

Applications of group 5 normally use TCP for reliable communication, often implementing custom solutions on top of it, according to their requirements. Publish-subscribe applications (group 6) usually take advantage of several popular solutions, such as JMS and MSMQ, to implement the interaction pattern they need. Applications in group 7, typically use HTTP, FTP and NFS, all running over TCP. Applications of group 8 (e.g., online banking) use HTTP over TCP, despite being very different in specifications and requirements. Finally, peer-to-peer applications (group 9) use TCP for data transmission.

From the analysis of the last column of Table 2.5, it is evident that most of the solutions used, on their own, do not match the reliability requirements of the previous column, a fact that we regard as remarkable. A clear example are the typical implementations of on-line banking and shopping, which use HTTP over TCP, thus being far from offering the required semantics. By opting for these solutions developers must add custom-made mechanisms, to deliver the proper reliability semantics.

The generalized preference of developers for mature and popular technologies, rather than middleware that was designed to offer superior reliability guarantees, is quite clear. TCP and HTTP are almost ubiquitous, but offer little guarantees. On the other hand, other protocols, such as FTP, NNTP, and SMTP offer additional levels of reliability but are not generic and address communication issues for very specific applications (respectively for file servers, news servers, and email servers).

2.4 Discussion

We initiated the chapter by presenting the main concepts involved in distributed communication in Section 2.1, where we synthesized and discussed key aspects of reliable distributed communication, such as the main type of interactions, semantics, types of failures, reliability targets, and mechanisms. We then identified existing solutions for reliable communication and mapped them into the concepts previously discussed. Finally, we applied this knowledge to categorize real cases of applications that need

reliable communication, in which we observed a large diversity of scenarios and a generalized mismatch between application requirements and what current reliable communication solutions can offer.

In this section, we highlight and discuss the main findings that resulted from this survey work. We also identify, based on the analysis and discussion, what we believe is currently the main open research line for us to pursue in the field of reliable distributed communication. Regarding the reliable communication solutions analyzed in the chapter (please refer to Tables 2.1, 2.2, 2.3, and 2.4), we identified the following relevant aspects:

- There is currently a large variety of options to achieve reliable communication, which target different reliability cases, but there is an even greater number of real solutions, in many cases with overlapping goals. The differences in such cases are, in practice, reduced to different designs or implementations.
- Selecting the right reliable communication solution is not an easy task, not only considering the variety of existent solutions, but also considering the different requirements, which are not always trivial to identify.
- Regarding the acceptance of solutions, we observed that only a very small number did actually succeed to gain wide acceptance among developers. Furthermore, some of the protocols were tailor made for specific applications (SMTP or NNTP) and cannot be easily adopted elsewhere.
- Considering stream-oriented solutions, TCP is the only widely used solution. This is quite curious as, in fact, it is the simplest and the least reliable of all solutions discussed in this chapter. TCP has implementations for a huge number of platforms, its API is very simple and well known, and it is quite mature. This probably makes its disadvantages relatively less important.
- TCP has significant limitations when the goal is to provide reliable communication (e.g., TCP does not possess mechanisms to handle connection crashes). Developers many times assume that the simple use of TCP brings in fully reliable communication, which is not the case (Ekwall et al., 2002).
- Among message-oriented solutions, only JMS and MSMQ became quite popular. These solutions are not only popular, but also quite rich in terms of reliability

features. However, they are only suitable to be used in asynchronous communication. The remaining solutions in this category, which can be suitable to be used in synchronous communication, did not gain any special acceptance until now.

- RPC and RMI are quite well-known solutions, but have poor reliability mechanisms. Their invocation model (i.e., blocking and non-pipelined invocations of remote objects) does not fit the requirements of many real applications.
- None of the solutions listed in Table 2.4 gained acceptance among developers for implementing conversation-based applications, despite being quite powerful from a reliability point of view.

Taking as a reference the diversity of scenarios, where reliable communication is required, even within a single application group (please refer to Table 2.5 in Section 2.3), it is easy to observe that most of the solutions available do not fit the applications' requirements very easily. With some exceptions, such as publish/subscribe applications, which have excellent supporting middleware, most of the cases are quite complex to implement. We identified the following relevant aspects during this analysis:

- There is a large variety of applications that require reliable communication, but at the same time have quite distinct objectives and require different interaction patterns and reliability features.
- Many distributed applications require one-way messaging, but due to the lack of solutions that support reliable one-way messaging, in many cases, applications use request-response protocols (e.g., HTTP) to achieve their goals.
- Many of the applications analyzed require reliable message transmission. Among the rest, it is easier to find applications that require reliable stream-based communication than reliable conversation-based communication. The strongest reliability semantics (i.e., exactly-once) is required for only a few groups of applications that are either business-critical or safety-critical, and are typically associated with reliable message or conversation.
- Despite being very different in specifications and requirements, most of the applications analyzed resort to the HTTP and TCP protocols for reliable communication. Although HTTP and TCP do not provide many reliability guarantees,

frequently they are selected and used in conjunction with custom mechanisms to meet the applications' requirements. This shows the importance of maturity and popularity in the process of selecting a solution for achieving reliable communication.

- Due to the lack of matching reliability solutions for specific types of applications, it is usually up to developers to come up with their own solutions. This is obviously an error-prone process, which can actually impair reliability, particularly for business and safety critical applications that require very strong reliability semantics (i.e., exactly-once).

The above analysis exposes a clear mismatch between the features offered by communication solutions and the features needed by applications, thus declaring that standards and implementations are lagging behind real application requirements. This strongly suggests that there is open space for further research in the field of reliable distributed communication.

Most of the solutions in the long list discussed in Section 2.2 did not gain enough acceptance among developers to be used in real applications, mainly due to their complexity (e.g., HotSwap, HTTPR, and EOS2) or to the fact that they are too specific (e.g., CoRAL, .NET Remoting, and Phoenix). We argue that a different approach to address reliability issues in distributed systems is needed, rather than developing new libraries. **Research in design-based solutions** (Gamma et al., 1994) can definitely help to reduce the growing number of custom solutions that try to deliver reliable communication by directing developers to design patterns. The need for reliable communication is shared among many distributed applications, independently of the platform or the programming language. Using design patterns to guide such implementations is an excellent solution to handle common cases, such as connection crashes (Hanmer, 2013; Hohpe and Woolf, 2003).

2.5 Conclusion

Achieving reliable distributed communication can be a difficult task. This is especially true if we consider the variety of requirements needed by applications and the lack of proper solutions that match those requirements. The applications needs greatly vary in

many dimensions, such as the semantics needed (e.g., at-most-once, ordered delivery), the types of failures to handle, or what is the target of the reliable communication (e.g., a byte stream, or an entire message). The available reliability mechanisms (e.g., retransmission, filtering) are also diverse and many times their selection is not trivial (i.e., the applications requirements can be complex and difficult to map into specific mechanisms).

Research and development efforts have produced a great number of solutions for reliable communication. Unfortunately, the number of solutions does not considerably simplify the task for developers, which understandably, prefer well-known, mature, although less reliable solutions. In fact, only a very small number of the solutions for reliable distributed communication did actually succeed and became popular. This was the case of JMS, MSMQ and just a few more. These solutions are both popular and rich in reliability features.

We analyzed a wide variety of applications requiring a large number of different communication and reliability features. The use of less reliable (but mature) solutions is evident and widespread. This exposes a mismatch between the features offered by communication frameworks and the features needed, thus suggesting that standards and implementations are lagging behind real applications.

The evidence that solutions do not match the needs of applications suggests that further research efforts are required. We believe, based on the state of the art, that such efforts should occur at a different scale (e.g., design). The complex scenarios involving reliable communication also suggest that developers need better tools and techniques for understanding applications needs in terms of reliability. Only with such knowledge, it is possible to make informed decisions, when the goal is to select a proper solution for an application. Stemming from the mismatch between the applications needs and the reliability solutions currently being used in real scenarios, we believe that there is also a strong need, not only for techniques that allow developers to assess and compare different solutions, but also for better understanding applications requirements in terms of reliable communication.

Despite the large knowledge and technical base in reliable distributed communication, there is a huge open space for research in this area. The research line identified,

regarding design-based solutions, involve significant challenges and open research opportunities. Based on the issues discussed in this chapter, we expect to see significant efforts in this area in the near future.

Chapter 3

A Reliable Stream-Based Solution for Distributed Interactions

The Transmission Control Protocol (TCP) plays a major role in building reliable communication in distributed applications. Despite offering reliability against dropped and reordered packets, the widely adopted TCP provides no recovery options for connection crashes that may occur due to, for example, long-term network outages.

Technically, a TCP connection fails when the operating system aborts a connection, for one of the following reasons: 1) when data in the send buffer is not acknowledged after a given number of retransmissions; 2) when the application waits for reading from the receive buffer for a period of time that exceeds the timeout defined for the read operations; 3) when an underlying network failure is reported by the network layer; 4) when the IP address changes; and 5) when the application layer requests to abort the connection implicitly (Zandy and Miller, 2002).

When a TCP connection fails, resuming communication between peers is quite challenging, even when both endpoints are still running, because the application has no means to determine which data did or did not reach the other endpoint, thus making recovery impossible if no additional mechanism is used. When the connection fails, developers must rollback the application to some coherent state on their own, many times using custom error-prone solutions. Overcoming this limitation of TCP is, therefore, a challenging and deeply investigated problem. However, none of the existing solutions has succeeded to gain wide acceptance, because, in general, most of them impact TCP's

simplicity, performance or ubiquity (Bicakci and Kunz, 2012; Burton-Krahn, 2002; Jin et al., 2003; Marwah et al., 2003; Shenoy et al., 2000; Zandy and Miller, 2002). The rest of the solutions are often not mature or require specific computational platforms (Ekwall et al., 2002; Liao et al., 2008).

In this chapter, we propose a solution, named Connection Handler Design Pattern, to overcome TCP's shortcoming in handling connection crashes. This design pattern allows developers to implement distributed applications that are able to tolerate connection crashes without loosing any data, independently of their platform and programming language, alleviating developers from creating custom error-prone solutions for a recurring problem. Then, based on the Connection Handler Design Pattern, we present a design solution to stream-based applications (e.g., video streaming), requiring reliable communication for transmission of byte streams. Finally, we improve our design by adding a combination of the Acceptor-Connector and Leader-Followers patterns, in order to build highly scalable reliable applications.

The remainder of this chapter is organized as follows. Section 3.1 presents the general design of a connection-based application. Section 3.2 presents the proposed solution for recovery from connection crashes. Section 3.3 presents our reliable solution to the stream-based connection-oriented applications. Section 3.4 presents a highly concurrent design solution to the reliable connection-based applications. Finally, Section 3.5 concludes this chapter.

3.1 Basic Design for Connection-Based Applications

In a connection-based communication, two peers establish a connection or a session before starting to exchange any data. This connection is created using a transport handle like a TCP Socket. One peer initiates the connection by sending a connection request to the other peer. The initializer of a connection is called "client", and the peer that accepts the connection request is called "server". Figure 3.1 presents the design of a connection-based application. The remainder of this section describes the components of this design and collaboration between them in detail.

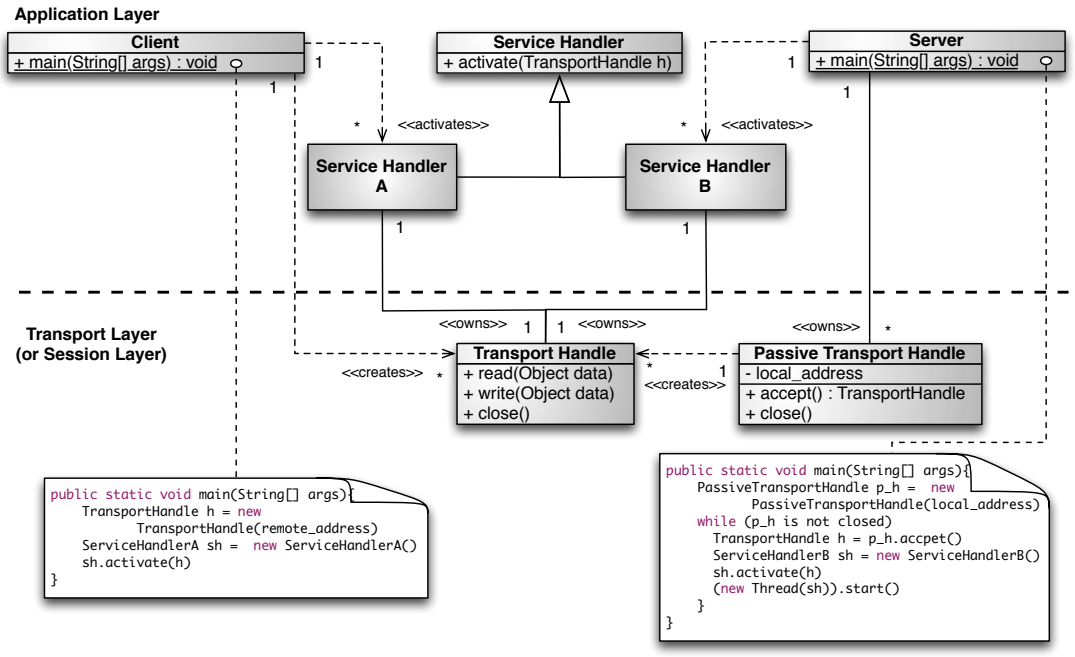

FIGURE 3.1: Basic design of a connection-based application

3.1.1 Components

As shown in Figure 3.1, each distributed connection-based application includes the following components:

- **Transport Handle** provides an interface to the applications to establish a connection (or session), write data, read data, and close the connection. This component belongs to the transport or session layer. A very well known example of a transport handle is a TCP Socket.
- **Passive Transport Handle** is a passive-mode `Transport Handle` that is bound to a network address (i.e., an IP address and a port number) below the application layer. It is used by a server to receive and accept connection requests from clients.
- **Service Handler** implements application services and business logic, typically playing two different roles as sender versus receiver (or as client versus server). For this reason we have two different components which extend the `Service Handler`, namely `Service Handler A` and `Service Handler B`. However, we may have

applications whose peers play both roles (e.g., sender and receiver) in different circumstances during a session. In addition, each `Service Handler` owns a `Transport Handle`, such as a socket, to exchange data with its connected peer.

- **Client** implements the actions to start establishing a connection to the `Server`, and then to initialize and activate a `Service Handler`.
- **Server** keeps checking the arrival of new connection requests and may own one or more `Passive Transport Handle`s. Upon arrival of a new connection request, a new `Transport Handle` is created. The `Server` then initializes and activates the `Service Handler` by passing the new transport handle.

3.1.2 Collaboration Between the Components

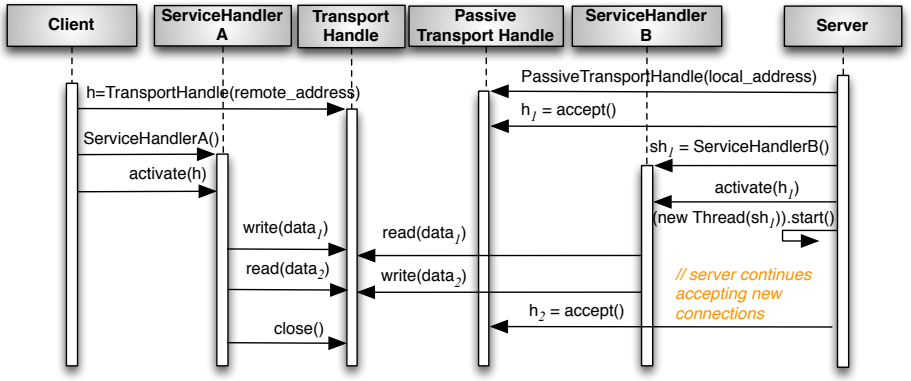

FIGURE 3.2: Collaboration between the components of the connection-based application

As shown in Figure 3.2, the `Client` starts establishing a connection to the server by creating a `Transport Handle` (h) and passing the server's network address (*remote_address*). On the other side, the `Server`, which already initialized a `Passive Transport Handle`, waits in a loop, for a connection request, by calling the method *accept()* of the passive handle. Upon reception and acceptance of the connection request on the server side, a `Transport Handle` is created (h_1). Afterwards, the `Service Handler` is initialized and activated, in both client and server (respectively *sh* and sh_1), by passing the transport handle (respectively h and h_1) through the method *activate()*. Then, after the successful connection establishment phase, they both can start

to exchange data ($data_1$ and $data_2$). Pseudocode presented in Figure 3.1, is a simplified description and implementation of the process explained above for connection establishment and service initialization.

3.2 Connection Handler Design Pattern

In a connection-based application, whose design was presented in Figure 3.1, the service handlers may suffer from connection crashes, when their transport handle is aborted by the operating system, due to the reasons previously mentioned in the beginning of this chapter. This is a quite challenging issue, first because the transport handle does not tackle connection crashes, and second because the transport handle does not provide any means for the application to access information regarding the data sent and received.

In this section, a solution, named Connection Handler design pattern, is proposed for recovery from connection crashes, which can be used by any component using a transport handle like TCP Socket. According to the basic design presented in the previous section, this component is a service handler that owns a transport handle to communicate with a remote peer. Figure 3.3[1] presents our solution, which is presented independently of the application's logic, programming language (or object-oriented programming languages) and the platform on which the application is running. The Connection Handler design pattern, its components, and the collaboration between its components are described in this section.

3.2.1 Reliable Endpoint

`Reliable Endpoint` is one of the components of the Connection Handler design pattern, which owns a transport handle (*handle*) to communicate and exchange data with its remote peer. According to the design of the connection-based application (Figure 3.1), the `Reliable Endpoint` can be a `Service Handler`, or even a middleware (i.e., might

[1] In all figures presented in this thesis, regarding design patterns, we use blue color for the components that are part of our solutions (usually in the session layer) and gray color to show the existing components (usually in the application and transport layers). We also use a light blue, to distinguish the components that have already been explained from the components that are being explained in that section.

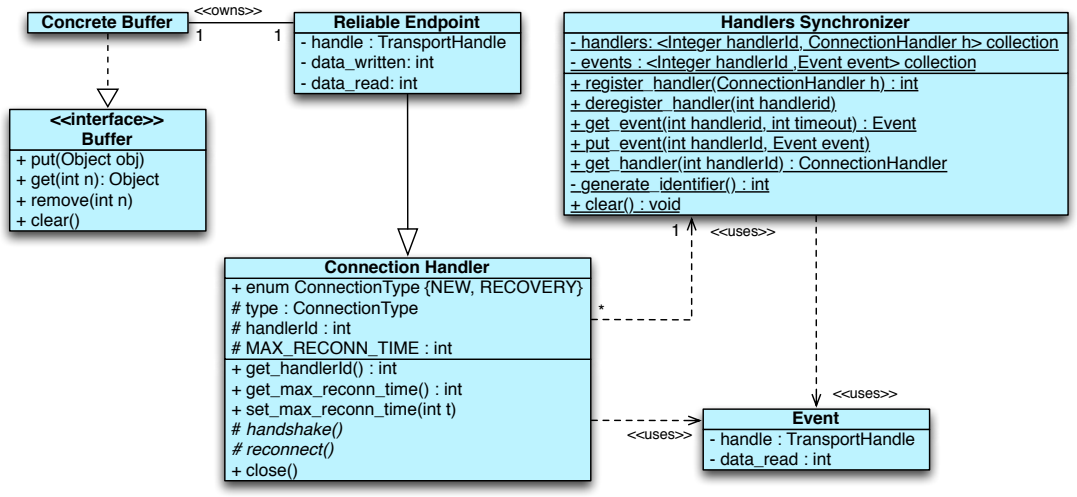

FIGURE 3.3: Connection Handler design pattern

be used for different reasons) underneath the `Service Handler`. This component stores the sent data into a buffer, to enable retransmission should a connection crash occur.

The `Reliable Endpoint` also keeps the track of the data sent and received (i.e., in *data_written* and *data_read* respectively). Depending on the type of data exchanged between the application's peers, the value of these attributes may represent different meanings; for example, it may represent the number of bytes sent or received when the data type is an array of bytes, or the identifier of the last message sent and received, when the data type is a message.

To reestablish a failed connection and retransmit the data that has been lost due to connection crashes, the `Reliable Endpoint` needs to implement some extra actions that are defined and encapsulated in a new component, named `Connection Handler`.

3.2.2 Buffer

Each `Reliable Endpoint` owns a `Concrete Buffer`, to keep the data sent, because all data in transit (e.g., data in the sender's send buffer and receiver's receive buffer) may be lost due to a connection crash. The `Concrete Buffer` implements the interface of `Buffer`, which allows saving, retrieving, and removing the acknowledged data, respectively through the methods *put()*, *get()* and *remove()*. The method *clear()* is used to remove all data from the buffer when it is not needed anymore (e.g., when the connection is closed by the application). The `Concrete Buffer` must be implemented

properly depending on the type of data (e.g., bytes or message) that the peers use for communication. For this reason, the term "Object" is used, in the design pattern, as the type of the data, stored in the `Buffer`, to generalize the design for all kinds of applications including stream-based, message-based, or object-based.

3.2.3 Connection Handler

The `Connection Handler` provides properties and functionalities, to implement all actions required to establish a connection and reestablish a failed one. Each instantiated `Connection Handler` has a unique identifier that is generated by the server, to distinguish a brand new connection from a connection that was established for recovery purposes. The unique identifier is exchanged between peers by means of a handshake process, implemented in the method *handshake()*, once a connection is successfully created.

The handshake process used to exchange the identifier works as follows. Upon establishment of a new connection, the client sends 0 as its identifier, which allows the server to identify that the connection is new. Then, the connection *type* is set to NEW and a unique identifier is generated and sent back to the client. When the client establishes a connection for recovery purposes, this identifier (*handlerId*) is sent to the server, so that the server can identify that the connection is created to replace a failed one. In this case, the connection type is set to $RECOVERY$. The actions to reconnect and resend the lost data must be implemented in the method *reconnect()*. Moreover, the `Connection Handler` allows the application to define the maximum time (MAX_RECONN_TIME) permitted for recovery process through the method *set_max_reconn_time()*.

3.2.4 Handlers Synchronizer

The most challenging part of the recovery process on the server side is to replace a failed connection with a new one. To do so, the server keeps the list of all open connection handlers with their identifier (i.e., in *handlers*), to enable the delivery of an event (new connection), from a new connection handler to an appropriate connection handler waiting for a new connection. The synchronization between the connection handler waiting for a connection and the connection handler created for recovery purposes is

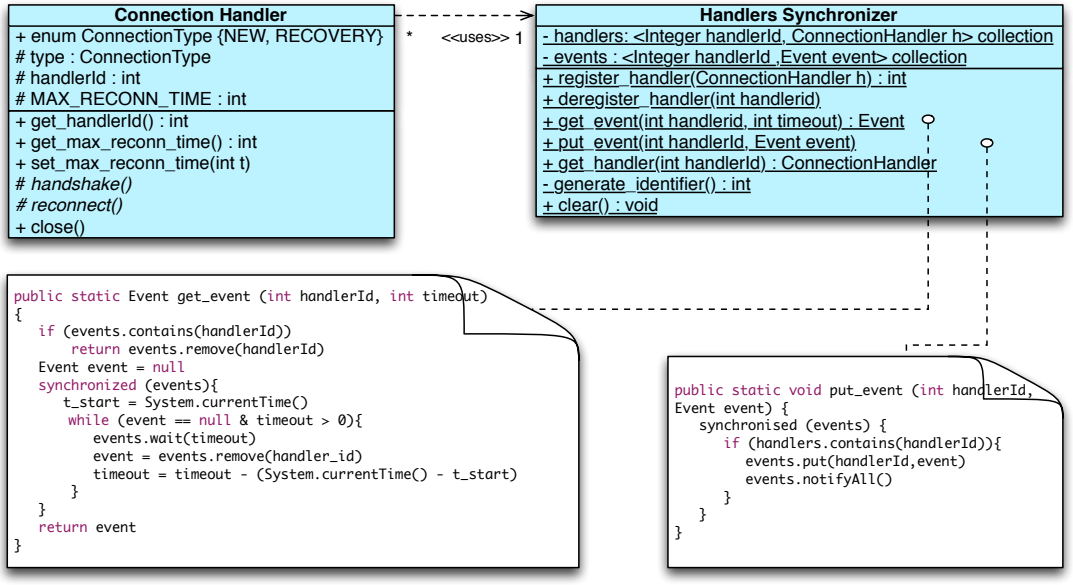

FIGURE 3.4: Connection handlers synchronizer

done through a central component, called `Handlers Synchronizer`. Figure 3.4 presents this component and its partial implementation details as pseudocode.

The `Handlers Synchronizer` provides an interface, allowing the connection handlers to: 1) register and deregister themselves into/from the list of the handlers; 2) put an event for another handler (i.e., used by the connection handler, created for recovery); and 3) wait for an event coming from a new connection handler (used by the connection handler with a failed connection). When a connection handler is being registered, the `Handlers Synchronizer` generates a unique identifier and returns it back to the connection handler.

The registered handlers can wait for an event by calling the method *get_ event()* and passing their identifier and a timeout, which represents the maximum time permitted to wait for a new event. The connection handlers can also leave events for other registered handlers by calling the method *put_ event()* and by passing the event and the identifier of the destination handler. These two operations are synchronized over a collection that includes the events and their destination handler (*events*). Once the method *get_ event()* is invoked, the calling handler blocks until a new event (i.e., a new connection) is inserted into the collection *events* for it, or the timer goes off. Once an event is inserted by the new connection handler, created for recovery, the blocked handler is notified to get the event from the *events*. The `Handlers Synchronizer` only

keeps the last event for each handler, which means that if a client's connection handle attempts several reconnections, only the last one will succeed.

According to the above explanation, the main objective of the `Handlers Synchronizer` in Connection Handler design pattern is to synchronize connection handlers upon connection crashes and reconnection process.

3.2.5 Event

The events that are exchanged between the `Connection Handler`s through the `Handlers Synchronizer` in the server, are of the type `Event`. The Event contains a transport handle and information about the data read in the remote peer.

When a connection is established for recovery purposes, a new transport handle is generated and a new connection handler is initialized. Upon initialization of the connection handler, a handshake request, including the identifier of the connection and the information about the data read, is received. At this point, the connection handler builds an `Event`, out of the new transport handle and the information about the data read, and asks the `Handlers Synchronizer` to give this event to the right connection handler by providing its identifier. The handshake will be completed (i.e., a handshake response is sent back) by the old connection handler (whose connection failed) after receiving the event and replacing the failed connection.

3.3 Connection Handler In Stream-Based Applications

In this section, we use the Connection Handler design pattern to tackle connection crashes for stream-based applications, which use stream of bytes as data to be exchanged between the application's peers (e.g., video streaming). To do so, the `Buffer` and `Reliable Endpoint` must be implemented.

3.3.1 Stream buffer

When 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 the number of bytes available in internal TCP buffers. Therefore, to obtain this information, we need to implement our own layer of buffering over TCP. To avoid explicit acknowledgments, we resort to a circular buffer, which is based on an idea of Zandy and Miller (Zandy and Miller, 2002). We name this buffer as `Stream Buffer`.

To explain how this works, we depict three buffers in Figure 3.5: a sender application's buffer, a sender's TCP send buffer and a receiver's TCP receive buffer. As shown in Figure 3.5, we assume that the receiver got m bytes so far, whereas the sender has a total of n bytes in the buffer, and the connection fails right at this point. Since the contents of both TCP buffers disappear due to crashes, the receiver needs to send the value m (the green part in the figure) to the sender after reconnection, in order to let it know the number of bytes that are successfully received. Then the sender must resend the last $n - m$ bytes (the blue and red parts in the figure) it has in the buffer.

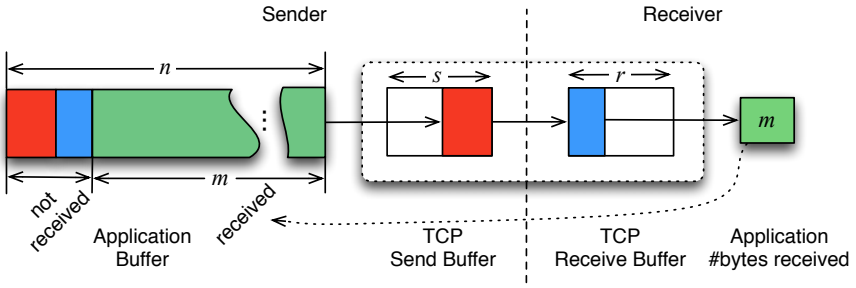

FIGURE 3.5: Sender and receiver buffers

The size of the application buffer can be limited if the application knew the number of bytes read by the receiver, which allows it to delete them from the 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, which 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. Using this mechanism we can simply avoid explicit acknowledgments to the received bytes [2].

[2]Interestingly, we can avoid any modulus 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. Note that apart from these limits, the buffers can have arbitrary sizes, according to the sender plus receiver TCP buffer sizes.

To implement this idea in practice, peers have to exchange the size of their receive buffer, through a handshake procedure, right after establishing the connection and before exchanging any data. Despite being simple and efficient, this buffering scheme has a shortcoming in the Web environment. In fact, this buffering mechanism cannot withstand proxies, which are a frequent element in the Internet. In fact, these intermediate nodes can keep an arbitrary amount of data outside their own buffers, causing the data in transit to exceed the $b = s + r$ bytes available on the `Stream Buffer`.

Figure 3.6 shows a simple sender-receiver scenario, which involves a proxy, and depicts the internal data buffers involved. As we can see, there is extra buffering of data at the proxy. While our main idea stands on having a `Stream Buffer` as large as the TCP send and receive buffers combined, now we have a total of five points in the traffic that can serve as buffers: the sender TCP send buffer, the proxy TCP receive buffer, the proxy internal state, the proxy TCP send buffer, and the receiver TCP receive buffer. The size of the buffers is now $b1 + b2 + b3 + b4 + b5$, much more than the $b1 + b5$ that the `Stream Buffer` was prepared to take. The problem becomes quite serious as we cannot know the sizes of most of these buffers and thus do not know how much data should be kept to be re-sent in case of crashes.

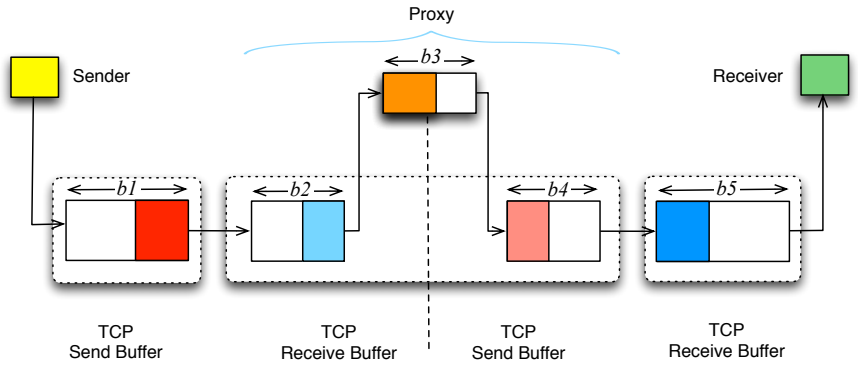

FIGURE 3.6: Buffers in a sender-receiver communication model with proxies

To solve the extra-buffering problem, we use a combination of explicit and implicit acknowledgment. When no proxy exists, client and server can rely on the implicit acknowledgment as explained. In contrast, when a proxy exists, the buffering and acknowledgment scheme must become explicit, because the sender must never allow the amount of data in transit to exceed the size of its `Stream Buffer`. In order to exchange acknowledgment messages when there is a proxy, a control channel is created and used by connection handlers.

To enable explicit acknowledgment, we need to mark the beginning and end of the `Stream Buffer`, to identify whether the buffer is full or if it has enough space for new data to be sent. By having these markers, the `Stream Buffer` of a given peer is considered to be *empty* when the pointer to the end of the buffer points to one place before the beginning of the buffer; and it is considered to be $full$ when the pointers to the beginning and end of the buffer, both, point to the same place. The pointers to the beginning and end of the buffer are respectively updated when a data is stored in the buffer and when acknowledged data is deleted from the buffer.

In the scenarios with proxy, whenever a `Stream Buffer` is becoming full, the peer should acknowledge reception of data, to allow the sender to release some space in its buffer to be able to send the next data without interruption. For example, consider that a server is sending a large file, with a size greater than the buffer size, to the client. If the server does not receive an early acknowledgment from the client, its buffer will become full and it needs to wait for an acknowledgment to release some space and send the rest of the file. To enable early acknowledgments, once a peer receives a number of bytes equal or greater than half the size of the peer's `Stream Buffer`, an acknowledgment should be sent. This allows the peer to clean its buffer, thus allowing it to proceed. Implementation of this idea is possible, because the peers can exchange the size of their send buffer as well as their receive buffer, allowing each other to simply calculate the size of the remote `Stream Buffer`.

Figure 3.7 presents more details of the `Stream Buffer` from the implementation perspective. Each `Stream Buffer` owns an array of bytes ($buffer$), pointers to *start* and *end* of the buffer, and a boolean attribute, named $write\_constraints$, which indicates if the buffer needs to keep the pointer to the end of the buffer (i.e., used in the communication scenarios with proxy). Methods *put()* and *get()* are respectively used to save data (i.e., array of bytes) in the buffer and return data that is lately sent. Methods *has_ space()* and *release_ space()* are used in the scenarios with proxy, to check whether the buffer has enough space to write over old data (i.e., acknowledged data).

3.3.2 Reliable Transporter

By considering the aforementioned challenge, regarding the proxies and the given solution, the `Reliable Endpoint` must be extended and implemented to support the

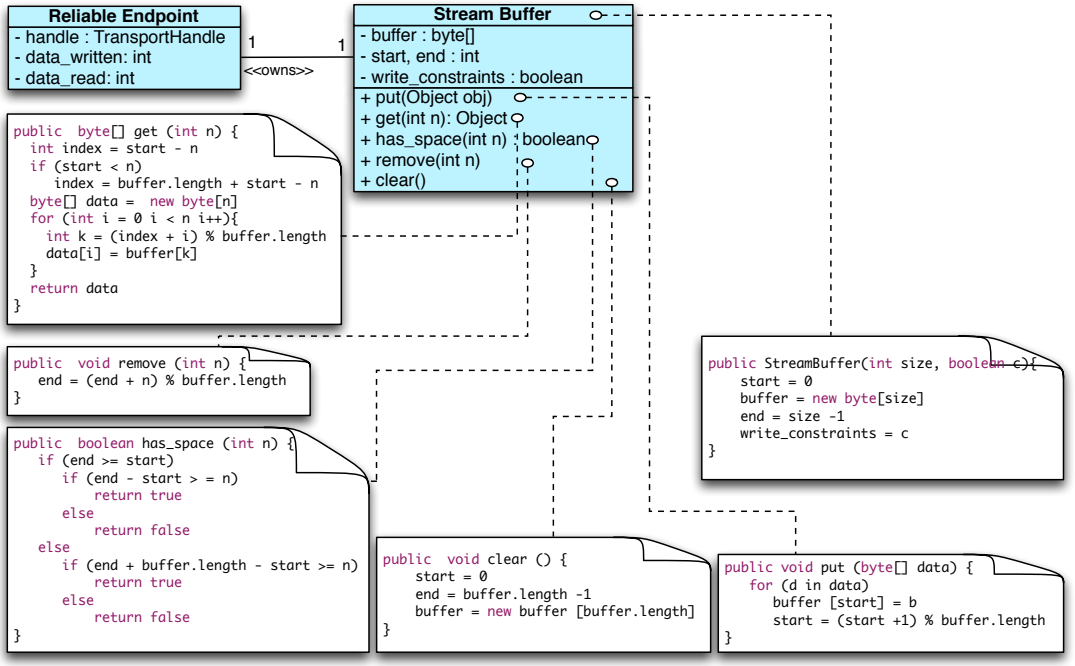

FIGURE 3.7: Stream Buffer and its implementation details

implementation of the explicit acknowledgment through a control connection. This extended version of the reliable endpoint for stream-based applications is called `Reliable Transporter`.

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

Moreover, each `Reliable Transporter` owns one `Control Connection` when the communication involves proxies. Each `Control Connection` is shared between all connections created from the same client. In the scenarios with proxies, the `Reliable`

`Transporter` also needs to keep the size of the remote `Stream Buffer` (*remoteBufferSize*) and the number of bytes read so far after the last acknowledgment (*numOfBytesReadAfterLastAck*). Figure 3.8 presents the `Reliable Transporter`, its attributes and connected components.

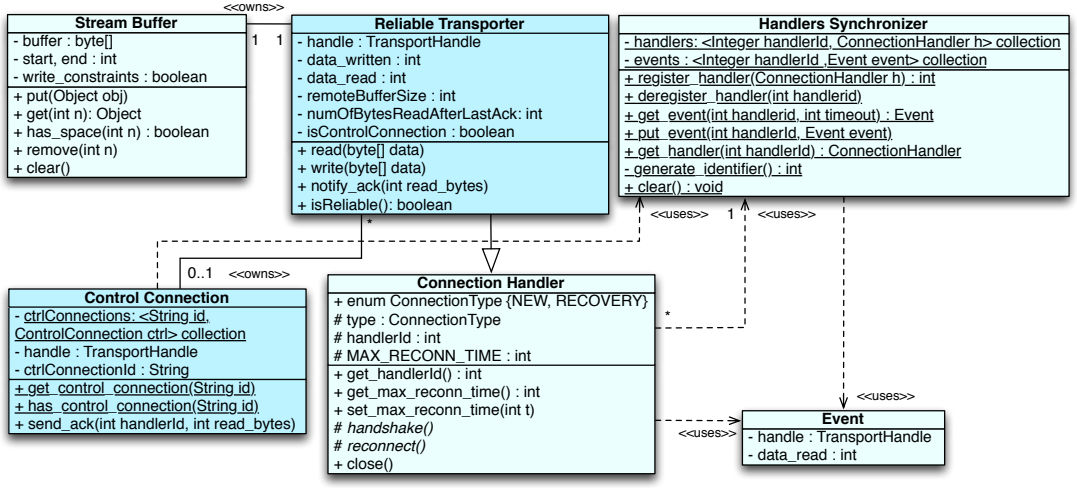

FIGURE 3.8: Reliable Transporter and connected components

To accomplish the recovery process transparently from the application layer, we need to insert the `Reliable Transporter`, between the transport layer (`Transport Handle`) and the application layer (`Service Handler`). Thus, this component besides owning a `Transport Handle` to exchange application data, provides *write()* and *read()* operations to the application to do the following actions underneath the application: 1) store the data sent by the application into the stream buffer; 2) count the number of bytes written and read; 3) intercept the read and write operations for detecting a connection crash; and finally 4) reconnect and retransmit the lost data when a connection crash is detected. In addition to these actions, the handshake process is also performed transparently from the application layer, once a `Reliable Transporter` is initialized.

The handshake is necessary to exchange the identifier of the connection and the size of the TCP receive buffer, which is necessary for calculating the size of the local `Stream Buffer`. Moreover, to support the scenarios including proxies, the handshake is used to carry the information needed to detect the existence of proxy and the size of the TCP send buffer, which is required to calculate the size of the remote `Stream Buffer`. Furthermore, the handshake process should also address the following two important issues. First, we must not expect endpoints to adhere to a specific reliable communication mechanism, which means that a solution for reliable communication should

ensure that inter-operation with legacy software is possible. Thus, our solution for reliable communication should also consider the presence of legacy endpoints. This is especially important in the Web environment, which comprises a very large base of legacy software that must not be prevented from communicating with reliable peers. Second, considering the case where the proxy performs a security function, in particular content-based filtering, as for example in HTTP-based applications; it will very likely filter out non-HTTP messages. This means that the critical handshake step may fail in the presence of content-based filtering proxies, if they do not follow the format of the application layer protocol being used(e.g., HTTP request and responses).

Given the above explanation, a handshake process, which is performed right after establishing a connection and before starting the exchange of application data, aims to identify: 1) if the peer is legacy software that does not support our reliability mechanism; 2) if the connection is brand new, or created for recovery purposes; 3) if there is any proxy in the middle of the connection between the endpoints; 4) the size of the local stream buffer, when both peers implement the reliability solution and the connection is new; and 5) the size of the remote stream buffer, when there is a proxy.

The handshake messages follow a predefined configurable format, which is shown in Figure 3.9. It includes a header line that can be configured differently on the client and server sides, depending on the application layer protocol. Then we can have several lines that carry the necessary information for the handshake. Each line is separated from the other lines using a separator (e.g., $\backslash r \backslash n$).

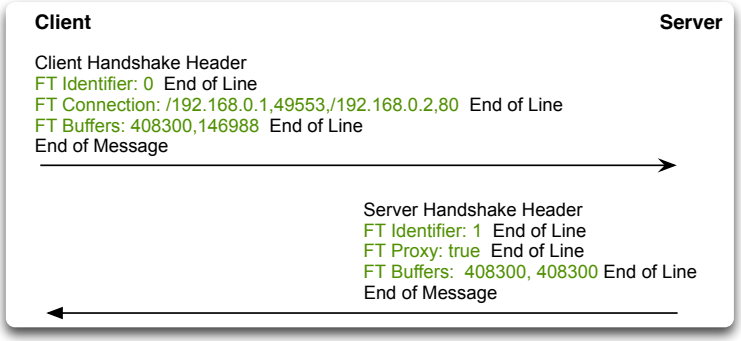

FIGURE 3.9: Handshake message format in Connection Handler

Considering the client handshake message, the *FT Identifier* header carries the identifier of the connection, which is used on the server side to identify whether the connection is new or created for recovery. For setting up a new connection, the client sets the

identifier to 0, and the server generates a new immutable identifier for the connection in the response. Once a reconnection occurs, the client sends this identifier and the number of bytes it received up to the connection crash. 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 crashes. The *FT Connection* header carries the network address of the client and server, which are used on the server side to identify if there is any proxy. The *FT Buffers* carries the size of the TCP send buffer, and the size of the TCP receive buffer, which are used in the server to calculate the size of its `Stream Buffer` and, if necessary, the size of client's `Stream Buffer`.

Regarding the server's handshake message, the corresponding headers are present, including the identifier of the connection, which, in this case, is 1. The *FT Proxy* informs the client whether a proxy was detected by the server or not. The server detects the presence of a proxy if the address sent in the *FT Connection* header is different from the address of the TCP connection it owns. If the existence of a proxy is identified, the client creates a new `Control Connection` to the server, for exchanging the acknowledgment messages. Each client uses just one control connection for all connections it may have to the same server, thus a control connection is identified by the server's address in the client and by the client's address in the server (*CtrlConnectionId*). A handshake message, including *FT Control* header, is sent by the client, which allows the server to distinguish a data connection from a control connection (*isControlConnection* is set when a connection is created as a control connection).

During the communication, the `Control Connection` is used (only if there is a proxy) by the `Reliable Transporter`, to send acknowledgment messages. The `Control Connection` keeps the references to the existing control connections (in *ctrlConnections*) and provides an interface to the `Reliable Transporter` to check the existence of a control connection to a specific peer (*has_control_connection()*) and access it (*get_control_connection*). It also provides an interface for sending acknowledgment messages (*send_ack()*). A `Control Connection`, which might be shared among several handlers, checks for the arrival of acknowledgment messages and delivers them to the appropriate handler (i.e., the `Control Connection` gets its reference from the `Handlers Synchronizer`) through the method *notify_ack()*, provided by the `Reliable Transporter`.

The acknowledgment messages have the same format as the handshake messages, but including an FT ACK header, carrying the number of bytes read so far. The `Reliable`

`Transporter`s need to count the number of the bytes read after the last acknowledgment message sent (*numOfBytesReadAfterLastAck*) and compare it with the size of the remote buffer (*remoteBufferSize*), to send early acknowledgment messages before the remote buffer becomes full.

Furthermore, it is worth noting that, the above handshaking mechanism, in addition to its main objectives, resolves two other aforementioned issues, regarding the inter-operation of non-reliable and reliable peers and content-based proxies. First, both non-reliable and reliable clients and servers are able to inter-operate. Using the above handshaking scheme, a legacy client will simply not receive any handshake message from the server because in the reliable server the `Reliable Transporter` simply detects that the client is legacy, so it turns to behave like a simple transport handle and avoid sending any handshake message. In contrast a legacy server after receiving a handshake message from a reliable client, either ignores it or replies, which the client can use to detect that the server is legacy. Hence, all the combinations of legacy/reliable client and server work. The application layer can always check whether the communication is reliable or not by calling the method *isReliable()*. Second, the configurable format we defined in our design for handshake messages allows the developers to adjust it according to their application layer protocol. As an example, Figure 3.10 shows handshake messages configured for the HTTP protocol.

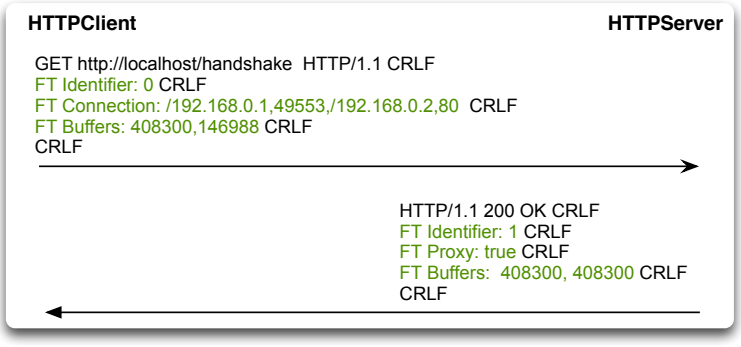

FIGURE 3.10: Handshake message configured for HTTP protocol

3.3.3 Reliable Stream-Based Application

Given the basic model of a connection-based application, presented in Figure 3.1, the design of a reliable stream-based application using the Connection Handler design pattern looks like Figure 3.11. As shown in this design, all the actions required to be taken

to tolerate connection crashes, in the presence or absence of proxies, are done in the session layer transparently from the service handler.

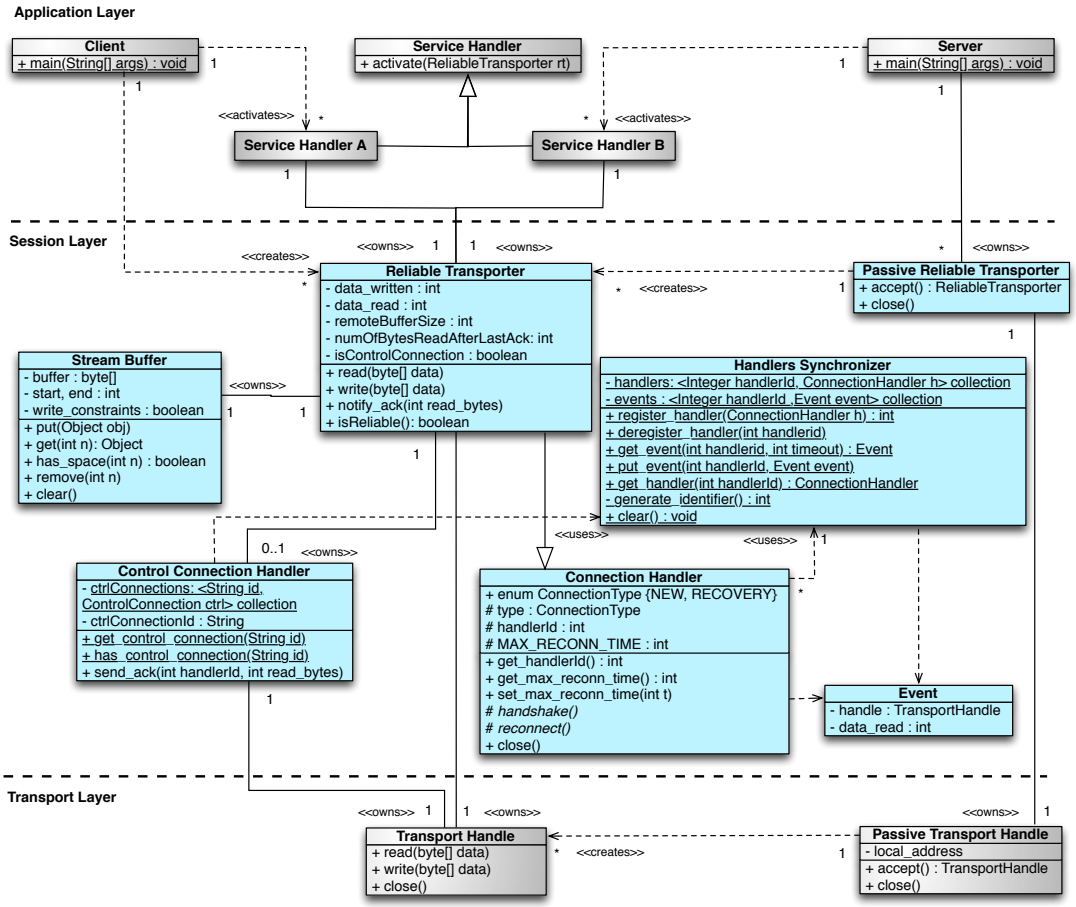

FIGURE 3.11: Reliable Transporter design pattern

The application layer includes a `Service Handler` and interacts with its connected peer through the session layer, which includes the `Reliable Transporter` implementing `Connection Handler`. The most common applications run directly over the transport layer, by means of a `Transport Handle` (e.g., TCP socket). In our case, the interaction between the `Service Handler` and the `Transport Handle` is accomplished through a session layer offering transparent recovery of byte streams from connection crashes. Here we explain the collaboration between the components of the above design in a failure free scenario, that is presented in Figure 3.12.

To initialize connections, the server creates one (or more, depending on the number of ports defined and assigned to the application server) `Passive Reliable Transporter` and binds it to the local network address (IP address and port number). Then the server

waits for a new connection, by invoking the method *accept()* of this passive handle. On the other side of the communication, the client initializes a `Reliable Transporter` by giving the network address of the server in order to establish a new connection. This will internally create a `Transport Handle`.

Upon reception and acceptance of a connection request in the server, a `Reliable Transporter` is generated. Right at this point, a handshake procedure is taken place to complete the initialization of the `Connection Handler`. The client's `Reliable Transporter` sends a handshake request including the identifier of the connection (zero in this scenario), the local and remote address of the connection, and the size of its TCP's send and receive buffers.

The server's `Reliable Transporter`, identifies that the connection is new (because the identifier is zero), and registers itself into the `Handlers Synchronizer` through the method *register_handler()*, which returns a unique identifier. It can also identify the existence of a proxy using the given information about the local and remote addresses and comparing them with its own information about the connection. A handshake reply is sent back to the client, which includes the unique identifier of the handler, the size of the buffers on the server side, and information about the existence of a proxy. At this point, both, client and server, can initialize their `Stream Buffer` with the appropriate configuration, depending on the information exchanged between them.

When no proxy exists, the peers initialize and activate service handlers, by passing the previously created `Reliable Transporter` (rt in the client and rt_1 in the server). This means that the client and server's `Service Handler` can start writing and reading data. After a successful write operation, the `Reliable Transporter`s put the data into the `Stream Buffer` and update the value of *written_data*. After a successful read operation they update the value of *data_read* (refer to part (a) of Figure 3.12).

In contrast, when there is some proxy, the `Reliable Transporter`s require a control connection to exchange acknowledgment messages in both sides (Refer to part (b) of Figure 3.12.). Since the control connection is shared between several connections created by the same client, client and server check the `Control Connection` for an existing connection, by specifying an identifier that is equal to their peer's address. If a connection already exists, they simply get it from the list and use it, otherwise the client must create a new one. When a control connection is successfully created, the client sends a handshake request including the *FT Control* header with the local address of

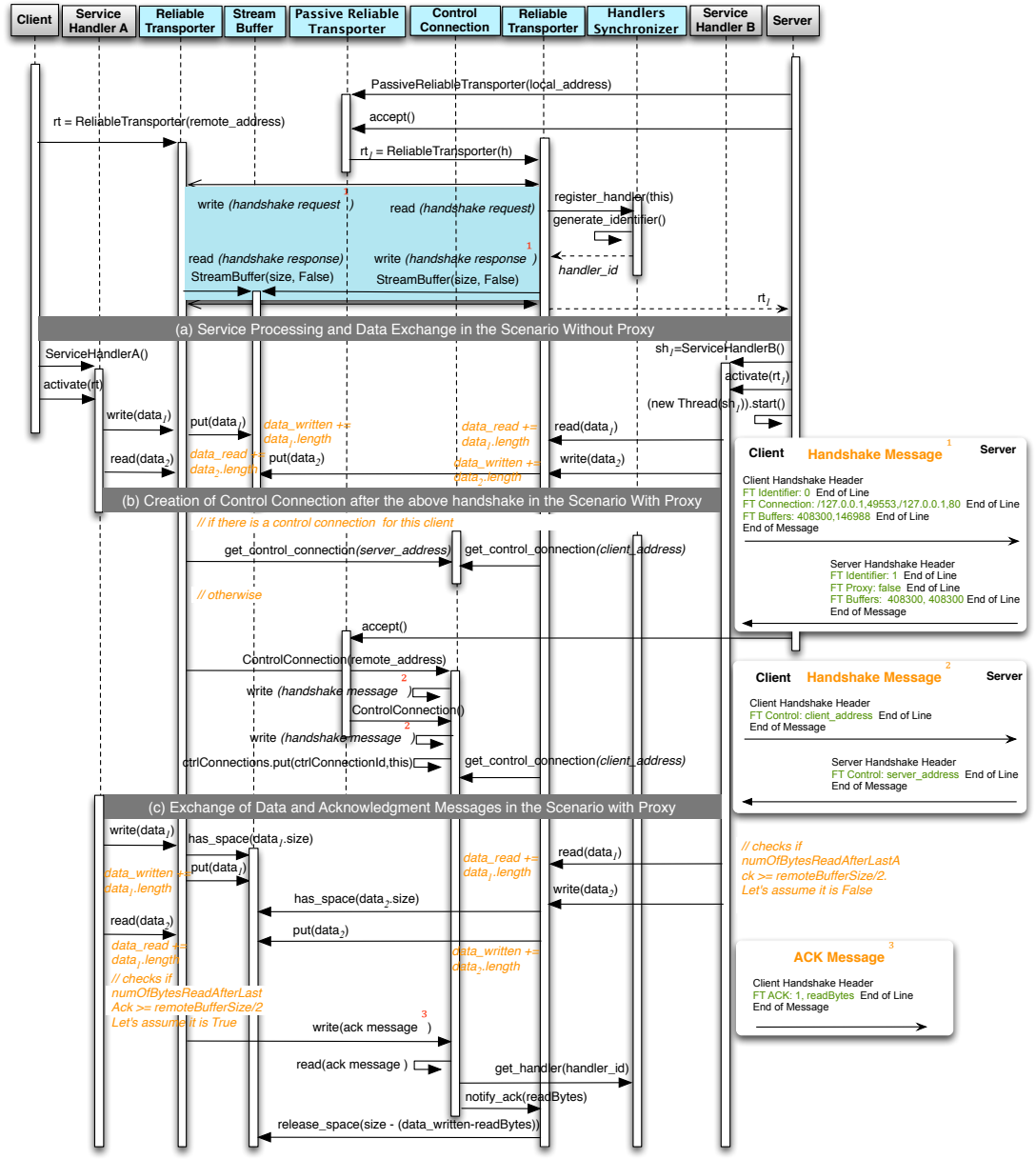

FIGURE 3.12: Collaboration between the components of the Reliable Transporter design pattern

the client, which will be used by the server as the identifier of the control connection. The server sends a handshake reply back to the client including the *FT Control* header with the IP address used by the server, which will be used as the identifier of the control connection on the client side. Both client and server store the reference of the control connection in a list (*ctrlConnections*), to be used with other `Reliable Transporters` if necessary.

The exchange of data is quite different when there is a proxy. The `Reliable Transporter` checks if there is enough space in the `Stream Buffer` before writing the data, and checks if the number of bytes read, after the last acknowledgment message, exceeds the half of the remote buffer. If so, it sends an acknowledgment through the control connection. Figure 3.12, part (c), shows a scenario where an acknowledgment is sent from the client. As shown in the figure, this message carries the identifier of the connection handler and the number of bytes read so far. The `Control Connection` delivers the read message to the appropriate `Reliable Transporter`, which is accessed by means of the `Handlers Synchronizer`, through the method *notify_ ack()*. This lets the `Reliable Transporter` release some space from the `Stream Buffer`.

3.4 Concurrent Connection Handling

To improve the design of Reliable Transporter, we now leverage on existing work and design principles for concurrent connections, to propose a highly scalable Reliable Transporter Design Pattern. We refine the basic model of a connection-based application, by adding the combination of the Acceptor-Connector and Leader-Followers design patterns explained in the previous chapter (Chapter 2), to enable efficient handling of a large number of concurrent connections. We call this design as **Multi-Threaded Acceptor-Connector**. We then modify the Reliable Transporter Design Pattern, by adding the Multi-Threaded Acceptor-Connector.

3.4.1 Supporting High Concurrency

It is quite challenging to implement large-scale concurrent applications. Considering a multi-tier architecture, the applications should allow front-end and back-end servers to perform their functions in an independent and efficient manner. The Acceptor-Connector (Schmidt, 1996) design pattern decouples processes and enables the creation of multiple concurrent connections, by separating event dispatcher from connection setup and service handling, but it is single-threaded, and is, therefore, unfit for modern servers. 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 (Schmidt et al., 2000), which combines the event dispatcher with a set of threads. Indeed, the combination of the `Dispatcher` and `Thread Set` allows applications to handle a large number of concurrent connections. Thus, we have refined the basic design of a connection-based communication, by adding the combination of Acceptor-Connector and Leader-Followers design patterns. The result of this combination is presented in Figure 3.13. 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 threads can execute concurrently.

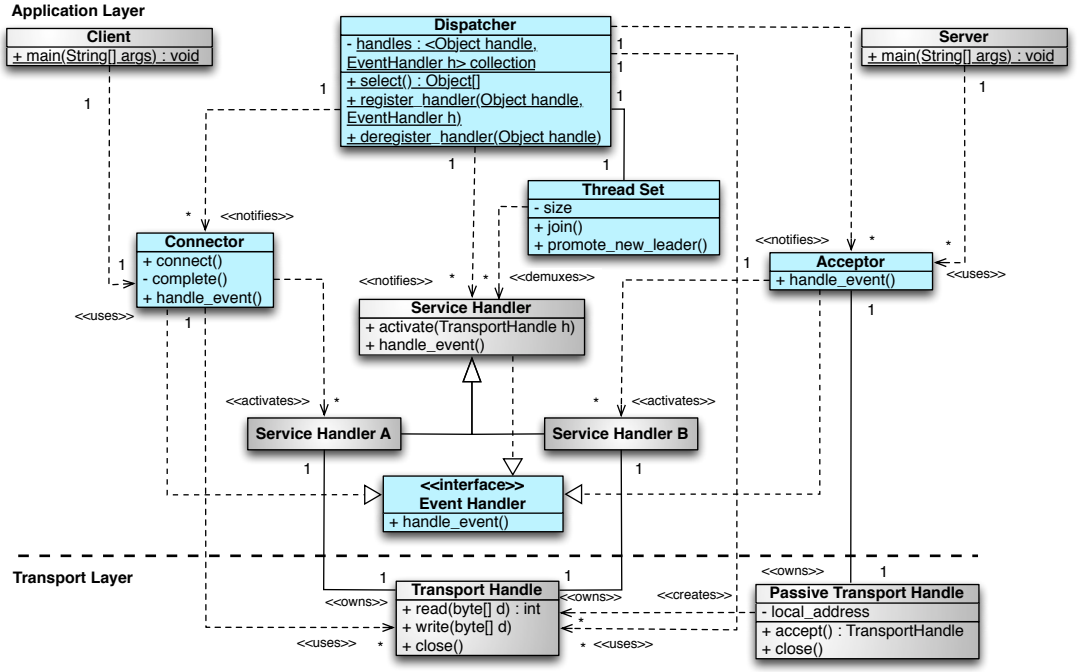

FIGURE 3.13: Multi-Threaded Acceptor-Connector design pattern

In other words, the threads are not permanently assigned to the `Service Handlers`, but a thread from the `Thread Set` is assigned to a `Service Handler` only when an event occurs on its `Transport Handle`, and released when the event is processed. In fact, the association of threads to services is unbounded, which means that any thread can process any event that occurs on any `Transport Handle`. This solves the problem with the unbounded growth of threads when the number of concurrent connections increases. The remaining problem is to adjust the size of the `Thread Set`, depending

on the available resources on the host the application is running on. Figure 3.14 shows the interactions between the components of the Multi-Threaded Acceptor-Connector design pattern. To simplify the description of this interaction, we divide it into three phases: connection initialization, service initialization, and service processing.

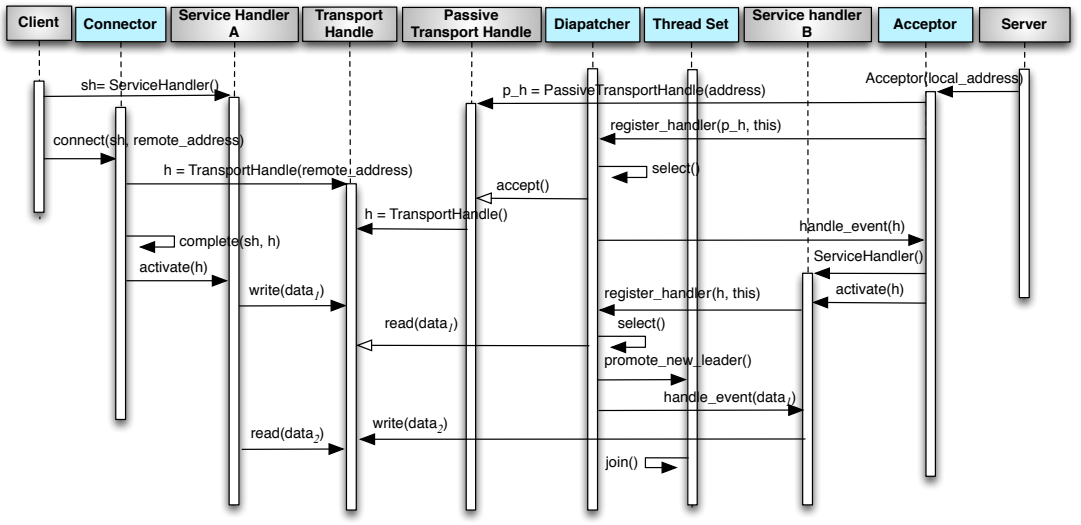

FIGURE 3.14: Collaboration between the components of the Multi-Threaded Acceptor-Connector design pattern

Phase 1: Initialization of a Connection

The server owns one `Dispatcher` and one `Thread Set`. To receive connection requests, the server can create one (or more) `Acceptor`(s). When an `Acceptor` is initialized, a passive-mode transport handle (`Passive Transport Handle`) is created and bound to a network address (p_h). The `Acceptor` registers this handle with an attachment, which is a reference to itself, in the `Dispatcher` using the method *register_ handler()*. Then, the `Dispatcher` continuously checks for arrivals of new events on the previously registered passive transport handles, which are selected by calling the method *select()*. It does a non-blocking invocation of the *accept()* method on the passive handles (non-blocking invocations are shown in Figure 3.14, Figure 3.16, and Figure 3.17 by arrows with white head). Upon arrival of a new connection, the `Dispatcher` dispatches it to the appropriate acceptor (attached to the handles), through the method *handle_ event()*.

On the other end of communication, a client starts establishing the connection by passing an instance of `Service Handler` and a remote address to the `Connector` through

the method *connect()*. This method creates a `Transport Handle` and blocks the thread of control, until the connection completes synchronously. At this point, a new handle (identified by h in Figure 3.14) exists on both sides.

Phase 2: Initialization of Service

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` in the same way. The `Service Handler` then registers the given handle and a reference to itself in the `Dispatcher`. This allows the `Dispatcher` to deliver the data received on the registered handle to the appropriate `Service Handler`.

Phase 3: Service Processing

Finally, 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`[3]. On the other hand, the server's `Service Handler` waits for the `Dispatcher` to call it back for new data. To receive data, the `Dispatcher` invokes the *read()* operation of the selected `Transport Handle` in a non-blocking manner. Then, the data is given to the appropriate `Service Handler`, and the leader thread is assigned to handle (process) the event (data). 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`.

3.4.2 Scalable Design of Reliable Stream-Based Applications

Here we refine the Reliable Transporter design pattern by adding the Multi-Threaded Acceptor-Connector design pattern in the application layer. With this refinement, we aim to achieve a solution that, besides correctly recovering from connection crashes, has the following features:

[3]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.

- Efficiently handles a large number of connections;
- Decouples the failure handling process from the service handling;
- Decouples the failure handling process of different connections from each other;
- Enables the possibility of adding new types of services, new service implementations, and new communication protocols, without affecting the existing connection handling phase.

To achieve these objectives, we resort to the Multi-threaded Acceptor-Connection. The result is presented in Figure 3.15. In general, each endpoint owns one `Dispatcher` to register handles and attach appropriate handlers for call back if necessary. It also owns one `Thread Set` to control the number of threads created for handling the connections. Here we consider that the client does not use these two components and handles the connection using a single thread. We present the collaboration between the components of the above design in two scenarios: failure free scenario (Figure 3.16) and with a failure (Figure 3.17).

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 Reliable Transporter` is created and bound to a network address. Then the `Acceptor` registers the passive handle and its reference in the `Dispatcher`.

On the other side of the communication, as shown before, the client uses the `Connector` to establish a new connection to the server. Upon reception and acceptance of the connection request in the server, a `Reliable Transporter` is created on both server and client sides (rt in the client and rt_1 in the server). A handshake procedure is followed to complete the connection initialization. At this point, the `Dispatcher` gives the `Reliable Transporter` (rt_1) to the `Acceptor` attached to the passive handle, to which the connection request arrived.

After the initialization of a `Reliable Transporter`, the `Connector` completes the process using the method `complete()`, which activates the `Service Handler`, by passing the `Reliable Transporter` through the method `activate()`. On the other side, the `Acceptor` creates a new instance of a `Service Handler` (sh_1) and activates it. Then, the `Service Handler` registers the `Reliable Transporter` and its reference in the `Dispatcher`.

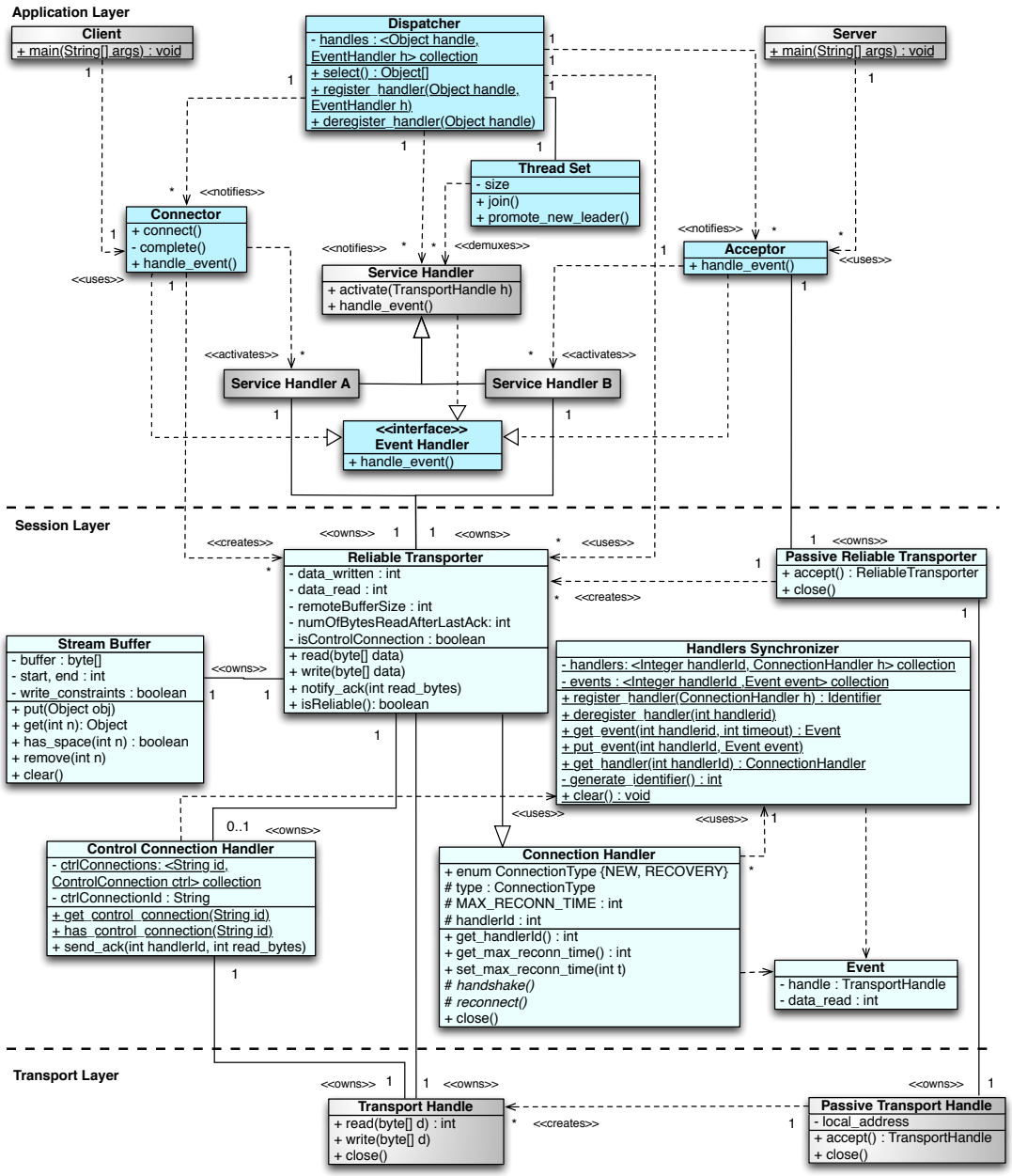

FIGURE 3.15: Scalable design of a reliable stream-based application

Once the connection is established and the `Reliable Transporter`s and `Service Handler`s are initialized successfully, the service-processing phase begins. The `Service Handler`s could use the `write()` and `read()` methods to send and receive data through the `Reliable Transporter`. All the actions related to the saving of data sent in the `Stream Buffer` and updating the number of bytes sent and received are transparently done in the `Reliable Transporter`. In the case of the server, a `read()` operation works

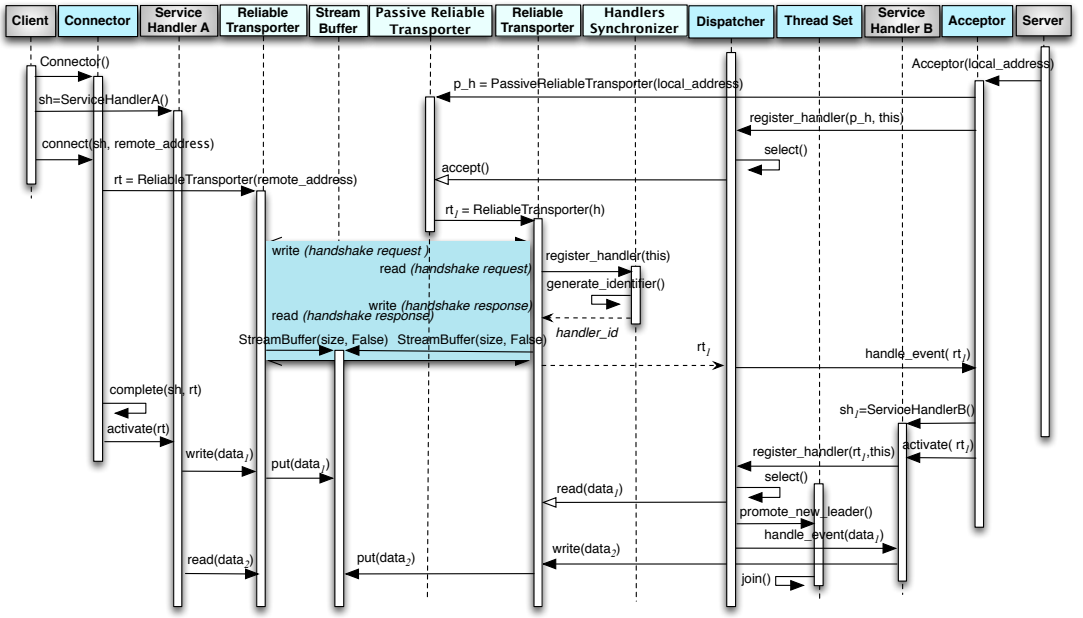

FIGURE 3.16: Connection establishment and service initialization with Multi-Threaded Acceptor-Connector design pattern

differently. The `Dispatcher` reads data in a non-blocking way (this may involve multiple read attempts, until the entire data is read)[4], and delivers it to the attached `Service Handler`. When the `Dispatcher` delivers a data to the `Service Handler` through the method `handle_event()`, the leader thread from the `Thread Set` is assigned to process the data, and a new thread is promoted to become the leader.

Figure 3.17 presents failure handling details. Once a `Reliable Transporter` fails completing a read or write operation, it transparently tries to reconnect. The reconnection is accomplished differently in the client and server's `Reliable Transporter`. As shown in the figure, neither the client's `Service Handler`, nor the server's are involved on the recovery procedure, to ensure the separation between service and failure handling.

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

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

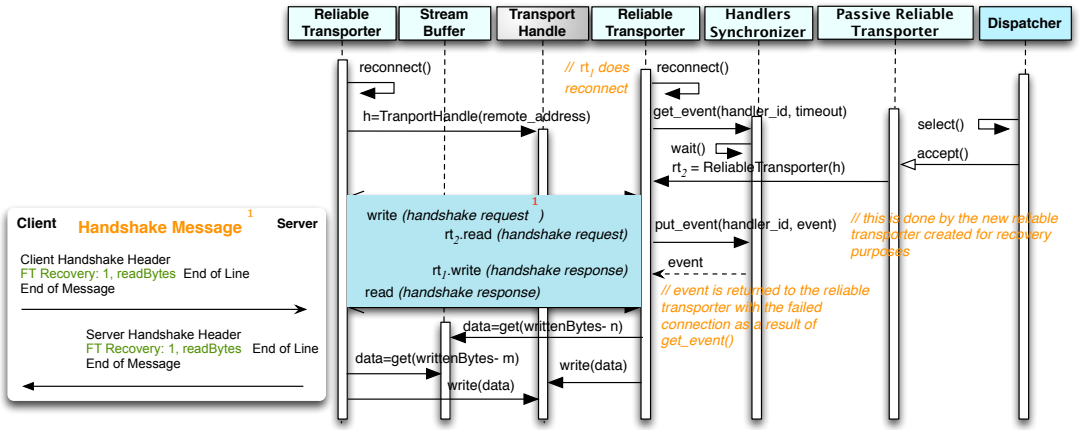

FIGURE 3.17: Collaboration between the components in the presence of connection crashes

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

3.5 Conclusion

In this chapter, we addressed one important limitation of TCP in building reliable communication. Despite being very popular, TCP does not handle connection crashes and does not provide any information regarding such connections, to help applications in the recovery process. In this chapter, we proposed a solution based on buffering and reconnection mechanisms. We built the Connection Handler design pattern allowing the developers to easily implement this solution using any programming language in any platform. Then, based on the Connection Handler design pattern, we built the Reliable Transporter design pattern for stream-based applications, requiring reliable

communication for transmission of byte streams. At the heart of the Reliable Transporter design pattern, there is a circular buffer that eliminates the need for explicit acknowledgments. We also addressed several challenges regarding legacy software and proxies in the Reliable Transporter.

Finally, we resorted to the Acceptor-Connector and Leader-Followers design patterns, to build a highly scalable design to the applications. We believe that the multi-threaded design proposed in this chapter provides the following benefits to the applications and developers: 1) allowing the developers to correctly handle connection crashes without losing any data; 2) decoupling the connection handling from service handling; 3) decoupling the failure-related processing from the connection and service processing; 4) efficiently recovering the state of a failed connection by applying a circular buffer and sharing the control connection between several data connections; 5) allowing interoperation between reliable and legacy peers; 6) efficiently supporting multi-threading, by applying the Leader-Follower design pattern; 7) providing flexible behavior by allowing developers to configure the reconnection procedure (maximum time for reconnection is configurable); 8) finally, allowing to configure the format of control messages (e.g handshake and acknowledgment messages) according to the application layer protocol to remove the effect of content-based filtering.

Chapter 4

Reliable Message-Based Solutions for One-way Interactions

Reliable messaging lies at the heart of many distributed systems, and is often found in two basic forms: one-way and request-response. One-way messaging, which is the simplest form of communication, is extremely useful in event-based systems, where the information flows in one single direction and does not require any response. Email and chat applications (Herring et al., 2013), multi-player games (Veljkovic et al., 2013), social networks (Canning, 2012), group communication systems (Birman, 2012; Chockler et al., 2001), and complex event processing systems (Buchmann and Koldehofe, 2009; Gharbi et al., 2013) are some of the examples that conceptually use this messaging paradigm.

Over the last few decades, the stream-based Transmission Control Protocol (TCP) (Postel, 1981) has been the most common option for providing reliable communication over the Internet, even for message-based applications. TCP provides a full-duplex communication channel, well suited for many applications, but not all communication patterns fit into this spectrum. Despite being very popular, TCP has several limitations for reliable message-based communication, especially for those that require a one-way interaction pattern.

TCP is a stream-based protocol, which means that it has no means to place application layer data into an envelope, in order to be sent and received as a "Message". Also, TCP's reliability guarantees are unfit for one-way messaging because they do not

provide any means for applications to know if application-layer messages are received or correctly processed. Thus, many applications (e.g., online multi-player games) use request-response protocols, despite being inherently one-way, to ensure that their messages reach the destination. For instance, in many web-based applications, the web client does not need a reply (although it expects the server to process the request), but still communicates using a request-response protocol, such as he HyperText Transfer Protocol (HTTP) (Fielding et al., 2009). Using a request-response paradigm in one-way applications, to implement confirmations over TCP, besides changing the interaction pattern from one-way to request-response, undoubtedly slows down performance, due to the waiting time needed for each response. Finally, as explained in Chapter 3, TCP does not provide any means for recovering from connection crashes and peers cannot determine which messages did or did not reach the destination, should the TCP connection fail.

In this chapter, we contribute to overcome these limitations by proposing three design solutions, namely Messenger, Trackable Messenger and Reliable Messenger design patterns, facilitating the implementation of reliable message-based communication for developers. The Messenger design pattern represents a general design of a message-based application, in which the distributed peers use enveloped messages for communication in a TCP-like full-duplex communication. This design includes a session-based component that implements the actions required to build (when being sent) and rebuild (when being received) a message, transparently and independently of the application layer protocol being used.

The Trackable Messenger design pattern adds feedback information of messages to the Messenger. This is enabled by a multi-level acknowledgment scheme, which informs the sender on the precise status of each message: *i*) if it left the sender Messenger already, *ii*) if it reached the peer Messenger, *iii*) or if it was already processed by the peer application. Finally, the Reliable Messenger extends the Trackable Messenger's functionalities, by applying the Connection Handler design pattern, presented in Chapter 3, to provide the ability for recovering from connection crashes. Thus, besides providing a full-duplex message-oriented communication (implemented in the Messenger), and enabling message tracking along messages' life cycle (implemented in the Trackable Messenger), the Reliable Messenger tolerates connection crashes, thus enabling reliable communication over the faulty Internet (offered by the Connection Handler design pattern).

Our patterns are not tied to any specific operating system or programming language. They present different solutions with increasing levels of functionality (and complexity), to problems that emerge frequently in distributed systems. We believe that using one of these design patterns is a light-weight middle ground between custom-made solutions and more complex middleware such as Java Message Service implementations or Microsoft's Message Queueing (Horrell, 1999; Richards et al., 2009), which typically use a persistent broker between the peers.

The remainder of this chapter is organized as follows. Section 4.1 presents an overall overview of the design process. Section 4.2 presents the Messenger design pattern, demonstrating a general session-based design for a message-based application. Section 4.3 presents the design of a trackable messaging service, named Trackable Messenger, which allows the senders to know about the status of their messages. Then, Section 4.4 presents the Reliable Messenger, which, besides the functionalities of the Trackable Messenger, is able to transparently recover from connection crashes. Finally, Section 4.5 concludes this chapter.

4.1 Overview of the Design Solutions

In this section, we present an external overall view our design solutions. We assume the existence of a service handler, running in the application layer, that implements the business logic of the application. We also assume the existence a transport handle, implemented in the transport layer, used by the service handler to exchange messages with its peer. We then gradually overcome the limitations or gaps found in the implementation of reliable one-way interactions, discussed in the previous section, and propose three session-based solutions. These solutions are design patterns named Messenger, Trackable Messenger, and Reliable Messenger, which are placed between the application and transport layers.

Figure 4.1 shows the organization of the three solutions proposed, and also represents the overall design process, which was carried out in an incremental manner (from the Messenger to the Reliable Messenger). The Messenger, presented in Figure 4.1 (*a*), aims to build a message-based session layer on top of stream-based transport layer and provide a simple interface to the applications enabling them to easily and independently of application layer protocol build, send and receive messages. The Trackable

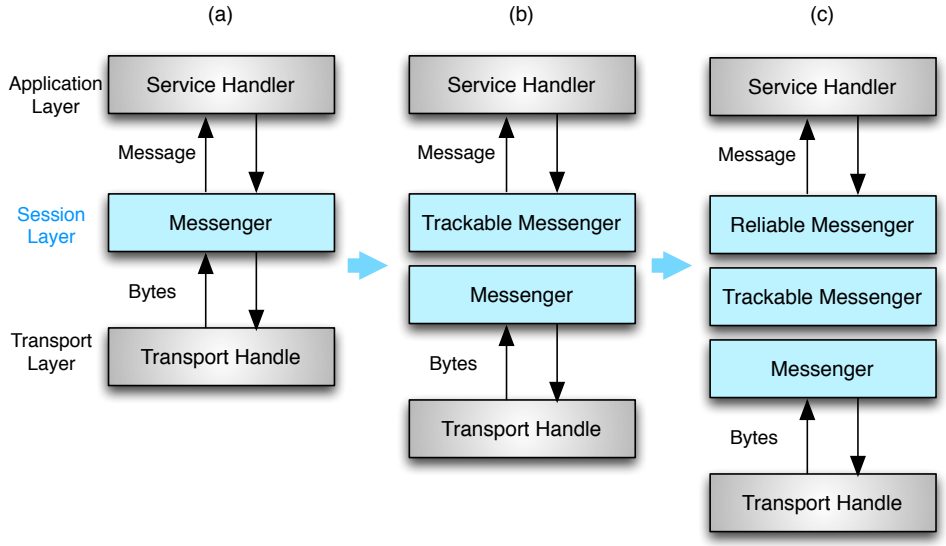

FIGURE 4.1: External view of the design process of the solution

Messenger, also depicted in Figure 4.1 (*b*), uses the functionalities of the Messenger for sending and receiving messages and aims to provide reliability support to applications using the one-way messaging pattern. It allows a sender to track its messages by exchanging multi-level acknowledgments at the send, receive, and processing points. Finally, the Reliable Messenger (Figure 4.1 (*c*)) builds on the functionality provided by the Trackable Messenger to enable recovery from connection crashes. The Figure emphasizes the decoupling of the patterns, to the point of allowing these three configurations to be used separately in real applications. Such configuration can better fit different requirements. The next sections present and discuss each of these designs in detail.

4.2 Messenger Design Pattern

In message-based communication, the data exchanged between the application peers is a discrete message that can be sent in several chunks. A message, which usually includes a header and a body, is placed into an envelope in a predefined format when sent, and it should be exactly the same when it is read (Hohpe and Woolf, 2003).

Since the most popular transport protocols are stream-based (e.g., TCP), it is up to the application layer to determine whether a message has been completely received or

not. There are various message-based protocols (e.g., HTTP, SMTP) that define their own messaging format, which besides achieving the functional goal of the protocol, restricts applications to build messages that respect the determined format. This format agreement allows the peers to rebuild the messages, after being received, from the stream of bytes.

In this section, we present a design solution, named Messenger design pattern, for synchronous message-based applications that is independently of the application layer protocol, programming language (as with design patterns in general, it targets object-oriented programming languages), and underlying platform (e.g., operating system). This solution helps simplifying the development process of applications with message-based interactions.

4.2.1 Components of the Messenger Design Pattern

Figure 4.2 presents the Messenger design pattern and its partial implementation details as pseudocode. This design is intended to be used by any synchronous message-based application and includes three layers: application, session and transport. The application layer includes `Service Handler`s to implement the business logic of the application; and `Client` and `Server` to initialize and activate the application peers. The transport layer includes a `Transport Handle` for transmitting byte streams over the network and a `Passive Transport Handle` for receiving connections on the server.

The session layer includes the following components. A `Message` is simply a serializable data structure that encapsulates application layer data and any associated meta-data. A message can be interpreted as data, as the description of a command to be invoked, or as the description of an event that occurred (e.g., a mouse click). The `Message` actually includes two parts, a header to carry meta-data and a body to carry data. The header of a message contains meta-data about the message (e.g., identifier, size) and any information required for communication, many times depending on the communication protocol used between the peers (in the application or session layer). This information is stored into a structure comprised of various fields (or attributes) and their corresponding values. While the header can be used by the application and session layer, the body contains the application's data and is ignored by the session layer.

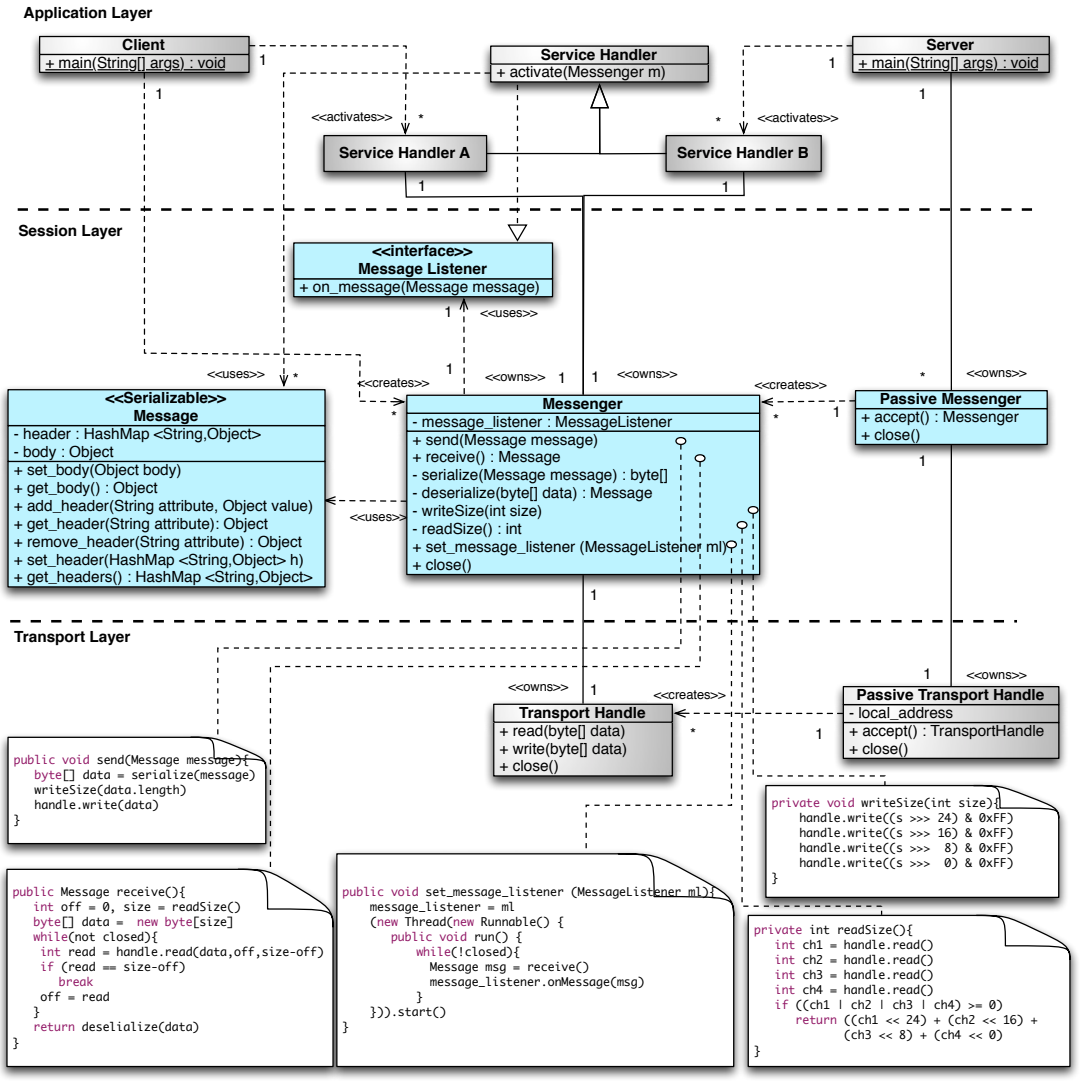

FIGURE 4.2: Messenger design pattern for synchronous message-based applications

The `Messenger` component is dedicated to take the necessary actions for sending and receiving the application's messages through the `Transport Handle`. Indeed, the `Messenger` is responsible for sending a `Message` as an array of bytes through the stream-based `Transport Handle`, and also for delivering an array of bytes, read from the `Transport Handle`, to the `Service Handlers` as a `Message`.

When a `Message` is given to the `Messenger` through the method *send()*, the messenger

converts (or serializes) the message to an array of bytes, writes the size of the message, [1] and then sends the serialized message. On the receiver side, when the method *receive()* is invoked by the application, the receiver reads the size of the incoming message, receives the message completely, deserializes it from an array of bytes to the `Message`, and delivers it to the `Service Handler`.

Messages can also be delivered to the application using a callback method defined in `Message Listener`. To enable the automatic delivery of messages, rather than having explicit requests for read (through the method *receive()*), the `Service Handler`s must implement the method *on_ message()* of the `Message Listener` and pass the reference of this method to the `Messenger` through the method *set_ message_ listener()*. When this happens, the `Messenger` internally dedicates a new thread for reading the messages and delivering them to the service handler through the method *on_ message()*.

4.2.2 Message Flow Diagram

Figure 4.3 presents the life cycle of a message in a one-way interaction pattern, considering a communication stack that includes the `Messenger`. In this context, message life cycle refers to the sequence of steps taken since a message is sent by the sender until it is processed by the receiver.

As shown in the figure, in the very first step, the sender generates a message (m at t_0), then it sends the message (m) to the receiver through its messenger (t_1). The sender's messenger serializes the message to a stream of bytes (m -> m'), writes its size (t_2), and finally sends it to the receiver (t_3). After reading the size of the message (at $t_{3'}$), the receiver's messenger reads the message (m') completely (at t_4), deserializes it (m' -> m), and delivers it to the receiver (t_5). Finally, the receiver processes the message (t_6).

4.3 Trackable Messenger Design Pattern

In this section, we advance the design presented in the previous section to build a one-way messaging service that enables tracking of messages along their life cycle. In

[1] Although there are several mechanisms to determine the end of each message (e.g., defining a unique marker in the beginning or end of each message), here we use the size of message for the sake of simplicity.

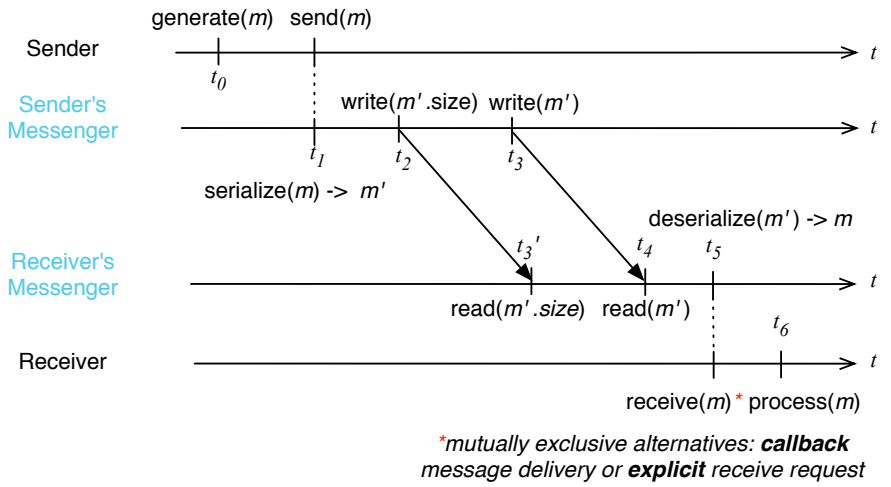

FIGURE 4.3: Message flow diagram in a one-way communication using the Messenger

a trackable one-way communication, the sender can always confirm the reception and successful processing of its messages at the intended destination. This idea of message tracking is also supported in several existing message-based middleware, such as JMS implementations and MSMQ.

We argue that the right solution for achieving this trackable messaging service is in-between the extremes of having no-feedback (i.e., never sending any feedback from the receiver to the sender for any messages), to having a request-response messaging paradigm (i.e., sending feedback for each message). On one hand, closing the loop and letting the sender know the result of its invocations enables the creation of more reliable applications. On the other hand, we must not do it on a synchronous single-message basis (i.e., one response for each request), because this is too costly. The approach we propose offers end-to-end reliability to one-way operations, by using an asynchronous acknowledgment mechanism. This decouples the sender from the receiver and lets them progress independently.

Having an additional receiver confirmation is a price worth paying. At first sight we are simply turning a one-way interaction into a request-response one, but this change is nearly transparent to the application (i.e., the acknowledgments are used below the application layer). Moreover, the receiver can send periodical collective acknowledgments (i.e., one acknowledgment for several messages) and the sender may receive asynchronous feedback, as it certainly does not want to wait for each single reply.

Our design solution, which is called Trackable Messenger design pattern, allows the sender to simply check the status of its messages at any time after the message is sent. Considering the message flow diagram presented in Figure 4.3, the status of a message is `SENT` when the message is written to the channel by the sender's messenger (t_2); `RECEIVED` when the message is read by the receiver's messenger (t_5); `PROCESSED`, when the message is successfully processed by the receiver (t_6); and `ERROR` when the message could not be successfully processed (t_6). The rest of this section explains the Trackable Messenger design pattern, presented in Figure 4.4, in detail. We first describe the components and their relations and then explain the collaboration between those components.

4.3.1 Components of the Trackable Messenger Design Pattern

In the core of the Trackable Messenger design pattern, presented in Figure 4.4, there is a component, named `Trackable Messenger`, which extends the functionalities of the `Messenger` to provide a trackable messaging service for message-based applications. The Figure 4.4 also presents a partial implementation detail of this design pattern.

This component, besides providing an interface that allows the applications to send and receive messages, assigns a unique identifier to each serializable `Message`, sent by the application layer, in order to keep track of them during their life cycle. The `Trackable Messenger` always keeps the identifier of the last message sent (for generating the next identifier and also for local session layer acknowledgment) and the identifier of the last message received (for remote session layer acknowledgment). The `Trackable Messenger` also allows the sender's `Service Handler` to request its peer (receiver's `Service Handler`) for a confirmation of processing of a particular message. In this case, the `Messenger` needs to keep the identifier of the last message processed or any error messages resulted from an unsuccessful processing of the messages. This information is exchanged between the peers either by being piggybacked into the application messages or by periodical acknowledgment messages generated by the `Trackable Messenger`s.

The `Trackable Messenger` uses a central timer, named `Acknowledgment Timer`, to efficiently perform periodical asynchronous acknowledgment. The acknowledgment intervals can be defined by the application, possibly depending on its messaging rate (i.e., the number of messages exchanged per unit of time). The `Acknowledgment Timer`, which

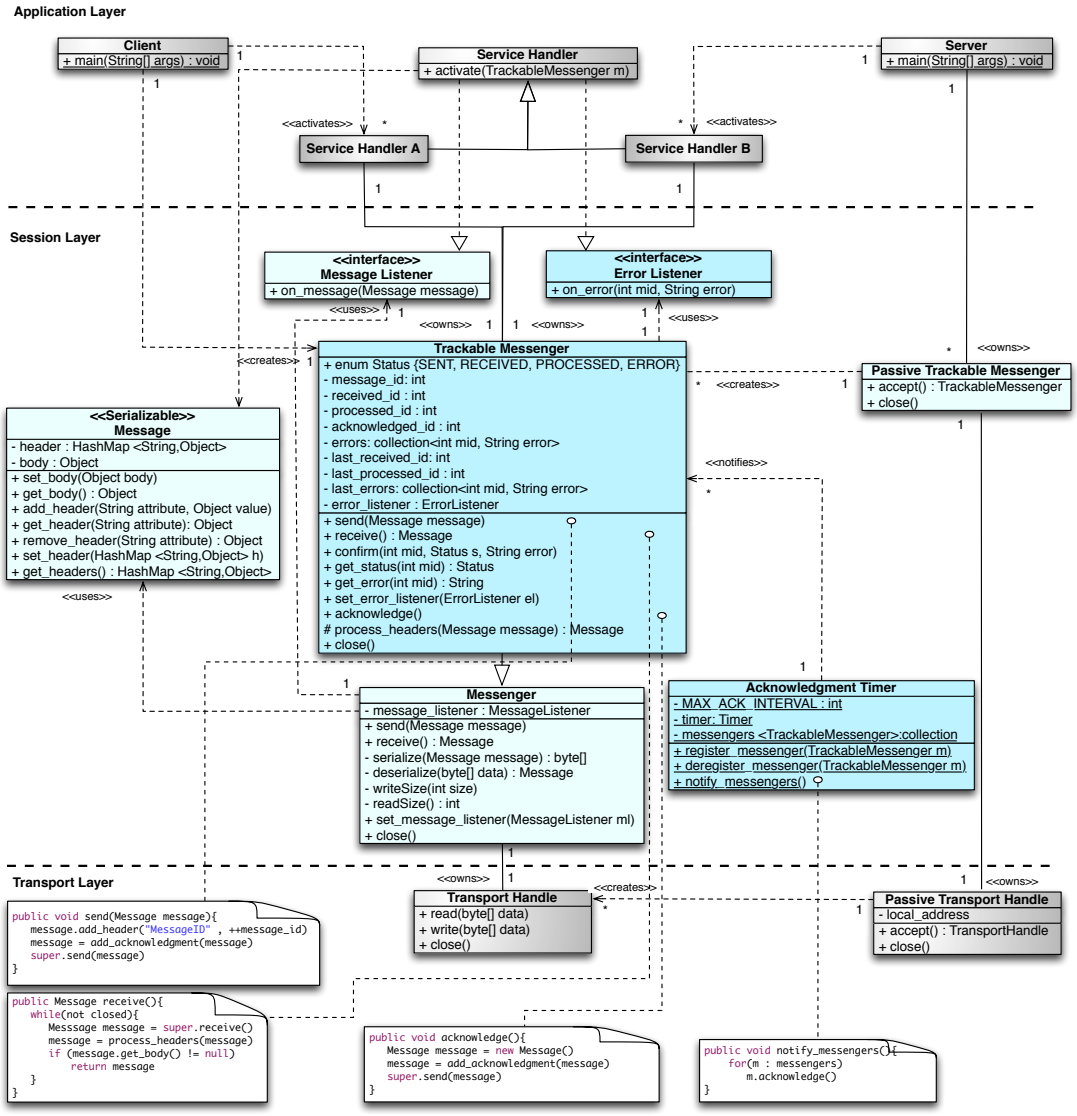

FIGURE 4.4: Trackable Messenger design pattern

implements the Observer design pattern (Hohpe and Woolf, 2003), is used to periodically trigger all `Trackable Messengers` (belonging to different concurrent connections that may exist, especially on the server side) for sending an acknowledgment if there are any unacknowledged messages. We dedicated just one central timer for all messengers to reduce the memory utilization especially in the server. We must emphasize that the acknowledgments can also be piggybacked in the header of the application's messages to reduce overhead on the network caused by extra messages.

`Error Listener` is another component of this design pattern, which, similarly to the

`Message Listener`, provides an interface that must be implemented by the `Service Handler`, if it wants the `Trackable Messenger` to notify any error messages resulted from the processing of messages at its peer. In other words, the `Error Listener` enables the service handlers to give the reference of their callback method (*on_ error()*) to the messenger to be used, if necessary, for notifying the errors.

4.3.2 Collaboration between the Components

Upon creation of a connection and after initialization of both client and server, the service handlers start exchanging messages. They send their messages by invoking the method *send()* of the `Trackable Messenger`, which in turn assigns a unique identifier to each message. The message identifiers are sequential integers starting from 1 (the value of the last identifier is kept in *message_id*).

The `Trackable Messenger`, in addition to the unique identifier, inserts some other information into the header of the message sent to piggyback the confirmation of the unacknowledged messages. These headers include the identifier of the last message received (*received_id*) for session-layer acknowledgment; the identifier of the last message successfully processed (*processed_id*); and the error messages (*errors*) for application-layer acknowledgment. Once the headers are inserted, the message is serialized, the size of the message is written, and finally the serialized message is sent.

The sender's service handler will later know what happened to its message, depending on the option it selected for that specific message at sending time. The `Service handler` can receive a session-layer acknowledgment, or an application-layer acknowledgment. In the former case, the sender's messenger acknowledges whether the message is sent, and the receiver's messenger acknowledges if the message is received. In the latter case, the receiver application (receiver's `Service Handler`) explicitly acknowledges using the method *confirm()* after the message is processed. To receive an application layer acknowledgment, the sender application asks its peer to confirm the successful processing of a specific message, by adding the attribute *Confirmation* into the header and setting it as *true*. For example, for a given message m, the application does *m.add_header("Confirmation", true)* to ask for the peer application-layer acknowledgment.

To receive messages, the method *receive()* of the `Trackable Messenger` is invoked, which, in turn, reads the size of the message, reads the message, deserializes the message, updates the identifier of the last message received (*received_id*), updates the information of the acknowledged messages (if any information is piggybacked) and delivers the message to the service handler. After processing the message, the service handler must check if a given message requires confirmation (by checking the value of the attribute *Confirmation*). If so, it must confirm its successful or unsuccessful processing (i.e., by identifying the message's status and sending an error message if the status is *ERROR*), through the method *confirm()*. Otherwise, when the attribute *Confirmation* is not set for a message, no action should be taken for application-layer acknowledgment.

The `Trackable Messenger` also sends acknowledgments periodically if some messages remain unacknowledged. The `Acknowledgment Timer` periodically triggers the messengers, by calling the method *acknowledge()*, to see if there are any unacknowledged messages left. To enable this, the messengers must register in the `Acknowledgment Timer` upon initialization of connection and deregister upon closure of the connection, respectively using the method *register_messenger()* and *deregister_messenger()*.

The information extracted from the acknowledgment messages is stored in the attributes *last_received_id*, *last_processed_id*, and *last_errors*. This information can be asked by the application through the method *get_status()* or *get_error()*. As explained before, the service handler can get the errors in two different ways. It either passes the reference of its callback method *on_error()* to the messenger through the *set_error_listener()* for automatic delivery, or directly invokes the method *get_error()* of the messenger.

4.4 Reliable Messenger Design Pattern

In this section, we further advance the Trackable Messenger design pattern to be able to tolerate connection crashes. To achieve this objective, we use the Connection Handler design pattern presented in Chapter 3, and the resulting solution is named Reliable Messenger design pattern. Figure 4.5 presents the complete design of a reliable message-based application, which extends the functionalities of the Trackable Messenger design pattern and the Connection Handler design pattern. In this section, we explain how the

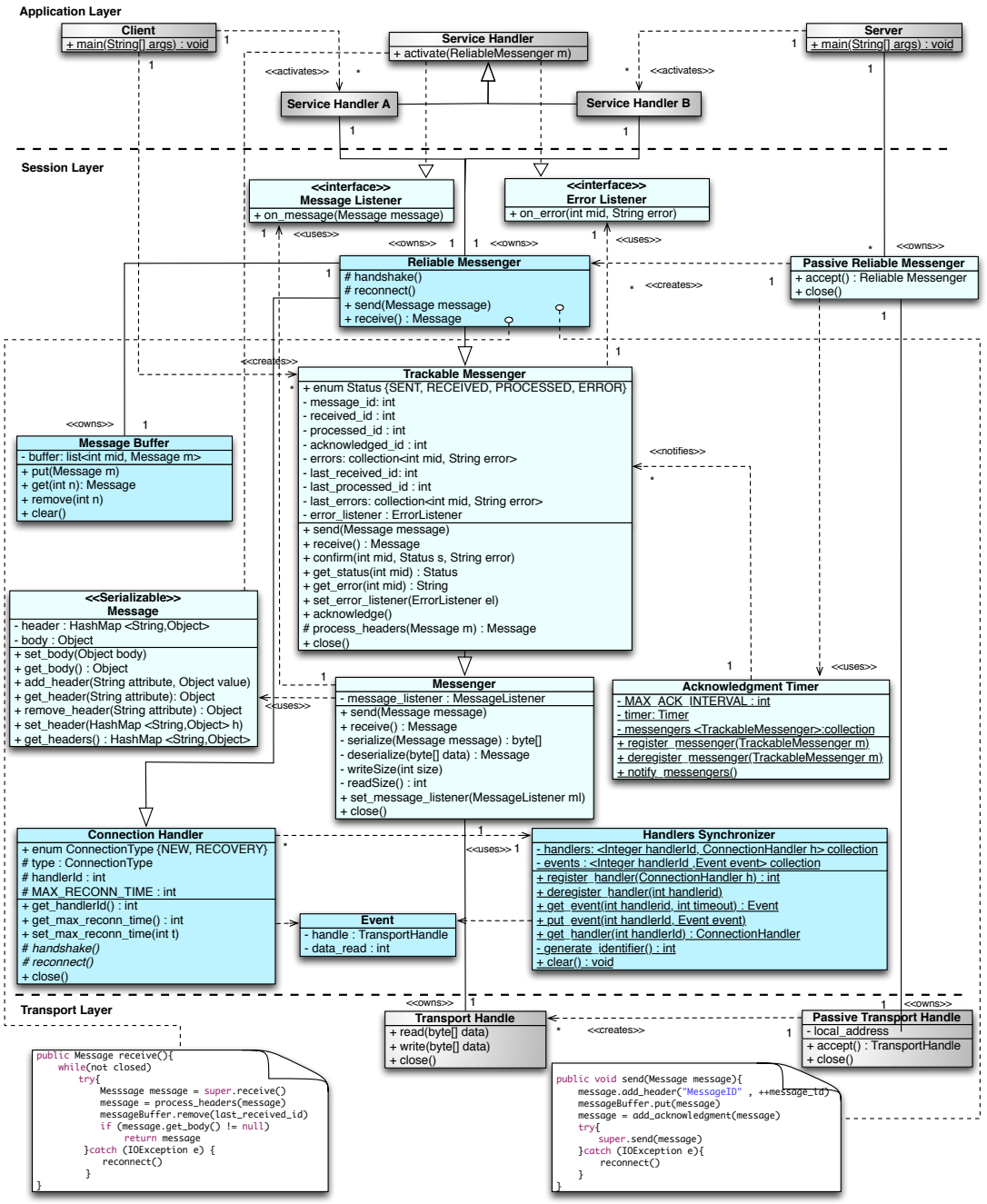

FIGURE 4.5: Reliable Messenger design pattern

Connection Handler design pattern is incorporated and integrated with the Trackable Messenger to ensure recovery from connection crashes.

4.4.1 Components of the Reliable Messenger Design Pattern

To be able to recover from connection crashes by using the Connection Handler design pattern, we need a reliable endpoint that inherits the properties of the `Connection Handler` and implements its handshake and reconnection processes. This reliable endpoint, which is called `Reliable Messenger`, also extends the functionalities of the `Trackable Messenger` to enable message tracking [2].

We also need a simple mechanism of buffering and retransmission of messages, so that we are able to keep both peers in a consistent state after recovery. Thus, each `Reliable Messenger` owns one `Message Buffer` that implements the interface of the `Buffer` component of the Connection Handler design pattern. Furthermore, the `Reliable Messenger` must modify the *send()* and *receive()* operations of the `Trackable Messenger`, to implement the actions that are necessary for buffering the messages (before sending them), removing the acknowledged messages from the buffer (after receiving an acknowledgment), and intercepting a connection crash (while writing or reading into/from the channel). To implement the actions that are necessary for reconnection after crashes, and retransmission of lost messages, two abstract methods *handshake()* and *reconnect()* of the `Connection Handler` must be also implemented.

4.4.2 Message Flow Diagram

Figure 4.6 presents the message flow diagram, for both data messages (application-layer messages) and control messages (acknowledgments), in a one-way messaging pattern with the `Reliable Messenger`. The sender starts by generating a message (m at $t0$), then it sends a message ($t0$). The sender's reliable messenger assigns an identifier to the message (mid), stores it in the `Message Buffer`, serializes it (m -> m'), writes its size ($t2$), and then writes the serialized message into the channel ($t3$). At this point, if the application asks for the message status, the messenger returns "SENT" to the application (at $t_{4'}$).

[2]Since multiple inheritance is not supported in several languages (e.g., Java) and, therefore, the `Reliable Messenger` cannot directly extend both `Trackable Messenger` and `Connection Handler`, when implementing the pattern in such languages, we need to extend the `Connection Handler` in the `Messenger`, to pass its properties and functionalities to the `Reliable Messenger` through an indirect inheritance.

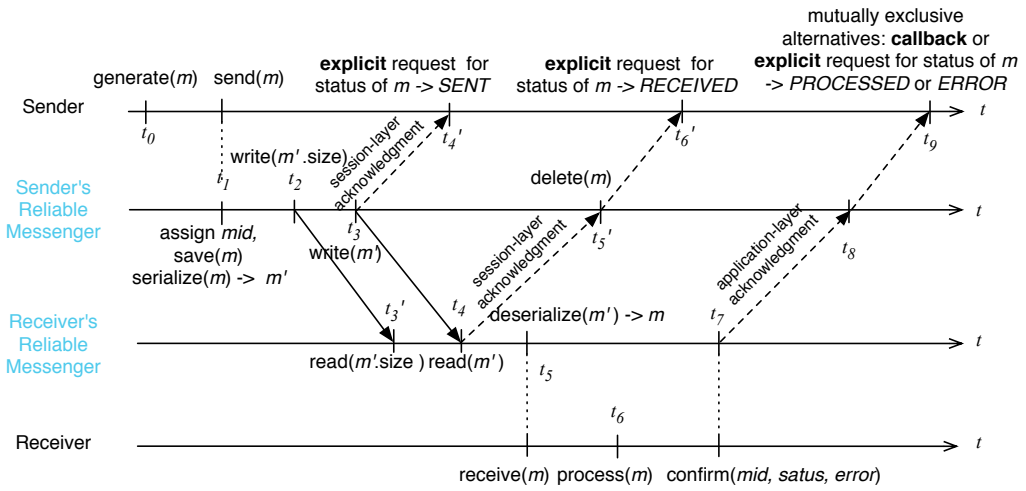

FIGURE 4.6: Message flow diagram in a one-way communication using the Reliable Messenger

On the receiver side, when the method *receive()* is invoked (either directly by the service handler or by an internal thread, which is created by the messenger when a message listener is set), the messenger first checks the channel to read the size of the incoming message ($t_{3'}$) and then to read the entire message (t_4). Upon receiving the message (m'), the messenger executes the following operations in order: 1) deserializes the message (m' -> m); 2) updates the value of the *received_id*; 3) extracts the acknowledgments from the message and updates the value of the *last_received_id*, *last_processed_id*, and *last_errors*; 4) removes the acknowledged messages from the buffer; and 5) delivers the message to the application layer (t_5). At this point, if an acknowledgment is sent to the sender, the status of the message will be changed to "RECEIVED" ($t_{6'}$).

The receiver's application processes the message (t_6), and if the message needs confirmation (requested by the sender of the message), it confirms the successful or unsuccessful processing of the message (t_7). If the message is processed successfully, the application passes the identifier of the message through the method *confirm()* and sets its status to "PROCESSED", otherwise the status should be set to "ERROR" and an error message is also passed to the messenger.

When the sender's reliable messenger receives the application-layer acknowledgment (t_8), including an error message, either it notifies the application by using the callback method *on_error()*, or delivers the error message when the status of the message is

explicitly asked by the sender's application (t_9). Although the sender application may discard any error notifications, we believe that having the option of receiving error notifications is an advantage.

4.4.3 Handling Connection Crashes

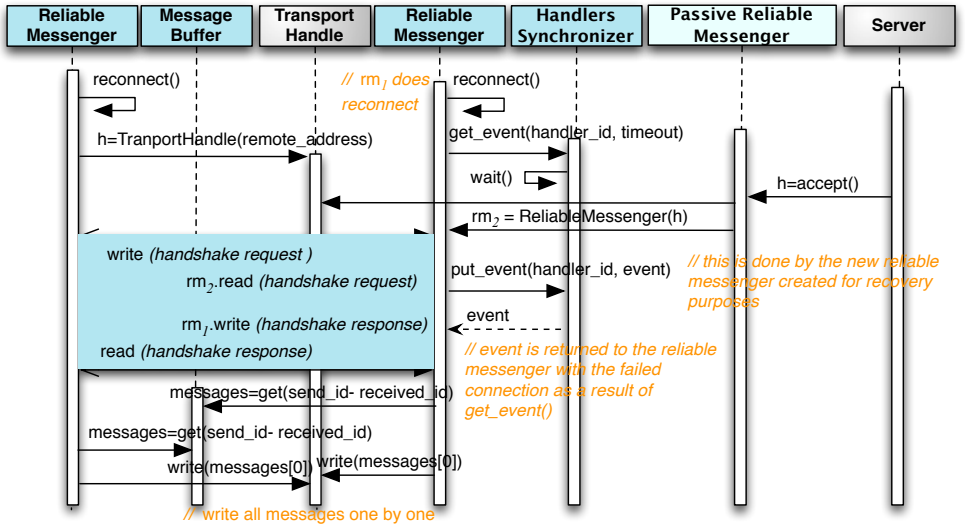

FIGURE 4.7: Recovery from connection crashes with the Reliable Messenger

When a connection fails, both client and server `Reliable Messengers` try to reconnect by invoking the *reconnect()* method of the `Connection Handler` class. Figure 4.7 presents the recovery process in detail.

The initiative to reestablish a connection always belongs to the client's messenger (due to the possible presence of NAT schemes or firewalls), while the server's messenger (rm_1) waits for a recovery connection to arrive. In order to distinguish new connections from the recovery connections (created for recovery purposes), each connection requires a unique identifier (*handler_id*), which is defined in the `Connection Handler`. To generate this identifier, the `Reliable Messengers` need to perform a handshake before starting the exchange of data messages, when the connection is created for the very first time. When a crash occurs, and after a successful reconnection, the client sends a handshake request, including this identifier, to the server to let the server know that the connection is created for recovery purposes. In addition to the connection's identifier (*handler_id*), the identifier of the last message received (*received_id*) is also exchanged between the peers through the handshake procedure. This allows the

`Reliable Messenger`s to re-transmit the messages that were lost in transit due to the connection crash.

The replacement of the failed connection with the new one occurs using the `Handlers Synchronizer`. The `Connection Handler`, created for recovery purposes (rm_2), asks the `Handlers Synchronizer` to deliver an `Event`, which includes the new transport handle and the identifier of the last message received, to the appropriate `Connection Handler` (rm_1). Once the failed handle is replaced with the new one, a handshake response is sent back to the client. At this point, both `Reliable Messenger`s can get the lost messages from the `Message Buffer` and retransmit them.

4.5 Conclusion

This chapter presented three design solutions for message-based communication: Messenger, Trackable Messenger, and Reliable Messenger. We progressively tackle several limitations of TCP for reliable message-based distributed applications, in which the one-way interaction pattern is used for communication.

We first built a message-based session layer on top of TCP with the goal of providing facilities for applications to easily send and receive enveloped messages, without being involved in the process of building the message envelope and extracting a message from an envelope. This is transparently done in the Messenger design pattern, which can be used in any message-based application with one-way or request-response interaction and independently of the application layer protocol. Then, we addressed TCP's limitation regarding the one-way interaction pattern, in the Trackable Messenger, to allow a sender to simply track its messages by exchanging multi-level acknowledgments at the send, receive, and processing points. We further advanced our solution to address TCP's limitations regarding connection crashes in message-based applications using the Connection Handler design pattern. We integrated the Connection Handler design pattern with the Trackable Messenger, and built the Reliable Messenger design pattern. With Reliable Messenger, message-based one-way applications can track the status of the messages sent, and transparently recover from connection crashes.

By being reliable and by providing synchronous one-way trackable communication, we believe that the Reliable Messenger design pattern closes the gap that exists between

the existing message-oriented solutions, that mostly support loosely-coupled communication, and many one-way applications that require reliable tightly-coupled communication (e.g., multi-player games, chat, stock market, social networks). Moreover, to cover a wide range of application and systems, our solution is presented as a design pattern, thus helping developers to implement more reliable message-based applications, independently of the programming language and platform.

Chapter 5

A Reliable Conversation-Based Solution for Request-Response Interactions

The request-response messaging paradigm is perhaps the most popular form of interaction in conversation-based distributed applications, where the accomplishment of an interaction depends on the exchange of several messages. We can find plenty of examples, where people interact with web sites to purchase some good, transfer money, pay bills, or perform diverse business activities. These applications consist of a request-response interaction pattern, where the server performs an action on behalf of the client and sends the result back.

Reliable conversation-based applications may require different reliability guarantees depending on the type of their operations. At-most-once, at-least-once, or exactly-once delivery and execution of messages are the main reliability semantics that can be required by the applications whose reliability target is a "conversation". A conversation, as explained in Chapter 2, refers to a sequence of messages that must be exchanged between two peers to complete an interaction. The reliability target in such applications is usually to ensure the delivery and execution of all of the messages involved in an interaction, which means that the conversation (or interaction) should be repeated from the beginning, if one of the messages is lost or corrupted (i.e., perhaps due to connection or endpoint crashes). For example, online banking transactions, such as money transfer,

usually involve exchanging several messages (operation details, confirmation, and, at the end, a security code), the loss of which causes the failure of the transaction, leading the user to repeat the transaction from the beginning.

With at-most-once semantics, the operations are not allowed to be processed more than once but may or may not be executed. To achieve this guarantee, in its simplest form, the client sends a request and its response may or may not arrive. In a more reliable form, the client may re-invoke the request, but the server has to filter duplicate requests to avoid processing the same request more than once. The at-least-once semantics is on the other end of the spectrum, in which a request must be executed and it is admissible to be processed more than once. To achieve this guarantee, the client must re-send the same request until it receives a response, even in the presence of crashes.

Neither the at-most-once, nor the at-least-once semantics can be used in business or safety-critical applications, such as online banking, airline reservation, and air traffic control systems. First, their operations are *non-idempotent* and should not be executed more than once. Furthermore, a lack of response is also not acceptable, because clients need to know whether the operation succeeded or not. Thus, only the exactly-once semantics, which is somehow the combination of both at-least-once (i.e., at-least-once delivery of messages) and at-most-once (i.e., at-most-once execution of messages) semantics, is entirely appropriate for such applications.

Exactly-once semantics is very hard to achieve, because any component may crash in the middle of an interaction and having the same knowledge of the interaction in all components is very difficult (Halpern, 1987). Also, many times, it is difficult or impossible to distinguish a crash from slow transmission or slow processing of messages (Fischer et al., 1985).

In this chapter, we aim to propose an exactly-once request-response protocol and a design solution facilitating the implementation of the protocol. To do so, we first explain the at-least-once and the at-most-once protocols and then accordingly propose a protocol for achieving the exactly-once guarantee in the request-response interactions. Our exactly-once protocol unambiguously defines the boundaries where the client may resend the request, upon recovering from a crash, without taking any chances of repeating the operation. Next, we adapt the protocol for the applications that may need middleware to implement the exactly-once request-response interaction and propose a session-based exactly-once protocol and design solution.

The remainder of this chapter is organized as follows. Section 5.1 explains how the at-least-once guarantee can be achieved. Section 5.2 describes the at-most-once request-response protocol and explains how a reliable request-response interaction with at-most-once semantics should be implemented. Section 5.3 proposes a protocol to achieve exactly-once semantics, using the protocols presented for at-most-once and at-least-once. Section 5.4 adapts the protocol proposed as a session-based protocol and proposes a design solution for implementing an exactly-once middleware. Finally Section 5.5 concludes this chapter.

5.1 At-Least-Once Request-Response Interaction

Before going to the details of the protocol we need to clarify some assumptions. Here we assume that any component involved in the interaction, including client, server and channel, may crash, but they eventually work in a fault-free period for a sufficiently long time to complete the at-least-once interaction (i.e., their crash mode is crash-recovery). Channels are assumed to be unreliable, but fair, which means that they may lose messages, but will deliver a message that is sent a sufficient number of times. They neither reorder the messages nor change their content. We also assume the existence of stable storage on the client, to save the state of the interaction in case of crashes.

To ensure the at-least-once semantics, the clients must re-invoke unreplied requests. It must repeat this action after a predefined period of time, until receiving the response. This implies that the client needs to keep the request and its timestamp into a buffer and set a timer for retransmission.

Figure 5.1 presents a reliable request-response protocol for achieving at-least-once semantics. The protocol starts with the client generating a request (req_1), assigning it a unique identifier, assigning it a timestamp, which indicates when the request is sent, and saving the request into both volatile (memory) and stable storage (t_0). Then it sends the request (t_1) to the server. The server receives the request (t_2); processes it, generates a reply (rep_1) for the request, and assigns it the same identifier of the request (t_3). Then it sends the reply to the client (t_4). The client, after receiving the reply (t_5), can delete the request from both memory and stable storage (t_6). In the case no response is received for the request, after the time limit, the protocol restarts from t_1, by updating the request's timestamp and retransmitting it.

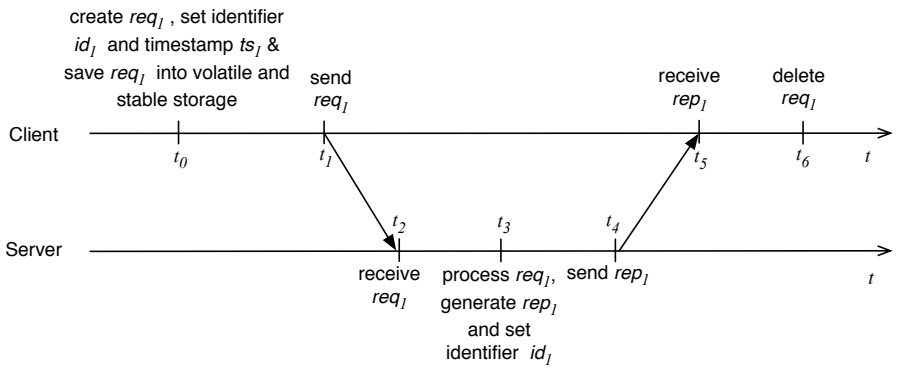

FIGURE 5.1: At-least-once request-response protocol

In this protocol, the requests and their responses must be uniquely identified, to allow the client to filter duplicate responses, which can be caused due to slow transmission or slow processing of the requests, and therefore, may cause failure at achieving at-least-once guarantee. Consider the following scenario using the above protocol without uniquely identifying the requests and responses. The client sends req_1 to the server. Then, since no response is obtained after the time limit, the client retransmits the same request. It then receives rep_1, which belongs to the first request, but arrived too late (after the time limit). In this case, the client is not able to identify that this is a delayed response, so it deletes req_1 and sends the next request (req_2). After sending req_2, it receives rep_1, which belongs to the second attempt of the req_1, but the client is not able to identify it, so it is considered as a reply of req_2, which lets the client to delete this request from the storage. At this point, the at-least-once semantics can be violated if req_2 has never been processed at the server, because the client will never re-transmit it.

Given the fact that several responses may arrive for the same request, the client needs to match each response with the corresponding request, to identify exactly which requests must be reinvoked (when its response is missing). As shown in the protocol, the client must set a unique identifier for each request and the server must use the same identifier of the request in its response, enabling the client to easily distinguish different responses from each other and then match them with the requests.

It is worth mentioning that there is another solution to address the above challenge regarding delayed duplicate responses and violation of at-least-once semantics. The client can send the request with a timestamp to the server. The server has to send back the reply with the same timestamp to the client. This allows the client to detect and

filter delayed responses by checking the difference between the timestamp and current time. Although this solution solves the above problem in an easier manner (since it does not need to generate unique identifier), it is not as effective as the previous solution that uses identifiers. This is due to the fact that the client may enter the loop of retransmission for a given request, not because its response did not arrive, but only because of the slow transmission and processing of the request. To avoid this situation, the time limit for retransmission has to be defined properly (i.e., it should be large enough to avoid retransmission of the requests whose responses are delayed). This is not a practical solution, because on one hand, it is difficult to define a small time limit that always works, and on the other hand a large time limit can negatively affect the performance.

5.2 At-Most-Once Request-Response Interaction

Comparing to at-least-once semantics, the assumptions and protocol to achieve at-most-once guarantee have three main differences: 1) the at-most-once client may or may not re-invoke a request, while an at-least-once client must re-invoke requests, for which it obtained no response; 2) the at-most-once server must prevent duplicate execution of the same request (in the case it is re-invoked), while an at-least-once server does not need to prevent duplicate execution of the same request; and 3) for achieving at-most-once, in contrast to at-least-once, the crash-mode at the client and communicating network does not need to be crash-recovery, which means that the client does not need any stable storage to store the state.

There are two strategies for achieving at-most-once guarantee. To ensure that each (non-idempotent) operation is not executed more than once, either the client must never repeat the same request (even when the first request is lost and never processed by the server), or the client re-invokes the requests, but the server must use a filtering mechanism to prevent duplicate execution of the repeated requests. The former strategy is not considered to be a reliable solution, because a lack of response that might be caused by several reasons including request loss, unsuccessful processing, or response loss, ends an interaction forever in the client. This occurs because all the aforementioned reasons for lacking a response are the same for the client (i.e., there is no mean in the client to differentiate them from each other), so it cannot risk duplicate execution of the request by re-invoking it. For this reason, we chose the second strategy for achieving

at-most-once guarantee, in which the client can re-invoke the requests, but the server must avoid re-execution of the same requests.

To implement the second strategy, the client must keep a request and its timestamp (i.e., it holds the time in which the request is sent) into a buffer until its reply is received. Since we are not assuming a crash-recovery client, a volatile memory is enough for buffering the requests. To enable the retransmission of the requests (if necessary), the client needs a timer. The server must distinguish the requests from each other (e.g., using a unique identifier assigned to each request) and filter duplicate requests, to ensure that each request is processed only once.

Using identifier is necessary to filter duplicate requests, but it is not enough for ensuring at-most-once. There are still two challenges left that need to be addressed properly. First, since the server filters the requests that have been already processed, the client may enter the loop of resending a request that is processed at the server but its reply is lost. To avoid this situation, the server needs to keep the responses in a buffer until it is certain that the client received the reply. This allows the server to resend the responses of the repeated requests without processing them again. The client can remove the requests from its buffer whenever it receives their response, but in order to allow the server to safely remove the responses, the client must send an acknowledgment to the server.

The second challenge is raised by crashes on the server side, that may cause re-execution of the same request, due to loosing the information stored in volatile storage about the request processed and its response. Thus to solve this problem, the server needs to save the state of the interaction into a stable storage, in order to recover it after crashes. The processing of the request, that may cause some changes in the state of the system, and storing the state of the request, including its identifier and response, must happen atomically. Otherwise, since the server may crash after processing the request and before saving the response, a repeated request that has been already processed and therefore made some changes in the general state of the system, might be processed again because the server has no information stored allowing to filter it.

Figure 5.2 presents the protocol of a reliable at-most-once request-response interaction. As shown in the figure, the client generates a request (req_1), assigns it a unique identifier (id_1), and sets a timestamp (ts_1), indicating when the request is sent, and keeps the request and its information (t_0). Then, it sends the request (t_1) to the server. The

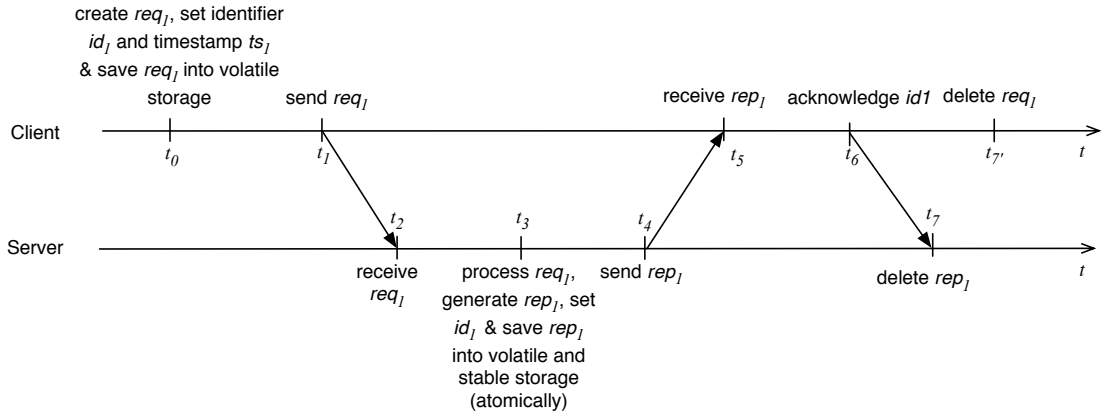

FIGURE 5.2: At-most-once request-response protocol

server receives (t_2) and processes the request, generates an appropriate reply (rep_1) for the request, assigns it the same identifier, and atomically stores the reply into both volatile and stable storage (t_3). The server then sends the reply to the client (t_4). After receiving the reply (t_5), the client sends an acknowledgment including the identifier of the request to the server(t_6), which allows it to safely delete the response (t_7). At this point the client can also delete the request ($t_{7'}$).

5.3 Exactly-Once Request-Response Interaction

Despite being ubiquitous, making the request-response messaging pattern work reliably and guaranteeing exactly-once semantics in the presence of crashes is everything but simple. From one hand, the client cannot simply invoke the request again, because this may cause the server to repeat *non-idempotent* operations, such as making a second flight reservation for the same person or ordering a given item twice. On the other hand, there are cases that have even more stringent restrictions that exceed the *at-most-once* semantics, as they actually require *exactly-once* guarantee. As an example, employers need to make sure that they issue the paychecks once and only once; the bank must then ensure that it deposits the money once and only once; the same for the corresponding withdrawal operation.

The exactly-once semantics was first addressed by Spector in (Spector, 1982). Spector presented request-response-acknowledge (RRA) protocol that enables the server to release memory, by letting the server know that the client received the response. In

this section, we explore this idea and propose our exactly-once protocol tolerating all crashes that may occur during a request-response interaction.

Unfortunately, even the most widely used communication technologies cannot easily tolerate crashes. TCP provides the greatest freedom to the programmer, but the programmer must bear the costs of recovering from TCP connection crashes. To recover from endpoint crashes, servers usually use replication, which although increasing the availability of the server, it cannot guarantee the exactly-once execution of invocations. Some developers use distributed transactions to handle crashes. With the use of distributed transactions, they can ensure that either the client and server agree on the positive outcome of an operation, and then, they commit it; or they both give up. Unfortunately, this kind of solution has several drawbacks: 1) it is difficult to use, as it involves a fairly complex configuration and Application Programming Interface (API); 2) it is slow, due to the several steps involved in protocols like the two-phase commit; 3) it is heavy, because it involves a coordinator process; and 4) it is blocking, because the participants block until a commit or rollback is received, so the transaction never is resolved when the coordinator fails permanently.

In this section, the at-most-once and at-least-once protocols are combined and refined to build an exactly-once request-response protocol. We use buffering (i.e., save messages in memory or volatile storage), logging (i.e., save messages and interaction's state in stable storage), and retransmission mechanisms to ensure that each messages is delivered at-least-once and we add a filtering mechanism, to ensure that each message is processed at-most-once. Saving the state of an interaction into stable storage allows the client and server to return back to the last coherent state, after recovery from endpoint crashes.

5.3.1 Exactly-Once Protocol

To ensure that each request is executed once and only once: 1) the client must re-invoke un-replied requests; 2) the server must prevent duplicate execution of the same request; 3) the server must resend the response of a repeated request to the client; and 4) all of these must happen even in the presence of crashes (e.g., endpoint and connection). Thus, to guarantee exactly-once, we first need to assume that the crash mode of any components involved in a request-response interaction must be crash-recovery. We further need to assume the existence of stable storage on both client and server sides, to keep interaction state to be used in the case of crashes. Here by

considering these assumptions we describe our protocol, presented in Figure 5.3. Note that despite being important for the practical implementation, the presence or absence of a previous handshake to setup a session, like a TCP connection, is irrelevant for our discussion at this point.

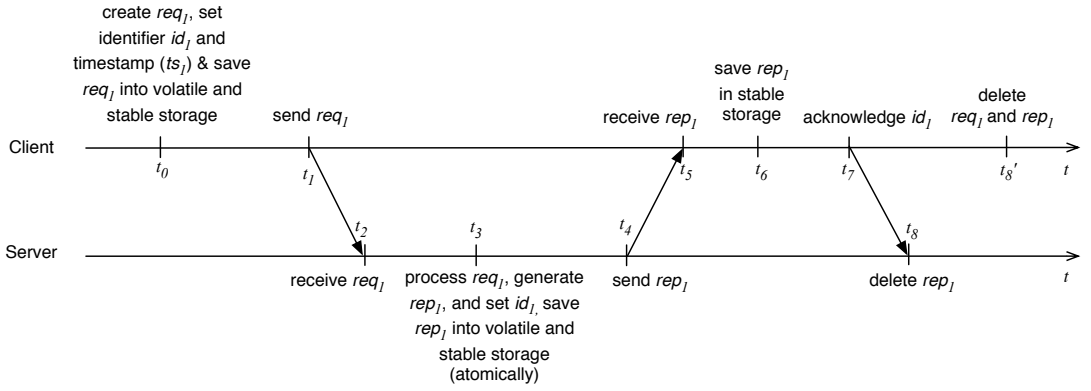

FIGURE 5.3: Exactly-Once request-response protocol

The client starts by creating a request message (req_1), assigning it a unique identifier (id_1) and storing it with its identifier and timestamp into memory and stable storage (t_0). The timestamp enables the client to resend the request if it does not receive a reply within some time limit. Next, it sends the request to the server (t_1), which receives it at time t_2.

Since the semantics is exactly-once, we assume that the server cannot generate the same reply again without changing its internal state, therefore, the server processes the request, generates the response (rep_1), assigns it the same identifier, and saves it into its stable storage atomically (t_3). This is often simple to do, if the server has some operations that use a database, for instance. The application must keep the same identifier to avoid repeating the same operation for a repeated request. Then the server replies (t_4).

The client receives the response (t_5), saves it into stable storage (t_6) and sends acknowledgment to the server (t_7), allowing it to delete the response (t_8). Finally, the client can delete the request and the response ($t_{8'}$). The client can use the response between t_6 and $t_{8'}$.

With this sequence we can ensure the exactly-once semantics, even if the client or the server crash and later recover. Let us examine the client actions. The client creates and stores the request into stable storage, sends it, receives the reply, acknowledges

the server and deletes the request from stable storage. We do not care for crashes for time $t < t_0$ because the request has not been generated yet. For $t > t_{8'}$ the request no longer exists, so we also do not care for it. The client can safely discard any response arriving after that point in time. We, thus, focus on the client actions, after recovery from crashes, for time $t_0 < t < t_{8'}$.

For time $t_0 < t < t_6$, upon recovery, the client cannot determine exactly at which state it crashed, so it resends the request. If the crash occurs at $t_0 < t < t_1$, the client never actually sent the request, so it can just send it for the first time when it recovers. For time $t_1 < t < t_3$, the client will resend the request, but the server can detect the duplicate identifier and avoid re-executing the request. If the client crashes in the interval $t_3 < t < t_6$, upon recovery, the client will resend the request as it does not have the corresponding response in its stable storage. But, since the server stored the state and response, it can filter duplicate requests and resend the response. After $t > t_6$, the client has a copy of the response, so it no longer needs to resend the request. For time $t_6 < t < t_7$, the client just picks the response it saved. For time $t_7 < t < t_{8'}$, the client will resend an acknowledgment, after recovery. The server could then receive an acknowledgment for a response it no longer has, but it can safely discard the acknowledgment.

Now, let us do a similar analysis for the server. If the server crashes before $t = t_3$, upon recovery it can safely process the repeated request (i.e., sent by the client after a given timeout in this case), because it has no state stored for the request that has not been processed. For time $t_3 < t < t_4$, the client will resend the request since it has not received the response yet. In this case, the server must retrieve the reply from storage instead of re-executing the request. For time $t_4 < t < t_8$, the server cannot know whether or not the reply reached the client. It must wait for either the acknowledgment from the client or for a repetition of the request.

5.3.2 Demonstration of Correctness

In this section, we formally identify the properties of the exactly-once request-response interaction pattern and demonstrate that they are held by using our protocol. We assume that the channel does not create, change, or reorder (First In First Out – FIFO) any messages. We assume that channels, client, and server eventually will be correct for a sufficiently long time that enables their interaction to finish. Regarding safety

and liveness, we require the following properties for the exactly-once request-response pattern:

- **Safety 1** At-most-once execution of requests.
- **Safety 2** No invention of response.
- **Safety 3** No duplication of response.
- **Liveness 1** At-least-once reception of response.

Liveness says that all requests eventually have a response. This is the "at-least-once" part of the interaction. However, beside a live behavior, we must ensure some safety properties, to prevent double execution of the request (Safety 1) or reception of the message (Safety 3), and to prevent receiving a response for a request the client never did (Safety 2). Next, we demonstrate these properties.

Safety 1 (At-most-once execution of a request): At-most-once guarantee can be violated if the server receives a repeated request after processing it (after t_3). Since each request is univocally identified by a number, if the server checks its stable storage for this request's identifier, it may not execute the same request twice between t_3 and t_8. Thus we must demonstrate that after t_8 (after it deletes the response), the server may not execute the same request again. Once the client sends the acknowledgment (t_7), it will not send the same request again, because it saved the response in the stable storage at time $t = t_6$. Since we rely on FIFO channels, the server must not receive any repetition for the same request after $t = t_8$.

Safety 2 (No invention of response): This property derives from the fact the channel does not invent any messages (i.e., one of our assumptions).

Safety 3 (No duplication of response): This property depends on the implementation of the client, which must use the response during the interval $t_6 < t < t_{8'}$ and must atomically delete the response and finish the task it has to do with the response at time $t = t_{8'}$. In this case, if the client ever crashes for $t < t_6$, when it restarts it did not use the response. For time $t > t_{8'}$, it is no longer waiting for the response, so it

will discard it (it may receive two or more responses in fact). For time $t_6 < t < t_{8'}$, the effect of the response takes place only once.

Liveness 1 (At-least-once reception of response): First, we assume that the client periodically resends the request if it does not get any response, e.g., because the channels keep failing. Assuming these conditions, we need to prove that the client reaches the point where it saves the response (t_6). Since the client keeps resending the request, and the server always gives a response to this request (either by executing the action or getting the response from stable storage), the property follows from the assumption of correctness of the channels, client and server for a sufficiently long time.

5.4 Exactly-Once Request-Response Middleware

Since the implementation of exactly-once interactions, as shown in the previous section, is not an easy task, it is useful to use a middleware that provides exactly-once services to the application layer. Due to the details involved in the exactly-once protocol we first explain how the exactly-once protocol described in the previous section should be implemented underneath the application (i.e., in the session layer). Then, we describe a design of an exactly-once middleware implementing this session-based protocol.

5.4.1 Session-Based Exactly-Once Protocol

Apart from generating the request and the response, most actions that clients and servers perform are repetitive and can be handed over to library functions (middleware). We separate the endpoint applications in two layers: the application layer takes care of generating and using the messages, while the session layer (middleware) is partially responsible for message idempotence and guarantees the delivery of the requests and responses to the application layer. In Figure 5.4, we show the result of our approach to factor out some client and server actions from Figure 5.3.

The client starts by creating a request (req_1), assigning it a unique identifier, and saving it into stable storage (t_0). The unique identifier can also be generated by the middleware, but it must guarantee that the same identifier is generated for the same request, when the client resends a request after recovery. To ensure this, the client

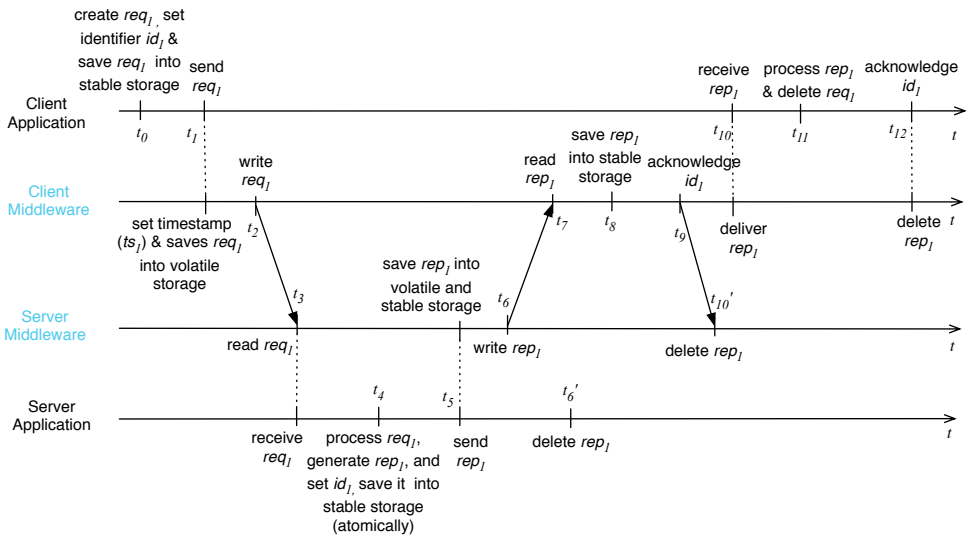

FIGURE 5.4: Session-based exactly-once request-response protocol

middleware can use the hash code of the request content as an identifier or even the request itself could serve as its identifier. In this case, if the client resends the same request, the middleware can determine if it is still processing that request or if it is new, and, at the same time, release the application from the burden of managing message identifiers. A shortcoming of this approach is that the client cannot generate a new request with the same data before acknowledging the previous one. Nevertheless, this limitation can be solved by adding some salt to change the request. In our solution, for the sake of simplicity, we assume that each request is sent with its identifier from the application layer. This identifier is simply a sequential integer.

After generating the request, the client sends it to the middleware (t_1). The client middleware assigns a timestamp to the request and stores it into the buffer (t_1). Next, it sends the request to the server middleware (t_2), which receives and delivers it to the server application (t_3). The server application processes the request, generates its response (rep_1), assigns it the same identifier as the request, and atomically saves it into the stable storage (t_4).

The server middleware must not send a new request for the same identifier, before the application replies (something that could happen if the client sends the same request twice or more) because this may cause a second execution of the same request. Up to $t = t_5$, when the server application sends the reply to the server middleware, this goes on unchanged. However, to enable the server application layer to delete all data

related to the request ($t_{6'}$), the server middleware must save the response (t_5). Then the server middleware replies to the client middleware (t_6), which receives (t_7) and saves the response (t_8). At this point an acknowledgment can be sent to the server middleware (t_8). This enables the server middleware to delete the response ($t_{10'}$). Then the response is delivered to the client application (t_{10}). The client application after receiving the response, uses it and deletes the request and its state from stable storage (t_{11}) and, at the end, asks the client middleware to delete the state of the response from stable storage too (t_{12}).

With middleware, the client application layer no longer resends requests when the connection or the server crash. Therefore, it does not need to generate timestamps for the requests. In the interval $t_1 < t < t_8$, the client middleware already has the request, so it can resend the request by itself. Moreover, the client application does not need to save the response, because this is taken care by its middleware. These are the most important differences for the client application layer. In case the client crashes, if it crashes between $t_0 < t < t_{11}$, the client application will resend the request after resuming. If the crash happens in the interval $t_8 < t < t_{11}$, the client middleware still has the response and does not need to resend the request to the server, otherwise ($t_0 < t < t_8$) the request will be sent to the server. After t_{11}, the client application no longer has the request, so it no further interacts with the middleware.

The server side also has the following important simplification: after delivering the response to the middleware, the application layer may delete the response. This allows the server application to use a typical API with a single blocking point, to receive requests, discarding the need for a second one to receive the acknowledgments. Moreover if a repetition of the request arrives during the interval $t_5 < t < t_{10'}$, the middleware itself will provide the response. Before that point ($t_4 < t < t_5$), the server application layer itself has the response and corresponding identifier and can filter duplicates.

5.4.2 Demonstration of Correctness

Here we demonstrate how the session-based exactly-once protocol ensures the safety and liveness properties.

Safety 1 (At-most once execution of a request): In the session-based protocol, the at-most-once can be violated only after t_4. If a repeated requests arrives at $t_4 < t < t_{6'}$, since the server application has the reply, it will not process it again. If the repeated request arrives at $t_{6'} < t < t_{10'}$, then, since the server middleware stores the response at $t = t_5$ and keeps it until $t = t_{10'}$, it will not give the repeated request to the server application. Now, assume that a repeated request arrives after $t_{10'}$, since the channel is FIFO, the client middleware must have resent the request after acknowledging it ($t > t_9$), when the response is deleted from the server middleware, which is impossible. Since the client middleware saved the response at time $t = t_8$ and deletes it only at $t = t_{12}$, the repeated requests originated by the client application layer will be replied by the client middleware and there is no need to send them to the server. After $t = t_{11}$ no request will be resent by the client because it has already deleted it from the storage.

Safety 2 & 3 (No invention of response and no duplication of response): These are very similar to the case of Figure 5.3, which we demonstrated before.

Liveness 1 (At-least-once reception of response): In addition to the fact previously explained for the Figure 5.3, we used the Spin model checker (Holzmann, 1997, 2004) to formally verify the liveness claims of our approach. Model checking is a method for verifying whether a specification is fulfilled by a model. A specification is a set of properties which a system is expected to satisfy, and a model is a formal description of the system's behavior, written in a modeling language, intended to preserve as much detail as necessary for the verification. Model checking tools such as Spin take a model and its specification as input. Their output is either an indication that the model is correct, or a case in which the correctness properties fail to hold.

The Spin model checker accepts a formal modeling language called Promela (Iosif, 1998). This language is appropriate to model distributed software systems. Interprocess communication can be specified using message channels, which can be either synchronous or asynchronous. We used asynchronous channels to model the communication between the client middleware and the server middleware. Synchronous channels were used to model the call-return interactions between the client application and the client middleware, as well as between the server application and the server middleware.

We modeled a system consisting of four processes, namely the client- and server-side applications and the corresponding middleware instances. An additional process was constructed in order to model failures. Two kinds of failure are considered: lost messages between the two middleware processes, and crashes of either client or server applications. Lost messages were modeled by snatching messages from the channels between client and server middleware. Crashes of the client and the server were modeled by resetting the client- or the server-side (including the application and the middleware) to the initial state. Only the content of the stable storage is assumed to be preserved after a crash.

The interaction among client application, client middleware, server middleware, and server application was modeled according to the specification of our approach. The model describes the behavior of the system for a single request. Given that multiple requests have no influence on one another and that all tiers are able to distinguish between different requests, this abstraction allows us to reduce the state-space needed to model the system.

Our analysis using model checking focused on the system's liveness. Given that the communication channel between the client and the server is asynchronous and that any messages may be lost, liveness can only be guaranteed under the assumption that eventually there is a fault-free period of execution. Moreover, as processes may crash, one must assume that these will also eventually remain fault-free for some period of the execution. To model this assumption, we allow the failure-injecting process to terminate its execution, and specify that a response is eventually received by the client if there are no more failures. In linear temporal logic (LTL) the specified property is the following:

$$\Box(\text{faultfree} \rightarrow \Diamond\text{terminates})$$

The formula is interpreted as follows. Whenever the system becomes fault-free (i.e., the failure injector ends its execution) the client will eventually receive the response and therefore terminate the execution of a request. The symbol "faultfree" was defined as the failure injector ending its execution and the symbol "terminates" was defined as the client application reaching its final statement.

The correctness of the model with respect to the liveness claim was checked using Spin version 6.1.0. The formula is found to be correct by the verifier in 2 hundredths of a second, for a system totaling 5.8×10^3 states and 2.2×10^4 transitions (with partial order

reduction enabled). This increases our confidence in the correctness of our approach regarding liveness.

5.4.3 Design of an Exactly-once Middleware

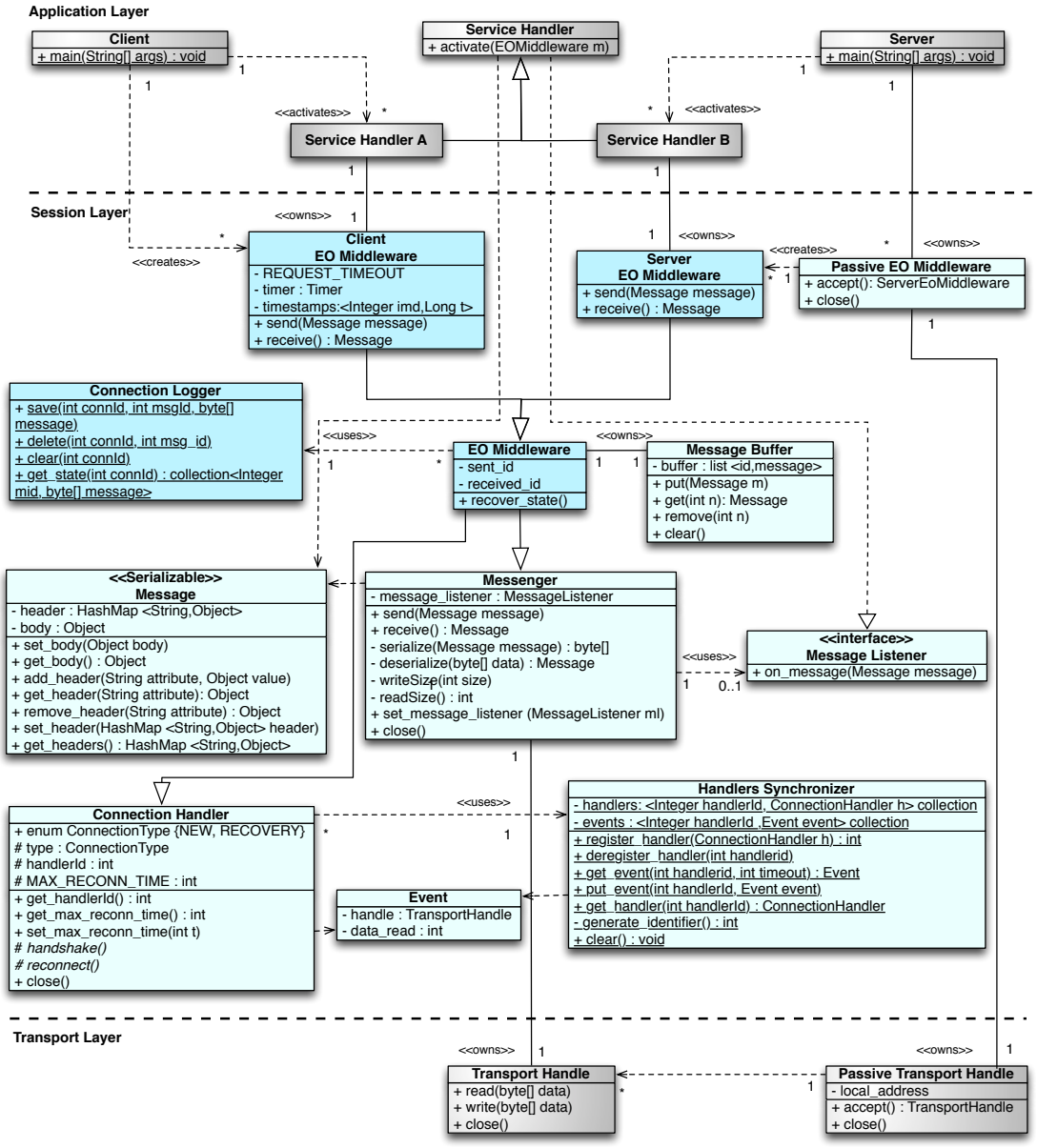

FIGURE 5.5: Design of exactly-once request-response middleware

Figure 5.5 presents the design of an application with an exactly-once request-once middleware implementing the protocol explained above. As shown, in this design, we

use the Messenger design pattern, presented in Chapter 4, to implement the message-based send and receive operations; the Connection Handler design pattern presented in Chapter 3, to handle connection crashes; and some extra components to implement the actions that must be taken for request retransmission, preventing request re-execution, and endpoint crashes.

As shown, the central component of the design is the `EO Middleware`, which extends the `Messenger`'s functionalities for exchanging messages and owns one `Message Buffer`, to keep the requests and responses until being respectively replied or acknowledged as explained in the protocol. Since the client and server's actions are different in the session-based exactly-once protocol, specially when they send and receive messages, the `EO Middleware` is extended by two different components: `Client EO Middleware` and `Server EO Middleware`. Moreover, Each `Client EO Middleware` owns a timer and keeps the timestamp of the sent requests in order to enable retransmission of requests after a predefined timeout ($REQUEST\_TIMEOUT$), while the `Server EO Middleware` does not need it.

In order to enable recovery from connection crashes, the `EO Middleware` extends the `Connection Handler`, so that its properties and functionalities are passed to the exactly-once middleware. `EO Middleware` keeps the identifier of the last message sent and the last message received respectively in *sent_id* and *received_id*, to support the recovery process. The abstract methods of *handshake()* and *reconnect* must be implemented properly in the `Client EO Middleware` and `Server EO Middleware`. During the handshake procedure, a unique identifier is assigned to the connection and is kept in $handlerId$, which is an attribute of the `Connection Handler`. After recovery from connection crashes the client and server middleware exchange the identifier of the connection, allowing the server to distinguish the new connection from a recovery connection, and the identifier of the last message sent and received, allowing the peers to retransmit the requests and responses that are lost due to crashes.

To enable recovery from endpoint crashes, the `EO Middleware` requires another component, the `Connection Logger`, to save the requests or responses in stable storage. This component provides an interface to save messages with their identifier and the identifier of the connection ($handlerId$), to which the messages belong; delete messages of a given connection; entirely remove the state of a given connection (for example, when it is closed); and finally re-build the state of a given connection (list of the messages that are not replied or acknowledged yet). The `Connection Logger` is shared between

several `EO Middleware`s, so it provides static and thread-safe methods to accomplish the aforementioned operations.

To recover from endpoint crashes, the `EO Middleware` provide the method *recover_state()* to the application layer, which is called to read and rebuild the state of the connection from the stable storage. Here, we assume that the application layer is responsible to keep the general state of the `Service Handler`s, including the connection identifier and destination address.

5.5 Conclusion

This chapter first focused on the at-least-once and at-most-once protocols for conversation patterns. Then, it focused on the strongest and most difficult reliability semantics, exactly-once. We first proposed an application-level protocol guaranteeing exactly-once semantics, by combining the at-least-once and at-most-once protocols. Then, the protocol is refined for a session-based solution, in order to partially eliminate the complexity from the application layer. The correctness of both exactly-once protocols is formally proven. Then, a session-based design solution is proposed, facilitating the development of request-response applications that require this reliability semantics. We believe that our solution releases developers from most of the complexities, deriving from crashes, in the development of their critical applications.

Chapter 6

Taxonomy of Reliable Request-Response Protocols

Communication in distributed systems often takes the form of request-response interactions, where a client uses a channel to send a request to a server that, in turn, sends back a response. Thus, such pattern typically involves three different roles (client, server, and channel), which must engage in a very rigid manner, to ensure that the interaction succeeds. However, the notion of **success** depends on the application, its business logic and reliability requirements. Some common reliability semantics used in this interaction pattern are at-least-once, at-most-once, and exactly-once, which respectively refer to the server executing the request once or more than once; once but not more than once; and once and only once.

Ensuring exactly-once semantics in a request-response interaction pattern is very difficult, because all the components involved in the interaction must work correctly even in presence of failures, which is difficult to ensure when there is no global knowledge that could be used to facilitate recovery. Due to the complexity involved in providing exactly-once semantics in request-response interactions, the importance of this reliability semantics in many businesses and peoples' lives, and the absence of appropriate solutions, we propose a knowledge base regarding the protocols offering strong reliability semantics, their characteristics and complexities.

One possible approach to build this kind of knowledge base is to collect information about existing protocols from real world applications, by analyzing their architecture

and implementation details or by studying their specifications. In general, this is an exhaustive and impractical task, first because many critical applications do not provide the necessary details about their systems, and second because we would be left with no coverage guarantees regarding all possible protocols. For these reasons, we chose a different approach, which is based on the formal definition of all reliable request-response protocols. To create the protocols, we define a set of valid actions for client and server, generate all possible sequences by interleaving the client and server actions, eliminate sequences that are invalid (from a reliability perspective), and organize the valid sequences based on their similarity. We then analyze and classify the protocols according to their reliability semantics, considering all crashes that may occur, and their memory requirements. Finally, we analyze several implementations of real online services that match protocols of our taxonomy.

The remainder of this chapter is organized as follows. Section 6.1 presents an overview of the approach used for generating reliable protocols for exactly-once request-response interactions and building the taxonomy. Section 6.2 describes the necessary definitions and assumptions used to generate all possible protocols. Section 6.3 explains how the protocols are generated, filtered, and organized in a tree and Section 6.4 presents the analysis and classification of the generated protocols based on two important aspects, reliability semantics and memory requirements. Section 6.5 presents the applicability of our taxonomy to real services. Finally, Section 6.6 concludes this chapter.

6.1 Approach Overview

Our approach to generate a comprehensive set of reliable request-response protocols includes the following key steps, which we overview in the next paragraphs and describe in detail in the following sections.

1. Stating definitions and assumptions;
2. Generating, filtering, and organizing protocols;
3. Analyzing and classifying protocols.

The first step consists of stating definitions and assumptions, which precisely define the scope of the work. They will later help setting rules or restrictions for obtaining the

final set of valid protocols. The definitions include basic aspects that should be defined "a priori", such as, a set of actions for client and server that might take place in an interaction. For example, in a request-response communication, the client generates a request, sends the request to the server, the server processes the request, and sends the result back to the client. The main challenge in this step is to define a minimum but complete set of all necessary client and server actions.

The assumptions help regulate the overall process of generating protocols, and the main challenge is that the assumptions and definitions must be made based on real considerations. Otherwise, the generated protocols cannot solve the problems of real world applications. As an example, we assume that servers eventually recover, but accept the possibility that clients may not recover (e.g., browsers). With this type of clients (i.e., crash-stop clients), the guaranteed reliability semantics that can be achieved by a reliable exactly-once protocol is at-most-once, because there is no way to ensure exactly-once when a client crashes. Thus, we consider, not only crash-recovery clients, but also crash-stop clients to generate both exactly-once and at-most-once protocols. This is a typical case that emulates a real scenario, and illustrates the necessary mapping to real environments.

In the second step, we generate all possible sequences, by alternating client and server actions. Here the main issue is that the number of the sequences generated by interleaving the client and server actions will be infinite because the client and server could keep exchanging messages forever. Hence, the main challenge, in this stage, is to define some restrictions to safely (i.e., the restrictions must not prevent the creation of reliable protocols that might exist in real applications) limit the number of the protocols.

We then eliminate all invalid (or unreliable) protocols, by applying some rules, which are defined by examining all scenarios that may lead to the violation of the desired reliability semantics. We then organize the remaining protocols, based on the similarities found in their sequences of actions. Finally, we classify the reliable protocols according to their reliability semantics (at-most-once vs. exactly-once) and memory requirements (e.g., bounded vs. unbounded).

6.2 Definitions and Assumptions

We assume the presence of a client, a server and a channel, all of which might be faulty. Client and server execute actions, such as generating a request or performing some computation, and each client or server may execute multiple consecutive actions. The term **protocol** refers to a sequence of interleaving client and server actions. For each of the interleaving points in a protocol, a message exchange is needed. We give the initiative of sending a message to the client and make the server acknowledge. Hence, the message exchange starts with the client sending a request to the server, then the server replies with another message. The exchange of messages continues until the protocol is completed. If the protocol terminates with a client action, no further message is required. If the protocol terminates with a server action instead, the server acknowledges to allow the client to remove the state associated with this interaction (this also lets the client to resend its message (e.g., request) if it misses a timely acknowledgment (e.g., response)).

We assume the following **crash modes**. The server must always recover from crashes; however, the client may or may not recover (i.e., the client may be either crash-recovery or crash-stop). Thus, we expect that a server that failed is recoverable, even if this implies manual intervention; but, we also expect that a client may not recover, which is also a reasonable assumption. For instance, in the case of web browsers, the user may simply give up on a particular interaction with a server. To be able to recover from crashes, crash-recovery servers and clients use stable storage to save their actions, so that the recovery process can resume from the last saved state. In the case of crash-stop clients, all data can be kept in volatile memory instead of stable storage.

Channels are assumed to be unreliable but fair, which means that they may lose messages, but will deliver a message that is sent a sufficient number of times. We also assume that the channel does not change the content of messages.

A subtle problem, that is of utmost importance in the context of this chapter, has to do with the ordering properties required from the communication channel. Some well-known protocols, such as the User Datagram Protocol (UDP), are inherently not FIFO; however, even TCP does not really offer a fully FIFO channel to applications. Consider the case where a client sends a request and later gets an exception informing that the TCP connection is no longer working. If the client tries to open a second TCP

connection *and* the server does not close the first connection on time, old messages may reach the server out of order. Likewise, in HTTP, a client may discard an HTTP session, while messages are still in transit. If the client creates a new session (e.g., the user restarts the browser or logs in again), before the server deletes the old one, the server may re-execute a request. Although careful implementations can avoid reordering problems, we believe that it is worthwhile to run exactly-once request-response protocols that are guaranteed to work over non-FIFO channels as well. Thus, we assume that the channel may reorder messages (i.e., it is non-FIFO).

Finally, we assume that the channel, the crash-recovery client, and server eventually work in a fault-free period for a sufficiently long time to complete the interaction. In fact, without this assumption, achieving exactly-once semantics is impossible (Fekete et al., 1993; Halpern, 1987).

With the goal of generating reliable protocols, we define a set of possible **client and server actions** that will be later aggregated and combined in sequences. Table 6.1 presents the complete set of client and server actions that are relevant for generating the reliable protocols. Each line of Table 6.1 includes a symbol that represents the action, the endpoint where that action takes place, and a short description of the meaning of the action. The client actions include: generating a request that will be later sent to the server (g); performing computation using volatile or stable storage (c); atomically saving a response received and any system state changes to stable storage (.); releasing all state related to a request from memory (r).

The client may need more than one (c) action if it needs to exchange multiple messages with the server. This action may result in changes to the system state, thus in this case crash-recovery clients must atomically perform computation and save the changes and responses in the storage. Note that we do not assume atomic save and send actions, which can be quite complex to implement. This means that saving to stable storage is one action and sending a message is a different separate action. The release (r) ends the request, by releasing any memory references associated with it (e.g., using a `free()` or equivalent operation) and also, in crash-recovery clients, by deleting the related state from storage. After the client atomically saves a response to stable storage (.), when it crashes and resumes it will not generate (and send) the original request again, since it already has the corresponding response in stable storage.

As we can see in Table 6.1, the server executes slightly different operations. When possible, we use capital letters to distinguish the server from the client. The "C" action refers to perform computation using volatile or stable storage that may or may not result in the generation of a message for the client; the ";" action refers to atomically saving a response and any system state changes to stable storage; and "D" refers to deleting all state associated with a given request from stable storage. Note that this latter action differs from the release operation (r), where the client releases all references to a given request (although crash-recovery clients may also optionally delete state from stable storage). Since a server must recover from failures, all state associated with a request is saved in stable storage, thus the "D" operation (used, for instance, when the server has been informed that the client received a response) must delete that information from the storage.

TABLE 6.1: Client and server set of actions

Action	Endpoint	Description
g	*client*	Generate a request.
c	*client*	Perform computation using volatile or stable storage.
.	*client*	Atomically save a response and any system state changes to stable storage (only in crash-recovery clients).
r	*client*	Release all memory references to a request. Crash-recovery clients may delete all state related to the request from stable storage.
C	*server*	Perform computation using volatile or stable storage, which may result in generating a response.
;	*server*	Atomically save a response and any system state changes to stable storage.
D	*server*	Delete all state related to a request from stable storage.

There is no symbol to identify a send operation because each interleaving point in the protocols means that a message is sent from one endpoint to another. A final remark regarding the unique identifier of the requests is necessary, at this point. Although we could expect the "g" action to generate the identifier, we found cases where the server creates such identifier (e.g., in a "C" operation).

In Table 6.2, we identify the actions that may require the use of stable storage, with respect to crash-stop clients (in at-most-once interactions), crash-recovery clients (in exactly-once interactions), and also the server. In Table 6.2, "Yes" means that stable storage must be used and "No" means that it is not used. The term "Maybe" means that

it may or may not be used, and in either case, the desired goal (guaranteed exactly-once or at-most-once semantics) is achieved. Note that, in addition to the normal case (i.e., no failure), we also consider (when applicable) the action in the context of a recovery procedure and this may have implications on the use of stable storage.

As we can see in Table 6.2, the request generation ("*g*") does not need stable storage in crash-stop clients. Their nature disallows them from saving requests or state (i.e., at least with the goal of using them for recovery) and, they will not read the request or any other data necessary to generate it from stable storage. In the case of crash-recovery clients, this operation may need to read some data (or even the entire request) from stable storage to regenerate the request, upon recovery. Since reading from stable storage is not mandatory (other mechanisms may be used), we use "Maybe" for exactly-once semantics in crash-recovery clients.

TABLE 6.2: Storage actions for the reliable protocols

Action	Crash-stop client (at-most-once)	Crash-recovery client (exactly-once)	Server
g	No	Maybe	—
. or ;	—	Yes	Yes
c or *C*	Maybe	Maybe	Maybe
r or *D*	No	Maybe	Yes

The "." action is used only in crash-recovery clients (to save state for recovery purposes), and not used in crash-stop clients. When recovering from failure, crash-recovery clients will go back to the previous preserved state, thus, the need for stable storage is mandatory in this type of clients. The server must also keep the state in stable storage to be able to safely recover after a failure. For example, if a failure has occurred after the ";" action, upon reception of the same request, the server must not do computation ("*C*") again, thus, the ";" action always requires stable storage to send the response of the repeated request without re-execution.

The computation actions "*c*" and "*C*" can either involve computation using stable storage or simply in-memory manipulation of data, this depends on the specific applications. Note that these actions are just generic computation and do not account for operations executed in a recovery procedure. For this reason, crash-stop clients, crash-recovery clients and server all are marked with "Maybe" for the computation action. In general, with crash-stop clients, since they do not try to recover from endpoint crashes, the

use of stable storage always depends on the application's business logic and not to the recovery procedure.

In crash-stop clients, the "r" action simply releases the references to a request from memory while in crash-recovery clients it may also delete state associated with the request from stable storage. The "D" also releases references from memory and permanently deletes the associated state from stable storage. Note that the endpoints stop re-sending a message once they delete all resources associated with a request.

6.3 Generation and Organization of Reliable Protocols

In this section, we present the process from generation to organization of all possible reliable request-response protocols, through the following key steps:

- **Generation of all protocols:** Starting with a set of basic constraints (e.g., the first action must be the generation of a request by a client), we define all possible sequences of actions for both client and server. We then exhaustively combine them in interleaved sequences (i.e., protocols based on alternating client and server sequences of actions).
- **Removal of invalid protocols:** We remove invalid protocols, which, for instance, cause multiple processing of the same request or lead peers into inconsistent states.
- **Organization of the protocols into a prefix tree:** The generated protocols are organized in a tree structure (each node of the tree is a protocol), according to the similarity of their actions. For instance, two protocols will be placed under the same parent if both share the same initial actions.

6.3.1 Generating the Protocols

The first obstacle to generate all possible protocols from interleaving the client and server actions is that their number is infinite, as the client and server could keep exchanging messages forever. To limit the number of protocols, we consider the following restrictions:

- The client must start with a "g" operation;
- The client and server can only save once ("." or ";") — this minimizes the number of operations that involve stable storage;
- Once both client and server save, they do not engage in more message exchanges (except possibly for releasing memory and deleting state);
- Their interaction prior to saving is limited to two rounds of exchanges, with the exception of faulty runs, in which client and server may repeat messages, thus engaging in more than two rounds of exchanges;
- The server does not perform any computation after saving response and deleting state.

The above-mentioned restrictions allow us to define "$gcc.cr$" as the largest sequence of client actions. A client may not execute some of the actions in this sequence (only the "g" must always be present). For example, crash-stop clients do not need to save data (.). Computation steps (c) and releasing memory (r) are also not mandatory. The final "cr" instead of simply "r" may provide a few more options for deletion.

A crash-recovery client may re-generate a given request (or the same identifier) if it crashes between "g" and "." (i.e., by returning back to the "g" operation). Otherwise, if the crash occurs after ".", the client will return to ".", but does not need to continue the protocol because the crash would work as a release (r). Thus, considering "$gcc.cr$" as the largest sequence of client actions, we can have the following variants: $CLIENT\_SEQ$ = {"g", "$gcccr$", "gc", "$gc.r$", "$gc.$", "$gcc.r$", "gcc", "$g.cr$", "$g.c$", "$gcc.$", "$gcc.cr$", "$g.$", "gr", "$gc.cr$", "$gc.c$", "$gcc.c$", "gcr", "$g.r$", "$gccr$", "$gccc$"}. As an example, a sequence like "gc" can be applicable to crash-stop clients, as they simply do computation, without saving state. On the other hand, a crash-recovery client could use "$gc.$", which means that the client does some computation (c) and saves the state or the result of the computation (.).

Considering the above-mentioned restrictions, we can define the largest sequence of server actions as "$CC; D$". Again, a server may not execute all actions, which results in the following possible sequences: $SERVER\_SEQ$ = {"$CC;$", "$C;$", "$CC; D$", "$C; D$"}. As we can see, in some cases the server does not delete state (e.g., when memory is unbounded).

In order to enumerate all protocols, we compute the Cartesian product between the client and server set of action sequences ($CLIENT\_SEQ$ and $SERVER\_SEQ$). For each element in the resulting set we generate combinations between client and server actions. These combinations follow two rules: 1) The first action must be executed by the client and followed by a server action; 2) The order of client and server actions must be the same before and after the combinations (although they may be interleaved). For instance, given the element $(gc., C;)$ generated by the Cartesian product, we may have the protocols "$gCc.;$", "$gC; c.$", "$gCc;.$" (among others), but not "$Cgc.;$" or "$g; Cc.$". An alternation of client and server actions implies an exchange of one message (e.g., $g \rightarrow C$). Thus, for each set in the Cartesian product, the number of protocols started by "g" and followed by the first action of the server is $\binom{ls+lc-2}{ls-1}$, where ls and lc are the length of the server and client sequences, respectively. Considering the entire product set we have 1646 possible protocols.

6.3.2 Eliminating Invalid Protocols

After generating the protocols (as described in the previous section), we eliminate invalid or redundant protocols. Although this step could be integrated in the first step, we have separated them to decrease implementation complexity. The (non mutually exclusive) rules for elimination are the following:

(a) The action "." must not happen before ";", because at save time (.) the client cannot be sure that the server will commit; for the same reason, if "." does not exist, the client cannot release (r) before ";" ("$gC.;$" or "$gCr;$" are incorrect);

(b) The server cannot delete (D) before the client saves (".") (e.g., "$gC; cD.$" is incorrect) and also before the client uses the result with "c" or "r" (e.g., "$gC; Dc$" is incorrect), because the client may re-send or regenerate the request causing a second execution on the server;

(c) Sequences that repeat actions on the same side, such as "$.c$", "cc", "$; C$" are useless and can be removed. We also remove sequences finishing with a "c", because the final c is implicit (the protocol must finish with a reply from the server and the client can continue with any computation from that point on).

Restrictions *a*), *b*), and *c*) respectively delete 855, 254, and 507 protocols, for a total of $1646 - 855 - 254 - 507 = 30$ protocols.

6.3.3 Organizing the Valid Protocols

In this step, we organize the protocols in a prefix tree, based on the similarity of their actions. Those with similar initial actions will be under the same branch (e.g., "$gC;.$", "$gC;cD$" , and "$gC;r$"). Figure 6.1 presents the prefix tree of protocols. As we can see, the prefix tree has three main branches that correspond to three families of protocols: "$gCc;$" on top, "$gCcCc;$" in the middle, and "$gC;$" below (which is similar to Request-Reply (Spector, 1982)). At level 2 (i.e., the direct children of the root), all protocols reach the point where the server commits and saves the state to stable storage (";") and may send a message to the client, if necessary. Then, level 3 adds one of three options for the client: ".", "c" or "r". From that level on we have different deletion variants. Their suffixes are "r", "D", "rD" and "Dr" and are common to all families.

6.4 Analyzing and Classifying the Reliable Protocols

In this section, we describe how the protocols presented in the prefix tree can be classified in meaningful categories. Since we considered both crash-stop and crash-recovery clients, one important aspect for classification is the reliability semantics offered by each protocol. Given the explanation regarding non-FIFO channels in Section 6.2, the classification of the protocols according to memory requirements is another important aspect in request-response interaction pattern with unreliable and non-FIFO channels. One of the difficult problems to address in this kind of interactions is the deletion of memory (or stable storage) used to keep information regarding the messages, as an improper deletion of a given message may easily violate the desired reliability semantics. Thus, the final part of this section presents and discusses time-based solutions for deleting memory when implementing the protocols.

6.4.1 Reliability Semantics

We first classify the protocols according to their reliability semantics, and distinguish them as exactly-once or at-most-once. The protocols where crash-stop clients do not

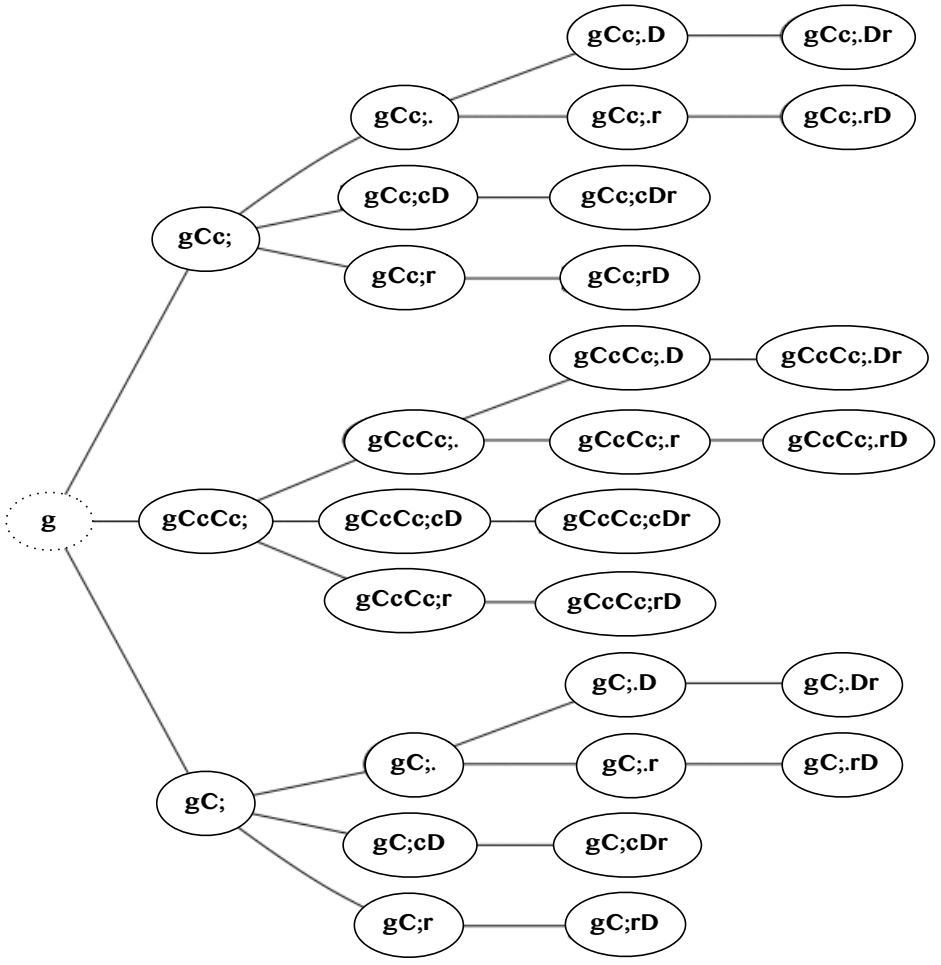

FIGURE 6.1: Organization of the reliable protocols in a prefix tree

save response and state (i.e., without ".") are shown as octagon nodes in Figure 6.2. Since the client does not save state, the exactly-once semantics can be violated when the client crashes, thus these nodes represent at-most-once protocols. The remaining nodes, where crash-recovery clients (that save state changes) are included, are exactly-once. As visible in Figure 6.2, the suffixes for at-most-once and exactly-once protocols are similar among families.

6.4.2 Memory Utilization

The second part of the analysis considers memory requirements and deletion of state in each protocol. Analysis according to memory requirements is quite important in reliable request-response interactions with unreliable and non-FIFO channels. In fact,

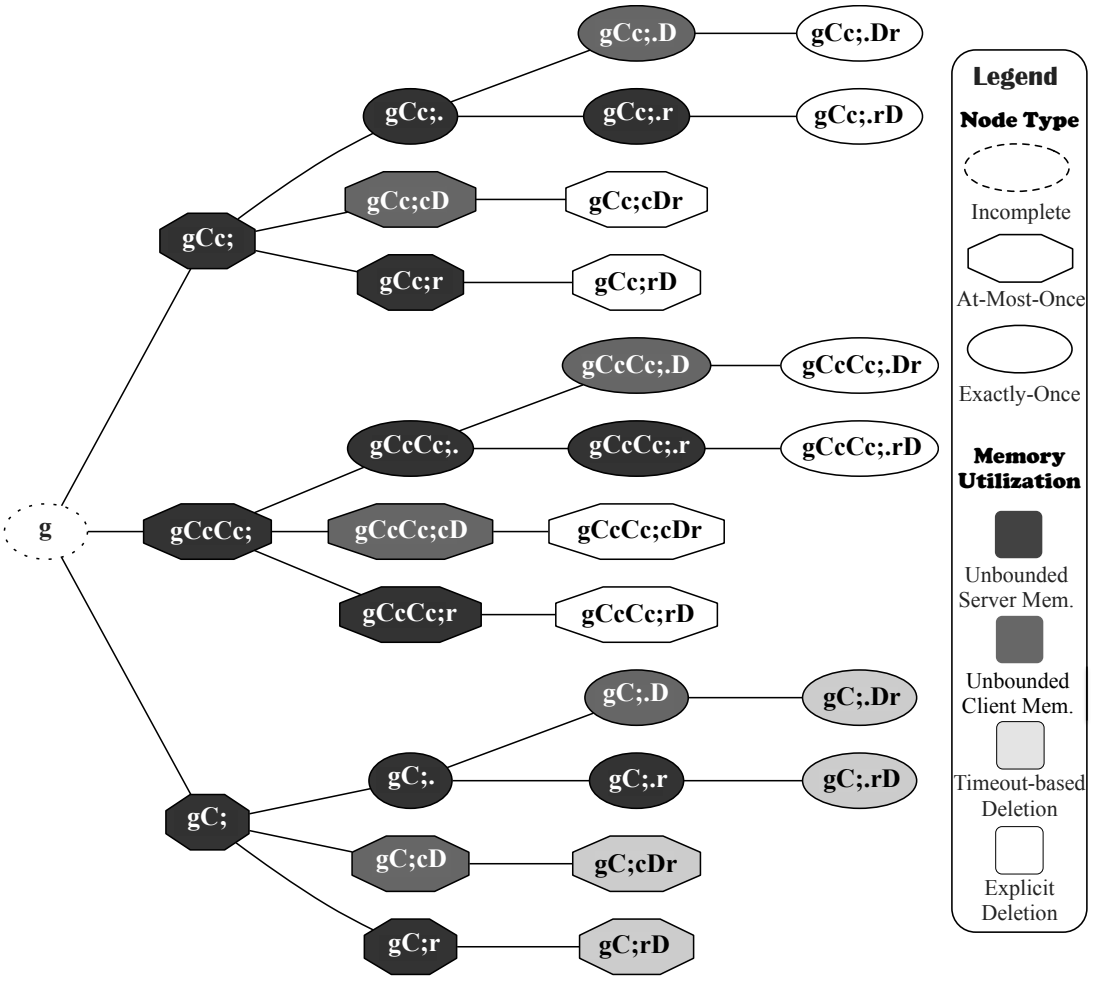

FIGURE 6.2: Taxonomy of exactly-once and at-most-once protocols

one of the main challenges to address is the deletion of memory used to keep information regarding the messages, as an improper deletion of a given message may easily violate the desired semantics. Thus, the following paragraphs discuss timeout-based solutions for deletion when implementing the protocols.

To avoid re-execution of a request, the server needs to save the state. It can delete the state whenever it knows that the client received the response and no duplicate request will arrive later on. The implementation of this scenario is easy using a FIFO channel, but most channels are non-FIFO. Here, we need to emphasize the common TCP and HTTP FIFO fallacy, as both, TCP and HTTP plus client sessions, may reorder messages in the presence of channel failures or client failures, respectively. This may break naive implementations of exactly-once request-response interactions.

If we consider a non-FIFO channel, it is impossible to know (excluding specific cases) if a duplicate request will arrive later or not. This turns deletion of state into a problem for the server, as it needs to avoid re-execution. Consider the following case with the "$gC;.D$" protocol: the initial "Request" (transition from g to C, written as $g \rightarrow C$) of the client does not pass through the channel. The client re-sends the request, receives the reply and acknowledges ($. \rightarrow D$), letting the server delete all associated state. Then, the first request finally arrives at the server, causing an undesired re-execution. On the other hand, memory conservation demands for the deletion of state concerning processed and completed requests on the server. Hence, we may say that deleting state is challenging, because: 1) channels may deliver repeated messages out of order and there is no guarantee that the server will not receive the same request after deleting its state; 2) even in protocols where the client sends a deletion order to the server ($\rightarrow D$) (e.g., "$gC;.D$"), deletion may not occur, because the client may crash before sending the message.

6.4.3 Timeout-Based Deletion of Interaction State

The solution for the problem described in the previous subsection is to use timeouts. Timeouts can help releasing memory in protocols where the deletion of state cannot be easily guaranteed (e.g., due to a client failure). In fact, clocks can be used for harmlessly deleting the state in the "$gC;$" family of protocols; obviously, only if they have bounded drifts and if client and server synchronize beforehand, using some mechanisms like NTP (Mills, 1992), or Christian's algorithm (Cristian, 1989).

In a timeout-based scenario, at the moment "$g \rightarrow C$" the client sends a timestamp with the request and keeps both. The server receives the message and associates a timeout past this timestamp. Once the timeout expires, the server may delete all data associated with the request and refuse to re-execute afterwards. The client can help the server releasing memory earlier if it knows that there are no pending messages in the channel for that request (e.g., because it received all replies). This corresponds to the "$\rightarrow D$" part of the protocols. Also, the server may use the timeout to delete the state, even if the client does not send the deletion message. In the next paragraphs, we prove that this "timeout-based deletion" of server state prevents re-execution of requests.

Theorem 1. In the "$gC;$" prefix protocols with timeout-based deletion, the server executes each request at most once.

Proof. For a request sent by the client at time t_c, the server sets a timeout that expires at $t_c + \Delta$, where Δ is the duration of the timeout. Before this timeout, the server has the state of the request and does not execute it a second time. After the timeout, the server discards the requests. Since the client cannot change the timestamp t_c of further copies, no duplicate execution can occur. □

Δ should be much larger than clock skews and channel delivery time, otherwise the request could fail to reach the server before $t_c + \Delta$ according to the server's clock. One should notice that if client and server clocks may differ by δ, the channel must not take more than $\Delta - \delta$ to deliver the message to the server or else the request may not get there in time.

The initial exchange of the "$gCc;$" family can offer better solutions for the deletion problem. When the server first replies ("$gC \rightarrow c$"), it can insert a deadline for the commit ("$gCc \rightarrow;$"). If duplicate messages of the same request arrive, the server will respond with the same timestamp (or even with the reply, if available). Any client commit order must include this timestamp. If the client fails to meet the deadline, the server aborts the request, deletes its state, and refuses to commit thereon. The server will also not commit if it has no previous information on the commit request. In this family, the server can delete the state as soon as it receives a delete order ("$\rightarrow D$"), which we name as "explicit deletion". Nonetheless, to limit memory utilization, the server may delete the state if the client fails to commit within a time frame Δ. For this reason, even crash-recovery clients must ensure that they do not repeat requests that arrive at the server spaced by an interval larger than Δ.

Theorem 2. In the "$gCc;$" prefix protocols with explicit deletion, the server executes each request at most once.

Proof. Assume that the server committed orders with timestamps t_{s_1} and t_{s_2} for the same request. Note that the server does not lose committed state even if it crashes. Hence, the server must have explicitly deleted the state of the request. If this resulted from a client delete message ("$\rightarrow D$"), by definition of the protocols, the client must not

generate new commit orders (even if it crashes). Any other commit order, must still have been in the channel, and would therefore not match any future timestamp set by the server for this request, in case the server received an old "$g \rightarrow C$" message, also still in the channel. If the server deleted the state after time $t_{s_1} + \Delta$, then $t_{s_2} > t_{s_1} + \Delta$. □

Demonstrating the at-least-once part (to ensure exactly-once) depends on many implementation details. We already assumed a fair channel and a crash-recovery server, but we need the following additional properties: 1) the client must recover from crashes or it must not fail; 2) the server must get the request within the timeout Δ; and 3) client and server must agree to execute the protocol. For instance, in banking operations, banks may request security codes before committing. If the client fails to provide the correct code, the request will go unanswered. Nonetheless, if we assume that all the three previous conditions hold, we can discard the intermediate "Cc" or "$CcCc$" operations before commitment and restrict our analysis to the "$gC;$" as the head of family (because the structure of the tree is the same in the three main families mentioned). In this case, the evaluation is simple: if the client crashes before sending the request, it will resume to re-generate and resend the request, according to the definition of "g". Since the channel is fair and the server is crash-recovery, it will receive and commit the request in ";" at least once.

We can now explain the different gray tones in Figure 6.2, from darker to lighter. Protocols that do not release server memory are the darkest; protocols that do not release client memory have the second darkest tone; protocols that release client memory but depend on timeout-based deletion at the server are next (these belong to the "$gC;$" family); finally, protocols that delete client memory and enable explicit deletion of server memory are white and have solid lines. Note that in this latter case, the server should also use timeouts, to clean data from clients that crash and do not recover.

As previously discussed, both TCP and HTTP with sessions can easily fail to provide FIFO ordering, when connections or endpoints crash. To avoid depending on properties that the channel cannot easily offer (which must be carefully added by the programmer), reliable protocols should resist to message reordering even when the server deletes the state of requests. We showed three families of exactly-once protocols that can do this, with different tradeoffs. The extra initial exchange of messages in the "$gCc;$" family (or "$gCcCc;$") enables immediate deletion of server data, at the cost of requiring a previous round of messages *per* request ("$g \rightarrow C \rightarrow c$"), whereas the "$gC;$" family requires

synchronized clocks and does not allow immediate deletion of data, usually forcing the server to keep data up to a timeout.

6.5 Reliable Protocols in Real Services

The protocols presented in our taxonomy match different types of real software and on-line services. The simplicity of the shortest family ("gC;"), makes it appealing to be used in many cases. For instance, the Exactly-Once E-Service Middleware (EOS) (Shegalov et al., 2002) uses the protocol "$gC;$." of this family. In the Shegalov et al. implementation, the server does not delete the state and the client, which is crash-recovery, saves all the requests but not their responses. Thus, the client has to send all the requests again upon recovery. Since the server does not delete the state, exactly-once is achieved.

Protocols with deletion, such as "$gC;.D$", which correspond to Spector's RRA (Spector, 1982), are implemented in our solutions for at-most-once and exactly-once middleware of Chapter 5. We are also aware of a metropolitan-scale ticketing system, where clients are pieces of equipment that periodically send data to a central database. Since this set of equipments does not grow too large, the server may keep a version number per client indefinitely. The server does not need to delete the last version number of each client's data.

We can find implementations similar to the sequences starting with "gCc;" in on-line services, although their main goal might be to prevent over-usage of the service and not to prevent duplicate executions of the same request. For example, many sites use captchas to prevent automatic submission of forms. The user requests a page (g), the server computes the page and sends a captcha (C), the user enters the data and the captcha text (c). Then the server processes the data and replies to the client. These captchas, which can be served as a unique identifier of invocations, are usually associated to a timeout window, which is similar to the Δ timeout mechanism of the "gCc;" family discussed earlier. Changing the identifier of a request is a pitfall that may disrupt any delivery guarantees.

We found one implementation of a "gCc;" protocol where the captcha changes on each reload, thus making all requests seem different and breaking any possibility of ensuring exactly-once semantics. In a cell phone operator (uzo.pt), we found a "gCc;"-like

implementation of a service providing on-line text messages (SMS). Users must fill the target phone number, the text, and a captcha that may be used as a unique identifier. The SMS submission page recreates the captcha on each reload of the page (possibly for security), which ends up providing a different identifier for the same message. This approach allows neither the user to understand if the request was successful, nor the site to correlate HTTP requests[1].

We tested the uzo.pt service as follows. After submitting an SMS and receiving a response, if we press the back button of the Safari 7.0.3 web browser and accept to reload the page, the forms are entirely filled as before, but the captcha text is not the same anymore (it changes on reload). This happens regardless of the response to the previous request. If we switch off the network and the response is an error message from the browser, the behavior is exactly the same, once we turn the network back on. If a page reload returned the same captcha, we would know that the previous message did not get through. This type of implementation is simply best-effort, from the point of view of the reliability semantics.

Longer sequences serve to provide better protection, by using the additional interactions to request security codes. Banking systems tend to use these more complex protocols, often of the "$gCcCc;$" family. In fact, once the client requests a money transfer (g), the bank will ask for the details (first C), the client will fill them in (first c), the server will respond with a test (second C) and the client will reply to that test (second c). After this point, the server can commit (;) and respond, to let the client change page (r). Only then should the server delete the request (D).

This can ensure an at-most-once semantics, by providing some clues to the user about the success of his previous attempts. Note however that the first goal of the developers is likely to be security[2].

We also tested an on-line banking site (name not disclosed due to security issues), to observe to which extent they force the security code repetition. We can refer that within the same login session, the bank keeps requesting the same code until one uses it. This lets the user know if the request got through and enables the server to filter

[1] We must emphasize that the developers of this system and web site may well have as their single goal to prevent automatic submission of messages. Eensuring at-most-once might not have been their primary goal. Nevertheless, given the nature of the service, we argue that the invocation semantics is important here.

[2] Again, the first goal of the developers is likely to be security, but it is difficult to pick a better case to demand for invocation reliability.

duplicates. However, if the user logs out and then logs in, or if he uses a different browser in a simultaneous login, the code will be different[3].

Many banks also include a case that our tree does not cover: the test may have two simultaneous messages to the client: a new page for the browser requesting credentials, and an SMS with a code to insert in the web page, to ensure that only a person with access to the cell phone can perform the transfer.

Another excellent study for the at-most-once semantics comes from big social networks, such as Facebook and Twitter. They do not delete server state, to ensure that each post is new. They apparently follow a simple "$gC;r$", where the "g" actually creates a unique message identifier (unlike the banking). However, we tried to submit the same message twice, faster than it would be possible to a human. For this, we wrote a browser extension in JavaScript that submits the same form twice within a configurable interval. With an interval of 10 ms, we managed to replicate posts in one social network (we omit its name for security reasons). Why exactly this happens is beyond the scope of this thesis, but hugely popular sites like Twitter or Facebook are typically backed by NoSQL databases that do not preserve all the ACID properties.

Some protocols used in real systems may elude our taxonomy. As we mentioned, banking sites may send two messages instead of one: one for the browser and a confirmation code for the user's cell phone. Many developers will also rely on hand-shaking, from TCP connections to HTTP cookies. They would first set up a session, before repeatedly invoking reliable operations. This sort of solution is halfway between the "$gCc;$" (or even "$gCcCc;$") and the "$gC;$" families, because it requires a *single* handshake, before invoking operations in *single* messages, possibly multiple times.

6.6 Conclusion

In this chapter, we presented an approach designed to generate a comprehensive set of reliable (exactly-once and at-most-once) protocols. The generated protocols are organized in a prefix tree and each node of the tree is classified based on the reliability semantics and memory requirements. We believe that this provides a detailed understanding of reliable request-response interactions and the challenges involved for

[3]Nevertheless, the bank had a security scheme that suspended the account of the user around the 5^{th} code without response.

ensuring exactly-once or at-most-once semantics, helping developers to build correct services. In summary, the main contributions of this chapter are as follows.

A Prefix tree of all reliable request-response protocols is presented, which is built upon a complete set of client and server actions. Both crash-stop and crash-recovery clients are considered, thus protocols either ensure exactly-once (i.e., with crash-recovery clients) or at-most-once semantics (i.e., with crash-stop clients). The prefix tree holds three different families of protocols that clearly match common real-world implementations, which might be used by developers for their future services and applications and also to understand if their current applications.

Moreover, an analysis of the protocols is presented by considering their use with unreliable and non-FIFO channels and with respect to memory requirements. Accordingly, some time-based solutions are presented allowing to safely delete the state of the interaction. This analysis and the time-based solutions can be vital for developers to implement reliable services according to specific requirements or resource restrictions.

Furthermore, we analyzed several implementations of real online services (a mobile phone operator, a social network application, and a bank) that match protocols of our taxonomy. The analysis showed the applicability of the taxonomy and pointed out several pitfalls in the implementation of these services.

Chapter 7

Experimental Evaluation and Discussion

In this thesis, we proposed several solutions for building reliable communication in distributed point-to-point applications. Our solutions address the main challenges of reliable stream-based applications (Chapter 3), one-way message-based applications (Chapter 4), and request-response conversation-based applications (Chapter 5). We also proposed a taxonomy of all possible reliable request-response protocols, offering exactly-once and at-most-once guarantees (Chapter 6). This chapter aims to evaluate these solutions.

The remainder of this chapter is organized as follows. Section 7.1 presents the general experimental setup. Section 7.2 presents the experiments and analyzes the results obtained for the stream-based solution using the Connection Handler design pattern. Section 7.3 presents the experiments and analyzes the results obtained for the Messenger, Trackable Messenger, and Reliable Messenger. Section 7.4 presents the experiments and analyzes the results obtained for the Exactly-once Middleware. Finally, Section 7.5 presents the evaluation performed on the taxonomy of reliable request-response protocols.

7.1 Experimental Setup

We implemented all the solutions proposed in this thesis in Java. Our implementations of reliable stream-based, message-based, and conversation-based solutions are freely available online and are respectively named **FSocket**[1] (fault-tolerant socket), **FTSL**[2] (fault-tolerant session-layer), and **EoMidd**[3] (exactly-once middleware).

In our tests, we use two versions for each application: a reliable one, using one of our reliability solutions, and an unreliable one, without any reliability mechanism. These applications contain three main operations, `Invoke1`, `Invoke2` and `Invoke3`. These operations receive a small string and return another small string (10 bytes). The major difference between them is that `Invoke1` replies immediately, `Invoke2` sleeps 1 millisecond (ms) before replying, whereas `Invoke3` sleeps 2 ms. Nevertheless, putting a thread to sleep and waking it up takes around 0.08 ms on the machine where we ran the server (and 0.15 on the client machine). To determine this number, we ran a single-threaded program that slept for 1 ms 1000 times. The reason why we used different sleep times essentially is to emulate scenarios where processing time is negligible or where there is some processing or access to a database involved.

We used two computers sharing the same Local Area Network (LAN), to run the client and server endpoints. We ran all the clients on a single process, using different threads on a Mac OS X, version 10.10.5, with a 2.4GHz Intel Core 2 Duo processor, 4GB of RAM and 3MB of cache. The server ran on a machine with Linux, version 2.6.34.8, with a 2.8 GHz Intel processor with four cores, 12 GB of RAM and 8 MB of cache. Refer to Table 7.1 for details.

Table 7.1: Systems used in the experiments

Endpoint	OS	CPU	Memory
Client	Mac OS X version 10.10.5	2.4 GHz Intel Core 2 Duo	4 GiB RAM, 3 MiB cache
Server	Linux version 2.6.34.8	2.8 GHz Intel(R) 4 Cores(TM) i7	12 GiB RAM, 8 MiB cache

[1] https://sourceforge.net/projects/fsocket/
[2] https://sourceforge.net/projects/ftsl/
[3] https://sourceforge.net/projects/eomidd/

The experiments carried out in this thesis mainly focus on four key aspects: correctness, performance, overhead, and implementation complexity. To evaluate **correctness**, we test the solutions, when they recover from crashes. To do so, we let client and server exchange data for 5 minutes (each test was repeated 100 times) and use `tcpkill` to cause connection crashes and `kill` to crash the client and server process at random instants during each test (three crashes per test). We then verify if communication is resumed without data losses, and whether the desired reliability semantics (e.g., exactly-once) is achieved.

To evaluate **performance**, we measure latency (round-trip-time of a data) and throughput (number of requests or operations per time unit). Latency is simply the round-trip-time of the request-response interaction, and includes the transmission time of the request, waiting time of the request before being served, processing time of the request, and transmission time of the response. To examine the latency of the proposed solutions, we send a request to the server and calculate the time taken from sending the request to receiving the reply from the server. We use the same approach even for the solutions given for one-way messaging in Chapter 4. In this case, we force the server to send an acknowledgment message to the client, and then we divide the time calculated for latency by two. All the results for latency are the average of 1000 trials. We also measure the latency for an unreliable application with the same operations to demonstrate performance degradation of the reliable version in comparison to the unreliable version. The latency degradation is computed according to Equation 7.1.

$$(Latency_{reliable} - Latency_{unreliable}) * 100/Latency_{reliable} \tag{7.1}$$

The throughput is defined as the number of messages processed in a unit of time (e.g., requests per second). To examine the throughput, we send a large number of requests (1000 in our tests) to the server without waiting for any response (a different thread takes care of that), and calculate the time taken from receiving the first request to sending the last reply (or to finishing the process of the last request, when the interaction pattern is one-way). We calculate the throughput degradation according to Equation 7.2.

$$(Throughput_{unreliable} - Throughput_{reliable}) * 100/Throughput_{unreliable} \tag{7.2}$$

To evaluate **overhead**, we measure the resource utilization in terms of memory and CPU. To examine the CPU and memory overhead, we set the client to send 100 requests per second to the server during 5 minutes, which we experimentally observed to be enough to show the usage of resources. Then, we ran the `ps` command to periodically read memory and CPU occupation on the server.

Finally, to evaluate implementation **complexity**, we measure three important complexity metrics, Lines of Code (LOC), Cyclomatic Complexity (CC), and Nested Block Depth (NBD) (Jorgensen, 2008). For Lines of Code, we simply count the number of lines in the source code of the solutions; for cyclomatic complexity, the number of linearly independent paths through the source code is measured (here it is applied to the methods of the classes); and for nested block depth, the average depth of nested blocks in the source code is measured (Fenton and Bieman, 2014). To perform these measurements, we used Metrics 1.3.6 (Sauer, 2013).

To minimize random and transitory effects on the experiments and possible warm-up periods, in the performance and overhead tests, we run 20 more tests and ignore the first 20 results. Moreover, we increase the number of clients from 1 to 1000, to evaluate the effect of concurrent clients on the performance and resource utilization.

7.2 Evaluation of the Stream-Based Solution

In this section, we present the experimental evaluation carried out on our reliable stream-based solution, named FSocket. This solution allows recovery from connection crashes, by using the Connection Handler design pattern, and enables reliable large-scale stream-based applications using the Multi-Threaded Acceptor-Connector design pattern, presented in Chapter 3. It works with legacy software and proxy, independently of the application layer protocol. Thus, there are several issues, including applicability, good performance, reliability, scalability, simplicity, and working with legacy software and proxies, that should be experimentally proven by the evaluation.

7.2.1 Demonstration of Applicability

To explore our solution in practice, we used FSocket in two existing servers, an FTP server and an HTTP server. The FTP server is the ANOMIC open source FTP

server (Christen). We called `ftANOMIC` (Fault-Tolerant ANOMIC) to the reliable version of this server, and we made its source code available online (Ivaki and Araujo, a). We also inserted FSocket into the Apache Tomcat 7.0.13 HTTP connector (Goodwill, 2002) included in JBoss AS 7.1.1 (Fleury and Reverbel, 2003).

To accomplish the modifications in these servers, we simply replaced every TCP `Socket` object by an `FSocket` object in their source code. We also used an equivalent passive handle in our implementation, called `ServerFSocket`, to replace all the `ServerSocket`s. Upon accepting a new connection, the `ServerFSocket` returns an `FSocket` 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` objects. These replacements are summarized below:

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

ServerFSocket serverFSocket = new ServerFSocket(port)
// instead of
ServerSocket serverSocket = new ServerSocket(port)

FSocket fsocket = ServerFSocket.accept()
// instead of
Socket socket =  serverSocket.accept()

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

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

In addition to these changes, we needed to do one more modification in the FTP server, as the server may listen on more than one port, to accept control and data connections. For self-containment, we briefly explain the active and passive modes of FTP servers. 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 the first TCP 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 FTP server in passive mode needed an important change. In this mode, although the server continuously checks on the command port (usually 21) for new control connections, it checks the data port for connections only once. This will cause a problem when the connection crashes, because the client's reconnection to the server will fail due to the lack of any listener socket on that port. To solve this problem, we forced the server to keep listening on the data port, until the data connection is closed.

7.2.2 Evaluation of Correctness

To verify the correctness of the Connection Handler design pattern, used by the stream-based solution, when recovering from connection crashes, we let client and server exchange data during 5 minutes (each test was repeated 100 times), and, as said, we use `tcpkill` to cause connection crashes at random instants during each test. We then verify if all data correctly arrive at the destination. The results showed no failures on the Connection Handler design pattern and its implementation for stream-based applications. For 100 repetitions of the test, we also observed that, while the first connection establishment to the server takes 15 ms in average, reconnection plus sending lost messages took an average of 26 ms, which, we believe, is quite fast.

To evaluate the correctness of our solution in compatibility with legacy software and proxies, we considered different HTTP client-server communication scenarios. In each scenario, we refer to reliable and non-reliable peers (i.e., client or server), respectively, as using or not using our reliable communication solution. The scenarios are as follows: 1) a reliable HTTP client communicating with a non-reliable (legacy) JBoss AS; 2) a non-reliable HTTP client communicating with a reliable JBoss AS; 3) a reliable HTTP client communicating with a reliable JBoss AS, without any proxy in the middle; 4) a reliable HTTP client communicating with a reliable JBoss AS via a proxy. Scenarios 1) and 2) are used to show that our solution is compatible with legacy and unreliable software; and scenarios 3) and 4) are used to show that our design pattern is able to tolerate connection crashes with and without proxies.

We first used a browser to generate HTTP requests for a set of typical web resources deployed in the non-reliable JBoss AS. We used those requests within our custom HTTP client and also used the responses as oracle for comparison with the responses obtained from the reliable JBoss AS during the tests. For each of the four scenarios, we let client and server exchange messages during 5 minutes (each test was repeated 10 times).

We observed that reliable and non-reliable peers were able to communicate perfectly in scenarios 1) and 2). To evaluate the ability to recover from crashes (scenarios 3 and 4) without and with proxy, we emulated connection crashes, and observed that all interactions worked correctly even in the presence of the crashes and all expected messages were correctly received.

7.2.3 Evaluation of Performance

To begin our performance evaluation, we first measured the latency and throughput of the application with three different invocations (`Invoke1`, `Invoke2`, and `Invoke3`), when using the single-threaded (plain) Acceptor-Connector design pattern for handling concurrent connections. We will use the results of this experiment as a baseline for comparison and analysis of the next evaluations, where the application uses the Multi-Threaded Acceptor-Connector design pattern to handle concurrent connections.

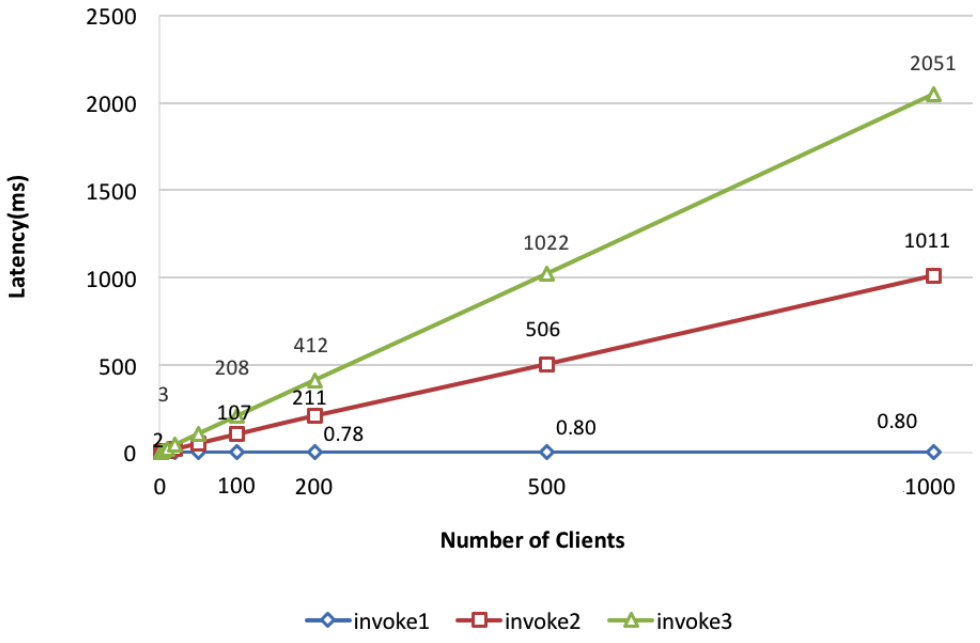

FIGURE 7.1: Latency of the application using the single-threaded Acceptor-Connector design pattern

Figure 7.1, shows the latency of the application. We get three plots with three different slopes for the three invocations due to the differences in their sleeping time (or

processing time). The slope for the first invocation, with processing time equal to 0, is close to 0, which means that the latency remains almost at the same level. When the processing time is zero, the server responds immediately after receiving each request, and therefore the waiting time of the next requests is almost zero too. In this situation, the latency only includes the transmission time of the request and response, which is not very influenced by the number of the clients.

For the other two invocations, the latency significantly increases with the number of clients that are continuously sending requests. As an example, for `Invoke3` the latency increases from 2.8 to 2051 ms, by increasing the number of clients from 1 to 1000. This happens because the server is single-threaded, which means that the waiting time of the requests will increase by increasing the number of requests arriving at the same time from concurrent clients.

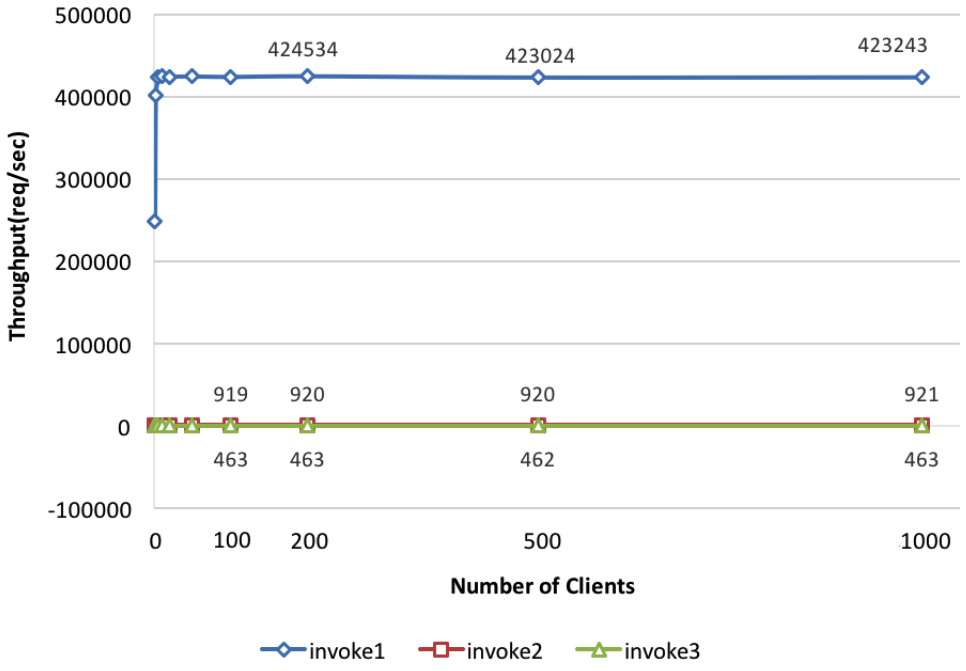

FIGURE 7.2: Throughput of the application using the single-threaded Acceptor-Connector design pattern

The last observation is that the transmission time is very low, because client and server share the same LAN. As previously said, the value shown in Figure 7.1 for the latency of the first invocation (`Invoke1`) can represent the round-trip transmission time, because the the processing time is zero. The comparison between the values shown for the

`Invoke2` and `Invoke3` confirms this observation, because the latency doubles (e.g., with 1000 clients, the latency is 1011 for `Invoke2` and 2051 for `Invoke3`) when the sleeping time doubles. It means that the latency, in our experiments, is not influenced by the transmission time, but rather influenced by the waiting and processing time in the server.

Figure 7.2 shows the throughput of the single-threaded application for three invocations. The results show different throughputs for the distinct invocations, which clearly is caused by their different sleeping time. Thus, the throughput is expected to be lowest for `Invoke3` and highest for `Invoke1`, which is proved by the results. With `Invoke2`, since its processing time is half of the `Invoke3`, the throughput is almost doubled (from 463 to 921 requests per second for 1000 clients). The throughput with `Invoke1` becomes extremely high (up to about 423000 requests per second), due to the fact that its processing time is zero.

The results show that the server behaves consistently during the experiments for these three invocations, as the throughput first increases radically and then stays almost constant, even when the number of clients grows. This is caused by the fact that, when the number of requests is low (less than the processing capacity of the server), the throughput varies with the number of the requests arriving on the server. When the number of the requests increases and saturates the processing capacity of the server, the throughput stays constant.

In the next step, we took similar measurements for an unreliable application that uses a TCP Socket and a reliable application that uses an FSocket for communication. Both applications do three invocations and use the Multi-Threaded Acceptor-Connector, for handling the concurrent connections. Figure 7.3 shows the latency of these two applications for different numbers of clients.

As the results for `Invoke2` and `Invoke3` show, the server behaves quite differently when it uses the multi-threaded Acceptor-Connector design pattern rather than the plain Acceptor-Connector. In both cases, latency increases very smoothly. In `Invoke3`, it just reaches 4 ms when the number of the clients is 1000. This is not the same for throughput, as we can see in Figure 7.4.

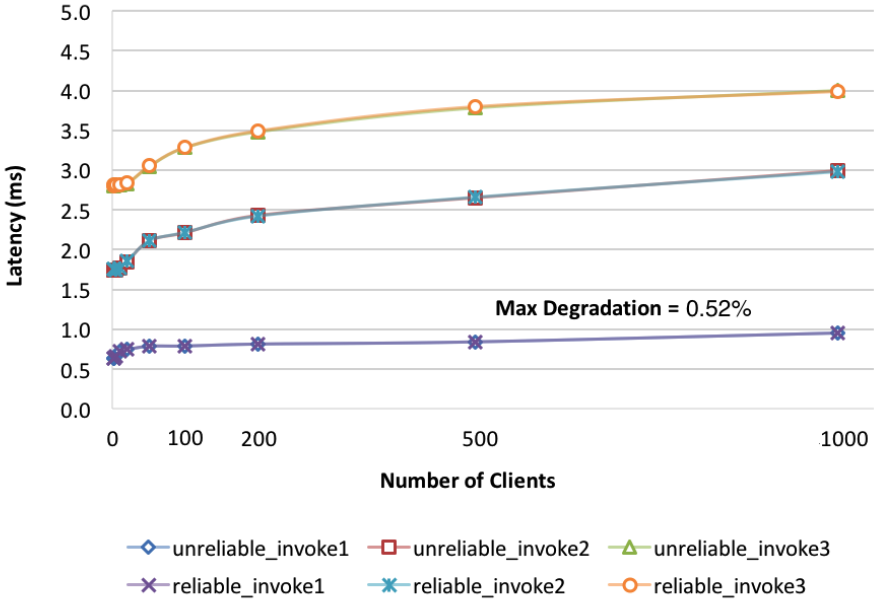

FIGURE 7.3: Latency of the unreliable (without FSocket) and reliable applications (with FSocket) using the Multi-Threaded Acceptor-Connector design pattern

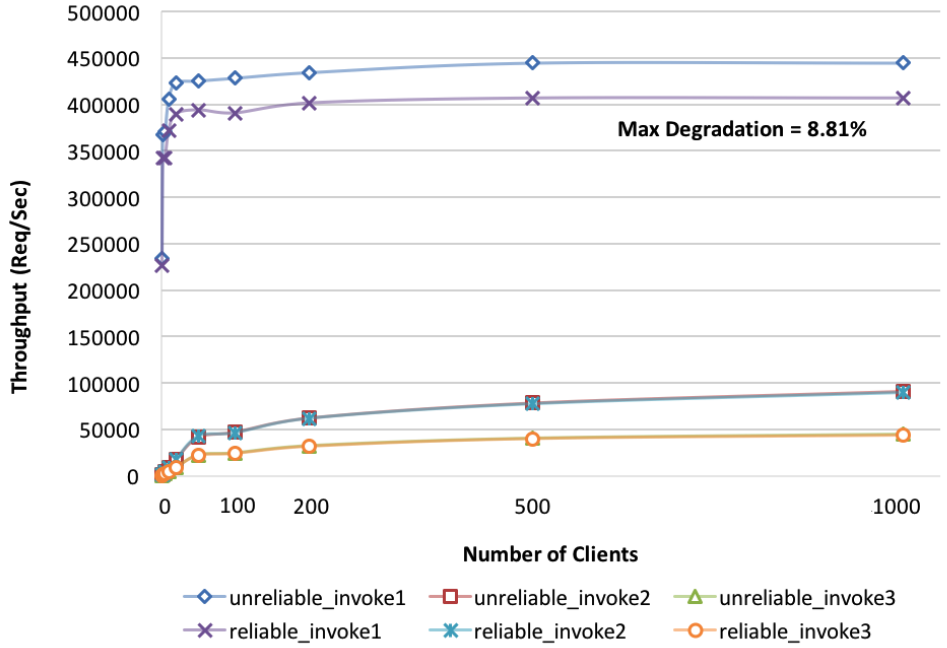

FIGURE 7.4: Throughput of the unreliable (without FSocket) and reliable applications (with FSocket) using the Multi-Threaded Acceptor-Connector design pattern

For both latency and throughput, the results show that the reliable application (with FSocket) is almost on par with the unreliable application (without FSocket) in performance. As shown in the plots, the maximum degradation we had in our evaluation for latency is less than 1 percent (0.52%). Throughput for the slower invocations, `Invoke2` and `Invoke3` is pretty much the same in both applications. `Invoke1` shows a higher degradation, but still below 10%. This invocation is indeed the worst case for measuring the FSocket's performance degradation, because `Invoke1` does nothing in the server, thus exposing all the communication overheads.

We also evaluated the performance with the HTTP client and server in the following four scenarios: 1) Non-reliable client and server interacting without proxy; 2) Non-reliable client and server with proxy; 3) Reliable client and server without proxy; 4) Reliable client and server with proxy. The proxy server used in our tests was Squid 3.1 (Saini, 2011).

Figure 7.5 and Figure 7.6 show the results obtained for latency and throughput. The plots also show the performance degradation for the reliable application, in comparison

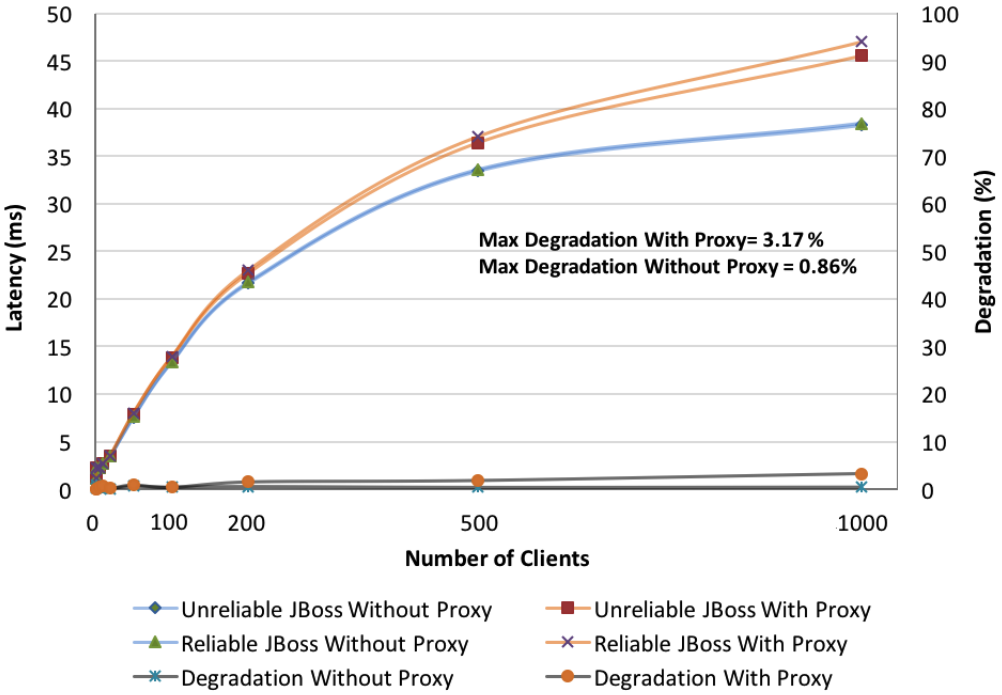

FIGURE 7.5: Latency of unreliable and reliable HTTP servers in the scenarios with and without proxy

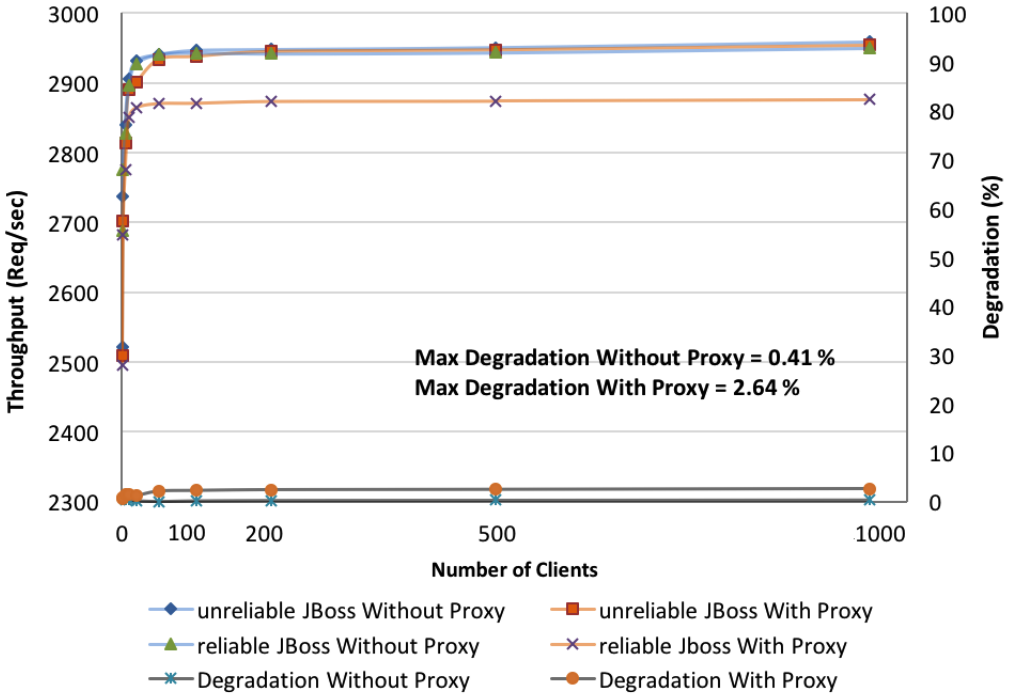

FIGURE 7.6: Throughput of unreliable and reliable HTTP servers in the scenarios with and without proxy

to the unreliable one on the right side vertical axis. As we can see, latency increases progressively in all cases; the same happens with throughput. In the scenarios with proxy, the latency is higher in comparison to the scenarios without proxy. The throughput in all scenarios increases rapidly in the beginning and then stays at the same level, as expected. The main observation is that the throughput of unreliable applications in both scenarios, with and without proxy, reaches the same level, although in the beginning it is slightly higher when there is no proxy. This does not happen for the reliable application. This difference is caused by the extra control connection and extra actions taken in FSocket when a proxy exists. However, the important aspect for both latency and throughput is that, when we compare the scenarios that use reliable peers with those that use the non-reliable peers, even with proxy, performance degradation show low values (about 3 percent). In fact, although we have all necessary mechanisms for reliable communication in place and in operation, performance degradation is quite small.

In addition to the above evaluation, we also used our reliable 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 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.

7.2.4 Evaluation of Complexity and Overhead

We used these complexity metrics to evaluate the implementation complexity of our solution. The measurements show that we used 485 extra lines of code in the application with FSocket in comparison with the plain unreliable application. In addition, the average cyclomatic complexity per method in both cases is around 1.87, while the depth of nested blocks of the reliable application is 1.4, close to the 1.26 of the unreliable application. Table 7.2 summarizes these numbers. These results demonstrate that providing reliability using our solution is quiet cheap. We took the same measurements for the FTP server with and without FSocket. 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.

Table 7.2: Implementation complexity of FSocket

Applications	Lines of Code (LOC)	Cyclomatic Complexity (CC)	Nested Block Depth (NBD)
With Plain TCP	572	1.873	1.26
With FSocket	1057	1.875	1.40

Regarding resource usage, we again increased the number of clients, and used each HTTP client to send 100 requests per second during 5 minutes. Figure 7.7 and Figure

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

7.8 show that the overhead in terms of memory (a) and CPU (b) is kept under acceptable limits.

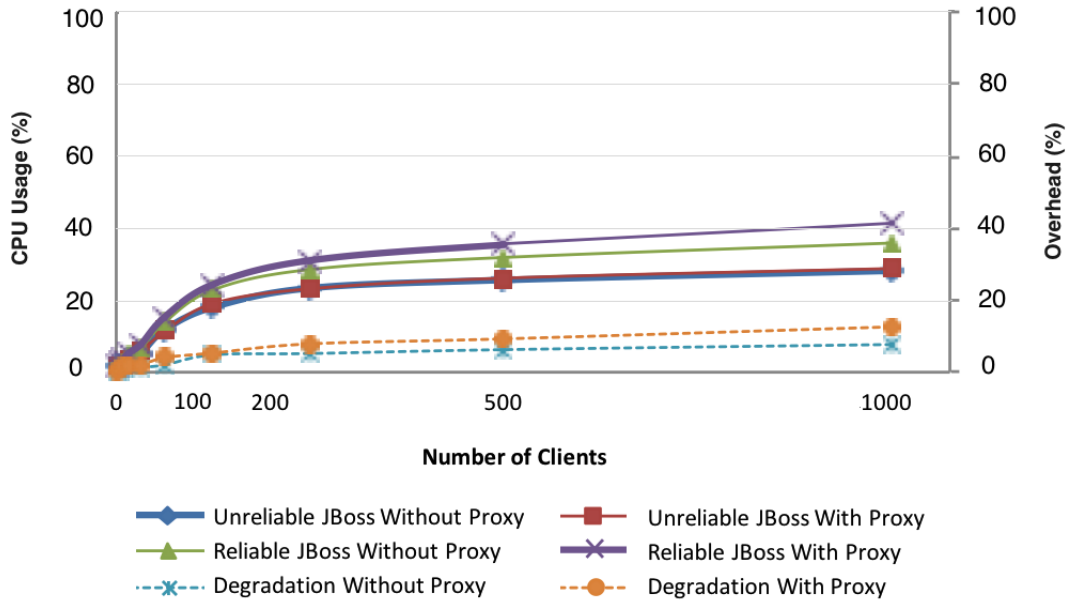

FIGURE 7.7: CPU utilization for unreliable and reliable HTTP servers in the scenarios with and without proxy

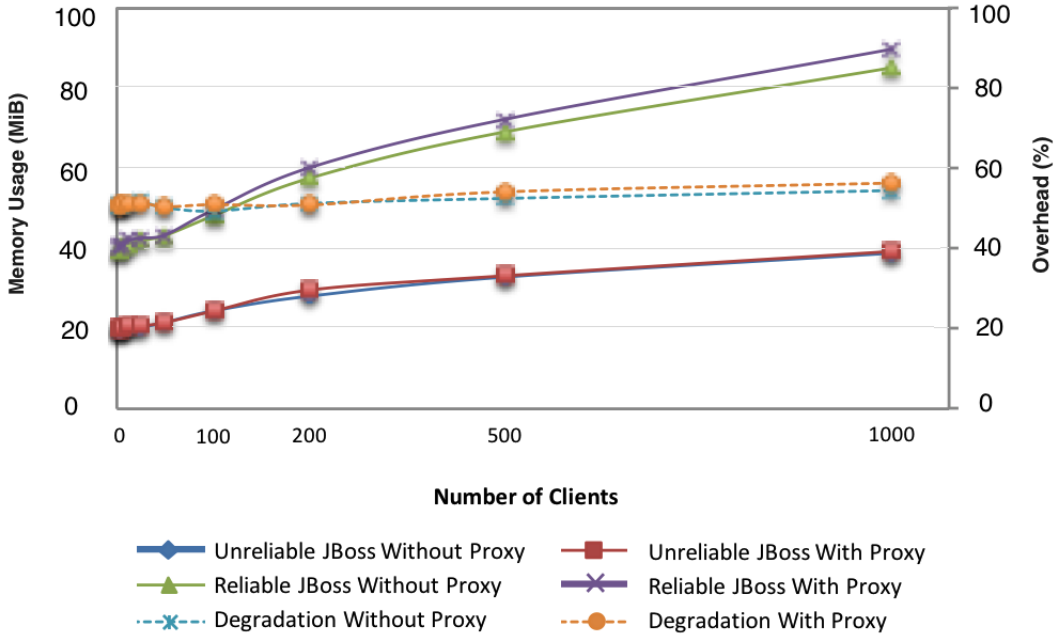

FIGURE 7.8: Memory utilization for unreliable and reliable HTTP servers in the scenarios with and without proxy

The memory used by our reliable server is, as expected, higher than the non-reliable one, with a maximum overhead of 60%, due to the extra buffering placed on top of TCP. The CPU overhead is again quite low (maximum of 15%), which is an excellent indication, as this resource can be many times of critical importance. Moreover, we can see that both CPU and Memory Usages are higher in the scenarios with proxy. This overhead is caused by the extra control channel and extra messaging (e.g., acknowledgment messages) of FSocket when proxies exist.

7.2.5 Discussion

FSocket is a session-based solution for the TCP's inability to handle connection crashes. FSocket provides full-duplex communication and supports the same interaction patterns as TCP. In this section, we showed that FSocket's performance degradation in all scenarios is negligible in comparison to TCP's performance.

TABLE 7.3: FSocket among other stream-based solutions presented in Chapter 2

Solutions	Reliability Mechanisms	Fault Tolerance	Reliability Semantics
TCP	Buffering, Acknowledgemnt and Retransmission, None (regarding connection or endpoint crashes)	None (regarding crashes)	Best-effort (regarding crashes)
RTP	None (regarding connection or endpoint crashes)	None (regarding crashes)	Best-effort (regarding crashes)
RSocket	Buffering and Explicit Acknowledgment	Connection Crashes	At-most-once
FSocket	Buffering and Implicit Acknowledgment	Connection Crashes	At-most-once
FT-TCP	Logging	Server Crashes	At-most-once
Rocks & Racks	Buffering and Implicit Acknowledgment, Checkpointing, and Migration	Connection Crashes and supports for server crashes	At-most-once
SCTP, cmpSCTP, MPTCP	Multiple connections	Connection Crashes	At-most-once
ST-TCP, HotSway, HydraNet-FT	Active Replication	Server Crashes	At-most-once
ER-TCP	Active Replication and Logging	Server Crashes	At-most-once

Among other reliable solutions, as presented in Table 7.3, FSocket shares more similarities with RSocket because both use similar mechanisms to tolerate connection crashes. As with RSocket, it uses an extra level of buffering, to ensure reliable communication. Its main difference to RSocket comes from the use of a buffering mechanism that does not always require explicit acknowledgments and an extra control channel.

We also did some simple experiments to compare FSocket with RSocket. Our initial results showed that RSocket is particularly slow due to Nagle's algorithm (Rencheng et al., 2010), which we could not switch off. To do a fair comparison, we have implemented that possibility in RSocket. We then observed that FSocket performance is a

little bit better than RSocket performance in the scenarios without proxy and on par with it, when there is a proxy.

7.3 Evaluation of the Message-Based Solution

In this section, we evaluate and compare the performance, complexity, and overhead of the message-based solutions proposed in Chapter 4, including Messenger (M), Trackable Messenger (TM), and Reliable Messenger (RM). We also verify the behavior of Reliable Messenger in the presence of connection crashes.

7.3.1 Evaluation of Correctness

To evaluate **correctness**, we performed a set of tests, to verify the correct functionality of Messenger, Trackable Messenger, and Reliable Messenger. In the case of Messenger, we carried out basic functionality tests targeting the correct delivery of messages in the tests. In Trackable Messenger, we programatically verified the correct exchange of multi-level acknowledgments and errors. As an illustrative example, to check if error messages are delivered to the sender's application, we let the client send messages while their $Confirmation$ attribute is set to $true$, and let the server to randomly confirm the messages with error.

In the case of Reliable Messenger, we verified its correct operation when recovering from connection crashes. In short, we let the client send messages to the server during 5 minutes, and generated connection crashes, by switching off the network interface at random instants during communication (we used `tcpkill` to cause connection crashes at random instants three times during each test). We then verify if communication is resumed without data losses. For 100 repetitions of the test, we observed that, even in the presence of connection crashes, all messages arrived correctly at the server.

7.3.2 Evaluation of Performance

Figure 7.9 shows the latency of three applications (with three invocations), which respectively use the Messenger, Trackable Messenger, and Reliable Messenger. The results obtained for the application using Messenger are used as baseline to measure the

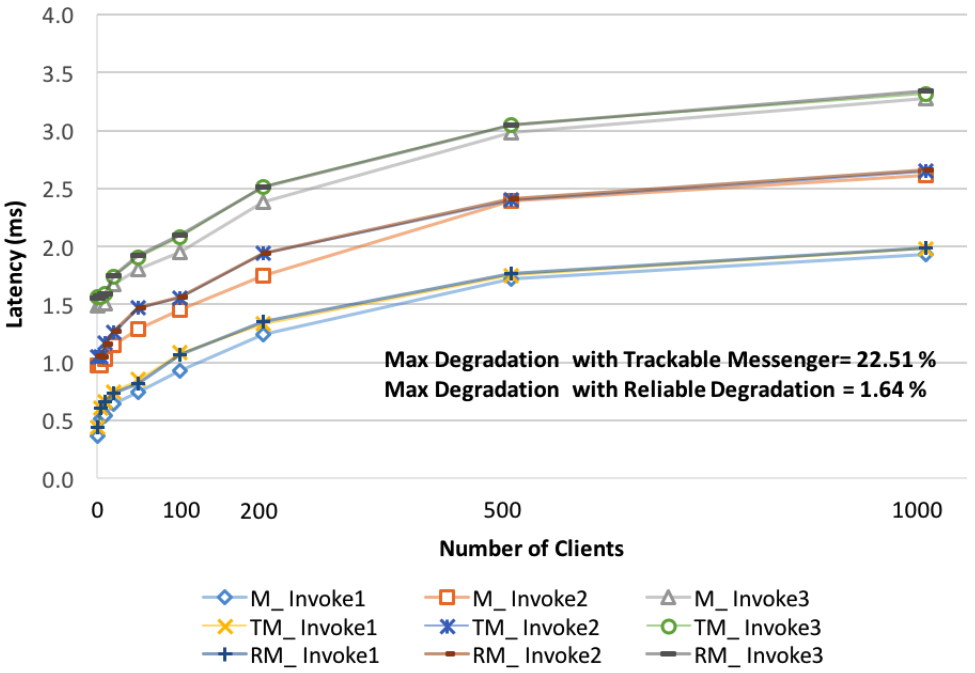

FIGURE 7.9: Latency of the Messenger, Trackable Messenger, and Reliable Messenger

overhead and performance degradation of the Trackable Messenger, and likewise, the results obtained for the Trackable Messenger are used to measure the overhead and performance degradation of the Reliable Messenger.

In all scenarios and with all invocations, the latency increases smoothly by increasing the number of clients. We see three different levels of latency for different invocations, due to the different processing time of the invocations. We calculated the latency degradation of the Trackable Messenger in comparison to the Messenger as well as the latency degradation of the Reliable Messenger in comparison to the Trackable Messenger. The results showed higher degradation in the former case and very low degradation in the latter case. The overhead of Trackable Messenger is mainly associated to the extra operations it has for acknowledgment: assign message identifier, application layer confirmation, piggyback the acknowledgment information, extract the acknowledgment information, and periodical acknowledgment.

It is worth mentioning that the maximum imposed degradation belongs to the `Invoke1` and when the number of the clients is very low (22.51%), which is the worst case for measuring the performance degradation. In fact, with all invocations, the degradation

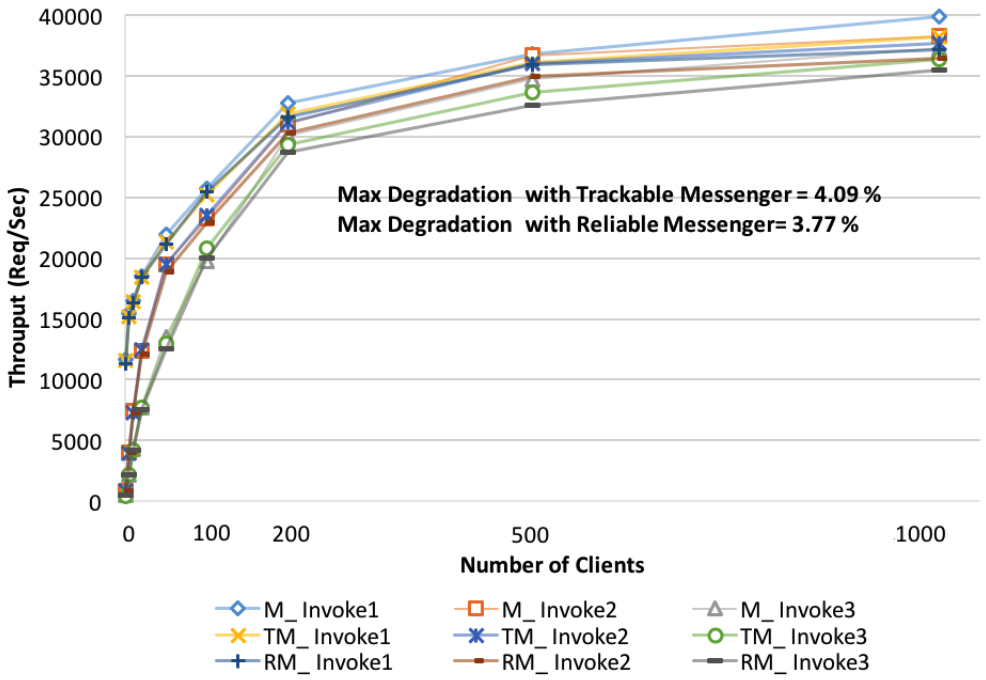

FIGURE 7.10: Throughput of the Messenger, Trackable Messenger, and Reliable Messenger

decreases as the number of clients increases. The maximum degradation in all scenarios is 2.57%, when the number of the clients is 1000. This observation shows that in highly concurrent applications, the latency imposed by Trackable Messenger will be very low.

The Reliable Messenger, in addition to the extra operations mentioned for the Trackable Messenger, buffers the messages on send and removes them from the buffer when acknowledged, but the results show that these operations do not impose a sensible overhead to the application, as the maximum degradation for latency we observed in the application using Reliable Messenger is only 1.64 %.

Figure 7.10 shows the results obtained for throughput of the applications using the different Messengers. Unlike latency that increases smoothly, the throughput increases rapidly in the beginning, by increasing the number of clients, and then pretty much levels out. Although our resources did not allow us to increase the number of clients to more than 1000, the figures show that the slope of the plots starts to decrease rapidly, by increasing the number of clients to more than 200 clients.

Throughput for `Invoke1` starts at a very higher value of more than 11000 requests per second, `Invoke2` at about 900 request per second, and `Invoke3` at about 450 requests per second). The plots show that the this difference remains until the end. However, the most interesting observation is that the performance degradation is very low with both Trackable Messenger (maximum 4.09%) and Reliable Messenger (maximum 3.77%) in all scenarios.

The final observation is that we obtained better performance for the application using FSocket, evaluated in the previous section, than the applications with the Messengers. We found out that the main source of overhead in Messengers belongs to the serialization and deserialization operations plus the operations required for writing and reading the size of the messages.

7.3.3 Evaluation of Complexity and Overhead

To evaluate complexity of the message-based solutions, we measure three important complexity metrics, Lines of Code (LOC), Cyclomatic Complexity (CC), and Nested Block Depth (NBD) for using Messenger, Trackable Messenger and Reliable Messenger.

TABLE 7.4: Implementation complexity of the messengers

	Lines of Code (LOC)	Cyclomatic Complexity (CC)	Nested Block Depth (NBD)
Messenger	178	1.33	1.38
Trackable Messenger	338	1.78	1.56
Reliable Messenger (FTSL)	447	1.98	1.58

The measurements, presented in Table 7.4, show that we used 178 lines of code to implement the Messenger design pattern, 160 extra lines of code to implement the Trackable Messenger, and 109 more lines of code than the Trackable Messenger, to implement the Reliable Messenger. In addition to LOC, the cyclomatic complexity increases from 1.33, to 1.78 in the Trackable Messenger, and to 1.98 in the Reliable Messenger. In all cases, the number of paths that go through each method, in average, stays between 1 and 2, which is a very good indication of low complexity in our designs and implementations (Jorgensen, 2008). The difference between the values for nested block depth, that increases from 1.38 in Messenger to 1.56 in Trackable Messenger and

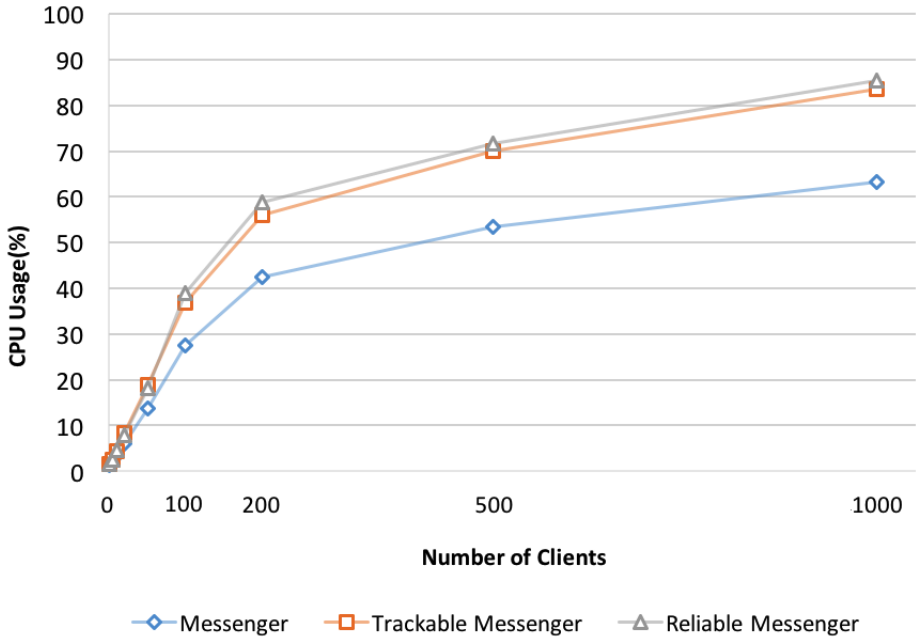

FIGURE 7.11: CPU utilization with Messenger, Trackable Messenger, and Reliable Messenger

to 1.58 in Reliable Messenger, confirms this fact. In general, these results show how simple our design solutions are, by taking their functionalities into consideration.

We also measure the CPU and memory usages in three applications that use Messenger, Trackable Messenger, and Reliable Messenger. Figure 7.11 presents the results obtained for CPU utilization.

As shown, the extra complexity of Trackable Messenger, in comparison to Messenger, imposes an extra price in the CPU utilization (maximum 37%) to the applications with Trackable Messenger and Reliable Messenger. The results for memory utilization, presented in Figure 7.12, show that the Trackable Messenger has very similar behavior as Messenger in terms of memory utilization, while the Reliable Messenger has a much higher overhead (maximum 68%), due to the extra buffering of messages. Furthermore, the results show that the memory overhead in the Reliable Messenger increases by increasing the number of clients. However, adjusting the time difference between periodical acknowledgments, which allows the peers to release some space in the buffer, can help to improve memory utilization, although it may impact on other performance and overhead parameters like CPU usage.

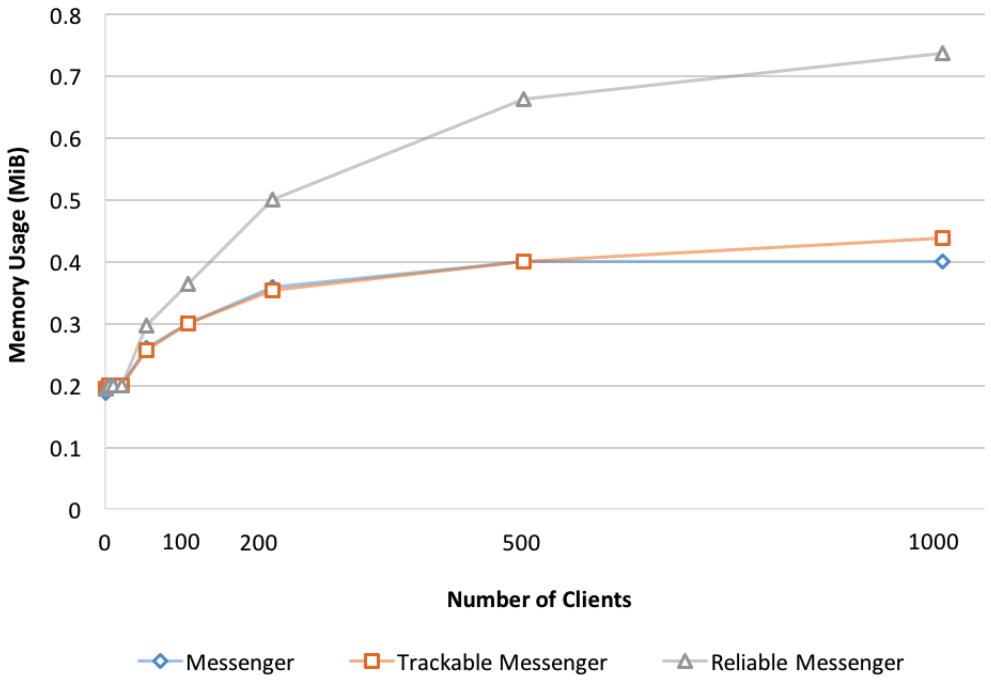

FIGURE 7.12: Memory utilization with Messenger, Trackable Messenger, and Reliable Messenger

7.3.4 Discussion

In general, there are two groups of message-based solutions that exist in the literature and that can offer reliability to the one-way interaction pattern. The first group includes messaging protocols used in communication, where there is no intermediate broker that could provide additional reliability guarantees. The second group of solutions include the presence of a broker that decouples communication and can provide additional reliable communication guarantees.

Our implementation of the Reliable Messenger is called FTSL, which is a message-oriented reliable protocol that addresses the limitations of TCP regarding connection crashes and provides special support for one-way communication. Our solution stands between the two main groups of solutions, presented in Table 7.5. It provides synchronous communication, like HTTP and HTTPR, with the difference that it is not based on the request-response interaction pattern, and on the other hand, it offers supports for one-way messaging, like JMS and MSMQ, but it does not use a broker. XMPP seems to be more similar, but it does not tolerate faults. The WS-Reliability

and WS-ReliableMessaging, that offer reliable delivery in the presence of failures, are designed only for SOAP messages. Our solution, besides providing full-duplex communication for message-based applications offers additional reliability services: it tolerates connection crashes, and it allows tracking of messages in one-way communication. It is general-purpose and independent of platform, technology and programming language. Moreover, neither request-response protocols, nor broker-based solutions for reliable one-way interaction can compete with our solution in terms of performance. In the former case, the single-message basis acknowledgment (i.e., one reply for each message sent), and in the latter case, the extra component in between the sender and receiver, negatively affect the performance.

TABLE 7.5: FTSL among other message-based solutions presented in Chapter 2

Solutions	Interaction Patterns	Reliability Mechanisms	Fault Tolerance	Reliability Semantics
HTTP	Request-Response, Synchronous, Transient	None (regarding crashes)	None (regarding crashes)	None (regarding crashes)
XMPP	One-way, Request-Response, Synchronous, Transient	None (regarding crashes)	None (regarding crashes)	None (regarding crashes)
HTTPR	Request-Response, Synchronous, Transient	Buffering, Logging & retransmission	Connection and endpoint crashes	Exactly-once delivery and at-most-once processing
CoRAL	Request-Response, Synchronous, Transient	Active replication and logging	Server crashes	At-most-once delivery
WS-Reliability, WS-ReliableMessaging	Request-Response, Synchronous, Asynchronous, Transient	Buffering and retransmission	Connection crashes	At-most-once delivery and processing
FTSL	One-Way, Synchronous, Transinet	Buffering and retransmission, and multi-level acknowledgement	Connection crashes	At-most-once delivery
ZeroMQ	One-way, Request-Response, Synchronous, Asynchronous, Transient , Persistent	-	None	At-most-once delivery
JMS, AMQP, MSMQ, WebSphere MQ, Oracle AQ	One-Way, Asynchronous, Persistent	Broker, acknowledgment, and support for transactions	Connection crashes and endpoint crashes	At-most-once delivery

Finally, the experimental evaluation showed that our reliable solution is simple and have a quite low overhead, considering the reliability services offered.

7.4 Evaluation of the Conversation-Based Solution

We implemented the protocol and design proposed for the exactly-once middleware of Chapter 5, in Java. To implement the logging operations of the `Connection Logger` with stable storage, we used SQLite 3.8.11, which is a self-contained, embeddable, zero-configuration SQL database engine (Owens and Allen, 2010). Our implementation of the exactly-once middleware is called EoMidd. In this section, we measure its correctness, performance, overhead and complexity.

7.4.1 Evaluation of Correctness

Here we evaluate the correctness of the EoMidd implementation, regarding its ability in recovery from connection, client and server crashes. For evaluating the recovery procedure, we run the client and server, three times for 5 minutes, to exchange messages, each time with a different failure scenario (with connection crashes, client crashes or server crashes). We use `tcpkill` to cause connection crashes and `kill` to crash the client and server processes at random instants during each test. Each test was repeated 10 times.

The results of our experiments showed that the EoMidd could successfully ensure the exactly-once execution of each request and delivery of its response in all scenarios.

7.4.2 Evaluation of Performance

To measure the performance of our solution, we implemented two request-response applications: a reliable application that uses EoMidd and an unreliable application that just uses the Messenger (shown by `M` in the figures), to implement its message-based communication. Figure 7.13 presents the latency of these two applications, with three types of operations, for an increasing number of clients. The results show that the operations with stable storage can significantly impact latency. The latency of the unreliable application increases from 0.72 ms for `Invoke1` with one client to a maximum of 6.56 ms for `Invoke3` with 1000 clients, while the latency of the reliable application starts from 1.12 for `Invoke1` with one client to 336.26 ms for `Invoke3` with 1000 clients. To make sure that the overhead is mainly caused by the logging operations, we temporarily eliminated these operations from the EoMidd and repeated the tests. We could observe that the latency of the reliable application without logging operations is on par with the latency of the unreliable application.

We also measured the throughput of the applications. Figure 7.14 presents the throughput for an increasing number of clients. Fortunately, the disk operations do not impose a big overhead on throughput. Unlike the latency case, we do not see a very big difference between the two applications. The maximum degradation we observed is 8.63%. In general, the performance degradation is small, by considering the stable storage operations involved in the exactly-once protocol.

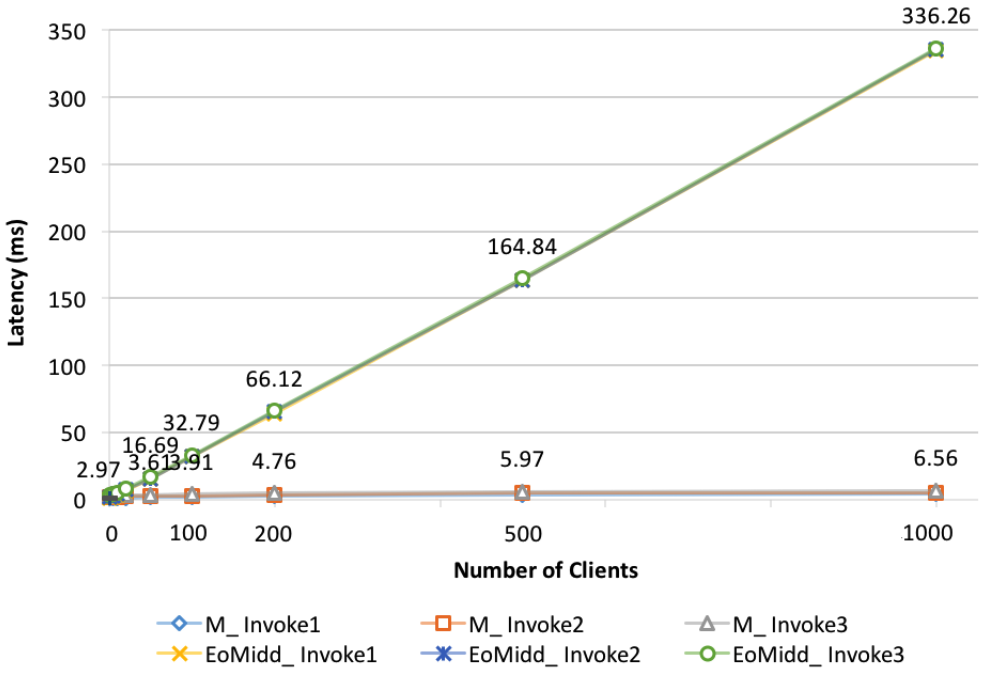

FIGURE 7.13: Latency without and with the Exactly-once Middleware

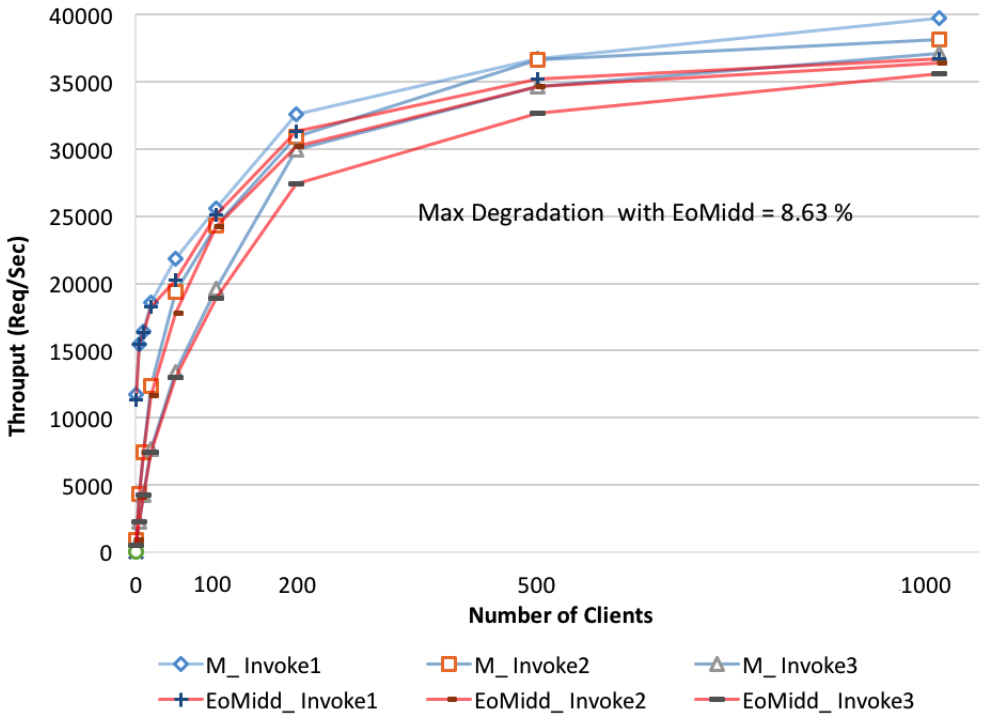

FIGURE 7.14: Throughput without and with the Exactly-once Middleware

7.4.3 Evaluation of Complexity and Overhead

Here, we measure the LOC, CC, and NBD of the EoMidd and compare it with the complexity of Messenger that is extended with the exactly-once middleware, to insert reliability.

TABLE 7.6: Implementation complexity of EoMidd

	Lines of Code (LOC)	Cyclomatic Complexity (CC)	Nested Block Depth (NBD)
Messenger	178	1.33	1.38
EoMidd	369	1.9	1.62

The measurements, presented in Table 7.6, show that we used 191 lines of code to implement reliability over the Messenger. In addition to LOC, the cyclomatic complexity increases from 1.33, to 1.90, and the average nested block depth increases from 1.38 to 1.62. These numbers are a good indication of the very low implementation complexity of EoMidd.

In addition to the implementation complexity, we measure the overhead of the EoMidd on CPU and memory utilization. The results, as presented in Figure 7.15 and Figure 7.16, show that both memory and CPU utilization increase by increasing the number of clients in both reliable and non-reliable scenarios. In both cases, a significant overhead is imposed by our reliability mechanism. In contrast to the proposed stream-based and message-based solutions evaluated in the previous sections, the results show a higher overhead on the CPU utilization than in memory utilization in this conversation-based solution. This difference is mainly caused by the logging operations.

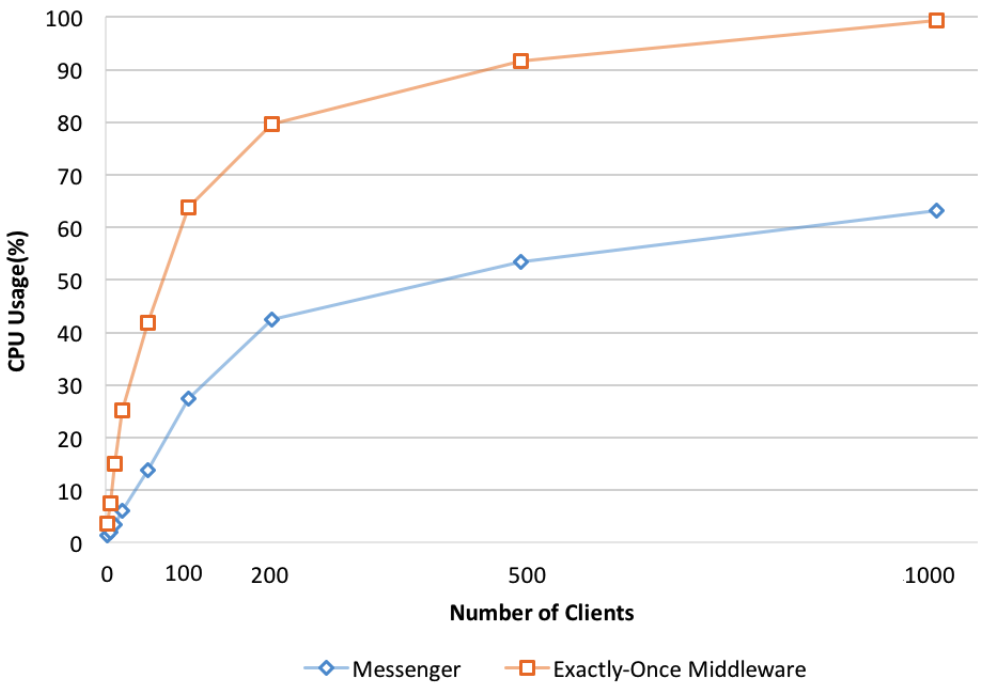

FIGURE 7.15: CPU utilization without and with the Exactly-Once Middleware

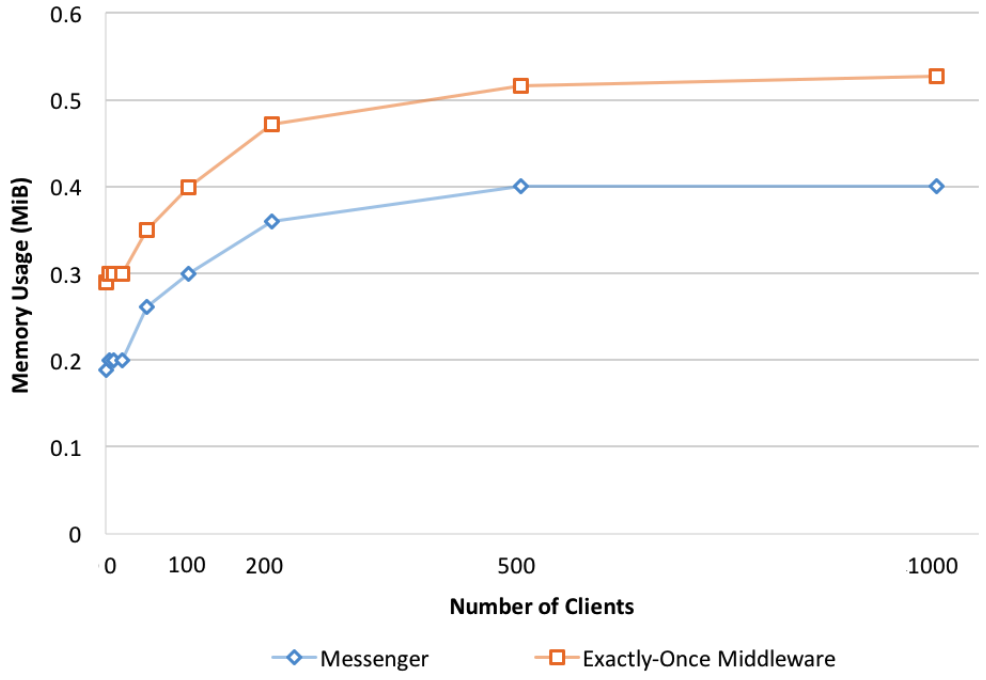

FIGURE 7.16: Memory utilization without and with the Exactly-Once Middleware

7.4.4 Discussion

In this section, we measured and presented the correctness, performance and overhead of the solution proposed for the exactly-once middleware. The results showed that implementing the exactly-once protocol proposed is not complex from the software engineering perspective, but its overhead on latency and CPU utilization is quite high. However, we believe that our session-based exactly-once protocol and design provide some clear advantages to developers, by making recovery from connection and endpoint crashes nearly or completely transparent, and second, because they do not compromise throughput.

TABLE 7.7: EoMidd among other conversation-based solutions presented in Chapter 2

Solutions	Reliability Mechanisms	Fault Tolerance	Reliability Semantics
EOS2	Logging	Endpoint crashes	Exactly-once
Phoenix	Logging and checkpointing	Endpoint crashes	Exactly-once
iSAGA	Logging	Client crashes	At-least-once
Exactly-once middleware	Buffering and Logging	Connection and endpoint crashes	Exactly-once

In comparison to the existing conversation-based solutions, as shown in Table 7.7, the Exactly-Once Middleware (EoMidd) uses a buffering and logging mechanism to enable recovery form both connection and endpoint crashes and guarantee exactly-once semantics. Moreover, EoMidd's protocol and design, unlike the other solutions, is presented independently of the platform and programming language.

7.5 Evaluation of the Taxonomy: Cost Analysis

In this section, we present a set of analysis and experiments, to evaluate the overhead of using a popular reliable request-response protocol in a typical services environment. We analyze the implementation complexity of the protocols and use a Java implementation of the TPC-C benchmark (jTPCC) (Barbosa and Tlemsani, 2005) to observe that the performance impact of running these reliable protocols on a server is negligible.

7.5.1 Analysis of Implementation Complexity

The commit operation (;) might be complex to implement, but developers may use a single commit to simultaneously change their state and save the response. Deletion of responses (D) is simpler, as it involves a single database table. Re-issuing a deletion order is not a problem for the server and, therefore, crashes do not raise any problem to the protocol.

On the client, the generation (g) may require a unique identifier, but the identifier may also come from the server (e.g., when the client is a browser). This case, which we described in the SMS and banking services will certainly involve the use of cookies. To implement the save operation (.), the client can first save and then send a message to let the server delete (if the protocol requires so), in a separate step. We never require an atomic disk write and message send neither in the client nor in the server.

We still need to know the cost to provide reliable invocations, from the server perspective. The cost for the client is clearer: in the "gCc;" family it involves two round-trip times, or even a third one, if the client must send a deletion order and does not use an additional thread for that; the "$gCcCc$;" will take even longer. Memory costs for the server depend on the size of the responses and on the time the server keeps them on disk (and possibly memory), before deleting them.

7.5.2 Evaluation of Performance

To evaluate the throughput degradation, we ran a simple benchmark with the "gCc;" reliable protocol. We did not consider "$gCcCc$;", because in the cases we are aware of, the extra "Cc" serves to ensure that the correct human is in the loop, thus making throughput less important. Furthermore, this extra round-trip might affect latency more than throughput. The other family (gC;) is too simple for conducting a realistic test.

To evaluate the impact of adding reliable interaction semantics to a service, we carried out an experimental evaluation using three versions of jTPCC v5.4, an implementation of the well-known TPC-C (Raab, 1993), which is a benchmark for Online Transaction Processing systems (OLTP). The versions used (and described in the following paragraphs) are:

(a) The default version of jTPCC;

(b) A Java RMI client-server version of jTPCC;

(c) A reliable version of jTPCC (two RMI calls, respective disk operations, plus a periodical deletion thread).

The standard form of jTPCC is a monolithic application (version a), with multiple terminals simulating operations on the database. We split the standard jTPCC, to run the client terminals and the server on different machines, using RMI for the communication (version b). The TCP connection that RMI first sets up between client and server is not a problem for us. On the contrary, RMI will try itself to ensure the at-most once semantics for each exchange of the protocol, for example, the "$g \rightarrow C$" part of the protocol, which involves a client-to-server message and its response.

In our reliable version of TPC-C (version c), all terminals at the client-side request transactions to the same remote object of the server. Notice that most RMI implementations, namely our Oracle Java 1.7.0_51 implementation, will provide multi-thread access to this remote object and will therefore create parallel requests to the OLTP system. Since we tried "gCc;", each client interaction occurs in two separate calls: a first one to get a timestamp, a second one to execute the operation. The typical server response to this operation is a text string with nearly 1 KiB. Since it must ensure at-most-once, the server saves the response on disk.

In some cases, we use a single transaction to run the operation and to save the response (committing in the end). In other cases, the requests involve multiple separate transactions (e.g., with independent queries), and we use an extra transaction to save. Since the server always needs a timer to delete replies to clients that abruptly cease interaction, we simply did not use any deletion on the protocol and resorted to a timer. The server deletes all responses with more than $\Delta = 1$ minute, every 20 seconds (see Theorem 2). The precise protocol we ran was "$gCc; r$" (with no "."), because we did not need to protect the client from crashes.

To accomplish the test, 10 simultaneous clients are used to invoke the TPC-C operations. Table 7.8 presents the results (in transactions per minute - TPM) obtained for an average of 40 tests. The throughput loss is around 3.5% for the reliable version. This value is small enough to suggest that the main additional cost of a reliable implementation is the extra round-trip times seen by the client.

TABLE 7.8: Throughput of exactly-once version of jTPCC compared with unreliable ones

TPC-C Version	TPM	Standard Deviation
a	389.11	42.28
b	389.03	22.69
c	375.37	15.00

As a summary, in this section, we analyzed the implementation complexity of the operations involved in a reliable request-response protocol. The analysis shows that the implementation cost of a popular protocol of the taxonomy can be quite low, comparing to solutions like distributed transactions or other protocols requiring agreement.

Chapter 8

Conclusion and Future Work

In this chapter, we summarize the thesis and briefly explain the main achievements of this research work. Then, we state the research directions that may follow.

8.1 Summary of the Thesis

These days, distributed applications support many businesses and services. However, given the unreliable nature of distributed systems, providing some guarantees to applications regarding delivery and utilization of data is a non-trivial and error-prone task. Tolerating crashes is a very difficult task, due to the incomplete and inconsistent knowledge in peers about the state of the interactions.

The point-to-point communication model is, by far, the most popular means of communication used to support critical services (e.g., e-commerce, banking, healthcare), where reliable communication is a primary concern. Within the point-to-point communication model, applications present largely different interaction patterns (e.g., one-way or request-response, synchronous or asynchronous) and require different reliability semantics (e.g., ordered delivery, or exactly-once execution), depending on their characteristics and goals. Thus, ensuring the reliability of point-to-point communication can become an extremely difficult task for developers, which have to make the right design and development choices (e.g., selecting and configuring the middleware that provides the right reliability guarantees). To make informed decisions, developers must

be supported by an appropriate knowledge base, or they might end up designing and deploying applications, whose reliability requirements have not been properly met.

This thesis started by presenting a survey on reliable distributed communication. In fact, the number of solutions offering some form of reliable communication in distributed systems is quite large. This is due to the increasing need for highly reliable distributed applications, but also due to the fact that tolerating distributed component crashes is quite difficult. Each solution supports a specific interaction pattern, uses some specific reliability mechanisms, and offers specific reliability semantics. Thus, the number of combinations available for developers is huge. Selecting the right reliability solution for an application is quite far from being a trivial task, due to the diversity of implementation solutions and configurations. In fact, the lack of synthesized information often leads developers to create their own ad hoc and error-prone solutions for reliable communication.

In this survey, we classified and analyzed available solutions for distributed communication, including protocols and middleware, as well as the applications, their specifications, and reliability requirements. The analysis is performed based on several key features, including the interaction pattern (e.g., one-way vs. request-response, synchronous vs. asynchronous), the reliability semantics (e.g., at-least-once, at-most-once), the reliability targets (e.g., stream, message, object, conversation), and the reliability mechanisms (e.g., Buffering, Acknowledgment and Retransmission). This helped us to find out the gaps between the state of the art solutions and the reliability requirements of the applications.

As a result of this survey, it is observed that, in most cases, the more elaborate communication solutions offering stronger or a larger number of guarantees are purely academic efforts that can, by no means, compete with the popularity, maturity and importance of older, more established, albeit poorer solutions, such as TCP. This survey also provides a knowledge base to researchers and developers that intend to work in the field of dependable distributed systems.

In the following chapters, this thesis addressed some of the main limitations of TCP for reliable distributed programming. Over the last few decades, TCP has been one of the cornerstones of the Internet, for building reliable communication, but it does neither handle connection crashes, nor provide information about the data sent and received to facilitate the recovery of crashed connections. Thus, it is totally up to developers to

rollback communicating peers to some coherent state. In this thesis, we proposed the Connection Handler design pattern, to facilitate the recovery process, independently of programming language or platform. This solution is also applicable to other protocols or middleware like Websockets.

The Connection Handler design pattern is then applied to reliable stream-based applications (e.g., multimedia streaming) and, by taking several challenges into consideration (e.g., legacy peers and proxies), a complete, reusable, extensible, and efficient design pattern, named Reliable Transporter Design Pattern, is proposed. This solution fills the gaps between the reliability requirements of a wide range of applications and the existing solutions, not by creating a new middleware or library, but by showing a detailed road map to enable correct and efficient implementation of more reliable communication.

This thesis also addressed some other limitations of TCP, regarding message-based communication, particularly, where application peers do not follow a request-response interaction pattern. TCP is a stream-based protocol and has no means to place application layer data into an envelope in order to be sent and received as a message. Moreover, in TCP-based one-way communication, the sender has no means to determine whether its messages were received or processed. Furthermore, the limitation of TCP in recovering from connection crashes are also relevant in this type of applications.

We proposed three design solutions, named Messenger, Trackable Messenger, and Reliable Messenger to overcome these limitations. The Messenger, besides offering a TCP-like full-duplex connection, provides a mean on top of transport layer for sending and receiving messages. The Trackable Messenger, which uses the Messenger design pattern to exchange messages, enables tracking of messages during its life cycle by using multi-level acknowledgments. The Reliable Messenger, besides using the Messenger and Trackable Messenger for exchanging and tracking messages, tolerates connection crashes by applying the Connection Handler design pattern.

This thesis further addressed the reliability issue in request-response interactions, in which the server must execute a given request once and only once (exactly-once semantics). This form of interaction is being used to support business and safety-critical operations in diverse sectors, such as banking or healthcare. Implementing this reliability semantics in request-response interactions is quite difficult, because the client, server, and communication channel may fail, potentially requiring diverse and complex

recovery procedures, which may result in duplicate messages being processed at the server. An exactly-once protocol is proposed and a design pattern, named Exactly-once Middleware, is presented to facilitate the development of this type of interaction.

We also developed a taxonomy of all possible reliable request-response protocols. To do this, valid sequences of client and server actions were generated, organized into a prefix tree, and classified according to their reliability semantics and memory requirements. The taxonomy reveals three families of protocols matching common real-world implementations that try to deliver exactly-once or at-most-once. Some solutions were also presented, to enable safe deletion of the stored state in all families of protocols. This taxonomy, with its strict organization of the protocols, provides a knowledge base and a solid foundation for creating correct services.

Several experiments are carried out in this thesis to demonstrate the applicability of the solutions proposed, and measure their correctness, performance, overhead, and implementation complexity. Moreover, some experiments are performed over the taxonomy of reliable request-response protocols, to explore its applicability in the real-world applications, and to evaluate the overhead exposed by these exactly-once protocols.

8.2 Future Work

We believe that this thesis increased the knowledge and experience on reliable communication and design patterns. However, due to time and space restrictions there were certain topics that we could not explore entirely, but that might be interesting research directions for the future:

- The Hypertext Transfer Protocol (HTTP) and the Transmission Control Protocol (TCP) are the most popular protocols used in the development of web-based applications. Despite their popularity, their use brings two limitations to applications and systems that require reliable interactive real-time communications. One of these limitations comes from HTTP itself, as it follows a strict request-response paradigm, between clients and servers. This interaction paradigm limits the interactive real-time communication between the web parties, because HTTP servers can only send messages back to clients in response to client requests. Hence, on one hand, distributed applications and systems are forced to work in this manner,

even if a reply is not actually needed for some requests; on the other hand, an HTTP-based server cannot send anything to a client if the client does not ask for it. Furthermore, TCP connection crashes can interrupt HTTP communication. Thus, we aim to use the Connection Handler design pattern to offer a solution for building Fault-Tolerant Bi-Directional Communication in Web-Based Applications. To make this possible, we can combine WebSocket (Fette and Melnikov, 2011) and the Connection Handler Design Pattern to create a fault-tolerant Websocket (FtWebSocket). A preliminary version of this idea is already presented in (Ivaki and Araujo, 2014). We believe that FtWebSocket can help web-based applications requiring reliability in their interactive real-time communication. Here we enumerate three target applications for FtWebSocket:

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

- We have already built a taxonomy of reliable request-response protocols. We intend to model and formally verify the protocols presented in this taxonomy.

- Strong reliability semantics, in particular exactly-once execution of non-idempotent operations, is very important in business and safety-critical applications. Ensuring this type of semantics can be extremely difficult and costly due to the complexity of the necessary mechanisms and operations with stable storage. One excellent solution for this type of applications could be to convert their

non-idempotent operations to idempotent operations, as these can tolerate duplicate invocations (Ramalingam and Vaswani, 2013). Thus, having the means that facilitate the design and implementation of idempotent operations can be vital in implementing applications where reliable communication is of critical importance.

- Designing and developing applications and systems that function in a hard-real-time environment with strict timing constraints (e.g., air-traffic control), is a non-trivial task. Most of these applications require real-time databases to manage time-constrained data transactions. With real-time databases, developers face different challenges when compared to general-purpose database systems (Bestavros and Fay-Wolfe, 2012). In general, real-time databases hide a major source of complexity to developers. We believe these complexities and challenges can be partially tackled, by building design patterns that help to correctly model and design real-time applications.

- Having design-based solutions does not exclude the need for implementation of such solutions. This means that the correctness of the implementation must be evaluated, which brings in the specific need for testing techniques that allow evaluating reliable communication implementations. Such techniques might involve fault injection (e.g., dropping packets, duplicating messages), code analysis, just to name a few options. The goal should be to provide developers with a way of verifying if their implementations actually meet the requirements in this specific scenario. Thus, we believe that using model-based testing (Utting et al., 2012) can be the right approach to address this problem. Ideally each design pattern could be presented with a model to test its implementations.

- The difficulty in selecting the right reliable communication solution, which is shown by the common fallback to the use of a popular but less reliable solution, like TCP , also suggests that there is a strong need for techniques that help developers evaluating and selecting solutions. Benchmarking might help developers with a way of assessing how good a given solution is. In the end, this assessment should allow for comparison of different solutions, thus helping to make the right selection. However, despite perfectly fitting the needs described, creating this type of benchmark involves significant challenges, mostly related to the complexity involved in these scenarios.

Bibliography

Abie, H., Savola, R. M., and Dattani, I. (2009). Robust, secure, self-adaptive and resilient messaging middleware for business critical systems. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pages 153–160. IEEE.

Aghdaie, N. and Tamir, Y. (2009). CoRAL: A transparent fault-tolerant web service. *Journal of Systems and Software*, 82(1):131–143.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A pattern language: towns, buildings, construction*, volume 2. Oxford University Press.

Alvisi, L., Bressoud, T., and El-Khashab, A. (2001). Wrapping Server-Side TCP to mask connection failures. *IEEE International Conference on Computer Communications (INFOCOM)*.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

Azaiez, M. N. and Bier, V. M. (2007). Optimal resource allocation for security in reliability systems. *European Journal of Operational Research*, 181(2):773–786.

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

Barbosa, R. and Tlemsani, M. (2005). Open source java implementation of the TPC-C benchmark.

Barga, R., Lomet, D., Paparizos, S., Yu, H., and Chandrasekaran, S. (2003). Persistent applications via automatic recovery. In *Proceedings of the Seventh International Database Engineering and Applications Symposium*, pages 258–267. IEEE.

Bestavros, A. and Fay-Wolfe, V. (2012). *Real-Time Database and Information Systems: Research Advances*, volume 420. Springer Science & Business Media.

Bicakci, M. V. and Kunz, T. (2012). TCP-Freeze: Beneficial for virtual machine live migration with ip address change? In *The 8th International on Wireless Communications and Mobile Computing Conference (IWCMC 2012)*, pages 136–141. IEEE.

Bilorusets, R., Box, D., Cabrera, L. F., Davis, D., Ferguson, D., Ferris, C., Freund, T., Hondo, M. A., Ibbotson, J., Jin, L., et al. (2005). Web services reliable messaging protocol (WS-ReliableMessaging). *Specification, BEA, IBM, Microsoft and TIBCO*.

Birman, K. P. (1997). *Building secure and reliable network applications*. Springer.

Birman, K. P. (2012). Group communication systems. In *Guide to Reliable Distributed Systems*, pages 369–405. Springer.

Birrell, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59.

Boutros, B. S. and Desai, B. C. (1996). A two-phase commit protocol and its performance. In *Proceedings of the Seventh International Workshop on Database and Expert Systems Applications*, pages 100–105. IEEE.

Braden, R., Borman, D., and Partridge, C. (1989). Computing the internet checksum. *ACM SIGCOMM Computer Communication Review*, 19(2):86–94.

Brown, N. and Kindel, C. (1998). Distributed component object model protocol–dcom/1.0. *Online, November*.

Buchmann, A. and Koldehofe, B. (2009). Complex event processing. *it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 51(5):241–242.

Burton-Krahn, N. (2002). HotSwap-Transparent server failover for linux. In *LISA*, volume 2, pages 205–212.

Canning, R. (2012). *Real-Time Web Technologies in the Networked Performance Environment*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library.

Carvalho, N., Araujo, F., and Rodrigues, L. (2005). Scalable QoS-based event routing in publish-subscribe systems. In *Fourth IEEE International Symposium on Network Computing and Applications*, pages 101–108. IEEE.

Chakradhar, S. T. and Raghunathan, A. (2010). Best-effort computing: re-thinking parallel software and hardware. In *Proceedings of the 47th Design Automation Conference*, pages 865–870. ACM.

Chakravorty, S. and Kale, L. V. (2007). A fault tolerance protocol with fast fault recovery. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10. IEEE.

Chakravorty, S., Mendes, C. L., and Kale, L. V. (2006). Proactive fault tolerance in mpi applications via task migration. In *High Performance Computing-HiPC*, pages 485–496. Springer.

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75.

Chockler, G. V., Keidar, I., and Vitenberg, R. (2001). Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469.

Christen, M. AnomicFTPD: A freeware FTP server in java. `http://anomic-ftp-server.e-programy.pl/`.

Coulouris, G. F., Dollimore, J., and Kindberg, T. (2005). *Distributed systems: concepts and design*. pearson education.

Crispin, M. R. (2003). Internet message access protocol-version 4rev1.

Cristian, F. (1989). Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158.

Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78.

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

Dantas, W. S., Bessani, A. N., and Correia, M. (2009). Not quickly, just in time: Improving the timeliness and reliability of control traffic in utility networks. *networks*, 9:12.

Douglass, B. P. (2003). *Real-time design patterns: robust scalable architecture for real-time systems*, volume 1. Addison-Wesley Professional.

Downing, T. B. (1998). *Java RMI: remote method invocation*. IDG Books Worldwide, Inc.

Driscoll, K., Hall, B., Sivencrona, H., and Zumsteg, P. (2003). Byzantine fault tolerance, from theory to reality. In Anderson, S., Felici, M., and Littlewood, B., editors, *Computer Safety, Reliability, and Security*, number 2788 in Lecture Notes in Computer Science, pages 235–248. Springer Berlin Heidelberg.

Dutta, K., VanderMeer, D., Datta, A., and Ramamritham, K. (2001). User action recovery in internet sagas (isagas). In *Technologies for E-Services*, pages 132–146. Springer.

Egwutuoha, I. P., Chen, S., Levy, D., Selic, B., and Calvo, R. (2012). A proactive fault tolerance approach to high performance computing (HPC) in the cloud. In *The Second International Conference on Cloud and Green Computing (CGC)*, pages 268–273. IEEE.

Ekwall, R., Urbán, P., and Schiper, A. (2002). Robust TCP connections for fault tolerant computing. In *Proceedings of the Ninth International Conference on Parallel and Distributed Systems*, pages 501–508. IEEE.

Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131.

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

Feather, C. D. (2006). Network news transfer protocol (NNTP), rfc 3977, https://tools.ietf.org/html/rfc3977.

Fekete, A., Lynch, N., Mansour, Y., and Spinelli, J. (1993). The impossibility of implementing reliable communication in the face of crashes. *Journal of the ACM (JACM)*, 40(5):1087–1107.

Feng, W.-c., Liu, M., Krishnaswami, B., and Prabhudev, A. (1998). Priority-based technique for the best-effort delivery of stored video. In *Electronic Imaging'99*, pages 286–300. International Society for Optics and Photonics.

Fenton, N. and Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC Press.

Ferrari, D. (1990). Client requirements for real-time communication services.

Fette, I. and Melnikov, A. (2011). The websocket protocol.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (2009). Rfc 2616, hypertext transfer protocol–http/1.1, 1999. *URL http://www. rfc. net/rfc2616. html*.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382.

Fleury, M. and Reverbel, F. (2003). The JBoss extensible server. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 344–373. Springer-Verlag New York, Inc.

Galdun, J., Takac, L., Ligus, J., Thiriet, J., and Sarnovsky, J. (2008). Distributed control systems reliability: consideration of multi-agent behavior. In *The 6th International Symposium on Applied Machine Intelligence and Informatics*, pages 157–162. IEEE.

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

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54.

Garro, A. and Tundis, A. (2015). On the reliability analysis of systems and sos: the RAMSAS method and related extensions. *IEEE Systems Journal*, 9(1):232–241.

Gartner, F. C. (1999). Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26.

Gawand, H., Mundada, R., and Swaminathan, P. (2011). Design patterns to implement safety and fault tolerance. *International Journal of Computer Applications*, 18(2):6–13.

Gawlick, D. (1998). Messaging/queuing in oracle8. In *IEEE 29th International Conference on Data Engineering (ICDE)*, pages 66–66. IEEE Computer Society.

Gharbi, G., Alaya, M. B., Diop, C., and Exposito, E. (2013). Aoda: an autonomic and ontology-driven architecture for service-oriented and event-driven systems. *International Journal of Collaborative Enterprise*, 3(2/3):167–188.

Goodwill, J. (2002). *Apache jakarta tomcat*, volume 1. Springer.

Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Gray, J. N. (1979). *A discussion of distributed systems*. IBM Thomas J. Watson Research Division.

Guerraoui, R. and Schiper, A. (1996). Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies-Ada-Europe'96*, pages 38–57. Springer.

Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317.

Halpern, J. Y. (1987). Using reasoning about knowledge to analyze distributed systems. *Annual Review of Computer Science*, 2(1):37–68.

Hanmer, R. (2013). *Patterns for fault tolerant software*. John Wiley & Sons.

Hart, J. (2003). Web Sphere MQ: connecting your applications without complex programming. *IBM WebSphere Software White Papers*.

Helland, P. (2012). Idempotence is not a medical condition. *Communications of the ACM*, 55(5):56–65.

Herring, S., Stein, D., and Virtanen, T. (2013). *Pragmatics of computer-mediated communication*, volume 9. Walter de Gruyter.

Hintjens, P. (2013). *ZeroMQ: Messaging for Many Applications*. " O'Reilly Media, Inc.".

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

Hohpe, G. and Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on software engineering*, (5):279–295.

Holzmann, G. J. (2004). *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading.

Horrell, S. (1999). Microsoft message queue. *Enterprise Middleware*.

Huebscher, M. C. and McCann, J. A. (2008). A survey of autonomic computing-degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7.

Iosif, R. (1998). The promela language.

Ivaki, N. and Araujo, F. Fault-tolerant AnomicFTPD: A fault-toleant freeware FTP server in java. `https://sourceforge.net/projects/ftanomic/`.

Ivaki, N. and Araujo, F. An open-source java implementation of the exactly-once middleware. `https://sourceforge.net/projects/eomidd/`.

Ivaki, N. and Araujo, F. An open-source java implementation of the reliable messenger. `https://sourceforge.net/projects/ftsl/`.

Ivaki, N. and Araujo, F. An open-source java implementation of the stream-based reliable transporter design pattern. `https://sourceforge.net/projects/fsocket/`.

Ivaki, N. and Araujo, F. (2014). Fault-tolerant bi-directional communications in web-based applications. In *The 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 833–836. IEEE.

Ivaki, N., Boychenko, S., and Araujo, F. (2014). A fault-tolerant session layer with reliable one-way messaging and server migration facility. In *IEEE 3rd Symposium on Network Cloud Computing and Applications (NCCA 2014)*, pages 75–82. IEEE.

Ivaki, N., Laranjeiro, N., and Araujo, F. (2015). A taxonomy of reliable request-response protocols. In *The 30th ACM/SIGAPP Symposium On Applied Computing (SAC)*. ACM.

Ji, X., Ma, Y., Ma, R., Li, P., Ma, J., Wang, G., Liu, X., and Li, Z. (2015). A proactive fault tolerance scheme for large scale storage systems. In *Algorithms and Architectures for Parallel Processing*, pages 337–350. Springer.

Jin, H., Xu, J., Cheng, B., Shao, Z., and Yue, J. (2003). A fault-tolerant TCP scheme based on multi-images. In *IEEE Pacific Rim Conference on Communications, Computers and signal Processing (PACRIM 2003)*, volume 2, pages 968–971. IEEE.

Johnson, D. B. (1989). Distributed system fault tolerance using message logging and checkpointing. Technical report, DTIC Document.

Jones, S., Wilikens, M., Morris, P., and Masera, M. (2000). Trust requirements in e-business. *Communications of the ACM*, 43(12):81–87.

Jorgensen, P. C. (2008). *Software Testing: A Craftsman's Approach*. Auerbach Publications, 3rd edition.

Knight, J. C. (2002). Safety critical systems: challenges and directions. In *Proceedings of the 24rd International Conference on Software Engineering (ICSE 2002)*, pages 547–550. IEEE.

Koloniari, G., Ntarmos, N., Pitoura, E., and Souravlias, D. (2011). One is enough: distributed filtering for duplicate elimination. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 433–442. ACM.

Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., and Senft, C. (1989). Distributed fault-tolerant real-time systems: The mars approach. *IEEE Micro*, 9(1):25–40.

Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. (1992). Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Lamport, L. (1987). Leslie lamport's home page. `http://research.microsoft.com/en-us/um/people/lamport/`.

Lann, G. L. (1997). An analysis of the ariane 5 flight 501 failure-a system engineering perspective. In *Proceedings of International Conference and Workshop on Engineering of Computer-Based Systems*, pages 339–346. IEEE.

Laverdiere, M.-A., Mourad, A., Hanna, A., and Debbabi, M. (2006). Security design patterns: Survey and evaluation. In *Canadian Conference on Electrical and Computer Engineering (CCECE'06)*, pages 1605–1608. IEEE.

Lee, P. A. and Anderson, T. (2012). *Fault tolerance: principles and practice*, volume 3. Springer Science & Business Media.

Liao, J., Wang, J., and Zhu, X. (2008). cmpSCTP: An extension of SCTP to support concurrent multi-path transfer. In *IEEE International Conference on Communications (ICC 2008)*, pages 5762–5766. IEEE.

Linington, P. F., Milosevic, Z., Tanaka, A., and Vallecillo, A. (2011). *Building enterprise systems with ODP: an introduction to open distributed processing*. CRC Press.

Lui, M., Gray, M., Chan, A., and Long, J. (2011). Scaling your spring integration application. In *Pro Spring Integration*, pages 529–559. Springer.

Maier, M. W. (1996). Architecting principles for systems-of-systems. In *INCOSE International Symposium*, volume 6, pages 565–573. Wiley Online Library.

Marwah, M., Mishra, S., and Fetzer, C. (2003). TCP server fault tolerance using connection migration to a backup server. In *null*, page 373. IEEE.

Marwah, M., Mishra, S., and Fetzer, C. (2005). A system demonstration of ST-TCP. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005)*, pages 308–313. IEEE.

McLean, S., Williams, K., and Naftel, J. (2002). *Microsoft. net remoting*. Microsoft Press.

Mills, D. (1992). Network time protocol (version 3) specification, implementation and analysis.

Mok, A. K. (1983). Fundamental design problems of distributed systems for the hard-real-time environment.

Myers, J. and Rose, M. (1996). Post office protocol-version 3. Technical report, STD 53, RFC 1939, May.

Natarajan, B., Gokhale, A., Yajnik, S., and Schmidt, D. C. (2000). DOORS: Towards high-performance fault tolerant CORBA. In *International Symposium on Distributed Objects and Applications*, pages 39–48. IEEE.

Northcutt, J. D. and Kuerner, E. M. (1992). System support for time-critical applications. In *Network and Operating System Support for Digital Audio and Video*, pages 242–254. Springer.

Nowicki, B. (1989). Nfs: Network file system protocol specification. Technical report.

Owens, M. and Allen, G. (2010). *SQLite*. Springer.

Papazoglou, M. P. (2003). Service-oriented computing: Concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003)*, pages 3–12. IEEE.

Paterson, I., Smith, D., Saint-Andre, P., and Moffitt, J. (2010). Xep-0124: bidirectional-streams over synchronous HTTP (BOSH). *Draft Standard, July*.

Pietzuch, P. R. and Bacon, J. M. (2002). Hermes: A distributed event-based middleware architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, pages 611–618. IEEE.

Pleisch, S., Kupšys, A., and Schiper, A. (2003). Preventing orphan requests in the context of replicated invocation. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 119–128. IEEE.

Popescu, A., Constantinescu, D., Erman, D., and Ilie, D. (2007). A survey of reliable multicast communication. In *Proceedings of the 3rd EURO-NGI Conference on Next Generation Internet Networks*.

Postel, J. (1981). RFC793: transmission control protocol. *Information Sciences Institute*, 27:123–150.

Postel, J. (1982). Simple mail transfer protocol, rfc 5321, https://tools.ietf.org/html/rfc5321. *Information Sciences*.

Postel, J. and Reynolds, J. (1985). Rfc 959: File transfer protocol.

Protocol, U. D. (1980). RFC 768 j. postel ISI 28 august 1980. *Isi*.

Raab, F. (1993). TPC-C - The Standard Benchmark for Online transaction Processing.

Ramalingam, G. and Vaswani, K. (2013). Fault tolerance via idempotence. *ACM SIGPLAN Notices*, 48(1):249–262.

Reis, D. and Miranda, H. (2012). FTRMI: fault-tolerant transparent RMI. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 511–518. ACM.

Rencheng, J., Xiao, M., Lisha, M., and Liding, W. (2010). A design of efficient transport layer protocol for wireless sensor network gateway. In *The 2nd International Conference on Signal Processing Systems (ICSPS 2010)*, volume 1, pages V1–775. IEEE.

Rentsch, T. (1982). Object oriented programming. *ACM Sigplan Notices*, 17(9):51–57.

Richards, M., Monson-Haefel, R., and Chappell, D. A. (2009). *Java message service*. " O'Reilly Media, Inc.".

Roman, R., Zhou, J., and Lopez, J. (2013). On the features and challenges of security and privacy in distributed internet of things. *Computer Networks*, 57(10):2266–2279.

Rushby, J. (1994). Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219.

Sadjadi, S. M. and McKinley, P. K. (2003). A survey of adaptive middleware. *Michigan State University Report MSU-CSE-03-35*.

Saini, K. (2011). *Squid Proxy Server 3.1: Beginner's Guide*. Packt Publishing Ltd.

Saint-Andre, P. (2011). Extensible messaging and presence protocol (XMPP): Core.

Sauer, F. (2013). Metrics 1.3. 6.

Scharf, M. and Ford, A. (2013). Multipath tcp (MPTCP) application interface considerations. Technical report.

Schmidt, D. C. (1995). Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Pattern languages of program design*, pages 529–545. ACM Press/Addison-Wesley Publishing Co.

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

Schmidt, D. C., O'Ryan, C., Kircher, M., Pyarali, I., et al. (2000). Leader/followers-a design pattern for efficient multi-threaded event demultiplexing and dispatching. In *University of Washington. http://www. cs. wustl. edu/˜ schmidt/PDF/lf. pdf*. Citeseer.

Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. (2003). RTP: a transport protocol for real-time applications. Technical report.

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., and Sommerlad, P. (2013). *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons.

Shao, Z., Jin, H., Cheng, B., and Jiang, W. (2008). Er-tcp: an efficient tcp fault-tolerance scheme for cluster computing. *The Journal of Supercomputing*, 43(2):127–145.

Shegalov, G. and Weikum, G. (2006). EOS 2: unstoppable stateful PHP. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1223–1226. VLDB Endowment.

Shegalov, G., Weikum, G., Barga, R., and Lomet, D. (2002). EOS: exactly-once e-service middleware. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 1043–1046. VLDB Endowment.

Shenoy, G., Satapati, S. K., and Bettati, R. (2000). HydraNet-FT: Network support for dependable services. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, pages 699–706. IEEE.

Sloane, N. J. and MacWilliams, F. J. (1981). *The theory of error-correcting codes*. North Holland.

Smith, J. (2007). *Inside microsoft windows communication foundation*. Microsoft Press Redmond.

So-In, C., Jain, R., and Dommety, G. (2009). PETS: persistent TCP using simple freeze. In *First International Conference on Future Information Networks (ICFIN 2009)*, pages 97–102. IEEE.

Spector, A. Z. (1982). Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(4):246–260.

Stankovic, J. A. and Ramamritham, K. (1989). *Tutorial: hard real-time systems*. IEEE Computer Society Press.

Stewart, R. and Metz, C. (2001). SCTP: new transport protocol for TCP/IP. *IEEE Internet Computing*, 5(6):64–69.

Stout, L., Moffitt, J., and Cestari, E. (2014). An extensible messaging and presence protocol (XMPP) subprotocol for websocket. Technical Report No. RFC 7395.

Tanenbaum, A. S. and Steen, M. V. (2006). *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2 edition edition.

Tay, B. H. and Ananda, A. L. (1990). A survey of remote procedure calls. *ACM SIGOPS Operating Systems Review*, 24(3):68–79.

Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. (2012). Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM.

Tindell, K. and Burns, A. (1994). Guaranteed message latencies for distributed safety-critical hard real-time control networks. *Dept. of Computer Science, University of York*.

Turner, D., Levchenko, K., Snoeren, A. C., and Savage, S. (2011). California fault lines: understanding the causes and impact of network failures. *ACM SIGCOMM Computer Communication Review*, 41(4):315–326.

Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312.

Vaysburd, A. and Yajnik, S. (1999). Exactly-once end-to-end semantics in CORBA invocations across heterogeneous fault-tolerant ORBs. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS)*, page 296.

Veljkovic, N., Punt, M., Bjelica, M. Z., and Crvenkovic, N. (2013). TV-centric multi-player gaming over the cloud for consumer electronic devices. In *Third International Conference on Consumer Electronics-Berlin (ICCEBerlin 2013)*, pages 1–3. IEEE.

Vinoski, S. (2006). Advanced message queuing protocol. *IEEE Internet Computing*, (6):87–89.

Waldby, J., Madhow, U., and Lakshman, T. (1998). Total acknowledgements: a robust feedback mechanism for end-to-end congestion control. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 274–275. ACM.

Wang, R., Salzberg, B., and Lomet, D. (2009). Transaction support for log-based middleware server recovery. In *IEEE 25th International Conference on Data Engineering (ICDE 2009)*, pages 353–356. IEEE.

Yoshioka, N., Washizaki, H., and Maruyama, K. (2008). A survey on security patterns. *Progress in informatics*, 5(5):35–47.

Zandy, V. C. and Miller, B. P. (2002). Reliable network connections. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 95–106. ACM.