X-Ability: A Theory of Replication

Svend Frolund, Rachid Guerraoui[1]
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-15
January, 2000

replication,
fault-tolerance,
high-
availability,
Exactly-once

Different replication mechanisms provide different solutions to the same basic problem. However, there is no precise specification of the problem itself, only of particular classes of solutions, such as active replication and primary-backup. Having a precise specification of the problem would help us better understand the space of possible solutions.

We present a formal definition of the problem solved by replication. We introduce x-ability (Exactly-once-ability) as a correctness criterion for replicated services. An x-able service has obligations to its environment and its clients. It must update its environment under exactly-once semantics. Furthermore, it must provide idempotent, non-blocking request processing and deliver consistent results to clients. X-ability is a local property: replicated services can be specified and implemented independently, and later composed in the implementation of more complex replicated services.

We illustrate the value of x-ability through a novel replication protocol that handles non-determinism and external side-effects.

[1] Swiss Federal Institute of Technology, Lausanne, Switzerland CH 1015

X-Ability: A Theory of Replication

Svend Frølund[1] Rachid Guerraoui[2]
[1] Hewlett-Packard Laboratories, Palo Alto, CA 94304
[2] Swiss Federal Institute of Technology, Lausanne, CH 1015

Abstract

Different replication mechanisms provide different solutions to the same basic problem. However, there is no precise specification of the problem itself, only of particular classes of solutions, such as active replication and primary-backup. Having a precise specification of the problem would help us better understand the space of possible solutions.

We present a formal definition of the problem solved by replication. We introduce x-ability (Exactly-once-ability) *as a correctness criterion for replicated services. An x-able service has obligations to its environment and its clients. It must update its environment under exactly-once semantics. Furthermore, it must provide idempotent, non-blocking request processing and deliver consistent results to clients. X-ability is a local property: replicated services can be specified and implemented independently, and later composed in the implementation of more complex replicated services.*

We illustrate the value of x-ability through a novel replication protocol that handles non-determinism and external side-effects.

1 Introduction

There is a significant body of literature about replication algorithms. Surprisingly, there is no precise specification of the general problem that these algorithms solve. There are well-known specifications of correctness for particular ways of implementing replication, such as primary-backup [BMST93] and active replication [Sch93]. However, these are specifications of replication solutions rather than a specification of the actual problem solved by replication. The very few abstract replication properties that we know about, e.g., [Aiz89] and [MP88], do not address correctness with respect to external side-effect. They only address consistency of state that is encapsulated within the service itself. In particular, there is no provisioning, in the specifications, for having a replicated service call a third-party entity, e.g., another replicated service. This form of interaction is however common in practice, especially for middleware application servers.[1]

This paper defines *x-ability (Exactly-once-ability)*, a correctness criterion for replicated services. X-ability is independent of particular replication algorithms. The main idea behind x-ability is to consider a replicated service correct if it somehow behaves like a single, fault-tolerant process. We develop a theory to express, in a precise manner, what it takes for a service to provide this illusion. Roughly speaking, an x-able service must satisfy a contract with its clients as well as a contract with third-party entities. In terms of clients, a service must provide idempotent, non-blocking request processing. Moreover, it must deliver replies that are consistent with its invocation history. The side-effect of a service, on third-

[1] Three-tier architectures are becoming mainstream for the Internet. In a three-tier architecture, a client typically invokes a middle-tier application server (which may be replicated), which itself invokes a back-end database.

party entities, must obey exactly-once semantics.

To deal with side-effects, x-ability is based on the notion of action execution. Actions are executed correctly (i.e., are *x-able*) if their side-effect *appears* to have happened *exactly-once*. The side-effect of actions can be the modification of a shared state or the invocation of another (replicated or not) service. Our theory represents the execution of actions as event histories. We formally define the notion of "appears to have happened exactly-once" in terms of history equivalence: an action history h is x-able if it is equivalent to a history h' obtained under failure-free conditions.[2]

We define history equivalence relative to the execution of two particular kinds of actions: *idempotent* and *undoable*. Essentially, the side-effect of a history with n incarnations of an idempotent action is equivalent to a history with a single incarnation. Writing a particular value to a data object is an idempotent action. An undoable action is similar to a transaction [GR93]: we can cancel its side-effect up to a certain point (the commit point), after which the side-effect is permanent. Thus, the side-effect of a history with a cancelled action is equivalent to the side-effect of a history with no action at all. Formalizing these intuitive properties of idempotent and undoable actions is the biggest technical difficulty in defining x-ability.

Our theory allows actions to be non-deterministic. Moreover, it is extensible: we can define equivalence rules for other types of actions without changing the basic notion of x-ability.

X-ability can be used in a recursive way to *locally* prove the correctness of composing replicated services. Let S_1 be a replicated service that is proven to be x-able, and S_2 be a replicated service that invokes S_1. We can prove the x-ability of S_2 by simply assuming that any interaction with S_1 is an idempotent action. In particular, we can reason about system correctness one service at a time. As long as all services satisfy their contracts, the system as a whole will be correct.

Being independent of particular replication protocols, x-ability introduces a unified framework to express and compare existing replication protocols. Because x-ability models services with side-effects, we can also use it to devise new replication protocols that involve third-party interaction. We designed a distributed protocol that handles the replication of services that execute non-deterministic actions with external side-effect. Interestingly, our replication protocol may vary at run-time and according to the asynchrony of the system, between some form of primary-backup and some form of active replication. In a companion paper [FG00], we use an incarnation of our protocol to replicate application servers in three-tier systems. The servers execute transactions on non-replicated, third-party databases. With our protocol, the servers provide exactly-once transaction semantics to frontend clients.

Roadmap. The rest of the paper is organized as follows. Section 2 describes our system model. Section 3 defines what it means for a history to be x-able. Section 4 defines what it means for a replicated service to be x-able. Section 5 illustrates the use of x-ability through our replication algorithm. Section 6 contrasts our work with related work. Appendix A contains more details about our x-ability theory and Appendix B contains more details about our replication protocol.

2 System Model

To formally introduce x-ability, we consider a general model where a set of process replicas implement a service. The functionality of the service is captured by a state machine. Each replica has its own copy of the state machine. Clients send requests to the service to invoke state-machine actions.

To describe the fault-tolerance semantics, and reason about correctness of a service, we associate events with the start and completion of actions.

[2]Being defined relative to failure-free executions, x-ability encompasses both safety and liveness. It is a safety property because it states that certain partial histories must not occur. It is also a liveness property since it enforces guarantees about what must occur (in fact, x-ability encompasses a notion of *wait-freedom* [Her91]).

Event histories convey the observable behavior of processes, i.e., the externally observable behavior of a service. Different *runs* of a service on the same input may produce different histories: the service may fail differently in different runs, actions may be non-deterministic, and the concurrency within the service may cause events to be interleaved differently. We use history *patterns* to abstract out some of these differences and capture structural properties of histories.

2.1 State Machines

A state machine exports a number of actions. An action takes an input value and produces an output value. In addition, an action may modify the internal state of its state machine and it may communicate with external entities. In contrast to [Sch93], our state machines may be non-deterministic. That is, the side-effect and output value of a specific action may not be the same each time we execute it, even if we execute it in the same initial state.

A client can invoke a replica's state machine by sending a request to the replica. A request contains the name of an action and an input value for the action. If no failures occur, the replica returns the action's output value to the client as a reply to the request. The execution of an action may fail (for example if the action manipulates a remote database and the database crashes), or the replica executing the action may fail. If the action fails, it returns an exception (or error) value as the execution result. Otherwise, we say that the action executes successfully.

Formally speaking, we model action names as elements of a set Action. We refer to elements of this set using the letter a. The set Value contains the input and output values associated with actions. Furthermore, we identify two sets, $\mathsf{Request}$ ($\mathsf{Request} \subseteq (\mathsf{Action} \times \mathsf{Value})$) and Result ($\mathsf{Result} \subseteq \mathsf{Value}$). A request is simply a pair value that contains an action name and an input value. We write pairs as "(a, v)" (this pair contains the action name a and the value v).

2.2 Events

A state machine represents the program that a service must execute. To reason about service correctness, we associate *events* with the execution of actions, and introduce a hypothetical event observer that can watch the occurrence of events and construct an event *history*. Events are subject to a total order that reflects the (relative) time at which they were observed.

We associate events with the start and completion of actions. The causal and temporal relationship between action execution and event observation is subject to the following axioms:

- An action's start event cannot be observed before the action is invoked.
- An action's completion event cannot be observed before its start event.
- If an action returns successfully, then its start and completion events have been observed.

In a failure-free run, the execution of an action gives rise to a start event and a completion event. If a failure occurs, an action may give rise to both events, a start event only, or no events at all.

We use events to reason about the side-effect of actions. A start event signifies that the side-effect may happen; a completion event means that the side-effect has happened (successfully).

We model events as elements of the set Event. Events are structured values with the following structure:

$$e \quad ::= \quad S(a, iv) \mid C(a, ov)$$

The event $S(a, iv)$ captures the start of executing the action a with iv as argument. The event $C(a, ov)$ captures the completion of executing the action a, and ov is the output value produced by the action.

2.3 Histories

A history is a sequence of events. The notion of a sequence captures the total order in which events are observed. We model histories as elements of the set History, and we consider histories to be structured values as defined by the following syntax:

$$h ::= \Lambda \mid e_1 \ldots e_n \mid h_1 \bullet \ldots \bullet h_n$$

The symbol Λ denotes the empty history—a history with no events. The history $e_1 \ldots e_n$ contains the events e_1 through e_n. The history $h_1 \bullet \ldots \bullet h_n$ is the concatenation of histories h_1 through h_n. The semantics of concatenating histories is to concatenate the corresponding event sequences.

The action a *appears* with input value iv in a history h if h contains a start event produced by the execution of a on iv. We write this as $(a, iv) \in h$.

2.4 Patterns

We typically consider histories that are produced by multiple processes. For example, we may want to reason about a history that is produced by a set of server processes that collectively implement a replicated service. Since processes execute concurrently, we end up with a "combined" history in which events produced by different processes are interleaved. In many cases, we want to consider this interleaving as "incidental" (or un-important), and reason about histories at a level of abstraction where histories that only differ in the particular interleaving are considered equivalent. We use history patterns (or simply patterns) to capture these higher-level structural properties.

In Figure 1, we define an abstract syntax for patterns. Formally speaking, patterns are elements of the set Pattern, and we use the letter p to refer to patterns.

$$\begin{array}{lcl} sp & ::= & [a, iv, ov] \mid ?[a, iv, ov] \\ p & ::= & sp \mid sp_1 \parallel_h sp_2 \end{array}$$

Figure 1: Abstract syntax for history patterns

$$\begin{array}{rll} & \triangleright \subseteq (\mathsf{History} \times \mathsf{Pattern}) & (1) \\ S(a, iv)C(a, ov) & \triangleright\ [a, iv, ov] & (2) \\ \Lambda & \triangleright\ ?[a, iv, ov] & (3) \\ S(a, iv) & \triangleright\ ?[a, iv, ov] & (4) \\ S(a, iv)C(a, ov) & \triangleright\ ?[a, iv, ov] & (5) \end{array}$$

Figure 2: Pattern matching rules for simple patterns

The only use for patterns is to match histories. A simple pattern sp matches single-action histories. The pattern $[a, iv, ov]$ matches a history that contains the events from a failure-free execution of an action a. The value iv is the input to a and ov is the output from a. The pattern $?[a, iv, ov]$ matches a history in which a may have failed. A matching history may be the empty history, it may contain a start event only, or it may contain both the start and completion event of a.

The pattern $sp_1 \parallel_h sp_2$ matches a history h' that contains an interleaving of three sub histories h_1, h_2, and h, where h_1 matches sp_1, h_2 matches sp_2, and h is an arbitrary history. The interleaving is constrained in the sense that the first event in h_1 must also be the first event in h' and the last event in h_2 must also be the last event in h'.

Formally speaking, pattern matching is a relation $\triangleright$ between elements of the set History and elements of the set Pattern. In other words, $\triangleright$ is a subset of History $\times$ Pattern (the set of all pairs from History and Pattern). We define this relation in figures 2 and 3.

$$\begin{array}{rclrcll} \mathsf{first}(\Lambda) & = & \Lambda & \mathsf{first}(e_1 e_2) & = & e_1 & (9) \\ \mathsf{first}(e) & = & e & \mathsf{second}(\Lambda) & = & \Lambda & (10) \\ \mathsf{second}(e) & = & e & \mathsf{second}(e_1 e_2) & = & e_2 & (11) \end{array}$$

Figure 4: The definition of first and second

A history that matches a simple pattern contains at most two events. We define two operators

$$\frac{h_1 \;\triangleright\; sp_1 \quad h_2 \;\triangleright\; sp_2}{(h_1 \bullet h \bullet h_2) \;\triangleright\; (sp_1 \parallel_h sp_2)} \tag{6}$$

$$\frac{h_1 \;\triangleright\; sp_1 \quad h_2 \;\triangleright\; sp_2}{(\mathsf{first}(h_1) \bullet h_3 \bullet \mathsf{second}(h_1) \bullet h_4 \bullet \mathsf{first}(h_2) \bullet h_5 \bullet \mathsf{second}(h_2)) \;\triangleright\; (sp_1 \parallel_{h_3 \bullet h_4 \bullet h_5} sp_2)} \tag{7}$$

$$\frac{h_1 \;\triangleright\; sp_1 \quad h_2 \;\triangleright\; sp_2}{(\mathsf{first}(h_1) \bullet h_3 \bullet \mathsf{first}(h_2) \bullet h_4 \bullet \mathsf{second}(h_1) \bullet h_5 \bullet \mathsf{second}(h_2)) \;\triangleright\; (sp_1 \parallel_{h_3 \bullet h_4 \bullet h_5} sp_2)} \tag{8}$$

Figure 3: Pattern matching rules for complex patterns

on such histories: first() and second(). We define those operators in Figure 4. The first operator returns the first element in a history, if any, and Λ otherwise. The second operator returns the second element in a history of length two, the only element in a history of length one, and the empty history otherwise.

3 X-Able Histories

To be fault-tolerant, a replicated service must be prepared to invoke the same action multiple times until the action executes successfully. To provide replication transparency, the service must have exactly-once semantics relative to its environment—the service must maintain the illusion that the action was executed once only. An x-able history is a history that maintains the illusion of exactly-once but possibly contains multiple incarnations of the same action.

3.1 History Reduction

We define a relation, $\Rightarrow$, on histories. If $h \Rightarrow h'$, then the execution that produced h has the same side-effect as an execution that produced h'. We refer to $\Rightarrow$ as a reduction operator because it is asymmetric, and h' always has fewer events than h. Essentially, a history is x-able if it can be reduced, under $\Rightarrow$, to a history that could arise from a system that does not fail.

In defining $\Rightarrow$, we consider two particular types of actions: idempotent and undoable. Informally speaking, n executions of an idempotent action has the same side-effect as a single execution of it. Thus, we write $h \Rightarrow h'$ if h contains n incarnations of an idempotent action and h' contains $n-1$ incarnations of the same action. Similarly, an undoable action is like a database transaction: it can be rolled back up to a certain point (the commit point), after which its effects are permanent. We also write $h \Rightarrow h'$ if h contains an undoable action that was rolled back and h' does not contain the action at all.

More precisely, we identify two subsets of Action: Idempotent and Undoable. The set Idempotent contains the names of idempotent actions. We use the notation a^i to indicate that the action a is idempotent. The set Undoable contains names of undoable actions. We use the notation a^u to indicate that an action a is undoable. An undoable action, a^u, has two associated actions: a cancellation action, a^{-1}, and a commit action, a^c. The commit and cancellation actions for an action a^u take the same arguments as a^u, and they return the value nil. Cancellation and commit actions are idempotent.

We then define the $\Rightarrow$ operator in terms of idempotent and undoable actions in Figure 5.

- The first inference rule (13) defines $\Rightarrow$ as a transitive relation.

- The second rule (14) captures the semantics of idempotent actions. If a history contains a successfully executed idempotent action a^i, then we can remove the events from a previous attempt to execute a^i. The events from the previous attempt and the success-

$$\Rightarrow \; \subseteq (\mathsf{History} \times \mathsf{History}) \tag{12}$$

$$\frac{h_1 \;\Rightarrow\; h_2 \qquad h_2 \;\Rightarrow\; h_3}{h_1 \;\Rightarrow\; h_3} \tag{13}$$

$$\frac{h \;\triangleright\; (?[a^i, iv, ov] \;\|_{h'}\; [a^i, iv, ov])}{h_1 \bullet h \bullet h_2 \;\Rightarrow\; h_1 \bullet h' \bullet (S(a^i, iv)C(a^i, ov)) \bullet h_2} \tag{14}$$

$$\frac{h \;\triangleright\; (?[a^u, iv, ov] \;\|_{h'}\; [a^{-1}, iv, \mathsf{nil}]) \qquad (a^u, iv) \notin h_1 \qquad (a^c, iv) \notin h'}{h_1 \bullet h \bullet h_2 \;\Rightarrow\; h_1 \bullet h' \bullet h_2} \tag{15}$$

$$\frac{h \;\triangleright\; (?[a^c, iv, \mathsf{nil}] \;\|_{h'}\; [a^c, iv, \mathsf{nil}]) \qquad (a^u, iv) \notin h'}{h_1 \bullet h \bullet h_2 \;\Rightarrow\; h_1 \bullet h' \bullet (S(a^c, iv)C(a^c, \mathsf{nil})) \bullet h_2} \tag{16}$$

Figure 5: Definition of history reduction

ful attempt can overlap. Moreover, there can be an interleaving history h' between these sets of events as well.

- The third rule (15) is concerned with cancellation of undoable actions. Intuitively, if we successfully cancel an undoable action, then we remove its side-effect (it appears as if the action was never executed). We can keep alternating between executing the action and cancelling it. But for the action to happen exactly-once, we must eventually execute it successfully and execute its commit action successfully. The rule captures when we can remove events that stem from an attempt to execute an action a^u and then cancel it.

  The sub-history h contains the events from such an action pair (a^u followed by a^{-1}). It also contains a history h' that is interleaved with the events from a^u and a^{-1}. One requirement is that h' must not contain the commit action of a^u: if we committed a^u before issuing a^{-1}, the cancellation would not take effect. Furthermore, we need this constraint on h' to ensure that an algorithm does not concurrently cancel and commit the same action.

  The requirement that $(a^u, iv) \notin h_1$ states that the preceding sub-history, h_1, cannot contain any events from a^u. Since $?[a, iv, ov]$ matches the empty history, we need to ensure that the cancellation events are not removed by themselves if they actually do cancel an action. If that is the case, we should also remove the action itself from the history. Thus, we create a constraint so that the $?[a, iv, ov]$ part of the pattern only matches the empty history if there are no events from a to the left of $?[a, iv, ov]$.

- The fourth rule (16) states that commit actions are idempotent. The requirement that $(a^u, iv) \notin h'$ ensures that the commit action and the action being committed do not overlap.

Appendix A illustrates the reduction rules by means of example history reductions.

3.2 Failure-Free Histories

A failure-free history is a history that could have been produced by a failure-free execution of a single state machine action. To define the notion of failure-free history, we define a function, called eventsof, on actions and their values. The eventsof function returns the failure-free history associated with the action and the values.

$$\begin{aligned}&\mathsf{eventsof}(a^u, iv, ov) = \\ &\quad S(a^u, iv)C(a^u, ov)S(a^c, iv)C(a^c, \mathsf{nil})\end{aligned} \tag{17}$$

$$\mathsf{eventsof}(a^i) = S(a^i, iv)C(a^i, ov) \tag{18}$$

Due to non-determinism, there are multiple failure-free histories which are possible for a given action a and a given input value iv. We define the set of all possible histories, $\mathsf{FailureFree}_{(a,iv)}$, as follows:

$$\begin{aligned}&\mathsf{FailureFree}_{(a,iv)} = \{h \in \mathsf{History} \mid \\ &\quad \exists\, ov \in \mathsf{Result} : h = \mathsf{eventsof}(a, iv, ov)\}\end{aligned} \tag{19}$$

A single-action history is x-able if it can be "reduced" to a failure-free history under the $\Rightarrow$ relation. We capture this through a predicate, x-able on histories:

$$\begin{aligned}&\mathsf{x\text{-}able}_{(a,iv)}(h) = \\ &\quad \begin{cases}\mathsf{true} & \text{if } \exists\, h' \in \mathsf{FailureFree}_{(a,iv)} : h \Rightarrow h' \\ \mathsf{false} & \text{otherwise}\end{cases}\end{aligned} \tag{20}$$

Notice that the predicate $\mathsf{x\text{-}able}_{(a,iv)}$ determines x-ability relative to a particular action-value pair.

3.3 History Signature

We need to ensure that the result delivered to the client corresponds to the server-side history. We introduce the notion of a history signature, which captures the client-side information (request and result) that is legal relative to a given server-side history. Because of non-determinism and server-side retry, a history can have multiple signatures. We define the set of signatures by the following inference rules:

$$\frac{h \;\Rightarrow\; S(a^u, iv)C(a^u, ov)S(a^c, iv)C(a^c, \mathsf{nil})}{(a, iv, ov) \in \mathsf{signature}(h)} \tag{21}$$

$$\frac{h \;\Rightarrow\; S(a^i, iv)C(a^i, ov)}{(a, iv, ov) \in \mathsf{signature}(h)} \tag{22}$$

3.4 Possible Reply Values

The execution of state machine actions may be non-deterministic. The same request may result in different reply values. For example, the state of the machine may determine the reply value, and this state may change over time.

We want to characterize the set of possible reply values for a given request. Since we do not know what state machine actions do, we cannot describe which specific values are possible. Instead, we assume the existence of a set PossibleReply that contains the possible reply values for a given request. To capture the history-sensitive nature of the set of possible replies, we define PossibleReply in the context of a request sequence $R_1 \ldots R_n$. The interpretation of PossibleReply in the context of a sequence is the set of possible replies to request R_n after the state machine has executed the requests $R_1 \ldots R_{n-1}$ one after the other. Thus, we write the set as: $\mathsf{PossibleReply}_{(R_1 \ldots R_n)}$.

Notice that the set PossibleReply is defined for state machines, not replicated services. Thus, there is no notion of failures or replication involved in its definition. The set is well-defined for state machines in general.

4 X-Able Services

We provide here a formal specification of replication that is independent of a particular replication protocol. We can implement the specification with various protocols, including protocols that have a primary-backup flavor and protocols that have an active-replication flavor. Moreover, the specification takes side-effects and non-determinism into account.

Formally speaking, a replicated service consists of a server-side state machine S and a client-side action *submit*. The state machine captures the functionality of the service. It is executed by a set of server processes $s_1 \ldots s_n$ that each has a copy of S. These are the only processes that have a copy of S. The action *submit* can be used by any process p to invoke the service. The action takes a value in the domain Request and, when executed, produces a value in the domain Result. We specify correctness relative to a single client C. Thus, we consider a system that consists of the processes $s_1 \ldots s_n$ and C only. The client submits one request at a time, and we can observe the server-side history for each request. The service is x-able if the following conditions hold:

R1. The action *submit* is idempotent.

R2. The client C will eventually be able to execute *submit* successfully.

R3. If the client submits a request (a, iv), then the server-side history for (a, iv) is either empty or it satisfies $\mathsf{x\text{-}able}_{(a,iv)}$.

R4. If the client receives a reply ov in response to a request (a, iv), and if the server-side history for executing this request is h, then $(a, iv, ov) \in \mathsf{signature}(h)$.

R5. If the client successfully submits $R_1 \ldots R_n$ and receives the reply R' in response to R_n, then R' is in $\mathsf{PossibleReply}_{(R_1 \ldots R_n)}$.

The first two requirements (R1 and R2) are concerned with the contract between a service and its clients. Clients use the action *submit* to invoke the service. Because *submit* is idempotent, clients can repeatedly invoke the service without concern for duplicating side-effects. The second requirement (R2) is a liveness property. The action *submit* is not allowed to fail an infinite number of times. The requirement also makes a service non-blocking in the sense that the *submit* is guaranteed to eventually return a value. In addition, *submit* is free to fail a finite number of times and return an error value (a value that does not belong to Result). The combination of the first two requirements facilitates composition of services. Since a replicated service can execute idempotent actions that eventually succeed, it can invoke another replicated service and view its invocation as an idempotent action.

The third requirement (R3) deals with the server-side side-effect of executing a request. The resulting server-side history must be x-able, that is, it must be equivalent (under history reduction) to a failure-free history.

The fourth requirement (R4) forces an algorithm to preserve consistency between the client-side view (request and reply) and the server-side view (the side-effect). This requirement, prevents the *submit* action from inventing reply values. It also prevents the service from inventing request values.

The fifth requirement (R5) forces the service to correctly maintain S's state, if any. The server-side history must be equivalent to a failure-free execution of the sequence $R_1 \ldots R_n$. But since R_1 may result in a transformation of S's state, the actions executed for R_2 may depend on this state transformation. So, a replication algorithm must ensure that the state resulting from R_1 is used as a context for executing R_2. The replication algorithm cannot assume that R_1 did not update the state of S, or that the state update is immaterial to the processing of R_2.

5 A Replication Algorithm

Being independent of particular replication protocols, x-ability introduces a unified framework to express and compare existing replication protocols. Because x-ability models services with side-effects, we can also use it to devise new replication protocols that involve third-party interaction. We sketch the principle of a distributed replication protocol that handles nondeterministic actions with external side-effect. We also discuss how traditional replication protocols can be viewed as specialized incarnations of our protocol.

At first glance, it may appear trivial to guarantee exactly-once execution for idempotent and undoable actions: we can always retry an idempotent action, and we can always cancel an undoable action, and try again, if the action appears to have failed. However, the trick is to coordinate the execution logic with the retry logic so that there is agreement on the result of a nondeterministic idempotent action and on the outcome (abort or commit) of an undoable action.

Interestingly, our protocol may vary, at runtime and according to the asynchrony of the system, between some form of active replication [Sch93], and some form of primary-backup [BMST93].

5.1 Protocol Description (sketch)

Our replication algorithm is round-based. Within each round, a particular replica behaves as a primary: it tries to compute the request submitted by the client and sends back a reply. Rounds are not synchronized, and there may be several primaries acting at the same time—one for each round. For presentation simplicity, we consider only the case of a single client, submitting a single request to the replicated service.

Basically, the client sends the request to a single replica and then waits until it either receives a reply from the replica or it suspects the replica to have failed, in which case it sends its request to another replica. In a "nice" run, where no replica crashes, or is suspected to have crashed, the protocol goes as follows. The replica that receives the client's request becomes the primary in the first round: it executes the corresponding state machine action, and replies to the client. In such a run, our replication scheme is very much like a primary-backup scheme applied to general actions that might have external side effect.

In a run where a replica q suspects the crash of a primary replica p, q tries to terminate the action executed by p: if p was executing an undoable action, q tries to abort the action; if p was executing a non-deterministic idempotent action, q prevents p from responding to the client. After terminating the possible ongoing action execution, q initiates a new round, and tries to become primary for that round.

Because of false failure suspicions, we may very well end-up in the situation where multiple replicas concurrently execute the same request (in different rounds): in such a configuration, our replication scheme is very much like an active replication scheme (applied to general actions that might be non-deterministic and have external side effect).

It is important to notice that since actions might fail, a primary typically needs to keep reissuing them until they succeed. Idempotent actions are simply repeated. For undoable actions, the procedure is slightly more complicated. If an undoable action fails, we apply its cancellation action first.

The complete protocol is given in Appendix B. As we point out in the appendix, the protocol tolerates crash failures and its correctness relies on the assumptions of (1) reliable channels; (2) the existence of at least one correct process; (3) a failure detector used by the client that eventually detects the crash of every replica; (4) an eventually perfect failure detector among the replicas [CT96]; and the existence of a consensus object [Her91] to ensure agreement among the replicas on results, outcomes, and primaries.

5.2 Active Replication and Primary-Backup

Our protocol is a general replication protocol. Traditional replication protocols can be viewed as special cases that handle particular subsets of actions.

If we assume a perfect failure detector (in the sense of [CT96]), we can optimize our protocol by eliminating the need for agreement on primaries. Furthermore, if we assume that actions only update local state and computes a result, then we can also eliminate the need for agreement on the outcome of undoable actions. Interestingly, the resulting protocol turns out to be very similar to a traditional primary-backup protocol.

If we assume that the client directly sends its request to all replicas, then every replica will actively try to execute the corresponding action and reply to the client. If we assume that actions are both deterministic and idempotent, with respect to their environment, then we can eliminate the need for our three agreement steps. In this case, the resulting protocol turns out to be very similar to a conventional active-replication scheme.[3]

[3]If we assume multiple clients, in order to ensure that actions indeed remain deterministic, some total ordering of messages will be needed, just as in [Sch93].

6 Concluding Remarks

The role of x-ability for replicated programs is similar to that of linearizability for concurrent objects [HW90] and serializability for concurrent transactions [Pap79]. It facilitates certain kinds of formal reasoning by transforming assertions about complex replicated behavior (resp. concurrent for [HW90, Pap79]) into assertions about simpler non-replicated (resp. sequential for [HW90, Pap79]) behavior.

Considering that a replicated program is correct if it can somehow be shown to be *equivalent* to a non-replicated program is an intuitive idea, and this idea has already been explored by different authors. The definition of equivalence is the main difference between the various approaches:

- In [MP88], an algebra of action sequences is used to define a correctness criterion for replication. The N replication of a base process is a replicated process, denoted by P^N. The replicated process P^N is correct if it is possible to *extract*, from every trace t^N of P^N, a trace t of P. The authors assume the existence of a generic *extract* function, and describe an implementation example of that function for deterministic pure server processes (that do not interact with third party entities). As pointed out by the authors, it is not clear how to devise such a function for non-deterministic programs. It is also not clear how to express it for services that invoke third-party entities.

- In [Aiz89], the author defines a reduction relation between programs in terms of *refinement mapping*, using temporal logic descriptions of state sequences. The author does not describe a mechanical way of performing the reduction, but rather suggests a methodology for transforming a non-replicated program into a replicated one.

Our reduction technique is much simpler than those considered in [MP88] and [Aiz89]: we describe simple rewriting rules that *mechanically* exploit idempotence and undoability properties of actions. In this sense, our theory is closer to the theory of 1-*copy serializability* [BHG87], which exploits the semantics of *read()* and *write()* operations: a replicated history is view-equivalent to a non-replicated one if they have the same *reads-from* relationships and final *writes*. There are however many differences between x-ability and 1-copy serializability. First, 1-copy serializability assumes replicated entities to be data servers on which *read()* and *write()* operations can be performed.[4] We more generally assume replicated entities to execute arbitrary actions, that may very well be operations on data objects, but also non-deterministic invocations of third party entities (which enables us to state interesting properties about replication composition). Second, we do not restrict ourselves to committed actions, and we integrate liveness in the x-ability theory. Third, x-ability does not directly handle concurrent invocations of a replicated service. More precisely, x-ability states constraints about the concurrency among replicas in the context of a given request ("intra-request" concurrency), but ignores the concurrency that originates from different requests (from different clients). The latter kind of concurrency is indirectly viewed in our case as a source of non-determinism of actions.[5] Finally, and as we pointed out, unlike serializability, but (somehow) like linearizability, x-ability is a local property of replicated services.

References

[Aiz89] J. Aizikowitz. Designing distributed services using refinement mappings. Technical Report CS TR 89-1040, Cornell University, 1989.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass., 1987.

[4]One could consider operations on more complex data objects, along the lines of [LMWF94], but the underlying model would remain that of replicated data servers.

[5]We believe the decoupling of concurrency and duplication to be an important step towards the design of more modular replication protocols.

[BMST93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*. Addison-Wesley, 1993.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[FG00] S. Frolund and R. Guerraoui. Implementing e-transactions with asynchronous replication. Technical report, Hewlett-Packard Laboratories, February 2000. To appear.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[HW90] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[LMWF94] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions.* Morgan-Kaufmann, 1994.

[MP88] L. Mancini and G. Pappalardo. Towards a theory of replicated processings. In *Formal Techniques in Real-time and Fault-tolerant Systems*, pages 175–192. LNCS (331), Springer Verlag, 1988.

[Pap79] C. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[Sch93] F. B. Schneider. Replication management using the state machine approach. In S. Mullender, editor, *Distributed Systems.* Addison-Wesley, 1993.

A Examples Of History Reduction

To illustrate the semantics of the rules in Figure 5, we show some example reductions. The goal of the rules is to formally capture our intuition about idempotent and undoable actions. We want the rules to allow elimination of "enough" events, but not "too many," relative to our intuition. We demonstrate both aspects. We show reductions that are possible, and we identify reductions that are impossible. We focus on undoable actions and their associated cancellation and commit actions. Although the rules are given as inference rules, we do not describe the corresponding proof trees. Instead, we describe reduction sequences, and argue in the text why a particular reduction rule is possible or why no reduction rule is possible.

Consider the reduction of history h_1 in Figure 6. We use [and] to demarcate the events that are eliminated in a reduction step. The first step uses the idempotence property of cancellation actions (14). We gather the events from the failed and successful cancellation action, and eliminate the events from the failed cancellation action. Then we apply the rule for undoable actions (15) to eliminate the undoable action a^u and the the cancellation action. The resulting history is now equivalent to a failure-free history.

The history h_2 in Figure 6 shows that we cannot have interleaving commit and cancel actions. We apply the idempotence rule for cancellation actions to gather the cancellation events into a consistent cancellation action. This is the first reduction step. However, we cannot apply the rule for undoable actions (15) to the resulting history. In the rule, the history h' must not contain any commit events. But in the above history, h' would be equal to $S(a^c, iv)$, which means that we cannot use the rule.

B A General Replication Algorithm

We present a general, asynchronous replication algorithm. The algorithm is general in the sense that it handles the replication of services that may execute actions that are non-deterministic

$$\begin{aligned}
h_1 &= S(a^u, iv)[S(a^{-1}, iv)S(a^{-1}, iv)C(a^{-1}, \mathsf{nil})]S(a^u, iv)C(a^u, ov)S(a^c, iv)C(a^c, \mathsf{nil}) \\
&\qquad \overset{(14)}{\Rightarrow} [S(a^u, iv)S(a^{-1}, iv)C(a^{-1}, \mathsf{nil})]S(a^u, iv)C(a^u, ov)S(a^c, iv)C(a^c, \mathsf{nil}) \\
&\qquad \overset{(15)}{\Rightarrow} S(a^u, iv)C(a^u, ov)S(a^c, iv)C(a^c, \mathsf{nil}) \\
h_2 &= S(a^u, iv)[S(a^{-1}, iv)S(a^{-1}, iv)]S(a^c, iv)[C(a^{-1}, \mathsf{nil})]S(a^u, iv)C(a^u, ov)S(a^c, iv)C(a^c, \mathsf{nil}) \\
&\qquad \overset{(14)}{\Rightarrow} S(a^u, iv)S(a^{-1}, iv)S(a^c, iv)C(a^{-1}, \mathsf{nil})S(a^u, iv)C(a^u, ov)S(a^c, iv)C(a^c, \mathsf{nil})
\end{aligned}$$

Figure 6: Examples of history reduction

and have external side-effect. It is asynchronous in the sense that it may vary, at run-time, and according to the asynchrony of the system, between some form of active replication [Sch93], and some form of primary-backup [BMST93]. We describe the algorithm and then prove its correctness, i.e., we show that every service replicated using this algorithm is x-able.

B.1 Overview

Our replication algorithm is mainly composed of two parts. A client part, described in Figure 7[6], and the replica part, described in Figure 8. For presentation simplicity, we consider only the case of a single client, submitting a single request to the replicated service. The replicated service is implemented by n replicas. Basically, the client sends the request to a single replica and then waits until it either suspects the replica to have failed or receives a result from the replica. All replicas execute the same protocol (Figure 8).

In a "nice" run, where no replica crashes or is suspected to have crashed, the protocol goes as follows. The replica that receives the client's request, executes the corresponding state machine action, and sends back the resulting reply to the client. In such a run, the replication scheme is very much like a primary-back scheme (applied to general actions that might have external side effect).

Any replica that suspects the crash of the primary tries to terminate the action execution by the primary: if the primary was executing an undoable action, the replica aborts this action; if the primary was executing a non-deterministic idempotent action, the replica prevents the primary from responding to the client. After terminating the possible ongoing action execution, the replica initiates a new round, and tries to become primary for that round.

Because of false failure suspicions, we may very well end-up in the situation where all replicas concurrently execute actions on behalf of the same clients (in different rounds): in such a configuration, our replication scheme is very much like an active replication scheme (applied to general actions that might be non-deterministic and have external side effect).

B.2 Assumptions

We assume that processes (client and replicas) fail by crashing. They do not recover after a crash, neither do they ever behave maliciously. A correct process is one that does not fail, and we assume the existence of at least one correct replica process. We assume that communication channels are reliable: there is no message creation or duplication and if a correct process sends a message to another correct process, then the message is eventually received. We also assume that every action is eventually successful. If we keep invoking an action, it will eventually execute to successful completion. Furthermore, we assume that a successfully executed undoable action can

[6]In fact, the figure actually describes the algorithm executed by the client's *stub*. In the presentation, we simply do not distinguish between the client and the client's stub.

be committed.

In order to ensure that the service is indeed x-able, we rely on two kinds of abstractions:[7]

1. *Failure detector* [CT96]. The failure detector is a distributed oracle that provides hints about failed processes. The client uses the failure detector to monitor the crashes of replicas, and every replica uses the failure detector to monitor the crashes of other replicas. We assume here that the client's failure detector satisfies the *strong completeness* property [CT96]: eventually, every crashed replica is suspected by the client. Among the replicas, we assume the failure detector to be *eventually perfect* [CT96]. Besides strong completeness, it also ensures *eventual strong accuracy*: eventually, no replica is suspected unless it has crashed. These assumptions are needed to guarantee progress. If a replica suspects another replica, it will try to clean up the execution state of the suspected replica. For undoable actions, this means cancelling the actions. Thus, if we forever have false suspicions, the same action could in principle be cancelled over and over again.

2. *Consensus object* [Her91]. The consensus abstraction is used for three kinds of synchronization: (1) to ensure agreement about which replica is primary for a given round, (2) to ensure agreement about the outcome of undoable actions (commit or abort), and (3) to ensure agreement about the replies of idempotent actions (these might be non-deterministic). The consensus abstraction is used here through two primitives: a *propose()* primitive which takes as input a value proposed for consensus, and returns the value decided, and a *read()* primitive that returns the value decided, if any, or $\perp$ if no such value has been decided.

[7]We simply assume here the existence of these abstractions, i.e., we do not discuss their implementation in a message passing system.

B.3 The pseudo-code

We discuss below the semantics of our C++-like pseudo-code we use in Figure 7, Figure 8, and Figure 9 to describe our algorithm

A channel is specified by two primitives: **send** and **receive**. For example, the statement "**send** [Request,req] **to** p_j" captures the action of sending the message [Request,req] to process p_j. A message [Request,req] is of type "Request" and contains the value req. We assume that messages are uniquely identified. In many cases, servers acknowledge receipt of messages. We assume that the receiver of an acknowledgment message can correlate it with the message being acknowledged. This can be achieved by appropriate tagging of acknowledgment messages. However, to simplify the presentation, we do not describe this tagging and correlation in our protocol. The statement "**receive** [Request,req] **from** p_i" captures the action of waiting for a message of type "Request" from process p_i. When such a message arrives, the variable req is assigned to the contents of the message, and the variable p_i is assigned to the sender's identity. We also use the receive primitive without a "from" part if we do not need to assign the sender's identity to a variable.

Besides message passing, we also use various synchronization primitives. We use "**await**" statements to wait for an event to occur. Events can be the reception of messages and detection of failures. We use **and** and **or** combinators to specify these event sets. Traditional control structures, such as branches and loops, are used with their usual semantics. In addition, we also use **cobegin** and **coend** to capture concurrent executions. The **cobegin** statement terminates when any of the contained activities terminates. We use "==" (resp. "!=") to compare values for equality (resp. non-equality) and ":=" for assignment. Finally, we abstract the suspicion information through a predicate *suspect()*. The execution of $suspect(p_i)$ by process p_j at t returns `true` if and only if p_j suspects p_i at time t.

```
Client {
  Process replicas[n];
  Int i = 1;

  Result submit(Request req) {
    Result res;
    send [Request,req] to replicas[i];
    await (receive [Result,res]) or
      suspect(replicas[i]);
    if(received [Result,res]) then
       return res;
    else
      i = (i +1) mod n;
      return failure;
    }
}
```

Figure 7: Client-side algorithm

B.4 Algorithm Description

The client part of the algorithm consist of the *submit* primitive described in Figure 7. The *submit* primitive sends a request to one of the replicas. It then waits for a result, and if it has suspected the replica, it returns an error. The primitive uses two "global" variables `replicas` and `i`. The variable `replicas` contains a list of the replicas. The variable `i` is the replica to contact next time *submit* is executed.

All replicas execute the same algorithm, and they all have a copy of the same state machine `S`. Rather than describe invocation of state machine actions directly, we assume that a state machine has a method, called `execute`, that "dispatches" a request. A request contains the name of an action and a list of input parameters for the action. One of these parameters is called `round`, and it keeps track of the current execution round of the request (the server-side algorithm increments this parameter when a new round is initiated). Having the round number as part of the parameters ensures that commit and cancellation actions are specific to a particular round. Thus, a cancellation action issued for round number n cannot cancel the action of round number $n+1$.

Round number one is initiated by a replica p_1, that receives a request from the client. This replica starts executing the requested state machine action. If it does not fail, and is not suspected to have failed, p_1 executes the action to completion and returns the result to the client. If another replica p_i suspects p_1 to have failed, p_i will start round two as a continuation of round one. Each round is owned by a single replica, and a replica only takes ownership of rounds greater than one if they suspect another owner to have failed.

In Figure 8, we show the behavior of the main part of a server replica. A server contains two activities: a thread to receive and execute requests and a thread to perform failure detection cleanup. Since the failure suspicion may be false—the replica may be suspected, but has actually not failed—we need to coordinate the actions taken by cleaner threads and replicas during request processing since they may execute in parallel.

Each replica has access to three arrays of consensus objects. The `owner-agreement` array contains consensus objects that control ownership of particular rounds. This array has a total of `max-round` objects. If a replica wishes to become the owner of round i, it will try to propose its own identity as the value of consensus object number i in the array. The `outcome-agreement` array contains consensus objects that implement the required coordination on the outcome (commit or abort) of undoable actions. Finally, the `result-agreement` array contains consensus objects that ensure agreement on the result of idempotent actions.

Different rounds can have a different outcome for the same undoable action. For example, we may have a number of rounds in which the outcome is abort followed by a single round in which the outcome is commit. Subsequent rounds will then not execute the action once it has been successfully committed. The `outcome-agreement` is indexed by requests, which have the round number as part of their parameters. The `owner-agreement` array is uni-dimensional because there is one owner per round, and `result-agreement` array is uni-dimensional

```
Server {
  Consensus(Process,Request,Process)
    owner-agreement[max-round];
  Consensus(Result)
    result-agreement[Request];
  Consensus(Outcome,Result)
    outcome-agreement[Request];
  Result result-store[Request];
  State-machine S;

  cobegin
    Request req; Process client;
    while true {
      receive [Request,req] from client;
      req.round := 1;
      this->process-request(req,client);
      }
  ||
    this->cleaner();
  coend;
}

Server::process-request(Request r,Process cl){
  Process id,tmp-client; Request tmp-val;
  (id,tmp-req,tmp-client) :=
    owner-agreement[r.round].propose(my-id,r,cl);
  if id == my-id then
    if result-store[r] != nil then
      res-val := result-store[r];
    else
      Result res-val := execute-until-success(r);
      res-val := result-coordination(r,res-val);
      result-store[r] := res-val;
    if res-val != empty-result then
      send [Result,res-val] to cl;
}

Server::cleaner(){
  while true {
    Process id,suspected-id,client; Request r;
    if suspect(suspected-id) then
      let last-round be the largest defined
      index in owner-agreement;
      (id,r,client) :=
        owner-agreement[last-round].read();
      if id == suspected-id then
        res-val :=
          result-coord(r,empty-result);
        if res-val == empty-result then
          r.round := last-round + 1;
          this->process-request(r,client);
    }
}
```

Figure 8: Main algorithm on server side

```
Result Server::result-coord(Request r,Value v){
  Result res-val; Outcome outcome;
  if S.is-idempotent(r) then
    res-val := result-agreement[r].propose(v);
  if S.is-undoable(r) then
    if val == empty-result then
      (outcome,res-val) :=
        outcome-agreement[r.round].
          propose(abort,val);
    else
      (outcome,res-val) :=
        outcome-agreement[r.round].
          propose(commit,val);
    if outcome == abort then
      this->execute-until-success(cancel(r));
    else
      this->execute-until-success(commit(r));
  return res-val;
}

Result Server::execute-until-success(Request r){
  while true {
    Result res-val;
    try res-val := S.execute(r);
    catch(failure)
      if S.is-idempotent(r) then
        continue;
      if S.is-undoable(r) then
        this->execute-until-success(cancel(r));
        continue;
    return res-val;
    }
}
```

Figure 9: Algorithms to execute and clean sequences of actions

because the result can be fixed the first time an idempotent action is successfully executed.

The local array, `result-store`, ensures that each replica only executes a round that owns once. A replica stores the result of a round into this array, and examines the array prior to executing any action. Due to the `owner-agreement`, a replica cannot re-execute a round owned by another replica.

The method called `result-coordination` in Figure 8 implements the required coordination of action results. The method can be used in two "modes:" cleaning mode and execution mode. In cleaning mode, the method is used to prevent a suspected primary from enforcing its action results. In execution mode, it is used to propose a value that is the result of a successfully executed action. The parameter `val` determines the mode. If it contains the value `empty-result`, we are in cleaning mode. If it contains a regular value, that value is used as the agreed upon result.

The method `execute-until-success` executes a state machine action until it succeeds. For idempotent actions, we simply keep reissuing the action. For undoable actions, the procedure is slightly more complicated. If an undoable action fails, we apply its cancellation action. We obtain the name of a cancellation action by using the primitive `cancel`. This primitive takes a request `r`, and returns a request that invokes the cancellation action of `r`. We construct commit actions in a similar manner by using the primitive called `commit`.

B.5 Protocol Correctness

To discuss the correctness of our protocol, we consider below, and separately, each of the properties of an x-able service.

Proposition (R1) The action *submit* is idempotent.

PROOF (SKETCH): Before returning any reply to the client, every replica first stores the reply in the consensus object `result-agreement`, and there is one such object per request. By the properties of consensus, even if a client invokes a request several times, and on different replicas, the same reply is returned. □

Proposition (R2) The client is eventually able to execute *submit* successfully.

PROOF (SKETCH): By the assumption of reliable channels and the wait-free property of consensus, no process remains indefinitely blocked waiting for a message or a consensus access. By the completeness property of the failure detector, every crashed process is eventually suspected. As we assume that some replica is correct, then eventually some correct process becomes primary and that process is not suspected. Since we assume that actions eventually succeed, then if the client keeps invoking the action *submit*, it eventually receives a reply back. □

Proposition (R3) If the client submits a request (a, iv), then the server-side history for (a, iv) is either empty or it satisfies $\textsf{x-able}_{(a,iv)}$.

PROOF (SKETCH): Consider a request that invokes an idempotent action. The server-side history only contains events from executing this action. The key requirement is that the history must end with a successful execution of the action. If this requirement is satisfied, we can eliminate the events from all previous executions, whether they are successful or not. All histories produced by the algorithm does indeed satisfy this requirement. We ensure this through the `result-agreement` consensus objects. Essentially, a replica only stores an action's result in this array if it executed the action successfully. Moreover, no replica retries an action whose result is stored in the array. Finally, a replica will eventually store a result in the array. This is due to the completion properties of actions and the eventually perfect failure detection between replicas.

For an undoable action, the server-side history may contain events from executing the action as well as events from executing its cancellation and commit actions. Our algorithm executes an undoable action in a number of rounds, and it satisfies the following properties. (1) A given round will contain events from either a commit or cancellation action, but not both. Moreover, (2) round number n will only be committed if round $n-1$ was aborted. And (3) the same round is only executed once. Finally, (4) the algorithm will eventually execute a commit round. These properties are due to use of the `outcome-agreement` consensus objects, the result store, and the completion properties of actions and failure-detection. In terms of reduction rules, each round of an undoable action can be analyzed separately: the events from different rounds will not match the same pattern. This is because each round has a unique input value (the round number is part of the input value). An aborted round produces a history with some events from the action itself as well as its cancellation action. We can completely remove the events from an aborted round using the idempotence rule for cancellation actions, and the rule for undoable actions. A committed round contains events from the action as well as its commit action. We can use the idempotence rule for commit actions to reduce these events to a history that contains only a successful execution of the action followed by a successful execution of its commit action. Thus, we can remove all the aborted rounds from the server-side history, and the committed round can be reduced to an x-able history. □

Proposition (R4) If the client receives a reply ov in response to a request (a, iv), and if the server-side history for executing this request is h, then $(a, iv, ov) \in \mathsf{signature}(h)$.

Proof (sketch): By the no-creation property of reliable channels, a client can only receive a reply that was computed by some replica. Conversely, a replica can only receive a request that was sent by the client.

Proposition (R5) If the client successfully submits $R_1 \ldots R_n$ and receives the reply R' in response to R_n, then R' is in $\mathsf{PossibleReply}_{(R_1 \ldots R_n)}$.

Proof (sketch): This property is trivially satisfied since we only consider a single request. □