A Taxonomy of Reliable Request-Response Protocols

Naghmeh Ivaki
CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
naghmeh@dei.uc.pt

Nuno Laranjeiro
CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
cnl@dei.uc.pt

Filipe Araujo
CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
filipius@uc.pt

ABSTRACT

Reliable request-response interactions, in which the server never executes a given request more than once, are being used to support business and safety-critical operations in diverse sectors, such as banking, E-commerce, or healthcare. This form of interactions can be quite difficult to implement, because the client, server, or communication channel may fail, potentially requiring diverse and complex recovery procedures, which may result in duplicate messages being processed at the server. In this paper we address the following question: could we provide a meaningful taxonomy of reliable request-response protocols? We generate valid sequences of client and server actions, organize the generated sequences into a prefix tree, and classify them according to their reliability semantics and memory requirements. The tree reveals three families of protocols matching common real-world implementations that try to deliver exactly-once or at-most-once. 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.

Keywords

Reliability, Exactly-Once, At-Most-Once, Taxonomy

1. INTRODUCTION

Most of the interactions in distributed systems are based on the request-response messaging paradigm, 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. Some common invocation semantics are at-least-once, at-most-once, and exactly-once, where the server executes the request once or more than once; once and not more than once; and once and only once, respectively.

SAC'15, April 13 - 17 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695898.

In its simplest form, the at-most-once semantics can be simple to achieve: the client sends the request and the response may or may not arrive. To achieve the at-least-once guarantee, the client must re-send the same request until it gets back a response, but risking multiple executions of the service. However, neither the simplest at-most-once, nor the at-least-once semantics can be used in bank transfers or online purchases. These operations are *non-idempotent* and should not occur more than once. A lack of response is also not acceptable, because clients need to know whether the operation succeeded or not. Only the exactly-once semantics is entirely appropriate for such operations. However, this semantics is hard to achieve, because, depending on the assumptions, failures might be difficult or impossible to distinguish from slow transmission and processing of messages [1].

Given the difficulty, the importance and the unreliability of solutions found in practice for the problem, we propose a taxonomy of reliable request-response protocols reaching beyond Spector's request-response-acknowledgment (RRA) [2]. In this taxonomy, we cover a large spectrum of reliable interactions over unreliable channels (including non-FIFO).

To create this taxonomy, we generate an exhaustive list of protocols based on a sequence of alternating client and server actions. We assume that servers eventually recover, but accept the possibility that clients may not recover (e.g., browsers). For this reason, we include, not only exactly-once, but also at-most-once interactions. We then eliminate all invalid protocols and organize the remaining into a prefix tree. Finally, we classify the protocols according to their reliability semantics (at-most-once or exactly-once) and memory requirements (bounded vs. unbounded).

We illustrate the application of our taxonomy to real cases and analyze several common implementations of online services that match protocols of our taxonomy tree. The analysis shows the applicability of the taxonomy and points out several pitfalls in the implementation of services that try to ensure at-most-once semantics. We also analyze the implementation complexity of the protocols and observe that the performance impact of running these reliable protocols on a server is negligible. In summary, the main contributions of this paper are the following:

- A taxonomy for reliable protocols with three different families that clearly match common real-world implementations that might be used by developers, to select the right protocol for their services.
- An analysis of the protocols considering their use with unreliable and non-FIFO channels and with respect to mem-

ory requirements (memory is used for keeping connection state). This analysis can be vital for developers to implement services according to specific requirements or resource restrictions.
- The application of the taxonomy to real on-line services (a phone operator, a social network, and a Bank), showing its usefulness in helping to analyze this type of services.
- An analysis of implementation complexity of the operations involved in a reliable request-response interaction, based on the jTPCC benchmarking tool [3].

This paper is organized as follows. In Section 2 we present related work. In Section 3 we present the approach we used to create the taxonomy. In Section 4 we analyze real services, match their communication mechanisms to our tree, and evaluate the throughput overhead of a popular family of reliable protocols. Finally, Section 5 concludes the paper.

2. BACKGROUND AND RELATED WORK

In this section we review the state of the art on protocols for reliable client-server communication and mechanisms for exactly-once execution of the requests at the server.

2.1 Reliable Communication Protocols

Developers usually resort to the Transmission Control Protocol (TCP) for reliability. However, despite providing important features, such as message integrity and message ordering, TCP cannot overcome long communication outages and endpoint crashes. This problem is so important that researchers proposed a large range of solutions at different levels of the communication stack.

At the transport layer, we can find efforts as SCTP [4], created to supersede TCP; and Multipath TCP [5] with the ability to explore multiple paths, when available. But communication failures may still exist when using these protocols. A few session-layer protocols [6, 7] try to mitigate the limitations of TCP, by introducing transparent buffering and retransmission mechanisms. At the application layer level we can find a quite large number of solutions. Web services [8] typically use HTTP to invoke remote operations, but HTTP cannot ensure invocation reliability. To overcome this, protocols like HTTPR [9] log messages and reliably transport them, even in the presence of network and endpoints failures. HTTPR also ensures that each message is delivered to its destination exactly once, or is reliably reported as undelivered. Other efforts that reached standardization were WS-Reliability [10] and its successor WS-ReliableMessaging (WS-RM) [11]. The WS-RM protocol supports a set of delivery assurances including at-least-once and exactly-once. However, these focus only on message delivery and ignore the (re)execution of requests.

The exactly-once semantics was addressed by Spector in [2]. Spector presented efficient communication techniques for high speed local networks and introduced a basic communication model. The work describes request (R), request-response (RR), and request-response-acknowledge (RRA) protocols. The latter enables the server to release memory, by letting the server know that the client received the response. This work was extensively explored later, namely in [12, 13], in which idempotent messaging protocols, that ensure message delivery, are described.

A subtle problem, that is of utmost importance in the context of this paper, has to do with the ordering properties required from the communication channel. Some well-known protocols, such as the User Datagram Protocol (UDP), are assumedly not FIFO; however, even TCP does not really offer a FIFO channel, because TCP connections may crash. 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.

2.2 Exactly-Once Execution of Requests

Any implementation providing exactly-once over non-FIFO lossy channels requires stable storage [14, 15]. As we will see in Section 3, this is the case of all protocols included in our taxonomy. Many authors resort precisely to stable storage to tolerate crashes of servers that maintain active TCP connections. As an example, a fault-tolerant TCP server that uses message logging is proposed in [16], as an approach to recover from faults. The server logs all requests and, in the case of failure, some other replica will replay the log to reach the same state of the original server. Phoenix/APP [17] and iSAGA [18] are also interesting cases that use logging mechanisms to release developers from the burden of managing faults occurring in business-oriented web applications.

ACID transactions and distributed ACID transactions [19], which ensure that multiple parties agree on the outcome of a given interaction, also play a key role in servers that must execute requests only once. The large body of work on the atomic commit and consensus problems [20] is a fundamental part of the solution for distributed transactions. A widely popular protocol for distributed transactions is the Two-Phase Commit [21] (2PC). However, 2PC is heavy and blocking, forcing all parties to lock the resources involved until they take the final decision.

To avoid the overhead of distributed transactions, Queued Transaction Processing techniques [22, 23] provide solutions that ensure server execution. For instance, in [24] the server queues each request, providing it a unique identifier. At commit time, the server saves the request identifier in the database. Upon recovery from a failure, the server reads the database to determine which requests remain unattended. Some authors extended this effort to ensure delivery for multiple subscribers in messaging systems [25].The idea of using a unique identifier explores the notion of idempotence, which is also quite common. Since many operations are not idempotent, developers often add identifiers to prevent their re-execution. For example, web developers may use hidden fields in HTML forms. In [26] JavaScript is used to provide logs that enable web browsers to resume from crashes and to relieve web developers from this effort. Preventing re-execution is also addressed in [27] in which generalized "idempotent" requests are used to enable persistence of execution state.

3. TAXONOMY OF PROTOCOLS

In this section we present the assumptions, definitions, and the approach used to build our taxonomy of reliable protocols. The approach includes 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.

- **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.
- **Classification and analysis of the protocols:** The identified protocols are classified based on reliability semantics (i.e., at-most-once or exactly-once) and analyzed according to memory requirements (bounded or unbounded).

3.1 Definitions and Assumptions

In this work, we consider 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. 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. This can be accomplished by giving the initiative of sending a message to the client and making the server acknowledge. Hence, the message exchange starts with the client sending a request to the server, then the server replies with another message and so on. 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 (this lets the client repeat the message if it misses a timely acknowledgment).

We assume the following **failure scenarios**. 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). To be able to recover from failures, crash-recovery clients and servers 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. We do not assume atomic save and send actions (which can be quite complex to implement): saving to stable storage is one action, sending a message is a different separate action. This means that endpoint crashes may cause duplicate messages.

Channels are assumed to be fair but unreliable. They may lose messages, but will deliver a message that is sent a sufficient number of times. However, this may result in message delivery failures, as endpoints stop re-sending a message once they delete all resources associated with a request. We also assume that the channel may reorder messages (i.e., it is non-FIFO), but does not change their contents. Finally, we assume that the channel, crash-recovery client, and server eventually work in a fault-free period for a sufficiently long time to complete the interaction. Without this assumption, achieving exactly-once semantics is impossible [28].

To generate the reliable protocols, we define a set of possible **client and server actions** (see Table 1) that will be later aggregated and combined in sequences. 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 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 may result in changes to the system state, thus crash-recovery clients must atomically save these changes and each response in the storage. 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 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; ";" 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 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 1: Client and server set of actions

Action	Endpoint	Description
g	*client*	Generate a request
c	*client*	Perform computation using volatile/stable storage
.	*client*	Atomically save a response and any system state changes to 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 request-related state 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 ir 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, see Section 4).

In Table 2 we identify the **actions that 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 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. 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 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 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), thus the need for stable storage is mandatory in this type of clients (and not used in crash-stop clients). The server must also keep the state in stable storage to be able to recover after a failure. Thus, ";" always requires stable storage. 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 are marked with "Maybe" (since they do not try to recover, use of stable storage depends on the application).

We also mark crash-recovery clients with "Maybe". When recovering from failure, these clients will go back to the previous commit point. Thus, although the commit operation and the recovery procedure require access to stable storage, the computation may not (its use depends on the application). The same applies to the server. Note that if a failure has occurred after a commit, upon reception of the same request, the server does no computation ("C") and simply sends a response. "r" simply releases the references to a request from memory and crash-recovery clients may also need to 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.

3.2 Generating the Protocols

A first obstacle to provide a taxonomy of exactly-once protocols 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: 1) the client must start with a "g" operation; 2) the client and server can only save once ("." or ";") — this minimizes the number of operations that involve stable storage; 3) once both client and server save, they do not engage in more message exchanges (except possibly for releasing memory and deleting state); 4) their interaction prior to saving is limited to two rounds of exchanges[1]; 5) the server does not do any computation after saving response and deleting state.

Based on the above restrictions, we can 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. A crash-recovery client may re-generate the request 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). 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 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 restrictions: 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. For instance, given the element $(gc., C;)$ generated by the Cartesian product, we may have the protocols "$gCc.;$", "$gC;c.$", "$gCc;.$" , but not "$Cgc.;$" or "$g;Cc.$". 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 $1,646$ possible protocols.

3.3 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: 1) "." 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); 2) 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; and 3)sequences that repeat actions on the same side, such as "$.c$", "cc", "$;C$" are useless and can be removed. In this step 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 1), 2), and 3) respectively delete 855, 254, and 507 protocols, for a total of $1646-855-254-507 = 30$ protocols.

[1]Note, however, that in faulty runs, client and server may repeat messages, thus engaging in more than two rounds of exchanges.

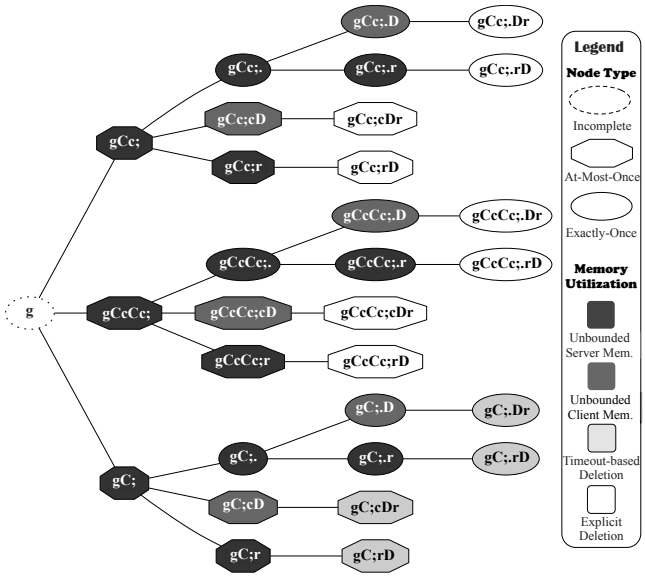

Figure 1: Exactly-once and at-most-once protocols

3.4 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 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 [2]). 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.

3.5 Classifying and Analyzing the Protocols

We first classify the protocols, according to their reliability semantics, as exactly-once or at-most-once. The protocols where crash-stop clients do not save response and state (i.e., without ".") are shown as octagon nodes. 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 1, the suffixes for at-most-once and exactly-once protocols are similar among families.

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

3.5.1 Causes for Unlimited Memory Utilization

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.

3.5.2 Timeout-Based Deletion of the State for Non-FIFO Channels

The solution for the problem described in the previous section 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 [29], or Christian's algorithm [30].

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. 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 (see Section 4). 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 1, 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.

4. CASE STUDIES AND COST ANALYSIS

In this section we present a set of experiments, which target the following goals: 1) show the applicability of our taxonomy to real services, including its usefulness in disclosing pitfalls and fallacies, which may occur when implementing the protocols; and 2) Understanding the overhead of using a popular reliable protocol.

4.1 Reliable Interaction in Real Services

The protocols presented in Figure 1 (Section 3) 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) [26] uses the protocol "$gC;.$" of this family. In the [26] 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 [2], are found implemented in [12]. 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, thus not needing to delete state.

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 are usually associated to a timeout window, which is similar to the Δ timeout mechanism of the "gCc;" family discussed earlier.

However, we can easily find cases where captchas change on reload, thus 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). 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. 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 invocation 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.

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 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[2].

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 paper, but hugely popular sites are typically backed by NoSQL databases that do not preserve all the ACID properties.

Some protocols used in real systems may elude our taxonomy. For example, 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.

4.2 Implementation and Overhead Analysis

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. 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). 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.

However, 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.

To evaluate the throughput overhead, we ran a simple benchmark with the "gCc;" reliable interaction. 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 benchmark [31]. The versions used are: a) the default version of jTPCC; b) a best-effort Java RMI client-server version of jTPCC; c) a reliable version of jTPCC.

Table 3: Systems used in the experiments

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

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

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

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.

Table 3 presents the systems used to execute the tests, which were configured to use 10 simultaneous clients that invoked the TPC-C operations. Table 4 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 4: Throughput overhead results

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

5. CONCLUSION

In this paper we presented an approach 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 showed the applicability of the protocols presented in the tree to real-world on-line services and discussed the likely fallacies and pitfalls that may take place when implementing these protocols. We also show, in a typical services environment, that the implementation cost of a popular reliable request-response protocol can be quite low. We believe that this paper provides a detailed understanding of reliable interactions and their challenges, helping developers to build correct services. It can help developers selecting the right type of interaction for future services and understand if their current ones are using the correct interaction. As future work, we intend to model and formally verify the protocols presented in this paper.

Acknowledgments

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

6. REFERENCES

[1] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 1985.

[2] Alfred Z. Spector. Performing remote operations efficiently on a local computer network. *Commun ACM*, 1982.

[3] Open source java implementation of the tpc-c benchmark. `http://jtpcc.sourceforge.net`.

[4] C. Metz R. Stewart. SCTP: new transport protocol for TCP/IP. *IEEE Internet Computing*, 2001.

[5] Sebastien Barre, Christoph Paasch, and Olivier Bonaventure. MultiPath TCP: from theory to practice. In *NETWORKING*, pages 444–457. Springer, 2011.

[6] N. Ivaki, F. Araujo, and F. Barros. Session-based fault-tolerant design patterns. *ICPADS*, 2014.

[7] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *MobiCom*, 2002.

[8] Ethan Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI, and WSDL*. O'Reilly Media, Inc., February 2002.

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

[10] C. Evans, D. Chappell, Bunting, et al. Web services reliability, ver. 1.0. *joint specification by Fujitsu, NEC, Oracle, Sonic Software, and Sun Microsystems*, 2003.

[11] D. Davis et al. Web services reliable messaging. Technical report, OASIS, 2006.

[12] N. Ivaki, F. Araujo, and R. Barbosa. A middleware for exactly-once semantics in request-response interactions. In *PRDC*, pages 31–40, November 2012.

[13] Jeremy Brown, J. P. Grossman, and Tom Knight. A lightweight idempotent messaging protocol for faulty networks. In *SPAA*, pages 248–257, 2002.

[14] Joseph Y. Halpern. Using reasoning about knowledge to analyze distributed systems. *Annual Review of Computer Science*, 2(1):37–68, 1987.

[15] H. Attiya, S. Dolev, and J.L. Welch. Connection management without retaining information. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, pages 622–631, 1995.

[16] Lorenzo Alvisi, Thomas C Bressoud, Ayman El-Khashab, Keith Marzullo, and Dmitrii Zagorodnov. Wrapping Server-Side TCP to mask connection failures. In *IEEE INFOCOM*, pages 329—337, 2001.

[17] R. Barga, D. Lomet, S. Paparizos, Haifeng Yu, and S. Chandrasekaran. Persistent applications via automatic recovery. In *International Database Engineering and Applications Symposium*, pages 258–267, July 2003.

[18] Kaushik Dutta, Debra E. VanderMeer, Anindya Datta, and Krithi Ramamritham. User action recovery in internet sagas (isagas). In *TES*, pages 132–146. Springer-Verlag, 2001.

[19] H. Vogler, T. Kunkelmann, and M.-L. Moschgath. Distributed transaction processing as a reliability concept for mobile agents. In *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, 1997*, pages 59–64, 1997.

[20] Leslie Lamport. Paxos Made Simple. *SIGACT News*.

[21] B.S. Boutros and B.C. Desai. A two-phase commit protocol and its performance. In *Seventh International Workshop on Database and Expert Systems Applications*, 1996.

[22] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[23] M. Richards, R. Monson-Haefel, and D. A. Chappell. *Java message service*. O'Reilly Media, 2009.

[24] James P. Emmond and Robert W. Griffin. Exactly-once semantics in a TP queuing system, 1990.

[25] S. Bhola, Robert Strom, S. Bagchi, Yuanyuan Zhao, and J. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *IEEE/IFIP DSN*, 2002.

[26] German Shegalov, Gerhard Weikum, Roger Barga, and David Lomet. EOS: Exactly-Once E-Service middleware. VLDB Endowment, M. Kaufmann, 2002.

[27] David B. Lomet. Generalized idempotent requests, September 2008.

[28] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)*, 37(3):549–587, 1990.

[29] D. Mills, Ed. J. Martin, J. Burbank, and W. Kasch. Network time protocol version 4. IETF, 2010.

[30] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[31] Francois Raab. TPC-C - The Standard Benchmark for Online transaction Processing. 1993.