X-Ability: A Theory of Replication

Svend Frølund[1] Rachid Guerraoui[2]
[1] Hewlett-Packard Laboratories, 1501 Page Mill Rd, Palo Alto
[2] Swiss Federal Institute of Technology, CH 1015, Lausanne

January 15, 2000

Abstract

This paper presents *x-ability (Exactly-once-ability)*: a correctness criteria for replicated services. *X-ability* provides the illusion that the actions executed by a replicated service are executed *exactly-once*, even if these actions have been actually executed several times and by various replicas. A client can treat a *x-able* replicated service as if it was not replicated, even if this service executes actions that are non-deterministic and have side-effects on the environment, e.g., invoke other services. *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 our theory through an asynchronous replication protocol that handles non-determinism and external side-effects. The replication protocol is asynchronous in the sense that it 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.

1 Introduction

Background. There has a been a significant body of literature in the last decade about replication algorithms. Surprisingly, there is no satisfactory specification of what it precisely means for a replication algorithm to be correct. 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 "schemes" rather than 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. In particular, there is no provisioning, in the specifications, of having a replicated service call a third party entity, e.g., another replicated service. This form of interaction is however common in practice.[1]

X-ability in short. This paper presents a correctness criteria for replicated action execution, which we call *x-ability (Exactly-once-ability)*. The notion of *x-ability* is independent of a particular replication algorithm: it can be viewed as the specification of the problem solved by the so-called *transparent* replication, where the aim is to provide the illusion that, as long as the functional

[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 (which may as well be replicated).

behavior of the service is concerned, replication is transparent and the client of the service typically invokes it as if was implemented by a single process.

Roughly speaking, a sequence of actions is executed correctly (i.e., is *x-able*) if their side-effect *appears* to have happened *exactly-once*. The side-effect of the action can be the modification of a shared state or the invocation of another (replicated or not) service. Actions can be non-deterministic as can the particular sequence of actions executed in response to a request. We formally define the notion of "appears to have happened *exactly-once*" through history reductions. An action history h is *x-able* if h can be reduced to a history h' where every action is executed *exactly-once*. We base history reductions on rewriting rules that exploit *idempotence* and *undoability* properties of actions. Basically, an idempotent action has the same side-effect whether executed once or multiple times, and an undoable action is one that either commits, or aborts and have its side-effect cancelled [GR93]. *X-ability* is a *local* property. It can be used in a recursive way to *locally* prove the correctness of composable replicated services. Let S_1 be a replicated service that is proved 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 interation with S_1 is an idempotent action.

X-ability in perspective. 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. *X-ability* is similar to *1-copy serializability* [BHG87] in that it considers a replicated program to be correct if it is "equivalent" to a non-replicated program. However, *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 does ignore 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.[2] On the other hand, our underlying model is more general than those considered for linearizability and serializability (or 1-copy serializability) in that we do not only consider operations on data objects, but arbitrary actions that may involve third party entities. Furthermore, *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 (*x-ability* encompasses a notion of *wait-freedom* [Her91]). Finally, and as we pointed out, unlike serializability, but (somehow) like linearizability, *x-ability* is a *local* property of replicated services.

X-ability in use. To illustrate the use of *x-ability*, we present a replication protocol where replicas may invoke non-deterministic actions on third-party entities. We consider here idempotent and undoable actions. 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 when to cancel an undoable action. Our replication

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

protocol has an asynchronous flavor. Unlike in primary-backup replication [BMST93], we do not make any assumption about the existence of a single leader at every given time and we tolerate unreliable failure detection. Unlike active replication [Sch93], we do not assume that replicas are deterministic. Our replication protocol is asynchronous: it 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.

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 asynchronous replication scheme. Section 6 contrasts our work with related work.

2 System model

To describe 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. 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 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 result of execution.

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, $\mathsf{Request}$ and Result, that are defined as follows:

$$\mathsf{Request} \subseteq (\mathsf{Action} \times \mathsf{Value}) \tag{1}$$

$$\mathsf{Result} \subseteq \mathsf{Value} \tag{2}$$

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 also need a way represent executions of this program. 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 will always give 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 can 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:

$$\begin{array}{rcl} h & ::= & \Lambda \\ & \mid & e_1 \ldots e_n \\ & \mid & h_1 \bullet \ldots \bullet h_n \end{array}$$

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:

$$\frac{h_1 = e_1 \ \ldots \ e_n \quad h_2 = e_{n+1} \ \ldots \ e_m}{h_1 \bullet h_2 \ = \ e_1 \ \ldots \ e_n \, e_{n+1} \ \ldots \ e_m} \tag{3}$$

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 $(a, iv) \ \in \ h$, and we formally define the semantics of $\in$ for histories as follows:

$$(a, iv) \ \in \ e_1 \ \ldots \ e_n \ = \begin{cases} \mathsf{true} & \text{if } \exists\, i : 1 \leq i \leq n \wedge e_i = S(a, iv) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

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

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_3, 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 Figure 2.

A history that matches a simple pattern contains at most two events. We define two operators on such histories: first() and second(). We define those operators in Figure 3. The first operator

$$\triangleright \subseteq (\mathsf{History} \times \mathsf{Pattern}) \tag{4}$$

$$S(a, iv)C(a, ov) \ \triangleright \ [a, iv, ov] \tag{5}$$

$$\Lambda \ \triangleright \ ?[a, iv, ov] \tag{6}$$

$$S(a, iv) \ \triangleright \ ?[a, iv, ov] \tag{7}$$

$$S(a, iv)C(a, ov) \ \triangleright \ ?[a, iv, ov] \tag{8}$$

$$\frac{h_1 \ \triangleright \ sp_1 \quad h_2 \ \triangleright \ sp_2}{(h_1 \bullet h \bullet h_2) \ \triangleright \ (sp_1 \parallel_h sp_2)} \tag{9}$$

$$\frac{h_1 \ \triangleright \ sp_1 \quad h_2 \ \triangleright \ sp_2}{(\mathsf{first}(h_1) \bullet h_3 \bullet \mathsf{second}(h_1) \bullet h_4 \bullet \mathsf{first}(h_2) \bullet h_5 \bullet \mathsf{second}(h_2)) \ \triangleright \ (sp_1 \parallel_{h_3 \bullet h_4 \bullet h_5} sp_2)} \tag{10}$$

$$\frac{h_1 \ \triangleright \ sp_1 \quad h_2 \ \triangleright \ sp_2}{(\mathsf{first}(h_1) \bullet h_3 \bullet \mathsf{first}(h_2) \bullet h_4 \bullet \mathsf{second}(h_1) \bullet h_5 \bullet \mathsf{second}(h_2)) \ \triangleright \ (sp_1 \parallel_{h_3 \bullet h_4 \bullet h_5} sp_2)} \tag{11}$$

Figure 2: Semantics of Pattern matching

$$\mathsf{first} : \mathsf{History} \rightarrow \mathsf{History} \tag{12}$$

$$\mathsf{second} : \mathsf{History} \rightarrow \mathsf{History} \tag{13}$$

$$\mathsf{first}(\Lambda) = \Lambda \quad \mathsf{first}(e_1 e_2) = e_1 \quad \mathsf{first}(e) = e \tag{14}$$

$$\mathsf{second}(\Lambda) = \Lambda \quad \mathsf{second}(e) = e \quad \mathsf{second}(e_1 e_2) = e_2 \tag{15}$$

Figure 3: The definition of first and second

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 it completes successfully. At the same time, 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 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 is *identical* to a history that is produced by system without failures.

Then main idea in defining $\Rightarrow$ is to 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 can say that $h \Rightarrow h'$ if h contains n incarnations of an 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. In terms of undoable actions, we can say that $h \Rightarrow h'$ if h contains an undoable action that was rolled back and if h' does not contain the action at all.

More formally speaking, 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.

$$\Rightarrow \subseteq (\mathsf{History} \times \mathsf{History}) \tag{16}$$

$$\frac{h_1 \Rightarrow h_2 \qquad h_2 \Rightarrow h_3}{h_1 \Rightarrow h_3} \tag{17}$$

$$\frac{h \rhd (?[a^i, iv, ov] \parallel_{h'} [a^i, iv, ov])}{h_1 \bullet h \bullet h_2 \Rightarrow h_1 \bullet h' \bullet (S(a^i, iv)C(a^i, ov)) \bullet h_2} \tag{18}$$

$$\frac{h \rhd (?[a^u, iv, ov] \parallel_{h'} [a^{-1}, iv, \mathsf{nil}]) \qquad (a^u, iv) \notin h_1 \qquad (a^c, iv) \notin h'}{h_1 \bullet h \bullet h_2 \Rightarrow h_1 \bullet h' \bullet h_2} \tag{19}$$

$$\frac{h \rhd (?[a^c, iv, \mathsf{nil}] \parallel_{h'} [a^c, iv, \mathsf{nil}]) \qquad (a^u, iv) \notin h'}{h_1 \bullet h \bullet h_2 \Rightarrow h_1 \bullet h' \bullet (S(a^c, iv)C(a^c, \mathsf{nil})) \bullet h_2} \tag{20}$$

Figure 4: Definition of history reduction

We then define the $\Rightarrow$ operator in terms of idempotent and undoable actions in Figure 4.

- The first inference rule (17) defines $\Rightarrow$ as a transitive relation.
- The second rule (18) 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 successful attempt can overlap. Moreover, there can be an interleaving history h' between these sets of events as well.
- The third rule (19) 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 the 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 preceeding 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 (20) 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.

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 $\mathsf{eventsof}$, on actions and their values. The $\mathsf{eventsof}$ function returns the failure-free history associated with the action and the values.

$$\mathsf{eventsof}(a^u, iv, ov) = S(a^u, iv)C(a^u, ov)S(a^c, iv)C(a^c, \mathsf{nil}) \tag{21}$$

$$\mathsf{eventsof}(a^i) = S(a^i, iv)C(a^i, ov) \tag{22}$$

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 as histories, $\mathsf{FailureFree}_{(a,iv)}$ as follows:

$$\mathsf{FailureFree}_{(a,iv)} = \{h \in \mathsf{History} \mid \exists\, ov \in \mathsf{Value} : h = \mathsf{eventsof}(a, iv, ov)\}$$

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, $\mathsf{x\text{-}able}$ on histories:

$$\mathsf{x\text{-}able}_{(a,iv)}(h) = \begin{cases} \mathsf{true} & \text{if } \exists\, h' \in \mathsf{FailureFree}_{(a,iv)} : h \Rightarrow h' \\ \mathsf{false} & \text{otherwise} \end{cases} \tag{23}$$

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

3.3 History Signature

Given a request (a, iv), we use the set $\mathsf{FailureFree}_{(a,iv)}$ to constrain the server-side processing of that request to have exactly-once side-effect. We also 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:

$$\frac{h \;\Rightarrow\; S(a^u, iv)C(a^u, ov)S(a^c, iv)C(a^c, \mathsf{nil})}{(a, iv, ov) \in \mathsf{signature}(h)} \tag{24}$$

$$\frac{h \;\Rightarrow\; S(a^i, iv)C(a^i, ov)}{(a, iv, ov) \in \mathsf{signature}(h)} \tag{25}$$

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 $R_1 \ldots R_{n-1}$. 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 idependent of a particular replication protocol. We can implement the specification with various protocols, including protcols that have a primary-backup flavor and protocols that have an active-replication flavor. Moreover, the specification takes side-effect into account. We use the notion of x-able histories to capture correct execution of actions with side-effect.

We consider here a single client submitting one request to a replicated service. Formally speaking, a replicated service consists of a sequencer S and an action *submit*. The sequencer captures the functionality of the service. It is executed by a set of server proceses $s_1 \ldots s_n$ that each have 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 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 C submits a sequence of requests $R_1 \ldots R_n$, and if C only submits R_{i+1} after R_i succeeds, then the history produced by $s_1 \ldots s_n$ satisfies either $\mathsf{x\text{-}able}_{(S,R_1 \ldots R_n)}$ or $\mathsf{x\text{-}able}_{(S,R_1 \ldots R_{n-1})}$.

R4. If C executes *submit* successfully on a request R, then *submit* returns a value in $\mathsf{Possible}(S, R)$.

The first two requirements 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. There must be a time after which *submit* does not fail. 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 "effect" of executing a sequence of requests. The resulting server-side history must be x-able, that is, it must be equivalent (under history reduction) to a failure-free history obtained from S. This forces the replication algorithm to correctly retry failed actions, and execute each action returned by S so that it appears to have happened exactly-once.

The fourth requirement forces *submit* to communicate with $s_1 \ldots s_n$, the *submit* action cannot locally, at C, ensure that the server-side history is x-able. Moreover, once a server process starts to execute the actions in response to a request, the service must execute the action sequence to completion (even if C fails).

Since C submits one request after another, only R_n can fail in the sequence $R_1 \ldots R_n$. If C itself fails before retrying R_n, the server-side history may not contain the events related to the processing of R_n: perhaps R_n was initially sent to a failed replica, and the processing never begun. Thus, because C can fail, we cannot guarantee that all submitted requests are indeed processed. What we can guarantee is that all successfully submitted requests are processed. Furthermore, if the service starts to process a request, then the processing will complete event if C fails. This gives an exactly-once guarantee for all successfully submitted requests and an at-most-once guarantee for all submitted requests.

The third requirement also 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. rep

Requirement R4 guarantees that the algorithm does not execute a correct sequence of actions and then return a meaningless result. The algorithm must return a result value that is in the set of possible values for each request.

5 A General Replication Algorithm

This section presents 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, or actions that 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.

5.1 Overview

Our replication algorithm is mainly composed of two parts. A client part, described in Figure 5[3], and the replica part, described in Figure 6. 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 6).

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).

5.2 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. We assume that communication channels are reliable: if a correct process sends a message to another correct process, then the message is eventually received, and it is only received once. We also assume that

[3]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.

every action is eventually successful. If we keep invoking them, they will eventually execute to successful completion. Furthermore, we assume that a successfully executed undoable action can be committed.

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

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 synchronizations: (1) to ensure agreement about which replica is leader for a given round, (2) to ensure agreement about the outcome of undoable actions (commit or abort), and (3) to ensure agreement on the results 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.

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

5.3 The pseudo-code

We discuss below the semantics of our C++-like pseudo-code we use to describe our algoritms in Figure 5, Figure 6, and Figure 7.

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,

```
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 5: Client-side algorithm

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.

5.4 Algorithm Description

The client part of the algorithm consist of the *submit* primitive described in Figure 5. 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 cancelletion 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 6, 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 four arrays. The `owner-agreement` array contains consensus objects that control onwership 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 because the result can be fixed the first time an idempotent action is successfully executed.

The method called `result-coordination` in Figure 6 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`.

6 Concluding Remarks

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 main differences between the various approaches have to do with how the notion of *equivalence* is defined. In [MP88], an algebra of action sequences is used to define a correctness criteria 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 neither how to express it for servers that invoke

```
Server {
  Consensus(Process,Request,Process) owner-agreement[max-round];
  Consensus(Result) result-agreement[Request];
  Consensus(Outcome,Result) outcome-agreement[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 req,Process client) {
  Process id,tmp-client; Request tmp-val;
  (id,tmp-req,tmp-client) := owner-agreement[req.round].propose(my-id,req,client);
  if id == my-id then
    Result res-val := execute-until-success(req);
    res-val := result-coordination(req,res-val);
    if res-val != empty-result then
      send [Result,res-val] to client;
}

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

Figure 6: Main algorithm on server side

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

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

Figure 7: Algorithms to execute and clean sequences of actions

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 here 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 equivalent to a non-replicated one if they have the same *reads-from* relationships and final *writes*. The main differences between x-*ability* and 1-*copy serializability* are twofold. 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.

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.

[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.

[DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, October 1998.

[FG99] S. Frolund and R. Guerraoui. Implementing e-transactions with asynchronous replication. Technical Report to appear, Hewlett-Packard Laboratories, December 1999.

[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.

[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.

[Lam89] L. Lamport. The part-time parliament. Technical Report 49, DEC Systems Research Center, 1989.

[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.

[PSB+88] D. Powell, D. Seaton, G. Bonn, P. Verissimo, and F. Waeselynk. The delta-4 approach to dependability in open distributed computing systems. In *International Symposium on Fault-Tolerant Computing Systems*. IEEE, June 1988.

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