Eventually-Serializable Data Services*

Alan Fekete† David Gupta‡ Victor Luchangco‡ Nancy Lynch‡ Alex Shvartsman§

December 1, 1998

Abstract

Data replication is used in distributed systems to improve availability, increase throughput and eliminate single points of failures. The cost of replication is that significant care and communication is required to maintain consistency among replicas. In some settings, such as distributed directory services, it is acceptable to have transient inconsistencies, in exchange for better performance, as long as a consistent view of the data is eventually established. For such services to be usable, it is important that the consistency guarantees are specified clearly.

We present a new specification for distributed data services that trades off immediate consistency guarantees for improved system availability and efficiency, while ensuring the long-term consistency of the data. An *eventually-serializable data service* maintains the requested operations in a partial order that gravitates over time towards a total order. It provides clear and unambiguous guarantees about the immediate and long-term behavior of the system.

We also present an algorithm, based on the *lazy replication* strategy of Ladin, Liskov, Shrira and Ghemawat [15], that implements this specification. Our algorithm provides the external interface of the eventually-serializable data service specification, and generalizes their algorithm by allowing arbitrary operations and greater flexibility in specifying consistency requirements. In addition to correctness, we prove performance and fault-tolerance properties of this algorithm.

Keywords: replication, weak coherence, consistency, distributed storage

*This work was supported by ARPA contract F19628-95-C-0118, AFOSR-ONR contract F49620-94-1-0199, AFOSR contract F49620-97-1-0337, and NSF contract 9225124-CCR. A preliminary version appeared in *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 300–309, May 1996.

†Basser Department of Computer Science, Madsen Building F09, University of Sydney, NSW 2006, Australia.

‡Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

§Computer Science and Engineering Dept., 191 Auditorium Rd. U-155, University of Connecticut, Storrs, CT 06269.

1 Introduction

Providing distributed and concurrent access to data objects is a fundamental concern of distributed systems. In this paper, we present a formal specification for a data service that permits transient inconsistencies while providing unambiguous guarantees about system responses to clients' requests, and ensuring the *eventual serialization* of all operations requested. We also present a distributed algorithm that implements the abstract specification. We prove the correctness of the implementation using invariants and simulations. By making simple assumptions about the timing of message-based communication, we also provide time bounds for the data service.

1.1 Replication: Trade-offs of Performance and Consistency

The simplest implementations of distributed data services maintain a single centralized object that is accessed remotely by multiple clients. While conceptually simple, this approach does not scale well as the number of clients increases. Systems address this problem by replicating the data object, and allowing each replica to be accessed independently. This enables improved performance and reliability through increased locality, load balancing, and the elimination of single points of failure.

Replication of the data object raises the issue of consistency among the replicas, especially in determining the order in which the operations are applied at each replica. The strongest and simplest notion of consistency is *atomicity*, which requires the replicas to collectively emulate a single centralized object. Methods to achieve atomicity include write-all/read-one [4], primary copy [1, 26, 23], majority consensus [27], and quorum consensus [11, 12]. Because achieving atomicity often has a high performance cost, some applications, such as directory services, are willing to tolerate some transient inconsistencies. This gives rise to weaker notions of consistency. *Sequential consistency* [16], guaranteed by systems such as Orca [3], allows operations to be reordered as long as they remain consistent with the view of isolated clients. An inherent disparity in the performance of atomic and sequentially consistent objects has been established [2]. Other systems provide even weaker guarantees to the clients [9, 5, 10] in order to get better performance.

Providing weaker consistency guarantees results in more complicated semantics. Even when the behavior of the replicated objects is specified unambiguously, it is more difficult to understand and to reason about the correctness of implementations. In practice, replicated systems are often incompletely or ambiguously specified.

1.2 Background for our Work: Lazy Replication

As it is important that our specification be applicable for real systems, we build heavily on the work of Ladin, Liskov, Shrira, and Ghemawat [15] on highly available replicated data services. They specify general conditions for such a service, and present an algorithm based on *lazy replication*, in which operations received by each replica are *gossiped* in the background. Responses to operations may be out-of-date, not reflecting the effects of operations that have not yet been received by a given replica. However, the user can indicate, for a newly requested operation, a set of previously completed operations on which the new one depends; the new operation may be applied at a replica only after the operations it depends on have been applied. If an operation is submitted without such dependencies, the system may respond with any value that is consistent with an arbitrary subset

of previously requested operations. This allows any causality constraints to be expressed. Two additional types of operations are defined to provide stronger ordering constraints, when causality constraints are insufficient to implement a data object: *forced* operations must be totally ordered with respect to all other forced operations, and *immediate* operations must be totally ordered with respect to all operations. Operations that are neither forced nor immediate are called *causal*. As long as most of the operations are causal, the algorithm of [15] is efficient.

The specification in [15] is tuned for their algorithm, and exposes some of the implementation details to the clients. This makes it difficult to ascertain which details are essential to the correctness of their algorithm, and which be may changed without significant effect. It is also difficult to compare their algorithm with similar algorithms that have slightly different interfaces. For example, their specification exposes the client to multipart timestamps, which are used internally to order operations. However, it is not clear which properties of their algorithm depend on their use of multipart timestamps, and which depend only on the lazy replication strategy. Also, their algorithm requires all operations to be either read-only *queries* or write-only *updates*. Whether an update is causal, forced or immediate is determined by the effect of that update, and so must be specified by the application programmer when the system is implemented, rather than by the user when the system is executing. Their algorithm requires that for any pair of non-commutative operations with effects on the state of the data, one must be specified as depending on the other. Without this, the algorithm can leave replicas inconsistent forever. That is, the apparent order on operations may not converge to a limiting total order.

1.3 Overview of this Paper

The *eventually-serializable data service* specification uses a partial order on operations that gravitates to a total order over time. We provide two types of operations at the client interface: (a) *strict* operations, which are required to be *stable* at the time of the response, i.e., all operations that precede it must be totally ordered, and (b) operations that may be reordered after the response is issued. As in [15], clients may also specify constraints on the order in which operations are applied to the data object. Our specification omits implementation details, allowing users to ignore the issues of replication and distribution, while giving implementors the freedom to design the system to best satisfy the performance requirements. We make no assumptions about the semantics of the data object, and thus, our specification can be used as the basis for a wide variety of applications. Of course, particular system implementations may exploit the semantics of the specific data objects to improve performance.

Our algorithm is based on the lazy replication strategy of [15]. We present a high-level formal description of the algorithm, which takes into account the replication of the data, and maintains consistency by propagating operations and bookkeeping information among replicas via *gossip* messages. It provides a smooth combination of fast service with weak causality requirements and slower service with stronger requirements. It does not use the multipart timestamps of [15], which we view as an optimization of the basic algorithm. By viewing the abstract algorithm as a specification for more detailed implementations, we indicate how this, and other optimizations, may be incorporated into the framework of this paper. We also establish performance and fault-tolerance guarantees of the algorithm.

The eventually-serializable data service exemplifies the synergy of applied systems work and distributed computing theory, defining a clear and unambiguous specification for a useful module

for building distributed applications. By making all the assumptions and guarantees explicit, the formal framework allows us to reason carefully about the system. Together with the abstract algorithm, the specification can guide the design and implementation of distributed system building blocks layered on general-purpose distributed platforms (middleware) such as DCE [24]. Cheiner implemented one such building block [6, 7], and used it to develop prototypes for diverse clients including a Web client, a text-oriented Unix client, and a Microsoft Excel client for Windows95.

The rest of the paper is organized as follows: Section 2 gives formal definitions and conventions used throughout the paper, including the definition of the data type. Section 3 defines the I/O automaton model used to formally specify the data service and algorithm. Section 4 characterizes the clients of the data service, and Section 5 gives the formal specification of the eventually-serializable data service, including some guarantees about its behavior. The algorithm is presented in Section 6, and Section 7 demonstrates several properties that are used in the simulation proof of Section 8, which shows that the algorithm implements the specification. The last three sections give initial steps to extend this work. Performance guarantees, under certain timing assumptions, are given in Section 9, together with some fault-tolerance considerations. Section 10 suggests several ways in which the algorithm can be modified to give better performance, or take into account some pragmatic implementation issues. Finally, Section 11 presents an overview of Cheiner's work, and discusses some applications which may use eventually-serializable data services.

2 Preliminary Definitions and Conventions

In this section, we introduce mathematical notation and conventions used in this paper. These are merely formal definitions; the motivation and intuition behind these definitions appear in the appropriate section later in the paper. We also state without proof several lemmas that follow easily from these definitions. Throughout the paper, whenever variables appear unquantified, there is an implicit universal quantification.

2.1 Functions, Relations and Orders

A ***binary relation*** R on a set S is any subset of $S \times S$; we sometimes write xRy for $(x, y) \in R$. The ***span*** of a binary relation R is $span(R) = \{x : xRy \vee yRx \text{ for some } y\}$. A relation R is ***transitive*** if $xRy \wedge yRz \implies xRz$. It is ***antisymmetric*** if $xRy \wedge yRx \implies x = y$. It is ***reflexive*** if xRx for all $x \in S$, and it is ***irreflexive*** if $(x, x) \notin R$ for all $x \in S$. The ***transitive closure*** of a relation R, denoted $\mathrm{TC}(R)$, is the smallest transitive relation containing R, and the ***reflexive closure*** is the smallest reflexive relation containing R. The relation ***induced by*** R on a set S' is $R \cap (S' \times S')$.

A binary relation is a ***partial order*** if it is transitive and antisymmetric. It is ***strict*** if it is also irreflexive. We say that x ***precedes*** y in a partial order R if xRy. For a set S, we denote the subset of elements that precede $x \in S$ in R by $S|_{Rx} = \{y \in S : yRx\}$. Two relations R and R' are ***consistent*** if $\mathrm{TC}(R \cup R')$ is a partial order. A relation R is a ***total order*** on S if it is a partial order on S with $xRy \vee yRx \vee x = y$ for all $x, y \in S$. A partial order R ***totally orders*** S if it induces a total order on S. If $\prec$ is a total order on S and X is a finite nonempty subset of S then we define $\min_{\prec} X$ to be the element $x \in X$ such that $x \preceq y$ for all $y \in X$ and $\max_{\prec} X$ to be the element $x \in X$ such that $y \preceq x$ for all $y \in X$, where $\preceq$ is the reflexive closure of $\prec$. We may omit the subscript when there is a single total order defined on S.

Lemma 2.1 Any irreflexive and transitive relation is a strict partial order.

Lemma 2.2 The relation induced by a partial order on any set is also a partial order.

Lemma 2.3 If R is a total order on S and R' is a partial order, then R and R' are consistent if and only if $xR'y \wedge yRx \implies x = y$.

A function $f : A \to B$ has ***domain*** A and ***range*** B. A function is ***null*** if its domain is the empty set. For a set B, we extend functions and relations on $B \times B$ to functions whose range is B. That is, if $f_1, f_2 : A \to B$, $g : B \times B \to C$, and R is a binary relation on B then we let $g(f_1, f_2)$ be the function $h : A \to C$ with $h(a) = g(f_1(a), f_2(a))$, and $(f_1, f_2) \in R$ if $(f_1(a), f_2(a)) \in R$ for all $a \in A$.

2.2 Data Types

The data service manages objects whose serial behavior is specified by some data type. This data type defines possible states of instantiated objects and operators on the objects. We use a definition similar to the variable types of [18]. Formally, a ***serial data type*** consists of:

- a set Σ of ***object states***
- a distinguished initial state $\sigma_0 \in \Sigma$
- a set V of ***reportable values***
- a set O of ***operators***
- a transition function $\tau : \Sigma \times O \to \Sigma \times V$

We use $.s$ and $.v$ selectors to extract the state and value components respectively, i.e., $\tau(\sigma, op) = (\tau(\sigma, op).s, \tau(\sigma, op).v)$. For the set O^+ of nonempty finite sequences of operators, we also define $\tau^+ : \Sigma \times O^+ \to \Sigma \times V$ by repeated application of τ, i.e., $\tau^+(\sigma, \langle op \rangle) = \tau(\sigma, op)$ and $\tau^+(\sigma, \langle op_1, op_2, \ldots \rangle) = \tau^+(\tau(\sigma, op_1).s, \langle op_2, \ldots \rangle)$, where $\langle \ldots \rangle$ denotes a sequence. In this paper, we assume that the serial data type is fixed, and often leave it implicit.

2.3 Operations

To access the data, a client of the data service issues a request, which includes the operator to be applied, a unique ***operation identifier***, and additional information that constrains the valid responses to the request. Formally, a client issues an ***operation descriptor*** consisting of:

- a data type operator op
- an operation identifier id
- a set $prev$ of operation identifiers
- a boolean flag $strict$

We often refer to an operation descriptor x simply as ***operation*** x, and denote its various components by $x.op$, $x.id$, $x.prev$ and $x.strict$. We denote by $\mathcal{O}$ the set of all operations, and by $\mathcal{I}$ the set of all operation identifiers. For a set $X \subseteq \mathcal{O}$, we denote by $X.id = \{x.id : x \in X\}$ the set of identifiers of operations in X. Thus $\mathcal{I} = \mathcal{O}.id$. If R is a partial order on $\mathcal{I}$, then $\prec_R = \{(x, y) \in \mathcal{O} \times \mathcal{O} : (x.id, y.id) \in R\}$ is a partial order on $\mathcal{O}$ such that $x \prec_R y$ if $(x.id, y.id) \in R$. We denote the reflexive closure of $\prec_R$ by $\preceq_R$.

An operation is ***strict*** if its *strict* flag has value true. For a set $X \subseteq \mathcal{O}$, we denote by $CSC(X) = \{(y.id, x.id) : x \in X \wedge y.id \in x.prev\}$ a relation on $\mathcal{I}$ expressing the ***client-specified constraints*** described by the *prev* sets of the operations. The interpretation of these is given in Section 4.

Lemma 2.4 If $X \subseteq Y \subseteq \mathcal{O}$ then $CSC(X) \subseteq CSC(Y)$.

Given a finite set $X = \{x_1, \ldots, x_n\}$ of operations and the strict total order $\prec = \{(x_i, x_j) : i < j\}$, we define the ***outcome*** of X from state $\sigma \in \Sigma$ with respect to $\prec$ to be $outcome_\sigma(X, \prec) = \tau^+(\sigma, \langle x_1.op, \ldots, x_n.op\rangle).s$, and the ***value*** of an operation $x \in X$ from σ with respect to $\prec$ to be $val_\sigma(x, X, \prec) = \tau(outcome_\sigma(X|_{\prec x}, \prec), x.op).v$. If $\prec$ is a partial order on X, we define $valset_\sigma(x, X, \prec) = \{val_\sigma(x, X, \prec') : \prec'$ is a strict total order on X consistent with $\prec\}$. When $\prec$ relates elements not in X, and $\prec'$ is the partial order induced by $\prec$ on X, we sometimes abuse notation by writing $valset_\sigma(x, X, \prec)$ for $valset_\sigma(x, X, \prec')$, and, if $\prec'$ is a total order on X, $val_\sigma(x, X, \prec)$ for the only element in $valset_\sigma(x, X, \prec)$, and $outcome_\sigma(X, \prec)$ for $outcome_\sigma(X, \prec')$. If σ is not explicitly specified, it is assumed to be the initial state σ_0.

Lemma 2.5 If $\prec$ is a partial order on X then $valset_\sigma(x, X, \prec) \neq \emptyset$ for all $x \in X$.

Lemma 2.6 If $\prec$ and $\prec'$ are partial orders on X such that $\prec \subseteq \prec'$ then $valset_\sigma(x, X, \prec') \subseteq valset_\sigma(x, X, \prec)$ for all $x \in X$.

Lemma 2.7 Suppose $X \subseteq Y \subseteq \mathcal{O}$, $\prec$ is a partial order on Y that induces a total order on X, and $x \prec y$ for all $x \in X$ and $y \in Y - X$. Then $valset_\sigma(x, Y, \prec) = \{val_\sigma(x, X, \prec)\}$ for all $x \in X$, and $valset_\sigma(y, Y, \prec) = valset_{\sigma'}(y, Y - X, \prec)$ for all $y \in Y - X$, where $\sigma' = outcome_\sigma(X, \prec)$.

3 Formal model

The specifications in this paper are done using a slight simplification of I/O automata [19], ignoring aspects related to liveness. We do not deal with liveness directly in this paper. Instead, we assume bounds on the time to perform actions, and prove performance guarantees that imply liveness under those timing assumptions.

A ***non-live I/O automaton*** A consists of:

- three disjoint sets of ***actions***: in(A), out(A), and int(A);
- a set $states(A)$ of ***states***;
- a nonempty subset $start(A)$ of ***start states***;
- a set $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ of ***steps*** such that there exists $(s, \pi, s') \in steps(A)$ for all $s \in states(A)$ and $\pi \in in(A)$.

We call the actions in $in(A)$, $out(A)$, and $int(A)$ the ***input***, ***output***, and ***internal*** actions respectively. The input and output actions are also called ***external actions***, and the set of external actions is denoted by $ext(A)$. We denote the set of all actions of A by $acts(A) = in(A) \cup out(A) \cup int(A)$. We write $s \xrightarrow{\pi}_A s'$ or just $s \xrightarrow{\pi} s'$ as shorthand for $(s, \pi, s') \in steps(A)$. We say an action π is ***enabled*** in s if there exists s' such that $s \xrightarrow{\pi} s'$. Notice that every input action is enabled in every state.

An ***execution fragment*** $s_0 \pi_1 s_1 \pi_2 s_2 \cdots$ is a finite or infinite sequence of alternating states and actions such that $s_{i-1} \xrightarrow{\pi_i} s_i$ for all i. The ***external image*** of an execution fragment α is the subsequence $\alpha|_{ext(A)}$ of its external actions. An ***execution*** is an execution fragment with $s_0 \in start(A)$. We denote the set of executions of A by $execs(A)$. A ***trace*** of A is the external image of an execution, and the set of traces is denoted by $traces(A)$. An ***event*** is an occurence of an action in a sequence. If an event π (strictly) precedes π' in α, then we write $\pi \prec_\alpha \pi'$. A state is ***reachable*** in A if it appears in any execution of A. An ***invariant*** of A is a predicate that is true of every reachable state of A.

We often want to specify a distributed system by specifying the components that constitute the system. The entire system is then described by an automaton which is the ***composition*** of the automata describing the components. Informally, composition identifies actions with the same name at different component automata. Thus, when an action is executed, it is executed by all components with that action. The new automaton has the actions of all its components. Some restrictions on the automata to be composed are necessary so that the composition makes sense. In particular, internal actions cannot be shared, an action can be the output action of at most one component, and actions cannot be shared by infinitely many components.

Formally, for any index set I, a set $\{A_i\}_{i \in I}$ of automata is ***compatible*** if $int(A_i) \cap acts(A_j) = \emptyset$ and $out(A_i) \cap out(A_j) = \emptyset$ for all $i, j \in I$ such that $i \neq j$, and no action is in $acts(A_i)$ for infinitely many $i \in I$. The ***composition*** $A = \Pi_{i \in I} A_i$ of a compatible set $\{A_i\}_{i \in I}$ of automata has the following components:

- $in(A) = \bigcup_{i \in I} in(A_i) - \bigcup_{i \in I} out(A_i)$
  $out(A) = \bigcup_{i \in I} out(A_i)$
  $int(A) = \bigcup_{i \in I} int(A_i)$
- $states(A) = \Pi_{i \in I} states(A_i)$
- $start(A) = \Pi_{i \in I} start(A_i)$
- $steps(A) = \{(s, \pi, s') : s_i \xrightarrow{\pi}_{A_i} s'_i \text{ or } \pi \notin acts(A_i) \wedge s_i = s'_i \text{ for all } i \in I\}$

We denote the composition of two compatible automata A and B by $A \times B$.

Communication between automata is done through shared external actions, which remain external actions of the composition. Sometimes it is useful to ***hide*** these actions, reclassifying them as internal, so they cannot be used for further communication and no longer appear in traces. Formally, if A is an I/O automaton and $\Phi \subseteq out(A)$, then the ***hiding operation*** on A and Φ produces an automaton A' identical to A except that $out(A') = out(A) - \Phi$ and $int(A') = int(A) \cup \Phi$.

I/O automata can be used as specifications as well as implementations. We say that an automaton A ***implements*** another automaton B, and write $A \subseteq B$, if $in(A) = in(B)$, $out(A) = out(B)$,

and $traces(A) \subseteq traces(B)$. We say that A and B are ***equivalent***, and write $A \equiv B$, if they implement each other.

Theorem 3.1 If $A_i \subseteq B_i$ for all $i \in I$ then $\Pi_{i \in I} A_i \subseteq \Pi_{i \in I} B_i$.

A standard way to show that one automaton implements another is to use *simulations*, which establish a correspondence between the states of the two automata. Formally, if A and B are automata with $in(A) = in(B)$ and $out(A) = out(B)$ then a ***forward simulation*** from A to B is a relation f between $states(A)$ and $states(B)$ such that:

- If $s \in start(A)$ then there exists some $u \in start(B)$ such that $f(s, u)$.
- For reachable states s and u of A and B, if $f(s, u)$ and $s \xrightarrow{\pi}_A s'$, then there exists some u' such that $f(s', u')$ and there is some execution fragment of B from u to u' with the same external image as π.

We denote $\{u : f(s, u)\}$ by $f[s]$, and typically write $u \in f[s]$ instead of $f(s, u)$.

Theorem 3.2 If there is a forward simulation from A to B then $A \subseteq B$.

4 Client Specification

We model a system as a service accessed by clients expected to obey certain conventions, called the *well-formedness assumptions*. In this section, we formally define these assumptions on the clients of the data service. The automaton *Users* in Figure 1 represents all clients, and uses shared state to encode the restrictions on the clients in a general and abstract way; in a real implementation, there need not be any shared state.

Signature

Input:
  $\mathsf{response}(x, v)$, where $x \in \mathcal{O}$ and $v \in V$
Output:
  $\mathsf{request}(x)$, where $x \in \mathcal{O}$

State

requested, a subset of $\mathcal{O}$, initially empty

Actions

Output $\mathsf{request}(x)$
  Pre: $x.id \notin requested.id$
    $x.prev \subseteq requested.id$
  Eff: $requested \leftarrow requested \cup \{x\}$

Input $\mathsf{response}(x, v)$
  Eff: None

Figure 1: *Users*: The well-formed clients

Clients access the data by issuing *requests* and receiving *responses* from the data service. The data type only specifies serial behavior, that is, the behavior when the operations are requested in

sequence. However, we allow clients to issue requests concurrently. To request an operation, a client specifies an operation descriptor x, which includes a unique identifier, and a *prev* set and *strict* flag which are intended to constrain the responses the client may receive from the data service for the requested operation. Informally, the *prev* set represents operations that must be done before the requested operation, and can only include operations requested earlier. The relation $CSC(requested)$ defines the ***client-specified constraints***.

The condition $x.id \notin requested.id$ ensures that the operation identifiers are unique, and the condition $x.prev \subseteq requested.id$ ensures that $\mathrm{TC}(CSC(requested))$ is a strict partial order.

Invariant 4.1 For $x, y \in requested$, $x = y \iff x.id = y.id$.

Invariant 4.2 $\mathrm{TC}(CSC(requested))$ is a strict partial order.

In any reachable state of *Users*, we define the partial order $\prec_c$ on *requested* so that $x \prec_c y$ if and only if $(x.id, y.id) \in \mathrm{TC}(CSC(requested))$.

This automaton only specifies the well-formedness assumptions on the clients; it does not place any restrictions on the responses it may receive. Given a set X of operations, we say that a $\mathsf{response}(x, v)$ event is ***consistent with*** a partial order $\prec$ on X if $v \in valset(x, X, \prec)$, and that a total order $\prec'$ ***explains*** the event if $v = val(x, X, \prec')$. We expect that every response corresponds to some request, and is consistent with the client-specified constraints. This is guaranteed by the data service specification in the next section.

5 ESDS Specification

In this section, we give the formal specification of an *eventually-serializable data service*. We first specify this as the automaton *ESDS-I*, and we then prove several properties of this automaton. We then give an alternative specification *ESDS-II*, which is equivalent to *ESDS-I*. We give two specifications because *ESDS-I* is simpler to understand, while *ESDS-II* is more convenient for showing that the specification is implemented by the abstract algorithm we define in Section 6.

5.1 Specification ESDS-I

We now define an *eventually-serializable data service*. The clients of the service may issue requests concurrently, and thus the responses are not uniquely defined by the data type specification. A sequentially consistent data service would require that there exist a total order on the operations consistent with all the responses of the service. This total order is called a *serialization*. However, for some systems, sequential consistency is too expensive to guarantee. The eventually-serializable data service specification permits more efficient and resilient distributed implementations by allowing some operations to be reordered even after a response has already been returned. However, it must always respect the client-specified constraints. In addition, an operation may *stabilize*, after which it may no longer be reordered.

Formally, an ***eventually-serializable data service*** is any automaton that implements *ESDS-I* in Figure 2. The input actions are the requests from the clients, and the output actions are the

Signature

Input:
  request(x), where $x \in \mathcal{O}$
Output:
  response(x, v), where $x \in \mathcal{O}$ and $v \in V$
Internal:
  enter(x, *new-po*), where $x \in \mathcal{O}$ and *new-po* is a strict partial order on $\mathcal{I}$
  stabilize(x), where $x \in \mathcal{O}$
  calculate(x, v), where $x \in \mathcal{O}$ and $v \in V$
  add_constraints(*new-po*), where *new-po* is a partial order on $\mathcal{I}$

State

wait, a subset of $\mathcal{O}$, initially empty; the operations requested but not yet responded to

rept, a subset of $\mathcal{O} \times V$, initially empty; operations and responses that may be returned to clients

ops, a subset of $\mathcal{O}$, initially empty; the set of all operations that have ever been entered

po, a partial order on $\mathcal{I}$, initially empty; constraints on the order operations in *ops* are applied

stabilized, a subset of $\mathcal{O}$, initially empty; the set of stable operations

Actions

Input request(x)
  Eff: $wait \leftarrow wait \cup \{x\}$

Internal enter(x, *new-po*)
  Pre: $x \in wait$
    $x \notin ops$
    $x.prev \subseteq ops.id$
    $span(\text{new-}po) \subseteq ops.id \cup \{x.id\}$
    $po \subseteq \text{new-}po$
    $CSC(\{x\}) \subseteq \text{new-}po$
    $\{(y.id, x.id) : y \in stabilized\} \subseteq \text{new-}po$
  Eff: $ops \leftarrow ops \cup \{x\}$
    $po \leftarrow \text{new-}po$

Internal add_constraints(*new-po*)
  Pre: $span(\text{new-}po) \subseteq ops.id$
    $po \subseteq \text{new-}po$
  Eff: $po \leftarrow \text{new-}po$

Internal stabilize(x)
  Pre: $x \in ops$
    $x \notin stabilized$
    $\forall y \in ops,\ y \preceq_{po} x \vee x \preceq_{po} y$
    $ops|_{\prec_{po} x} \subseteq stabilized$
  Eff: $stabilized \leftarrow stabilized \cup \{x\}$

Internal calculate(x, v)
  Pre: $x \in ops$
    $x.strict \implies x \in stabilized$
    $v \in valset(x, ops, \prec_{po})$
  Eff: if $x \in wait$ then $rept \leftarrow rept \cup \{(x, v)\}$

Output response(x, v)
  Pre: $(x, v) \in rept$
    $x \in wait$
  Eff: $wait \leftarrow wait - \{x\}$
    $rept \leftarrow rept - \{(x, v') : (x, v') \in rept\}$

Figure 2: Specification *ESDS-I*

responses to these requests. Because of the well-formedness assumptions, we expect that the client-specified constraints define a strict partial order on the requested operations. Although the automata is formally defined for any input, the following discussion assumes well-formed clients. The informal claims in this subsection are stated and proved formally in the next subsection.

The main idea is to maintain a strict partial order of the operations consistent with the client-specified constraints. In addition, the automaton maintains a set of ***stable*** operations, whose prefix in the partial order is total and fixed. If every operation is stable, the partial order is total, and we call this the *eventual total order*. Responses to strict operations must be consistent with the eventual total order.

The *wait* and *rept* variables are used to keep track of pending requests. The set *ops* contains the operations that have been *entered* (by the enter action); only these operations are used (by the calculate action) to compute the return values of operations. The variable *po* defines a strict partial order $\prec_{po}$ on the operations in *ops*, which restricts the order in which these operations may applied. This order must be consistent with the client-specified constraints given by the *prev* sets. The set *stabilized* contains the stable operations.

The request and response actions are the interface actions with the clients. They update *wait* and *rept* appropriately. For each operation x, the specification defines internal actions of the form enter(x, *new-po*), stabilize(x) and calculate(x, v). The enter(x, *new-po*) action adds sufficient constraints to *po* to ensure that the new operation follows every operation specified by the client in the *prev* set, and preserves the prefix of stable operations. That is, a new operation must be preceded in *new-po* by every operation specified by the client and by every stable operation. The stabilize(x) action can occur only if x is totally ordered with respect to other operations in *ops*, and all preceding operations are already stable. The calculate(x, v) action chooses some return value for x consistent with the constraints specified by *po*. Strict operations must be stable when a value is calculated for them, but nonstrict operations need not be. Thus, the responses to the clients for nonstrict operations need not be consistent with the eventual total order. Repeated calculate actions for a specific operation may produce different return values and the response action selects one of the values for the operation nondeterministically.[1]

In addition, there is an internal action add_constraints(*new-po*) which extends the partial order of constraints. Notice that the partial order can only be constrained further; once a constraint is imposed, it is never revoked.

Although informally we expect every request to get a response and every operation to stabilize, there are no formal liveness guarantees in this specification. Instead, in Section 9, we assume time bounds on the actions, and prove performance guarantees that imply liveness under these timing assumptions.

5.2 Properties of Eventually-Serializable Data Services

We now prove several properties of the composition *ESDS-I* $\times$ *Users* that are useful for writing applications that use the eventually-serializable data service.

The first lemma says that *stabilized*, *ops* and *po* only increase, and that only entered operations

[1] This is equivalent to an automaton that only allows a single calculate action for each operation, but this requires additional formal machinery (e.g., backward simulations [20]) to prove.

are stabilized.

Lemma 5.1 If $s \xrightarrow{\pi} s'$ then $s.stabilized \subseteq s'.stabilized \subseteq s.ops \subseteq s'.ops$, and $s.po \subseteq s'.po$.

Proof: Immediate from definition of *ESDS-I*. ■

The next invariant says that *po* orders only operations in *ops* and contains the client-specified constraints.

Invariant 5.2 $span(po) \subseteq ops.id$ and $CSC(ops) \subseteq po$.

Proof: We prove this by induction on the length of an execution. This is trivial in the initial state since *po* is empty. If the invariant holds in s and $s \xrightarrow{\pi} s'$ then only enter and add_constraints actions change *po* or *ops*:

1. If $\pi = \mathsf{enter}(x, s'.po)$ then $span(s'.po) \subseteq s.ops.id \cup \{x.id\} = s'.ops.id$, and $CSC(s.ops) \subseteq s.po \subseteq s'.po$ and $CSC(\{x\}) \subseteq s'.po$, so $CSC(s'.ops) = CSC(s.ops) \cup CSC(\{x\}) \subseteq s'.po$.
2. If $\pi = \mathsf{add\_constraints}(s'.po)$ then $span(s'.po) \subseteq s.ops.id = s'.ops.id$ and $CSC(s'.ops) = CSC(s.ops) \subseteq s.po \subseteq s'.po$.

■

The following two invariants say that stable operations can be compared with any entered operation, and thus, that *stabilized* is totally ordered by $\prec_{po}$.

Invariant 5.3 For all $x \in stabilized$ and $y \in ops$, we have $y \preceq_{po} x \vee x \preceq_{po} y$.

Proof: We prove this by induction on the length of an execution. This is trivial in the initial state since $stabilized = \emptyset$. If the invariant holds in s and $s \xrightarrow{\pi} s'$ then $y \preceq_{s'.po} x \vee x \preceq_{s'.po} y$ for all $x \in s.stabilized$ and $y \in s.ops$, since $s.po \subseteq s'.po$ by Lemma 5.1. If $x \in s'.stabilized - s.stabilized$ then $\pi = \mathsf{stabilize}(x)$ so $y \preceq_{s'.po} x \vee x \preceq_{s'.po} y$ for all $y \in s.ops = s'.ops$ by the precondition for $\mathsf{stabilize}(x)$. If $y \in s'.ops - s.ops$ then $\pi = \mathsf{enter}(y, s'.po)$ and $x \prec_{s'.po} y$ for all $x \in s.stabilized = s'.stabilized$. ■

Invariant 5.4 *stabilized* is totally ordered by $\prec_{po}$.

Proof: Immediate from Invariant 5.3 since $stabilized \subseteq ops$ (by Lemma 5.1). ■

The next invariant says that operations preceding stable operations are also stable.

Invariant 5.5 If $x \in stabilized$ then $ops|_{\preceq_{po} x} \subseteq stabilized$.

Proof: We prove this by induction on the length of an execution. This is trivial in the initial state since *stabilized* is empty. Suppose the invariant holds in s and $s \xrightarrow{\pi} s'$. Then for $x \in s.stabilized$, if $y \in ops|_{\preceq_{s'.po} x}$ then $x \not\preceq_{s.po} y$, since $s.po$ is a strict partial order and $s.po \subseteq s'.po$ by Lemma 5.1. By Invariant 5.3, $y \prec_{s.po} x$. Thus, by the inductive assumption and Lemma 5.1, $y \in s.stabilized \subseteq s'.stabilized$. If $x \in s'.stabilized - s.stabilized$ then $\pi = \mathsf{stabilize}(x)$ and thus, $ops|_{\preceq_{s'.po} x} = ops|_{\preceq_{s.po} x} \subseteq s'.stabilized$. ■

The next invariant says that there is a unique value for stable operations.

Invariant 5.6 If $x \in stabilized$ then $valset(x, ops, \prec_{po}) = \{val(x, ops|_{\preceq_{po} x}, \prec_{po})\}$

Proof: By Invariant 5.5, $ops|_{\preceq_{po} x} \subseteq stabilized$, so by Invariant 5.4, $\prec_{po}$ totally orders $ops|_{\preceq_{po} x}$, and by Invariant 5.3, $y \in ops - ops|_{\preceq_{po} x} \implies x \prec_{po} y$. Thus, by Lemma 2.7, $valset(x, ops, \prec_{po}) = \{val(x, ops|_{\preceq_{po} x}, \prec_{po})\}$. ■

We now give several guarantees on the behavior of the system that may be useful for applications. The first theorem says that for each operation x, there is a total order of the requested operations consistent with the client-specified constraints that explains the response for x and the response of every strict operation that receives a response before x is requested.

Theorem 5.7 Suppose β is a trace of *ESDS-I* $\times$ *Users*, and *reqs* is the set of operations requested in β. For each $\mathsf{response}(x, v)$ event in β, there exists a total order $to(x)$ on $reqs.id$ consistent with $CSC(reqs)$ such that $v = val(x, reqs, \prec_{to(x)})$ and for every $\mathsf{response}(y, v') \prec_{\beta} \mathsf{request}(x)$ with $y.strict$, $v' = val(y, reqs, \prec_{to(x)})$.

Proof: (Sketch) Let α be an execution of *ESDS-I* $\times$ *Users* with external image β. There must be a $\mathsf{calculate}(x, v)$ event in α preceding the $\mathsf{response}(x, v)$ event. Let s be the state of α immediately preceding this event. By the precondition, there is a total order $\prec$ on $s.ops$ consistent with $\prec_{s.po}$ such that $v = val(x, s.ops, \prec)$. If $\mathsf{response}(y, v') \prec_{\beta} \mathsf{request}(x)$ and $y.strict$ then let s' be the state immediately preceding the $\mathsf{calculate}(y, v')$ event. By Invariant 5.6, v' is the unique value for y consistent with $\prec_{s'.po}$, and by Lemma 5.1, $s'.po \subseteq s.po$, so $v' = val(y, s'.ops, \prec) = val(y, s.ops, \prec)$. Let $to(x)$ be such that $\prec$ is a prefix of $\prec_{to(x)}$, that is, all operations in $reqs - s.ops$ are ordered after the operations in ops. Then $val(y, reqs, \prec_{to(x)}) = val(y, s.ops, \prec)$ for $y \in s.ops$, establishing the theorem. ■

The next theorem says that there is an eventual total order that explains all responses to strict operations.

Theorem 5.8 Suppose β is a finite trace of *ESDS-I* $\times$ *Users*, and *reqs* is the set of operations requested in β. There exists a total order eto on $reqs.id$ consistent with $CSC(reqs)$ such that for every $\mathsf{response}(x, v)$ event in β with $x.strict$, $v = val(x, reqs, \prec_{eto})$.

Proof: (Sketch) Let α be a finite execution of *ESDS-I* $\times$ *Users* with external image β, and s be the final state of α. Let $\prec$ be a total order on $s.ops$ consistent with $\prec_{s.po}$. If $\mathsf{response}(x, v)$ is an event of α with $x.strict$ then let s' be the state immediately preceding the $\mathsf{calculate}(x, v)$ event. By Invariant 5.6, v is the unique value for x consistent with $\prec_{s'.po}$, and by Lemma 5.1, $s'.po \subseteq s.po$, so $v = val(x, s'.ops, \prec) = val(x, s.ops, \prec)$. Let eto be such that $\prec$ is a prefix of $\prec_{eto}$. Then $val(x, reqs, \prec_{eto}) = val(x, s.ops, \prec)$ for $x \in s.ops$, establishing the theorem. ■

The following corollary says that when all requests are strict, *ESDS-I* appears similar to an atomic object. The eventual total order from the previous theorem defines the serialization.

Corollary 5.9 Suppose β is a finite trace of *ESDS-I* $\times$ *Users*, *reqs* is the set of operations requested in β, and $x.strict$ for all $x \in reqs$. Then there exists a total order eto on $reqs.id$ consistent with $CSC(reqs)$ such that for every $\mathsf{response}(x, v)$ event in β, $v = val(x, reqs, \prec_{eto})$.

5.3 Specification ESDS-II

We now give an alternative specification of eventually-serializable data services, using a more nondeterministic automaton *ESDS-II*, and we show that *ESDS-I* and *ESDS-II* are equivalent. Although this automaton is more complicated than *ESDS-I*, it is easier to use as the specification in a simulation proof because it allows more nondeterminism. We use it in the simulation proof in Section 8.

There are three differences between the two automata, all in the preconditions of two actions, enter and stabilize. The new actions appear in Figure 3.

Internal enter(x, *new-po*)
Pre: $x \in wait$
$x.prev \subseteq ops.id$
$span(new\text{-}po) \subseteq ops.id \cup \{x.id\}$
$po \subseteq new\text{-}po$
$CSC(\{x\}) \subseteq new\text{-}po$
$\{(y.id, x.id) : y \in stabilized\} \subseteq new\text{-}po$
Eff: $ops \leftarrow ops \cup \{x\}$
$po \leftarrow new\text{-}po$

Internal stabilize(x)
Pre: $x \in ops$
$\forall y \in ops,\ y \preceq_{po} x \vee x \preceq_{po} y$
$\prec_{po}$ totally orders $ops|_{\prec_{po} x}$
Eff: $stabilized \leftarrow stabilized \cup \{x\}$

Figure 3: The enter and stabilize actions of *ESDS-II*

Two of the differences are minor: the clauses $x \notin ops$ and $x \notin stabilized$ are removed from the preconditions of the enter and stabilize actions respectively. This allows them to be done repeatedly for each operation. This is minor because a repeated enter(x, *new-po*) is equivalent to an add_constraints(*new-po*) action, and a repeated stabilize(x) does not change the state at all.

The third difference is more significant. When an operation is stabilized, instead of requiring that preceding operations already be stable, the stabilize action of *ESDS-II* only requires that they be totally ordered by $\prec_{po}$. This allows "gaps" between stable operations, which are impossible in *ESDS-I* by Invariant 5.5. All the other invariants, lemmas and theorems remain true for *ESDS-II*, with their proofs largely unchanged.

It is easy to see that *ESDS-I* implements *ESDS-II* since every execution of *ESDS-I* is an execution of *ESDS-II*. We can show that *ESDS-II* implements *ESDS-I* with a simple simulation proof. The simulation, given in Figure 4, relates states of *ESDS-II* to states of *ESDS-I* when the operations stable in the implementation are also stable in the specification, and all other state components are equal. Informally, this allows *ESDS-I* to "fill in the gaps" allowed between stable operations in *ESDS-II*.

The proof that this is a simulation is straightforward. Every action simulates itself except that the stabilize(x) action in *ESDS-II* simulates a (possibly empty) sequence of stabilize action in *ESDS-I*, one for each operation in $ops|_{\preceq_{po} x} - stabilized$. The key observation is that if an execution of *ESDS-II* stabilizes an operation that has preceding operations that have not yet stabilized, then the simulated execution of *ESDS-I* can stabilize all such operations first.

G is a relation between states in *ESDS-II* and states in *ESDS-I* such that $u \in G[s]$ if and only if:

- $u.wait = s.wait$
- $u.rept = s.rept$
- $u.ops = s.ops$
- $u.po = s.po$
- $u.stabilized \supseteq s.stabilized$

Figure 4: Forward simulation from *ESDS-II* to *ESDS-I*

6 Algorithm

We now present an algorithm that implements the eventually-serializable data service specification *ESDS-II* in the previous section. In later sections, we prove formally that the algorithm implements the data service.

The algorithm replicates the data, maintaining a complete copy at each ***replica***. We assume that there are at least two replicas. Each client uses a front end to the service that keeps track of pending requests and handles communication with the replicas. We model inter-process communication by point-to-point channels, and we assume that the processes and channels are reliable, but we make no assumptions about the order of message delivery. The algorithm can be modified to tolerate processor crashes and message losses and we discuss these considerations in a later section. We also assume that local computation time is insignificant compared with communication delays, so that a process is always able to handle any input it receives. This is reasonable if the computation required by each operation is not excessive.

This algorithm is based on the *lazy-replication* scheme of Ladin, Liskov, Shrira and Ghemawat [15], which uses *gossip* messages to maintain consistency among the replicas. Each replica maintains a *label* for each operation it knows about. These labels may be received by gossip, or generated by the replica if no label has been gossiped to it. The labels are totally ordered and are generated uniquely, and an operation's place in the eventual total order is determined by the system-wide minimum label for that operation.

6.1 The Channels

Point-to-point channels are used for request and response messages between front ends and replicas and for gossip messages between replicas. We assume that channels are reliable, but we do not assume that they are FIFO. A channel from process i to process j with message set $\mathcal{M}$ is modelled by the simple automaton in Figure 5. It has send_{ij} and $\mathsf{receive}_{ij}$ actions and a single state variable $channel_{ij}$ representing the messages in transit. For channels from a front end to a replica, the message set is $\mathcal{M}_{req} = \{\langle \text{“request”}, x\rangle : x \in \mathcal{O}\}$; from a replica to a front end, it is $\mathcal{M}_{resp} = \{\langle \text{“response”}, x, v\rangle : x \in \mathcal{O}, v \in V\}$. For channels between replicas, the message set is $\mathcal{M}_{gossip} = \{\langle \text{“gossip”}, R, D, L, S\rangle : R, D, S \subseteq \mathcal{O}, L : \mathcal{I} \to \mathcal{L} \cup \{\infty\}\}$, where $\mathcal{L}$ is a set of labels, formally defined in Section 6.3, used by the replicas to maintain consistency. For a gossip message $m = \langle \text{“gossip”}, R, D, L, S\rangle$, we use R_m, D_m, L_m and S_m to denote R, D, L and S respectively.

Signature

Input:
$\mathsf{send}_{ij}(m)$, where $m \in \mathcal{M}$
Output:
$\mathsf{receive}_{ij}(m)$, where $m \in \mathcal{M}$

State

$channel_{ij}$, a multiset of messages (from $\mathcal{M}$), initially empty

Actions

Input $\mathsf{send}_{ij}(m)$
Eff: $channel_{ij} \leftarrow channel_{ij} \cup \{m\}$

Output $\mathsf{receive}_{ij}(m)$
Pre: $m \in channel_{ij}$
Eff: $channel_{ij} \leftarrow channel_{ij} - \{m\}$

Figure 5: Automaton for channel from process i to process j with message set $\mathcal{M}$.

6.2 The Front Ends

When a client submits a request, its front end to the service relays the request to one or more replicas, which maintain a copy of the data object; when the front end receives a response, it relays that to the client. Although we have been modelling the clients as a single automaton, the replicas must distinguish between clients, so they can send responses to the appropriate front ends. For simplicity, we assume that the clients encode their identity into the operation identifier. Formally, if C is the set of clients, we assume there is a static function $client : \mathcal{I} \to C$. We use this to partition $\mathcal{O}$ into sets $\mathcal{O}_c = \{x : client(x.id) = c\}$.

The front end automaton is shown in Figure 6. The variables $wait_c$ and $rept_c$ keep track of pending requests from client c. The request and response actions are the interface actions with the client c, and they update $wait_c$ and $rept_c$ appropriately. The front end may send a message requesting a response for any pending operation, i.e., any operation in $wait_c$. Note that the $\mathsf{send}_{cr}(\langle \text{“request”}, x \rangle)$ action may be performed repeatedly, requesting a response from different replicas, or even repeatedly from the same replica.[2] When a response for a pending request is received from a replica, the front end records it in $rept_c$.

6.3 The Replicas

The replicas do not keep an explicit state of the data which is updated by each operation. Instead, they assign *labels* to the operations from a well-ordered set. These labels are used to compute the return values for the operations. To maintain consistency, the replicas “gossip” the labels, keeping the minimum label for each operaton. An operation is ***done*** at a replica if that replica has a label for that operation. An operation is ***stable*** at a replica r if r knows that it is done at every replica. The informal claims made in this section are stated and proved formally in the next section.

The automaton specification for a replica r is given in Figure 7. The set $pending_r$ keeps track

[2]Some implementations may do this for efficiency, or to compensate for faulty channels or servers, and we allow it as it does not affect correctness. However, we assume for now that no faults occur, and we are only concerned with safety. See Section 9 for a discussion of performance and fault-tolerance.

Signature

Input:
  $\mathsf{request}(x)$, where $x \in \mathcal{O}_c$
  $\mathsf{receive}_{rc}(m)$, where r is a replica and $m \in \mathcal{M}_{resp}$
Output:
  $\mathsf{response}(x, v)$, where $x \in \mathcal{O}_c$ and $v \in V$
  $\mathsf{send}_{cr}(m)$, where r is a replica and $m \in \mathcal{M}_{req}$

State

$wait_c$, a subset of $\mathcal{O}$, initially empty

$rept_c$, a subset of $\mathcal{O} \times V$, initially empty

Actions

Input $\mathsf{request}(x)$
  Eff: $wait_c \leftarrow wait_c \cup \{x\}$

Output $\mathsf{send}_{cr}(\langle \text{“request”}, x \rangle)$
  Pre: $x \in wait_c$
  Eff: None

Input $\mathsf{receive}_{rc}(\langle \text{“response”}, x, v \rangle)$
  Eff: if $x \in wait_c$ then $rept_c \leftarrow rept_c \cup \{(x, v)\}$

Output $\mathsf{response}(x, v)$
  Pre: $(x, v) \in rept_c$
    $x \in wait_c$
  Eff: $wait_c \leftarrow wait_c - \{x\}$
    $rept_c \leftarrow rept_c - \{(x, v') : (x, v') \in rept_c\}$

Figure 6: Automaton for the front end of client c

of pending requests. The set $rcvd_r$ contains every operation that this replica has received, either directly from a front end, or through gossip from another replica. The variable $done_r$ is an array of sets of operations, one for each replica, where $done_r[i]$ is the set of operations that r knows are done at replica i. Similarly, $stable_r$ is an array of sets of operations, where $stable_r[i]$ is the set of operations that r knows are stable at i. Note that $done_r[r]$ and $stable_r[r]$ are special in that r does not need to gossip with itself, so $done_r[r]$ is the set of operations that are done at r, and $stable_r[r]$ is the set of operations that are stable at r. The $label_r$ function keeps the minimum label r has seen for each operation identifier, where ∞ indicates that no label has been seen yet. Labels are taken from a well-ordered set $\mathcal{L}$, and this order is extended to ∞, so $l < \infty$ for all $l \in \mathcal{L}$. We use the $label_r$ function to derive lc_r, the ***local constraints*** at r, which is a strict partial order on $\mathcal{I}$.

The labels are partitioned into sets $\mathcal{L}_r$ for each replica r, and replica r only assigns labels from $\mathcal{L}_r$; this ensures that labels are assigned uniquely. For any finite set of labels and any replica r, there exists a label $l \in \mathcal{L}_r$ that is greater than any label in the finite set. This prevents a replica from “getting stuck” without a label to assign to an operation. Because labels are assigned uniquely to the operations done at r, lc_r totally orders $done_r[r]$.

The $\mathsf{receive}_{cr}(\langle \text{“request”}, x \rangle)$ action simply records that a new request from client c for an operation has been received. That request is pending, i.e., it is added to $pending_r$, even if the operation had been received previously.[3] The $\mathsf{do\_it}_r(x, l)$ action assigns a new label $l \in \mathcal{L}_r$ to x, and adds x to the set of done operations. This is allowed only if x is not yet done at replica r and all the operations specified by $x.prev$ set are. The new label is chosen to be greater than

[3]This is unnecessary if communication is reliable, as we assume, but it does not affect correctness, and is natural if the front ends issue multiple requests, as we allow. Some implementations may do this for performance or fault-tolerance. See Section 9 for a discussion of these issues.

Signature

Input:
$\mathsf{receive}_{cr}(m)$, where c is a client and $m \in \mathcal{M}_{req}$
$\mathsf{receive}_{r'r}(m)$, where $r' \neq r$ is a replica and $m \in \mathcal{M}_{gossip}$
Output:
$\mathsf{send}_{rc}(m)$, where c is a client and $m \in \mathcal{M}_{resp}$
$\mathsf{send}_{rr'}(m)$, where $r' \neq r$ is a replica and $m \in \mathcal{M}_{gossip}$
Internal:
$\mathsf{do\_it}_r(x, l)$, where $x \in \mathcal{O}$ and $l \in \mathcal{L}_r$

State

$pending_r$, a subset of $\mathcal{O}$, initially empty; the messages that require a response

$rcvd_r$, a subset of $\mathcal{O}$, initially empty; the operations that have been received

$done_r[i]$ for each replica i, a subset of $\mathcal{O}$, initially empty; the operations r knows are done at i

$stable_r[i]$ for each replica i, a subset of $\mathcal{O}$, initially empty; the operations r knows are stable at i

$label_r : \mathcal{I} \to \mathcal{L} \cup \{\infty\}$, initially all ∞; the minimum label r has seen for $id \in \mathcal{I}$

Derived variable: $lc_r = \{(id, id') : label_r(id) < label_r(id')\}$, a strict partial order on $\mathcal{I}$; the local constraints at r

Actions

Input $\mathsf{receive}_{cr}(\langle \text{“request”}, x\rangle)$
Eff: $pending_r \leftarrow pending_r \cup \{x\}$
$rcvd_r \leftarrow rcvd_r \cup \{x\}$

Internal $\mathsf{do\_it}_r(x, l)$
Pre: $x \in rcvd_r - done_r[r]$
$x.prev \subseteq done_r[r].id$
$l > label_r(y.id)$ for all $y \in done_r[r]$
Eff: $done_r[r] \leftarrow done_r[r] \cup \{x\}$
$label_r(x.id) \leftarrow l$

Output $\mathsf{send}_{rc}(\langle \text{“response”}, x, v\rangle)$
Pre: $x \in pending_r \cap done_r[r]$
$x.strict \implies x \in \bigcap_i stable_r[i]$
$v \in valset(x, done_r[r], \prec_{lc_r})$
$c = client(x.id)$
Eff: $pending_r \leftarrow pending_r - \{x\}$

Output $\mathsf{send}_{rr'}(\langle \text{“gossip”}, R, D, L, S\rangle)$
Pre: $R = rcvd_r$; $D = done_r[r]$;
$L = label_r$; $S = stable_r[r]$

Input $\mathsf{receive}_{r'r}(\langle \text{“gossip”}, R, D, L, S\rangle)$
Eff: $rcvd_r \leftarrow rcvd_r \cup R$
$done_r[r'] \leftarrow done_r[r'] \cup D \cup S$
$done_r[r] \leftarrow done_r[r] \cup D \cup S$
$done_r[i] \leftarrow done_r[i] \cup S$ for all $i \neq r, r'$
$label_r \leftarrow \min(label_r, L)$
$stable_r[r'] \leftarrow stable_r[r'] \cup S$
$stable_r[r] \leftarrow stable_r[r] \cup S \cup (\bigcap_i done_r[i])$

Figure 7: Automaton for replica r

the label for any operation already done at r, so that these operations precede x in the local constraints. If there is a pending request for an operation done at r, the $\mathsf{send}_{rc}(\langle \text{“response”}, x, v \rangle)$ action computes a return value for x according to the local constraints, and relays a message to the client that requested it. If x is strict, this action is enabled only if r knows x is stable at all replicas.

Replica use gossip messages to inform the other replicas about operations they have received and processed. When replica r sends a gossip message, it includes the set R of operations it has received, the set D of operations done at r, the set S of operations stable at r, and a function L giving the minimum label seen by r for each operation. When replica r receives a gossip message from r', it "merges" the information with its knowledge. Specifically, it adds R to its received operations, and adds D to $done_r[r']$. Since any operation that r knows is done at r' is done at r, D is also added to $done_r[r]$. Similarly, S is added to $stable_r[r']$ and $stable_r[r]$. Since S consists of operations stable at r' when the message was sent, every operation in S is done at every replica, so S is also added to $done_r[i]$ for all i. The $label_r$ function is updated to return for each operation, the smaller of the label already known to r and the label in the gossip message. Recall that ∞ indicates that an operation has no label. Finally, operations that r knows are done at every replica, i.e., the operations in $\bigcap_i done_r[i]$, are added to $stable_r[r]$.

6.4 The System

Let *ESDS-Alg* be the composition of all the front end, channel, and replica automata, with the send and receive actions hidden. It is convenient for the analysis of this algorithm to define several derived state variables for this automaton. These are summarized, together with the local constraints defined in the previous subsection, in Figure 8.

- $ops = \bigcup_r done_r[r]$, the set of operations done at any replica
- $ops|_l = \{x \in ops : label_r(x.id) = l \text{ for some } r \text{ or } L_m(x.id) = l \text{ for some } m \in \bigcup_{r,r'} channel_{rr'}\}$, the set of operations with label l
- $minlabel(id) = \min(\bigcup_r \{label_r(id)\})$, the system-wide minimum label for each operation
- $potential\_rept_c = \{(x, v) : \langle \text{“response”}, x, v \rangle \in \bigcup_r channel_{rc} \wedge x \in wait_c\}$, responses en route to c
- $lc_r = \{(id, id') : label_r(id) < label_r(id')\}$, the local constraints at replica r
- $mc_r(m) = \{(id, id') : \min(label_r(id), L_m(id)) < \min(label_r(id'), L_m(id'))\}$, for any $m \in \mathcal{M}_{gossip}$, the message constraints of m at replica r
- $sc = \left(\bigcap_r lc_r\right) \cap \left(\bigcap_{r,r'} \bigcap_{m \in channel_{r'r}} mc_r(m)\right)$, the system constraints
- po, the relation induced by $\mathrm{TC}(CSC(ops) \cup sc)$ on ops

---

Figure 8: Derived variables for *ESDS-Alg*

The set ops is the set of operations done at any replica, and the set $ops|_l$ is the subset of these operations with label l. The function $minlabel$ returns the system-wide minimum label for each operation.

It is also useful to consider the effects of messages in transit. We denote by $potential\_rept_c$ the change to the $rept_c$ set if all the response messages to c are received immediately. When a gossip

message is received by a replica, the local constraints at that replica may change. We define the ***message constraints*** $mc_r(m)$ of a gossip message m at a replica r to be the local constraints that r would have if it received m immediately.

Since each replica assigns labels to operations independently, the local constraints at different replicas need not be consistent. However, because the replicas use the minimum label for each operation, these constraints converge as the replicas gossip. To capture this, we define the ***system constraints*** sc to be those constraints agreed upon by all replicas, taking into account the message constraints. Together with the client-specified constraints, the system constraints define a partial order po that restricts the eventual total order.

7 Invariants

We now prove several invariants about the system $\mathcal{A} = \mathit{ESDS\text{-}Alg} \times \mathit{Users}$. These are used in the next section to show that the algorithm implements the specification when the clients are well-formed.

7.1 Basic Invariants

We first prove several invariants that capture some of the basic intuition about the state variables and the messages sent.

The first invariant says that every operation r knows to be done at any replica is also done at r, and every operation r knows to be stable at any replica is also stable at r.

Invariant 7.1 $done_r[r] = \bigcup_i done_r[i]$ and $stable_r[r] = \bigcup_i stable_r[i]$

Proof: Since $done_r[r] \subseteq \bigcup_i done_r[i]$, and $stable_r[r] \subseteq \bigcup_i stable_r[i]$, we only need to show that $done_r[i] \subseteq done_r[r]$ and $stable_r[i] \subseteq stable_r[r]$ for all i. We prove this by induction on the length of an execution. The base case is trivial because all the sets are empty. But notice that the $done_r[r]$ and $stable_r[r]$ never decrease, and any elements added to $done_r[i]$ or $stable_r[i]$ (when processing a gossip message) are also added to $done_r[r]$ or $stable_r[r]$ respectively. ■

The next invariant says $stable_r[r]$ contains exactly the operations r knows are done at every replica.

Invariant 7.2 $stable_r[r] = \bigcap_i done_r[i]$

Proof: We prove this by induction on the length of an execution. The base case is trivial because all the sets are empty in the initial state. If the invariant holds for s and $s \xrightarrow{\pi} s'$ then

1. If $\pi = \mathsf{do\_it}_r(x, l)$ then note that $x \notin s.done_r[r]$, and since there is at least one other replica r', $x \notin s.done_r[r'] = s'.done_r[r']$ by Invariant 7.1. So $s'.stable_r[r] = s.stable_r[r] = \bigcap_i s.done_r[i] = \bigcap_i s'.done_r[i]$.

2. If $\pi = \mathsf{receive}_{r'r}(m)$ then $s'.stable_r[r] = s.stable_r[r] \cup S_m \cup \bigcap_i s'.done_r[i] \supseteq \bigcap_i s'.done_r[i]$. Also, $s.stable_r[r] \cup S_m = \bigcap_i s.done_r[i] \cup S_m \subseteq \bigcap_i s'.done_r[i]$, so $s'.stable_r[r] \subseteq \bigcap_i s'.done_r[i]$. Thus, $s'.stable_r[r] = \bigcap_i s'.done_r[i]$.

3. Other actions do not change $done_r[i]$ or $stable_r[r]$, so the invariant continues to hold.

■

The next invariant says that the information in a gossip message is not more "up-to-date" than the state of the replica that sent it, and that the set of stable operations in a message is a subset of the set of done operations in that message.

Invariant 7.3 For any gossip message $m \in channel_{rr'}$, $R_m \subseteq rcvd_r$, $D_m \subseteq done_r[r]$, $L_m \geq label_r$, $S_m \subseteq stable_r[r]$, and $S_m \subseteq D_m$.

Proof: We prove this by induction on the length of an execution. The base case is trivial because there are no messages. If the invariant holds in s and $s \xrightarrow{\pi} s'$ then for any $m \in s.channel_{rr'}$, we have $R_m \subseteq s.rcvd_r \subseteq s'.rcvd_r$, $D_m \subseteq s.done_r[r] \subseteq s'.done_r[r]$, $L_m \geq s.label_r \geq s'.label_r$, $S_m \subseteq s.stable_r[r] \subseteq s'.stable_r[r]$, and $S_m \subseteq D_m$. And $s'.channel_{rr'} \subseteq s.channel_{rr'}$ unless $\pi = \mathsf{send}_{rr'}(m)$. But in this case, the new message has $R_m = s'.rcvd_r$, $D_m = s'.done_r[r]$, $L_m = s'.label_r$ and $S_m = s'.stable_r[r]$, and by Invariant 7.2, $S_m \subseteq D_m$. ■

The next invariant says that the information at r about the state of i is not more "up-to-date" than the state of i.

Invariant 7.4 $done_r[i] \subseteq done_i[i]$ and $stable_r[i] \subseteq stable_i[i]$

Proof: We prove this by induction on the length of an execution. The base case is trivial because all the sets are empty. If the invariant holds in s and $s \xrightarrow{\pi} s'$ then $done_i[i]$ and $stable_i[i]$ never decrease, and $done_r[i]$ and $stable_r[i]$ are unchanged unless $\pi = \mathsf{receive}_{r'r}(m)$ for some $r' \neq r$. There are two cases:

1. If $r' \neq i$ then $s'.stable_r[i] = s.stable_r[i]$. By Invariants 7.3 and 7.2, and the inductive assumption (applied for $r = r'$), $S_m \subseteq s.stable_{r'}[r'] \subseteq s.done_{r'}[i] \subseteq s.done_i[i]$, so $s'.done_r[i] = s.done_r[i] \cup S_m \subseteq s.done_i[i] = s'.done_i[i]$.

2. If $r' = i$ then by Invariant 7.3, $s'.stable_r[i] = s.stable_r[i] \cup S_m \subseteq s.stable_r[i] \cup s.stable_i[i] = s.stable_i[i] = s'.stable_i[i]$. By Invariant 7.3, $D_m \subseteq s.done_i[i]$, and following the reasoning above, $S \subseteq s.done_i[i]$, so $s'.done_r[i] = s.done_r[i] \cup D_m \cup S_m \subseteq s.done_i[i] = s'.done_i[i]$.

■

The next invariant says that operations have labels at replica r exactly when they are in $done_r[r]$, and they have labels in a gossip message m exactly when they are in D_m. Recall that ∞ indicates that an operation has no label.

Invariant 7.5 $done_r[r].id = \{id : label_r(id) < \infty\}$ and for any $m \in \bigcup_{r,r'} channel_{rr'}$, $D_m.id = \{id : L_m(id) < \infty\}$.

Proof: We prove this by induction on the length of an execution. The base case is trivial since there are no messages, and for all r, $done_r[r] = \emptyset$ and $label_r(id) = \infty$ for all id. If this invariant holds in s and $s \xrightarrow{\pi} s'$ then:

1. If $\pi = \mathsf{do\_it}_r(x, l)$ then $s'.label_r(x.id) = l < \infty$ and $x \in s'.done_r[r]$, and all other conditions hold from s.

2. If $\pi = \mathsf{send}_{rr'}(m)$ then we have $D_m.id = s.done_r[r].id = \{id : s.label_r(id) < \infty\} = \{id : L_m(id) < \infty\}$. All other conditions hold from s.

3. If $\pi = \mathsf{receive}_{r'r}(m)$ then since $S_m \subseteq D_m$ by Invariant 7.3,

$$\begin{aligned} s'.done_r[r].id &= s.done_r[r].id \cup D_m.id \\ &= \{id : s.label_r(id) < \infty\} \cup \{id : L_m(id) < \infty\} \\ &= \{id : \min(s.label_r(id), L_m(id)) < \infty\} \\ &= \{id : s'.label_r(id) < \infty\}. \end{aligned}$$

   All other conditions hold from s.

4. Other actions do not change $label_r$, $done_r[r]$ or $channel_{rr'}$, so the invariant continues to hold.

■

The next invariant says that operations in the system have been requested.

Invariant 7.6 $\bigcup_c wait_c \cup \{x : \langle \text{“request”}, x\rangle \in \bigcup_{r,c} channel_{cr}\} \cup \bigcup_r rcvd_r \cup ops \subseteq requested$.

Proof: We prove this by induction on the length of an execution. It is trivial in the initial state because all the sets are empty. Suppose the invariant holds in s and $s \xrightarrow{\pi} s'$. If x is added to $wait_c$ then $\pi = \mathsf{request}(x)$, and x is also added to $requested$. If $\langle \text{“request”}, x\rangle$ is added to $channel_{cr}$ then $\pi = \mathsf{send}_{cr}(\langle \text{“request”}, x\rangle)$, so $x \in s.wait_c$. If x is added to $rcvd_r$ then either $\pi = \mathsf{receive}_{cr}(\langle \text{“request”}, x\rangle)$, in which case $\langle \text{“request”}, x\rangle \in s.channel_{cr}$, or $\pi = \mathsf{receive}_{r'r}(m)$ with $x \in R_m$, in which case $x \in s.rcvd_{r'}$ by Invariant 7.3. If x is added to $done_r[r]$ then either $\pi = \mathsf{do\_it}_r(x, l)$ for some l, in which case $x \in s.rcvd_r$, or $\pi = \mathsf{receive}_{r'r}(m)$ with $x \in D_m \cup S_m$, in which case $x \in s.done_{r'}[r']$ by Invariant 7.3. Since $requested$ is never decreased, the invariant holds in s'. ■

The next invariant says that operations for which a response has been generated are done at some replica.

Invariant 7.7 $\{x : \exists v, (x, v) \in rept_c \cup potential\_rept_c\} \subseteq ops$.

Proof: We prove this by induction on the length of an execution. This is true in the initial state because all the sets are empty. If it is true in s and $s \xrightarrow{\pi} s'$ then it must be true in s' since ops never decreases, and $rept_c$ and $potential\_rept_c$ are only increased by the send and receive of response messages. But if $\pi = \mathsf{send}_{rc}(\langle \text{“response”}, x, v\rangle)$ then $x \in s.done_r[r] \subseteq s'.done_r[r] \subseteq s'.ops$, and if $\pi = \mathsf{receive}_{rc}(\langle \text{“response”}, x, v\rangle)$ then $(x, v) \in s.potential\_rept_c$, so $s'.rept_c \cup s'.potential\_rept_c \subseteq s.rept_c \cup s.potential\_rept_c$. ■

The next invariant says that requested operations that are no longer in $wait_c$ for any client c are done at some replica.

Invariant 7.8 $requested - \bigcup_c wait_c \subseteq ops$.

Proof: We prove this by induction on the length of an execution. The base case is trivial because $requested = \emptyset$. If this invariant holds in s and $s \xrightarrow{\pi} s'$ then:

1. If $\pi = \mathsf{request}(x)$ then the invariant continues to hold since although x is added to *requested*, it is also added to $wait_c$, where $c = client(x.id)$.

2. If $\pi = \mathsf{response}(x, v)$ then $(x, v) \in s.rept_c$ for $c = client(x.id)$, so by Invariant 7.7, $x \in s.ops$, and the invariant continues to hold.

3. Other actions decrease neither *ops* nor $\bigcup_c wait_c$, and do not increase *requested*.

■

7.2 System Constraint Lemma and Invariants

We now prove a lemma and several invariants about the system constraints. We begin with the following lemma, which states that the system constraints only increase.

Lemma 7.9 For any reachable state s, if $s \xrightarrow{\pi} s'$ then $s.sc \subseteq s'.sc$.

Proof: For $(id, id') \in s.sc$, we must show that for each replica r, $(id, id') \in s'.lc_r$ and $(id, id') \in s'.mc_r(m)$ for all $m \in \bigcup_{r'} s'.channel_{r'r}$. We do this by cases on π:

1. If $\pi = \mathsf{do\_it}_r(x, l)$ then $x.id \notin s.done_r[r].id$ by Invariant 4.1, since $x \notin s.done_r[r]$. But $id \in s.done_r[r].id$ by Invariant 7.5, since $s.label_r(id) < s.label_r(id') \leq \infty$. Thus, $s'.label_r(id) = s.label_r(id)$. If $id' = x.id$ then $s'.label_r(id') = l > s.label_r(id)$; otherwise, $s'.label_r(id') = s.label_r(id') > s.label_r(id)$. Thus, we have $(id, id') \in s'.lc_r$ since $(id, id') \in s.lc_r$.

   For $m \in \bigcup_{r'} s'.channel_{r'r} = \bigcup_{r'} s.channel_{r'r}$, we have $(id, id') \in s.mc_r(m)$, and thus

   $$\min(s.label_r(id), L_m(id)) < \min(s.label_r(id'), L_m(id')) \leq L_m(id').$$

   Since $\min(s'.label_r(id), L_m(id)) \leq s'.label_r(id) < s'.label_r(id')$, as shown above, we have $(id, id') \in s'.mc_r(m)$.

2. If $\pi = \mathsf{send}_{r'r}(m)$ then $s'.sc = s.sc \cap s'.mc_r(m)$. Since

   $$\begin{aligned} s.sc &\subseteq s.lc_r \cap s.lc_{r'} \\ &\subseteq \{(id, id') : \min(s.label_r(id), s.label_{r'}(id)) < \min(s.label_r(id'), s.label_{r'}(id'))\} \\ &= s'.mc_r(m) \end{aligned}$$

   we have $s'.sc = s.sc$.

3. If $\pi = \mathsf{receive}_{r'r}(m)$ then $s'.lc_r = s.mc_r(m) \supseteq s.sc$. For $m' \in \bigcup_i s'.channel_{ir} = \bigcup_i s.channel_{ir} - \{m\}$, we have $(id, id') \in s.mc_r(m) \cap s.mc_r(m')$. Thus,

   $$\min(s.label_r(id), L_m(id), L_{m'}(id)) < \min(s.label_r(id'), L_m(id'), L_{m'}(id'))$$

   and $(id, id') \in s'.mc_r(m')$.

4. All other actions leave $label_r(id)$, $label_r(id')$ and $\bigcup_{r'} channel_{r'r}$ unchanged, so $(id, id') \in s.lc_r = s'.lc_r$ and $(id, id') \in s.mc_r(m) = s'.mc_r(m)$ for all $m \in \bigcup_{r'} s'.channel_{r'r} = \bigcup_{r'} s.channel_{r'r}$.

■

The next invariant says that labels for operations in the *prev* set of an operation x are no greater than the label for x. The equality is allowed because there may not be a label for either x or the operation in its *prev* set, in which case both will have "labels" of ∞.

Invariant 7.10 If $(id, id') \in CSC(ops)$ then $label_r(id) \leq label_r(id')$, and $L_m(id) \leq L_m(id')$ for all $m \in channel_{rr'}$.

Proof: We prove this by induction on the length of an execution. The base case is trivial since *ops* is empty. If the invariant holds in s and $s \xrightarrow{\pi} s'$ then we show that it holds in s'. The invariant is maintained trivially except by the following actions:

1. If $\pi = \mathsf{do\_it}_r(x, l)$ then for $id \in x.prev \subseteq s.done_r[r].id$, we have $s'.label_r(id) = s.label_r(id) < l = s'.label_r(x.id)$. If $(x.id, id) \in CSC(s'.ops)$ for some $id \in s'.ops.id$ then $s'.ops = s.ops$. By the inductive assumption and Invariant 7.5, $s'.label_r(id) = s.label_r(id) \geq s.label_r(x.id) = \infty > l = s'.label_r(x.id)$.

2. If $\pi = \mathsf{send}_{rr'}(m)$ then $L_m(id) = s.label_r(id) \leq s.label_r(id') = L_m(id')$ by the inductive assumption.

3. If $\pi = \mathsf{receive}_{r'r}(m)$ then by the inductive assumption, $s'.label_r(id) = \min(s.label_r(id), L_m(id)) \leq \min(s.label_r(id'), L_m(id')) = s'.label_r(id')$.

■

The next invariant says that the local constraints at any replica are consistent with the client-specified constraints.

Invariant 7.11 $\mathrm{TC}(CSC(ops) \cup lc_r)$ is a strict partial order.

Proof: $\mathrm{TC}(CSC(ops) \cup lc_r)$ is transitive by definition, so we only need to show it is irreflexive. Suppose, for contradiction, that $(id, id) \in \mathrm{TC}(CSC(ops) \cup lc_r)$, and let $l = label_r(id)$. Then there exist $id_0, id_2, \ldots, id_k$ such that $id = id_0 = id_k$ and $(id_{i-1}, id_i) \in CSC(ops) \cup lc_r$ for $i = 1, \ldots, k$. By Invariant 7.10 and the definition of lc_r, $l = label_r(id_0) \leq label_r(id_1) \leq \cdots \leq label_r(id_k) = l$, so $label_r(id_i) = l$ for $i = 0, \ldots, k$. Thus, $(id_{i-1}, id_i) \notin lc_r$ for $i = 1, \ldots, k$, and so $(id_{i-1}, id_i) \in CSC(ops) \subseteq CSC(requested)$ for all i. However, this implies $(id, id) \in \mathrm{TC}(CSC(requested))$, which contradicts Invariant 4.2. ■

The next invariant is a corollary of the previous one. It says that the system constraints are consistent with the client-specified constraints.

Invariant 7.12 $\mathrm{TC}(CSC(ops) \cup sc)$ is a strict partial order.

Proof: This is immediate from Invariant 7.11 since $\mathrm{TC}(CSC(ops) \cup sc) \subseteq \mathrm{TC}(CSC(ops) \cup lc_r)$ for any r, and it is transitive by definition. ■

7.3 Invariants for Local Constraints

In this subsection, we show that at each replica, different operations have different labels. However, operations at different replicas may have the same label. Recall that the set of operations in ops with label l is $ops|_l$. We first show that operations with labels from $\mathcal{L}_r$ are done at r.

Invariant 7.13 If $l \in \mathcal{L}_r$ then $ops|_l \subseteq done_r[r]$.

Proof: We prove this by induction on the length of an execution. This is trivial in the initial state because ops is empty. If it is true in s and $s \xrightarrow{\pi} s'$ then $s'.ops|_l \subseteq s.ops|_l \subseteq s.done_r[r] \subseteq s'.done_r[r]$, unless $\pi = \mathsf{do\_it}_r(x, l)$, in which case, $s'.ops|_l = s.ops|_l \cup \{x\} \subseteq s.done_r[r] \cup \{x\} = s'.done_r[r]$. ■

The next invariant says that all operations with label l can be ordered so that if an operation has label l at a replica or in a message, then all earlier operations, according to the ordering, have smaller labels at that replica or in that message. This order corresponds to the order in which the operations are first assigned the label l.

Invariant 7.14 There is a total order $\prec_l$ on $ops|_l$ such that if $y \prec_l x$ then

$$\begin{array}{ll} label_r(x.id) = l \implies label_r(y.id) < l & \text{for all } r \\ L_m(x.id) = l \implies L_m(y.id) < l & \text{for } m \in \bigcup_{rr'} channel_{rr'} \end{array}$$

Proof: We prove this by induction on the length of an execution. This is trivial in the initial state since $ops|_l$ is empty. If it is holds in s and $s \xrightarrow{\pi} s'$ then $s'.ops|_l = s.ops|_l$ unless $\pi = \mathsf{do\_it}_r(x, l)$ where $l \in \mathcal{L}_r$. Let $\prec_l$ be a total order satisfying the invariant in s.

1. If $\pi = \mathsf{do\_it}_r(x, l)$ then $x \notin s.done_r[r] \supseteq s.ops|_l$ by Invariant 7.13. Since $s.label_r(y.id) < l$ for $y \in s.done_r[r]$, we have $\prec_l \cup \{(y, x) : y \in s.ops|_l\}$ is a total order on $s'.done_r[r]$ satisfying the invariant in s'.
2. If $\pi = \mathsf{send}_{rr'}(m)$ then for $y \prec_l x$, $L_m(x.id) = s.label_r(x.id) = l \implies L_m(y.id) = s.label_r(y.id) < l$ by the inductive assumption. So $\prec_l$ satisfies the invariant in s'.
3. If $\pi = \mathsf{receive}_{r'r}(m)$ then for $y \prec_l x$, if $s'.label_r(x.id) = l$ then $s.label_r(x.id) = l$ or $L_m(x.id) = l$, so by the inductive assumption, $s'.label_r(y.id) = \min(s.label_r(y.id), L_m(y.id)) < l$.
4. For all other actions, $\prec_l$ continues to satisfy the invariant in s'.

■

The next invariant says that the local constraints totally order the operations done at a replica.

Invariant 7.15 $\prec_{lc_r}$ totally orders $done_r[r]$.

Proof: For $x, y \in done_r[r]$, if $x \not\prec_{lc_r} y \wedge y \not\prec_{lc_r} x$ then $label_r(x.id) = label_r(y.id) = l < \infty$, by the definition of $\prec_{lc_r}$ and Invariant 7.5. Let $\prec_l$ be a total order satisfying Invariant 7.14. Then $x \not\prec_l y$ and $y \not\prec_l x$, so $x = y$. ■

The next invariant says values computed by a replica are consistent with both the client-specified constraints and the local constraints.

Invariant 7.16 For $x \in done_r[r]$, $valset(x, done_r[r], \prec_{lc_r}) = valset(x, ops, \prec_R)$, where $R = \mathrm{TC}(CSC(ops) \cup lc_r)$.

Proof: Since $lc_r \subseteq R$, by Lemmas 2.5 and 2.6, $\emptyset \neq valset(x, ops, \prec_R) \subseteq valset(x, ops, lc_r)$. By Lemma 2.7, $valset(x, ops, lc_r) = \{val(x, done_r[r], lc_r)\}$, since lc_r is a total order on $done_r[r]$ by Invariant 7.15, and $x \in done_r[r] \wedge y \in ops - done_r[r] \implies x \prec_{lc_r} y$ by Invariant 7.5. Thus, $valset(x, done_r[r], \prec_{lc_r}) = valset(x, ops, \prec_R)$. ■

7.4 Invariants for Strict Operations

Finally, we prove several invariants that guarantee that strict operations receive the "correct" values, that is, values consistent with the eventual total order.

The next invariant says that if an operation has label $l \in \mathcal{L}_r$ then the label for that operation at replica r is no larger than l.

Invariant 7.17 For $l \in \mathcal{L}_r$, if $label_{r'}(id) = l$ or $L_m(id) = l$ for some $m \in \bigcup_{i,i'} channel_{ii'}$, then $label_r(id) \leq l$.

Proof: We prove this by induction on the length of an execution. This is trivial in the initial state because $label_r(id) = \infty$ for all $id \in \mathcal{I}$. Suppose it holds in s and $s \xrightarrow{\pi} s'$. Since $s'.label_r(id) \leq s.label_r(id)$, it is sufficient to show that $l \geq s.label_r(id)$.

1. If $s'.label_{r'}(id) = l$ for $r' \neq r$ then either $s.label_{r'}(id) = l \geq s.label_r(id)$ by the inductive assumption, or $\pi = \mathsf{receive}_{ir'}(m)$ for some i and $m \in s.channel_{ir'}$ with $L_m(id) = l$. In this case, $L_m(id) = l \geq s.label_r(id)$, again by the inductive assumption.

2. If $L_m(id) = l$ for $m \in \bigcup_{i,i'} s'.channel_{ii'}$ then either $m \in \bigcup_{i,i'} s.channel_{ii'}$, or $\pi = \mathsf{send}_{ii'}(m)$ for some i and i' and $s.label_i(id) = l$. In either case, $l \geq s.label_r(id)$ by the inductive assumption.

■

Suppose r has its own label $l \in \mathcal{L}_r$ for an operation and some larger label for another operation. The next invariant says that anyone that knows that r has done the second operation has a label no larger than l for the first operation.

Invariant 7.18 If $label_r(id') = l \in \mathcal{L}_r$ and $l \leq label_r(id)$ then

$$
\begin{array}{ll}
id \in done_{r'}[r].id \implies label_{r'}(id') \leq l & \\
id \in D_m.id \implies L_m(id') \leq l & \text{for } m \in channel_{rr'} \\
id \in S_m.id \implies L_m(id') \leq l & \text{for } m \in \bigcup_i channel_{ir'}
\end{array}
$$

Proof: We prove this by induction on the length of an execution. This is trivial in the initial state because $label_r(id') = \infty$ for all $id' \in \mathcal{I}$. Suppose it holds in s and $s \xrightarrow{\pi} s'$ and that $s'.label_r(id') = l \in \mathcal{L}_r$ and $l \leq s'.label_r(id)$.

1. If $\pi = \mathsf{do\_it}_r(x, l')$ with $x.id = id$ then $x \notin s.done_r[r]$, so by Invariant 4.1, $id \notin s.done_r[r].id$. Thus, $id \notin s.done_{r'}[r].id$ by Invariant 7.4, and $id \notin D_m.id$ for $m \in s.channel_{rr'}$ by Invariant 7.3. Also, $S_m \subseteq s.stable_i[i] \subseteq s.done_i[r] \subseteq s.done_r[r]$ for $m \in \bigcup_i s.channel_{ir'}$ by Invariants 7.3, 7.2 and 7.4. So this invariant holds trivially in s'.

2. If $\pi = \mathsf{do\_it}_r(x, l)$ with $x.id = id'$ then $s.label_r(id) \geq l \implies id \notin s.done_r[r].id$, so following the reasoning above, this invariant holds trivially in s'.

3. If $\pi = \mathsf{receive}_{ir}(m)$ then $s'.label_r(id') \in \mathcal{L}_r \implies s'.label_r(id') = s.label_r(id')$, since by Invariant 7.17, if $L_m(id') \in \mathcal{L}_r$ then $s.label_r(id') \leq L_m(id')$. So this invariant continues to hold.

4. If $\pi = \mathsf{send}_{rr'}(m)$ then $D_m = done_r[r]$, and $L_m = label_r$, and $S_m \subseteq D_m$, so the invariant continues to hold.

5. If $\pi = \mathsf{receive}_{rr'}(m)$ then $id \in s'.done_{r'}[r].id = s.done_{r'}[r].id \cup D_m.id \implies s'.label_{r'}(id') = \min(s.label_{r'}(id'), L_m(id')) \leq l$, so the invariant continues to hold.

6. If $\pi = \mathsf{send}_{ir'}(m)$ for $i \neq r$ then $S_m = s.stable_i[i] \subseteq s.done_i[r]$ by Invariant 7.2, so by the inductive assumption (with $r' = i$), $id \in S_m.id \implies L_m(id') = s.label_i(id') \leq l$.

7. If $\pi = \mathsf{receive}_{ir'}(m)$ for $i \neq r$ then $id \in s'.done_{r'}[r].id = s.done_{r'}[r].id \cup S_m.id \implies s'.label_{r'}(id') = \min(s.label_{r'}(id'), L_m(id')) \leq l$, so the invariant continues to hold.

■

The final three invariants are about the labels for stable operations, including those that are ordered before operations in $stable_r[r]$, but not yet in $stable_r[r]$. The first says that r has the system-wide minimum label for any operation with a smaller label than any operation stable at r.

Invariant 7.19 If $id \in stable_r[r].id$ and $minlabel(id') \leq minlabel(id)$ then $label_r(id') = minlabel(id')$.

Proof: Since $id \in ops$, we have $minlabel(id') \leq minlabel(id) < \infty$, so $minlabel(id') = l \in \mathcal{L}_{r'}$ for some r' by Invariant 7.5. By Invariant 7.17, $label_{r'}(id') = l$. Since $l \leq minlabel(id) \leq label_{r'}(id)$ and, by Invariant 7.2, $id \in stable_r[r].id \subseteq done_r[r'].id$, we have $label_r(id') \leq l = minlabel(id')$ by Invariant 7.18. Thus, $label_r(id') = minlabel(id')$. ■

The next invariant says if every replica has the minimum label for an operation, and it is less than the minimum label for another operation, then the first operation precedes the second in the system constraints.

Invariant 7.20 If $label_r(id) = minlabel(id) < minlabel(id')$ for all r then $(id, id') \in \mathrm{TC}(CSC(ops) \cup sc)$.

Proof: For all r, we have $(id, id') \in lc_r$ since $minlabel(id') \leq label_r(id')$ for all r. And for all $m \in \bigcup_{r,r'} channel_{r'r}$, wea also have $(id, id') \in mc_r(m)$, since $minlabel(id') \leq L_m(id')$. Thus, $(id, id') \in sc \subseteq \mathrm{TC}(CSC(ops) \cup sc)$. ■

The next invariant says if an operation is stable at all replicas then operations are ordered to it by the system and client-specified constraints according to their minimum labels.

Invariant 7.21 If $id \in \bigcap_r stable_r[r].id$ then $(id, id') \in \mathrm{TC}(CSC(ops) \cup sc) \iff minlabel(id) < minlabel(id')$.

Proof: By Invariant 7.19, $label_r(id) = minlabel(id)$ for all r. If $minlabel(id) < minlabel(id')$ then $(id, id') \in \mathrm{TC}(CSC(ops) \cup sc)$ by Invariant 7.20. If $minlabel(id) = minlabel(id')$ then $label_r(id) = label_r(id')$ for some r, so by Invariants 7.15 and 4.1, $id = id'$, and $(id, id') \notin \mathrm{TC}(CSC(ops) \cup sc)$ by Invariant 7.12. Otherwise, $minlabel(id') < minlabel(id)$, so by Invariant 7.19, $label_r(id') = minlabel(id')$ for all r. Thus, by Invariant 7.20, $(id', id) \in \mathrm{TC}(CSC(ops) \cup sc)$, and by Invariant 7.12, $(id, id') \notin \mathrm{TC}(CSC(ops) \cup sc)$. ■

8 Simulation

To show that *ESDS-Alg* meets the specification *ESDS-II* when the clients are well-formed, we establish a simulation [20] from $\mathcal{A} = ESDS\text{-}Alg \times Users$ to $\mathcal{S} = ESDS\text{-}II \times Users$.

We begin by extending some earlier results about system constraints to the system-wide partial order po. Recall that po is the relation on ops induced by $\mathrm{TC}(CSC(ops) \cup sc)$. We first note that po is a partial order on ops.

Invariant 8.1 For $\mathcal{A}$: po is a strict partial order with $span(po) \subseteq ops$.

Proof: Immediate from the definition of po and Invariant 7.12. ■

The following lemma extends Lemma 7.9 to po.

Lemma 8.2 For any reachable state s of $\mathcal{A}$, if $s \xrightarrow{\pi} s'$ then $s.po \subseteq s'.po$.

Proof: Immediate from Lemmas 2.4 and 7.9 and the fact that $s.ops \subseteq s'.ops$. ■

The next invariant says that if an operation is stable at all replicas, its relation to other operations in po is determined by their minimum labels.

Invariant 8.3 For $\mathcal{A}$: If $x \in \bigcap_r stable_r[r]$ and $y \in ops$ then $x \prec_{po} y \iff minlabel(x.id) < minlabel(y.id)$.

Proof: Immediate from the definition of po and Invariant 7.21. ■

We now prove the main result, that $\mathcal{A}$ implements $\mathcal{S}$.

Theorem 8.4 The relation F in Figure 9 is a simulation from $\mathcal{A}$ to $\mathcal{S}$.

Proof: To show that F is a simulation from $\mathcal{A}$ to $\mathcal{S}$, we show that for each start state of $\mathcal{A}$, there exists a corresponding start state of $\mathcal{S}$, and that this correspondence is preserved by each step of $\mathcal{A}$.

If s is a start state of $\mathcal{A}$, then *requested* and *responded* are empty, as are $wait_c$ and $rept_c$ are empty for all c, and $rcvd_r$, $done_r[i]$, $label_r$, and $stable_r[i]$ for all replicas r and i. The start state

F is a relation between states in $\mathcal{A}$ and states in $\mathcal{S}$ such that $u \in F[s]$ if and only if:

- $u.requested = s.requested$
- $u.responded = s.responded$
- $u.wait = \bigcup_c s.wait_c$
- $u.rept = \bigcup_c s.rept_c \cup s.potential\_rept_c$
- $u.ops = s.ops = \bigcup_r s.done_r[r]$
- $u.po \subseteq s.po$, the partial order induced by $\mathrm{TC}(CSC(s.ops) \cup s.sc)$ on $s.ops$
- $u.stabilized = \bigcap_r s.stable_r[r]$

Figure 9: Forward simulation from the algorithm to the specification

of $\mathcal{S}$ corresponds to this state since it has *requested* and *responded*, and *ops*, *po*, and *stabilized* all empty, has *wait* and *rept* empty.

To establish that the simulation is preserved by every step of the implementation, suppose that s and u are reachable states of $\mathcal{A}$ and $\mathcal{S}$ respectively such that $u \in F[s]$ and that $s \xrightarrow{\pi} s'$. We show that there exists a state $u' \in F[s']$ such that there is an execution fragment of $\mathcal{S}$ from u to u' that has the same external image as π.

1. If $\pi = \mathsf{request}(x)$, this simulates the same action in the specification, which has the same external image. The $\mathsf{request}(x)$ action is enabled in the specification because $u.requested = s.requested$. The change in state of each automaton is exactly to add x to $wait_c$ in $\mathcal{A}$, to *wait* in $\mathcal{S}$, and to *requested* in both, preserving the simulation as required.

2. If $\pi = \mathsf{send}_{cr}(\langle \text{“request”}, x \rangle)$ then we show $u \in F[s']$, which appears the same externally since the send action is internal. This is true since the send action only adds $\langle \text{“request”}, x \rangle$ to $channel_{cr}$, which does not appear in the simulation.

3. If $\pi = \mathsf{receive}_{cr}(\langle \text{“request”}, x \rangle)$ then we show $u \in F[s']$, which appears the same externally since the receive action is internal. This is true since the receive action only deletes $\langle \text{“request”}, x \rangle$ from $channel_{cr}$, and adds x to $rcvd_r$, which do not appear in the simulation.

4. If $\pi = \mathsf{do\_it}_r(x, l)$ then we have two cases:

   (a) If $x \in s.wait_c$ for some c then we show that $\mathsf{enter}(x, s'.po)$ is enabled in u and $u' \in F[s']$ for u' such that $u \xrightarrow{\mathsf{enter}(x,s'.po)} u'$. First we verify that $\mathsf{enter}(x, s'.po)$ is enabled in u:
   - $x \in s.wait_c \subseteq u.wait$
   - $x.prev \subseteq s.done_r[r].id \subseteq \bigcup_i s.done_i[i].id = u.ops.id$
   - $span(s'.po) \subseteq s'.ops.id = s.ops.id \cup \{x.id\}$ by Invariant 8.1.
   - $u.po \subseteq s.po \subseteq s'.po$ by Lemma 8.2.
   - $CSC(\{x\}) \subseteq s'.po$ since $x \in s'.ops$ and $CSC(s'.ops) \subseteq s'.po$.
   - For $y \in u.stabilized = \bigcap_i s.stable_i[i]$, if $s'.minlabel(x.id) \leq s'.minlabel(y.id)$ then by Invariant 7.19, $s'.minlabel(x.id) = s'.label_r(x.id) = l > s.label_r(y.id) = s.minlabel(y.id)$, which is a contradiction. So $s'.minlabel(y.id) < s'.minlabel(x.id)$, and by Invariant 8.3, $(y.id, x.id) \in s'.po$, as required.

The actions have the same external image since both are internal. The do_it and enter actions do not change *wait*, *rept*, *potential_rept*, *stable* and *stabilized*, and $u'.ops = u.ops \cup \{x\} = \bigcup_i s.done_i[i] \cup \{x\} = \bigcup_i s'.done_i[i]$, and $u'.po = s'.po$, so $u' \in F[s']$ as required.

(b) Otherwise, we check that $u \in F[s']$. The *wait*, *rept*, *potential_rept*, and *stable* variables are unchanged by do_it. Since $x \in s.rcvd_r - \bigcup_c s.wait_c$, we have $x \in s.ops$ by Invariants 7.6 and 7.8, so $s'.ops = s.ops = u.ops$. Finally, $u.po \subseteq s.po \subseteq s'.po$ by Lemma 8.2.

5. If $\pi = \mathsf{send}_{rc}(\langle \text{“response”}, x, v\rangle)$ then let u' be such that $u \xrightarrow{\mathsf{calculate}(x,v)} u'$. These have the same external image because they are both internal.

   First we verify that $\mathsf{calculate}(x, v)$ is enabled in u:

   - $x \in s.done_r[r] \subseteq u.ops$
   - $x.strict \implies x \in \bigcap_i s.stable_r[i] \subseteq \bigcap_i s.stable_i[i] = u.stabilized$ by Invariant 7.4 and the simulation relation.
   - $v \in valset(x, s.done_r[r], \prec_{s.lc_r}) = valset(x, u.ops, \prec_{u.po})$ by Invariant 7.16 and Lemma 2.6, since $u.po \subseteq s.po \subseteq \mathrm{TC}(CSC(s.ops) \cup lc_r)$ and $u.ops = s.ops$.

   To see that $u' \in F[s']$, note that $s'.channel_{rc} = s.channel_{rc} \cup \{\langle \text{“response”}, x, v\rangle\}$. If $x \in s.wait_c \subseteq u.wait$ then $u'.rept = u.rept \cup \{(x, v)\}$, and otherwise, $u' = u$, each as required by the corresponding state change from s to s'.

6. If $\pi = \mathsf{receive}_{rc}(\langle \text{“response”}, x, v\rangle)$ then we show that $u \in F[s']$. This follows because $s'.rept_c = s.rept_c \cup \{(x, v)\}$ if $x \in s.wait_c$ and otherwise, $s'.rept_c = s.rept_c$.

7. If $\pi = \mathsf{response}(x, v)$ then this simulates the same action in the specification, which has the same external image. The $\mathsf{response}(x, v)$ action is enabled in the specification because $x \in s.wait_c \subseteq u.wait$ and $(x, v) \in s.rept_c \subseteq u.rept$, where $c = client(x.id)$. (Note that $x \notin \mathcal{O}_{c'} \supseteq s.wait_{c'}$ and $(x, v) \notin \mathcal{O}_{c'} \times V \supseteq s.rept_{c'}$ for $c' \neq c$, since a front end only keeps operations for its client.) The change in state of each automaton is to remove x from $wait_c$ in $\mathcal{A}$ and from *wait* in $\mathcal{S}$, and to remove all pairs (x, v') from $rept_c$ in $\mathcal{A}$ and from *rept* in $\mathcal{S}$, preserving the simulation as required.

8. If $\pi = \mathsf{send}_{rr'}(m)$ then we show that $u \in F[s']$. Since $s' = s$ except that $s'.channel_{rr'} = s.channel_{rr'} \cup \{m\}$, we need only check that $u.po \subseteq s.po \subseteq s'.po$, which follows from Lemma 8.2.

9. If $\pi = \mathsf{receive}_{r'r}(m)$ then let u' and u'' be such that $u \xrightarrow{\mathsf{add\_constraints}(s'.po)} u' \xrightarrow{\mathsf{stabilize}(x_1)} \cdots \xrightarrow{\mathsf{stabilize}(x_k)} u''$, where $\{x_1, \ldots, x_k\} = \bigcap_i s'.stable_i[i]$.

   By Lemma 8.2, we know that $\mathsf{add\_constraints}(s'.po)$ is enabled in u, and $u'.po = s'.po$. Since the stabilize action only changes the *stabilized* component, which is not used in the precondition, it suffices to check that $\mathsf{stabilize}(x)$ is enabled in u' for each $x \in \bigcap_i s'.stable_i[i]$. For any $y \neq x$, if $s'.minlabel(y.id) < s'.minlabel(x.id)$ then by Invariant 8.3, $y \prec_{s'.po} x$ and if $s'.minlabel(x.id) < s'.minlabel(y.id)$ then $x \prec_{s'.po} y$. So we have $y \preceq_{u'.po} x \vee x \preceq_{u'.po} y$. By Invariant 8.3, if $y \prec_{s'.po} x$ then $s'.minlabel(y.id) < s'.minlabel(x.id)$, and by Invariant 7.19, $s'.label_r(y.id) = s'.minlabel(y.id)$ for all r. Thus, for $y, z \in s.ops|_{\prec_{s'.po} x}$, if $minlabel(y.id) < minlabel(z.id)$ then by Invariant 7.20, $y \prec_{s'.po} z$.

■

9 Performance and Fault-Tolerance

We now derive time bounds on the response and stabilization time for requests, assuming time bounds on the time to do the underlying actions. Initially, we assume that local computation time is negligible, that the channels are reliable, and that there is a bound on the time to deliver messages and the time between sending gossip messages. Later, we consider some cases where these assumptions are relaxed, and also some methods to tolerate faulty processes and channels, and how these methods affect performance.

9.1 Basic Timing Definitions and Assumptions

To prove performance guarantees, we need to extend the model to include time. For a completely formal treatment, we could use a model such as the *general timed automaton* model [18, 21]. For this paper, however, a restricted treatment suffices, and allows us to avoid several technical details. For example, we only consider *admissible* executions, in which time advances to infinity.[4] Rather than augment the automata with time directly, we annotate executions with the times of each event.

Specifically, we define a ***timed execution*** of an automaton A by associating a non-negative real-valued time with each event in an admissible execution of A. Formally, $\alpha = s_0(\pi_1, t_1)s_1(\pi_2, t_2)\cdots$ is a timed execution of A if $s_0\pi_1s_1\pi_2\cdots$ is an execution of A, $t_i \leq t_{i+1}$ for all i, and $t_i \to \infty$ as $i \to \infty$. We say that the event π_i ***occurs*** at time t_i in α.

A predicate holds in α at time t_i if it holds on s_i. Because several events may occur at the same time, it is possible for contradictory predicates to hold at the same time. We also say that a predicate holds by time t if it holds at some time $t' \leq t$. We typically reserve this usage for predicates that once true, remain true.

We now formalize the timing assumptions for $\mathcal{A}$. Let d_{ij} be an upper bound on the time to deliver messages from i to j. That is, if a send_{ij} event occurs at time t, the corresponding $\mathsf{receive}_{ij}$ event must occur by time $t + d_{ij}$. Let d_{f} be the maximum of all d_{cr} and d_{rc} bounds, and d_{g} be the maximum of all $d_{rr'}$ bounds. Thus, d_{f} is an upper bound on the delivery time for messages between front ends and replicas, and d_{g} is an upper bound on the delivery time for gossip messages. We also define a quantity called the ***gossiping delay***. The gossiping delay $g_{rr'}$ for any two replicas r and r' is an upper bound on the time between successive sendings of gossip messages from r to r', and g is the maximum of all $g_{rr'}$ bounds.

We assume that local computation time is negligible, so that each front end immediately relays each request to some replica, and computing the results of each operation, and processing gossip messages is instantaneous. We also assume that replicas immediately send out response messages when possible, and that front ends immediately respond to clients when possible.

Formally, a timed execution α ***satisfies the timing assumptions*** in an interval I if for all $t \in I$:

1. If a $\mathsf{request}(x)$ event occurs by t then a $\mathsf{send}_{cr}(\langle\text{“request”}, x\rangle)$ event occurs by t for some r

[4] In the literature, admissible executions may be finite if only input actions are enabled in the final state. However, sending gossip messages is always enabled in *ESDS-Alg*, so we need not consider this possibility.

and $c = client(x.id)$.

2. If $\mathsf{do\_it}_r(x, \cdot)$ is enabled at t then x is done at r by t.
3. If $\mathsf{send}_{rc}(\langle \text{“response”}, x, \cdot \rangle)$ is enabled at t then a $\mathsf{send}_{rc}(\langle \text{“response”}, x, \cdot \rangle)$ event occurs by t.
4. If $\mathsf{response}(x, \cdot)$ is enabled at t then a $\mathsf{response}(x, \cdot)$ event occurs by t.
5. For all replicas r and r', if $t \geq g_{rr'}$ then at least one $\mathsf{send}_{rr'}(m)$ event occurs in $(t - g_{rr'}, t]$.
6. If a $\mathsf{send}_{ij}(m)$ event occurs by $t - d_{ij}$ then the corresponding $\mathsf{receive}_{ij}(m)$ event occurs by t.

The interval $[0, \infty)$ is assumed if no interval is explicitly specified.

Note that this definition also constrains events enabled before the interval. For example, if α satisfies the timing assumptions in $[t_1, t_2]$ and a message is sent from i to j at time $t < t_1$, then the message must be delivered by $\max(t_1, t + d_{ij})$ if $t + d_{ij} \leq t_2$.

If several operations are received but not done at a replica, doing some of them may allow others to be done. It is convenient to characterize these operations. We say that an operation is ***ready*** at replica r if $x \in rcvd_r$ and $y \prec_c x \implies y \in rcvd_r$ for all $y \in requested$. Thus, if x is ready at r then it has been received by r and all operations specified by its *prev* set are also ready at r. The following lemma says that an operation is done as soon as it is ready.

Lemma 9.1 In any timed execution of $\mathcal{A}$ that satisfies the timing assumptions in an interval I, if x is ready at replica r at time $t \in I$ then it is done at r by t.

Proof: Immediate from the second condition of the timing assumptions, since $\prec_c$ induces a strict partial order on $rcvd_r$. ■

9.2 Performance Without Failures

Assuming the local computation time is negligible and that there are no failures, the response time for a nonstrict request with an empty *prev* set is simply the roundtrip time between the front end and a replica. For strict requests, or requests with nonempty *prev* sets, the analysis is not so easy. The basic intuition is that an operation may need to wait for one round of gossiping to receive all the operations specified in its *prev* set. A strict operation must be in $\bigcap_i stable_r[i]$ for some replica r before it may generate a response. This may take two extra rounds of gossiping, one for all replicas to know that it is done at some replica, and thus to be stable at all replicas, and one more for r to learn this.

We first prove the following lemma, which bounds the time after an operation is requested until it is done at every replica, if the timing assumptions are satisfied.

Lemma 9.2 If α is a timed execution of $\mathcal{A}$ that satisfies the timing assumptions and x is requested by time t, then x is done at every replica by time $t + d_{\mathrm{f}} + g + d_{\mathrm{g}}$.

Proof: (Sketch) Suppose an operation x is requested by client c at time t. A request message is sent immediately to some replica r, so by time $t + d_{cr}$, we have $x \in rcvd_r$. For every other replica

r', there is at least one $\mathsf{send}_{rr'}(m)$ event in $(t+d_{cr}, t+c_{cr}+g_{rr'}]$ with $x \in R_m$. Therefore, r' receives x by $t + d_{cr} + g_{rr'} + d_{rr'}$, and every replica receives x by $t + d_{\mathrm{f}} + g + d_{\mathrm{g}}$.

Because the users are well-formed, any operation required to precede x must have been requested at time $t' \leq t$, and by the reasoning above, received by every replica by $t' + d_{\mathrm{f}} + g + d_{\mathrm{g}}$. So, by $t + d_{\mathrm{f}} + g + d_{\mathrm{g}}$, x is ready at every replica, and by Lemma 9.1, x is done at every replica by $t + d_{\mathrm{f}} + g + d_{\mathrm{g}}$. ■

For any operation x, we define the upper bound on the response time for x to be:

$$\delta(x) = \begin{cases} 2d_{\mathrm{f}} & \text{if } \neg x.strict \wedge x.prev = \emptyset \\ 2d_{\mathrm{f}} + g + d_{\mathrm{g}} & \text{if } \neg x.strict \wedge x.prev \neq \emptyset \\ 2d_{\mathrm{f}} + 3(g + d_{\mathrm{g}}) & \text{if } x.strict \end{cases}$$

Then we summarize the results in the following theorem:

Theorem 9.3 If α is a timed execution of $\mathcal{A}$ that satisfies the timing assumptions, and x is requested by time t, then a $\mathsf{response}(x, v)$ event occurs within $[t, t + \delta(x)]$ in α.

Proof: (Sketch) Suppose an operation x is requested at time t. A request message is sent immediately to some replica r, so by time $t + d_{\mathrm{f}}$, we have $x \in rcvd_r$. If $x.prev$ is empty then a response message is sent immediately, and a $\mathsf{response}(x, v)$ event occurs by $t + 2d_{\mathrm{f}}$. Otherwise, by Lemma 9.2, x is done at every replica by $t + d_{\mathrm{f}} + g + d_{\mathrm{g}}$. If x is not strict, r sends a response message immediately, and a $\mathsf{response}(x, v)$ event occurs by $t + 2d_{\mathrm{f}} + g + d_{\mathrm{g}}$.

For any two replicas i and i', there is at least one $\mathsf{send}_{i'i}(m)$ event with $x \in D_m$ in $(t + d_{\mathrm{f}} + g + d_{\mathrm{g}}, t + d_{\mathrm{f}} + 2g + d_{\mathrm{g}}]$. So by $t + d_{\mathrm{f}} + 2(g + d_{\mathrm{g}})$, we have $x \in \bigcap_{i'} done_i[i'] = stable_i[i]$ for every replica i. And again, we have $x \in \bigcap_i stable_r[i]$ by $t + d_{\mathrm{f}} + 3(g + d_{\mathrm{g}})$. Thus, a $\mathsf{response}(x, \cdot)$ event occurs by $t + 2d_{\mathrm{f}} + 3(g + d_{\mathrm{g}})$. ■

If a client only specifies dependencies on operations it requested, and its front end always communicates with the same replica, then every operation requested by that client is ready as soon as it is received by that replica, and so the delay for nonstrict operations is reduced to at most $2d_{\mathrm{f}}$.

9.3 Fault-Tolerance

The algorithm does not depend on any timing assumptions for correctness, nor does it restrict the order of delivery of messages. Thus, slow processes and delayed message delivery do not affect correctness. They do, of course, affect performance. However, the analysis in the failure-free case holds starting from any reachable state of the system. Thus, even if some part of the system fails for a period of time, as long as it does not make any false computations, then the performance analysis above holds. This is captured by the following theorem:

Theorem 9.4 Suppose α is a timed execution of $\mathcal{A}$ that satisfies the timing assumptions in the interval $[t, \infty)$. If x is requested by t then a $\mathsf{response}(x, v)$ event occurs within $[t, t + \delta(x)]$ in α.

Proof: (Sketch) Note that Lemma 9.2 is true even if the timing assumptions are only satisfied in the interval $[t, t + d_{\mathrm{f}} + g + d_{\mathrm{g}}]$, and that a request message is sent to some replica by time t. The rest of the proof follows exactly the proof of Theorem 9.3. ■

It is easy to see that even message loss does not affect any safety properties, because the algorithm cannot distinguish lost messages from merely delayed ones. Alternatively, we could show that all the invariants, and the simulation relation, are preserved with the addition of an action that simply removes a message from a channel. (This would be an internal action, otherwise identical to the receive action.) Similarly, it is easy to show that duplicate messages do not compromise any safety properties.

If replicas may crash and restart, but there is no volatile memory, then a crash is indistinguishable from message loss to any other process, and so safety is still preserved. If memory is volatile, most of the state can be reconstructed from the gossip messages. A replica recovers by requesting new gossip messages and waiting for a response from each replica before resuming the algorithm. The key to establishing correctness is that after recovery, the replica should have a label for each operation that is less than or equal to the label it had for that operation before the crash. This is only a problem if the smallest label it had prior to the crash was generated locally, so only those labels need to be kept in stable storage. Other methods can also be used to ensure this property, but these are beyond the scope of this paper.

10 Optimizations of the Abstract Algorithm

The algorithm we presented so far deals with the fundamental problems of maintaining consistency in a distributed, replicated data service, and is stated at a high level, ignoring important issues of local computation, local memory requirements, message size, and congestion. In this section, we explore some ways to improve the algorithm to address these issues better.

10.1 Memoizing Stable State

In defining the *ESDS-Alg* automaton, we were not concerned with modelling local computation, and the value returned by a replica is derived by computing all the preceding operations in the $label_r$ order each time a response is issued by that replica. Of course, this is computationally prohibitive, and a real implementation would do some sort of *memoization* of the state of the data to avoid redundant computation. In particular, once an operation has stabilized, as long as its value is remembered, it never needs to be recomputed since its place in the eventual total order is fixed. However, because a replica may temporarily misorder some operations, some recomputation of unstable operations may still be necessary.

Operations should be memoized in the order they appear in the eventual total order. Thus, an operation may be memoized by a replica only if its place in the eventual total order is already known at that replica. This is true not only of the stable operations, but also of those operations in the "gaps" between the stable operations. We say that an operation is ***solid*** at replica r if it is stable at r or if it is locally constrained to precede some operation stable at r. We introduce a derived state variable $solid_r = \bigcup_{y \in stable_r[r]} done_r[r]|_{\preceq_{lc_r} y}$ to express the set of operations solid at r. Notice that $solid_r$ does not have the "gaps" that $stable_r[r]$ might have.

Invariant 10.1 If $stable_r[r] \neq \emptyset$ then $solid_r = done_r[r]|_{\preceq_{lc_r} y}$, where $y = \max_{\prec_{lc_r}} stable_r[r]$.

Proof: By Invariant 7.15, $\prec_{lc_r}$ is a total order on $done_r[r]$, so $y = \max_{\prec_{lc_r}} stable_r[r]$ is well-defined.

If $x \in solid_r$ then $x \preceq_{lc_r} y'$ for some $y' \in stable_r[r]$. Thus, $x \preceq_{lc_r} y' \preceq_{lc_r} y$, so $x \in done_r[r]|_{\preceq_{lc_r} y}$. ■

The eventual total order is determined by the labels that the replicas ultimately agree upon for each operation. The following lemma says that once an operation is solid at a replica, its label at that replica does not change.

Lemma 10.2 For any reachable state s of $\mathcal{A}$, if $id \in s.solid_r.id$ and $s \xrightarrow{\pi} s'$ then $s'.label_r(id) = s.label_r(id)$.

Proof: This is immediate from the definition of the automaton unless $\pi = \mathsf{do\_it}_r(x, l)$ with $x.id = id$ or $\pi = \mathsf{receive}_{r'r}(m)$. The first case is impossible by Invariant 4.1, since $id \in done_r[r].id$ by Invariant 7.2. By the definition of $solid_r$, there exists $id' \in stable_r[r].id$ such that $s.label_r(id) \leq s.label_r(id')$. By Invariant 7.19, $s.label_r(id') = s.minlabel(id')$, and since $s.minlabel(id) \leq s.label_r(id) \leq s.minlabel(id')$, we also have $s.label_r(id) = s.minlabel(id)$. If $\pi = \mathsf{receive}_{r'r}(m)$ then by Invariant 7.3, $L_m(id) \geq s.label_{r'}(id) \geq s.minlabel(id) = s.label_r(id)$, and thus $s'.label_r(id) = s.label_r(id)$. ■

We modify the automaton for replica r as shown in Figure 10 to model such memoization explicitly. We augment the state of each replica with three variables, $memoized_r$, mv_r and ms_r.

Changes to State

Derived variable: $solid_r = \bigcup_{y \in stable_r[r]} done_r[r]|_{\preceq_{lc_r} y}$; the operations solid at r

$memoized_r$, a subset of $\mathcal{O}$; initially empty; the operations that have been memoized by r

$ms_r \in \Sigma$, initially σ_0; the state resulting from doing all the operations in $memoized_r$

$mv_r : memoized_r \to V$, initially null; the values of the memoized operations in the eventual total order

Changes to Actions

Output $\mathsf{send}_{rc}(\langle \text{“response”}, x, v \rangle)$
- Pre: $x \in pending_r \cap done_r[r]$
  $x.strict \implies x \in \bigcap_i stable_r[i]$
  if $x \in memoized_r$
  then $v = mv_r(x)$
  else $v \in valset_{ms_r}(x, done_r[r] - memoized_r, \prec_{lc_r})$
  $c = client(x.id)$
- Eff: $pending_r \leftarrow pending_r - \{x\}$

Internal $\mathsf{memoize}_r(x)$
- Pre: $x \in solid_r - memoized_r$
  $done_r[r]|_{\prec_{lc_r} x} \subseteq memoized_r$
- Eff: $ms_r, mv_r(x) \leftarrow \tau(ms_r, x.op)$
  $memoized_r \leftarrow memoized_r \cup \{x\}$

Figure 10: Memoizing operations: Changes to replica r

The set $memoized_r$ contains the operations that have been memoized by r. The mv_r function stores the values for all the operations in $memoized_r$, and ms_r reflects the state of the data after applying those operations. We modify the action that computes return values to use mv_r for the memoized operations, and to start from ms_r, rather than the initial state, for later operations.

We also add a $\mathsf{memoize}_r$ action which nondeterministically memoizes operations. An operation can be memoized by r if it is solid at r and all operations with smaller labels at r have already been memoized. This action computes the value for the operation being memoized, and updates ms_r appropriately.

Let *ESDS-Alg'* be the composition of the refined replica automata and the original front end and channel automata. It is not difficult to prove that *ESDS-Alg* and *ESDS-Alg'* are equivalent.

The key lemmas are the following invariants of *ESDS-Alg'*.

Invariant 10.3 $memoized_r \subseteq solid_r$.

Proof: This follows immediately from the automaton definition, by induction on the length of an execution. ■

Invariant 10.4 $ms_r = outcome(memoized_r, \prec_{lc_r})$, and $mv_r(x) = val(x, done_r[r], \prec_{lc_r})$ for all $x \in memoized_r$.

Proof: We prove this by induction on the length of an execution. This is trivial in the initial state since $memoized_r$ is empty and $ms_r = \sigma_0$. If the invariant holds in s and $s \xrightarrow{\pi} s'$ then by Invariants 10.3 and 7.19, $s.label_r(id) = s.minlabel(id)$ for all $id \in s.memoized_r.id$, so the partial order induced by lc_r on $s.memoized_r$ is unchanged. So unless $\pi = \mathsf{memoize}_r(x)$ the invariant continues to hold. If $\pi = \mathsf{memoize}_r(x)$ then $s'.lc_r = s.lc_r$ and $y \prec_{s.lc_r} x$ for all $y \in s.memoized_r$, so $s'.ms_r = \tau(s.ms_r, x.op).s = \tau(outcome(s.memoized_r, \prec_{s.lc_r}), x.op).s = outcome(s'.memoized_r, \prec_{s'.lc_r})$ and $s'.mv_r(x) = \tau(s.ms_r, x.op).v = val(x, s'.done_r[r], \prec_{s'.lc_r})$. ■

Invariant 10.5 $valset_{ms_r}(x, done_r[r] - memoized_r, \prec_{lc_r}) = valset(x, done_r[r], \prec_{lc_r})$ for all $x \in done_r[r] - memoized_r$.

Proof: Immediate from Invariant 10.4 and Lemma 2.7. ■

10.2 Reducing Memory Requirements

It is also possible to significantly reduce some local memory requirements implicit in the abstract algorithm. In particular, *ESDS-Alg* specifies that for every operation, all the information specified by the client, plus the minimum label, is maintained at each replica. However, the *prev* sets are only used by the do_it action, and once a replica has an operation in its $done_r[r]$ set, it may free that memory.

Memoizing stable state can also have a positive impact on the memory requirements. This follows from the same observation that led us to memoize the stable state to reduce local computation: stable operations do not have to be recomputed, as long as we remember their return values. This means that once an operation is memoized, all the information about it can be purged from the memory, except its identifier and return value. Furthermore, if a replica knows that it will never need to respond with the value of an operation again, it can purge even that from its memory. For example, if communication is perfectly reliable, then once a response is sent to a front end, it will never need to be sent again, even if another request for the same operation is received. When communication is not reliable, acknowledgements can be used to achieve the same effect. Thus, while *ESDS-Alg'* has more state variables than *ESDS-Alg*, a reasonable implementation of *ESDS-Alg'* will in practice be more memory efficient as well.

Unfortunately, the identifiers cannot be so readily dispensed with, since they are required in case they are included in the *prev* sets of future operations. However, by imposing some structure on these identifiers, it is possible to summarize them so they do not take linear space with the number

of operations issued. A simple time-based strategy can be used to achieve this. For example, if the identifiers included the date of request, and all operations are guaranteed to be stable within a certain time period, then all identifiers older than this time may be expunged from the memory. A more sophisticated approach can involve logical timestamps, such as the multipart timestamps of [15].

10.3 Exploiting Commutativity Assumptions

The algorithm of [15] is intended to be used when most of the operations require only causal ordering, but it allows two other types of operations which provide stronger ordering constraints. The ordering constraints on an operation must be determined by the application developer, not the client, based on "permissible concurrency". Otherwise, clients may cause, perhaps inadvertantly, an irreconcilable divergence of the data at different replicas. For example, suppose an "increment" and a "double" operation are requested concurrently, and are done in different orders at two replicas. If the value at both replicas was initially 1, then the replica that does the increment first will have a final value of 4, while the replica that does the double first will have a final value of 3, even after the operations stabilize.

In this section, we describe how to further reduce the need to recompute operations, when all operations have sufficient "permissible concurrency". We begin with a careful characterization of the relationship between data operators.

Suppose that $op_1, op_2 \in O$ are two operators of the data type. We say that they ***commute*** if $\tau^+(\sigma, \langle op_1, op_2 \rangle).s = \tau^+(\sigma, \langle op_2, op_1 \rangle).s$ for all $\sigma \in \Sigma$. We say that op_1 is ***oblivious*** to op_2 if $\tau^+(\sigma, \langle op_2, op_1 \rangle).v = \tau(\sigma, op_1).v$ for all $\sigma \in \Sigma$. We say that two operations are ***independent*** if they commute and are oblivious to each other.

We first state without proof some lemmas about how commutativity and independence restricts the possible results of operations consistent with a partial order on the set of operations. The first lemma says that if the partial order orders all operations that do not commute then the final outcome of applying these operations is determined. The second lemma says that if it orders all operations that are not independent, then the return values are also determined.

Lemma 10.6 If $\prec$ is a partial order on a finite set X of operations such that $x \prec y$ or $y \prec x$ for all $x, y \in X$ that do not commute, then for all $\sigma \in \Sigma$, $outcome_\sigma(X, \prec')$ is the same for all total orders $\prec'$ on X consistent with $\prec$.

Lemma 10.7 If $\prec$ is a partial order on a finite set X of operations such that $x \prec y$ or $y \prec x$ for all $x, y \in X$ that are not independent, then for all $\sigma \in \Sigma$, $|valset_\sigma(x, X, \prec)| = 1$ for all $x \in X$.

If we require that clients explicitly order every pair of operations that are not independent, then by Lemma 10.7, the return value is uniquely determined by the client-specified constraints. Thus, any values consistent with the client-specified constraints are also consistent with the eventual total order, and are the same values that would be returned by an atomic memory. So an implementation need not keep track of *stable* sets, or even *done* sets at other replicas.

Suppose we only require that clients explicitly order operations that do not commute.[5] Formally, we model this by adding a clause to the precondition of the request(x) action of *Users*. Call this new

[5]This condition is still very strong. A weaker variation may be sufficient for the algorithm of [15] since *updates*

automaton *SafeUsers*. We show how to modify the algorithm to take advantage of this restricted client using Lemma 10.6.

We again modify the automaton of each replica r. We augment the state with two additional state variables, cs_r and val_r. We do not need the mv_r function anymore, because we simply reassign the val_r function when an operation is memoized. The cs_r reflects all the operations in

---

Signature

Same as in Figure 7.

State

Same as in Figure 7.

Derived variable: $solid_r = \bigcup_{y \in stable_r[r]} done_r[r]|_{\preceq_{lc_r} y}$; the operations solid at r

$memoized_r$, a subset of $\mathcal{O}$; initially empty; the operations that have been memoized by r

$ms_r \in \Sigma$, initially σ_0; the state resulting from doing all the operations in $memoized_r$

$cs_r \in \Sigma$, initially σ_0; the state resulting from doing all the operations in $done_r[r]$

$val_r : done_r[r] \to V$, initially null; the value for $x \in done_r[r]$

Actions

Input $\mathsf{receive}_{cr}(\langle \text{“request”}, x \rangle)$
Eff: $pending_r \leftarrow pending_r \cup \{x\}$
$rcvd_r \leftarrow rcvd_r \cup \{x\}$

Internal $\mathsf{do\_it}_r(x, l)$
Pre: $x \in rcvd_r - done_r[r]$
$x.prev \subseteq done_r[r].id$
$l > label_r(y.id)$ for all $y \in done_r[r]$
Eff: $done_r[r] \leftarrow done_r[r] \cup \{x\}$
$cs_r, val_r(x) \leftarrow \tau(cs_r, x.op)$
$label_r(x.id) \leftarrow l$

Output $\mathsf{send}_{rc}(\langle \text{“response”}, x, v \rangle)$
Pre: $x \in pending_r \cap done_r[r]$
$x.strict \implies x \in \bigcap_i stable_r[i] \cap memoized_r$
$v = val_r(x)$
$c = client(x.id)$
Eff: $pending_r \leftarrow pending_r - \{x\}$

Output $\mathsf{send}_{rr'}(\langle \text{“gossip”}, R, D, L, S \rangle)$
Pre: $R = rcvd_r$; $D = done_r[r]$;
$L = label_r$; $S = stable_r[r]$

Input $\mathsf{receive}_{r'r}(\langle \text{“gossip”}, R, D, L, S \rangle)$
Eff: $rcvd_r \leftarrow rcvd_r \cup R$
$done_r[r'] \leftarrow done_r[r'] \cup D \cup S$
$done_r[i] \leftarrow done_r[i] \cup S$ for all $i \neq r, r'$
for $y \in D - done_r[r]$
(in any order consistent with $CSC(D)$)
$done_r[r] \leftarrow done_r[r] \cup \{y\}$
$cs_r, val_r(y) \leftarrow \tau(cs_r, y.op)$
$label_r \leftarrow \min(label_r, L)$
$stable_r[r'] \leftarrow stable_r[r'] \cup S$
$stable_r[r] \leftarrow stable_r[r] \cup S \cup (\bigcap_i done_r[i])$

Internal $\mathsf{memoize}_r(x)$
Pre: $x \in solid_r - memoized_r$
$done_r[r]|_{\prec_{lc_r} x} \subseteq memoized_r$
Eff: $ms_r, val_r(x) \leftarrow \tau(ms_r, x.op)$
$memoized_r \leftarrow memoized_r \cup \{x\}$

Figure 11: Automaton for replica r with current state

$done_r[r]$, and val_r is computed as each operation is added to $done_r[r]$, whether by a $\mathsf{do\_it}_r$ action, or by processing gossip received from another replica. The code for this new replica is given in Figure 11.

Let *Commute* be the composition of these new replica automata and the original channel and front end automata, with the send and receive actions hidden, and $\mathcal{C} = Commute \times SafeUsers$. We want to show that $\mathcal{C}$ implements $\mathcal{S}$. The proof for this follows the proof that $\mathcal{A}$ implements $\mathcal{S}$. It

and *queries* are handled differently, and operations may not atomically read and write the data.

is easy to check that every action has an equivalent or stronger precondition, and identical effects on the original state variables, so that all the invariants in Section 7 are invariants of $\mathcal{C}$.

There are two main changes to the simulation proof. First, we need to check that the memoize action preserves the simulation, which is handled in much the same way as in Section 10.1. Second, the calculate(x, v) action is now simulated by the action which assigns v to $val_r(x)$, instead of the send$(\langle$"response"$, x, v\rangle)$ action. This is either a do_it$_r(x)$ or receive$_{r'r}(m)$ action for nonstrict operations, and a memoize$_r(x)$ for strict operations. Lemma 10.6 is used to show that cs may be used to compute the return values for nonstrict operations.

10.4 Reducing Communication

There are also many possibilities for reducing communication overhead, or weakening the assumptions about the communication mechanism. These are largely orthogonal to the work in this paper, but we mention a few possibilities to give a sense of how this may be done.

In the abstract algorithm, replicas send gossip messages that include information previously gossiped. If the channels are reliable and FIFO, it is possible to reduce the gossip message sizes by sending only incremental information. The use of timestamps, including logical timestamps such as the multipart timestamps of [15], to summarize sets of operations, as noted above, also reduces the size of messages.

Also, the algorithm specifies that each replica sends a separate gossip message to every other replica, resulting in a quadratic number of messages for each "round" of gossip. However, the algorithm allows a replica to send the same gossip message to all other replicas, so an efficient broadcast protocol could greatly reduce the number of messages sent.

11 Implementation and Uses of the Eventually-Serializable Data Services

An important consideration in our work is that our specification be reasonable for real systems. We close this paper with an overview of an implementation of the eventually-serializable data service, and a discussion of some sample applications of this service.

11.1 An Experimental Implementation

The abstract algorithm presented in this paper was used by Cheiner [6, 7] as the basis for developing an exploratory implementation of the eventually-serializable data service. The main objectives of this implementation were to show that a modular implementation of the eventually-serializable data service can be used by dissimilar clients, and to obtain empirical data on the scalability of the implementation, and the trade-off between consistency and performance. We present only an overview of this work; the reader should refer to the papers cited above for details.

The implementation of the service runs on a network of Unix workstations and uses MPI [8] as its message passing mechanism. The implementation is coded in C++, and incorporates some of the optimizations described above. Because of its object-oriented design, it is easy to parameterize

the implementation for different serial data types and to integrate it with a variety of clients. The clients for which prototypes were developed include a Web client, a text-oriented Unix client and Microsoft Excel client for Windows95. This demonstrates the suitability of the service as a generic distributed system building block.

To evaluate its scalability, the implementation was tested using one to ten replicas. These tests used only nonstrict operations. As the number of replicas increased and the frequency of requests per replica held constant, the throughput of the system increased almost linearly.

To evaluate the effect of strict operations on performance, the average percentage of strict requests (determined randomly) was increased from 0% to 100%. It was observed that latency increased linearly as the proportion of strict requests increased. This provides evidence that the service indeed reflects a designed trade-off between consistency and performance.

11.2 Directory Services and Distributed Repositories

The eventually-serializable data service is well-suited for implementing distributed directory services. In any computing enterprise, naming and directory service are important basic services used to make distributed resources accessible transparently of the resource locations or their physical addresses. Such services include Grapevine [5], DECdns [17], DCE GDS (Global Directory Service) and CDS (Cell Directory Service) [24], ISO/OSI X.500 [14], and the Internet's DNS (Domain Name System) [13].

A directory service must be robust and it must have good response time for name lookup and translation requests in a geographically distributed setting. Access to a directory service is dominated by queries and it is unnecessary for the updates to be atomic in all cases. Consequently, the implementations use redundancy to ensure fault-tolerance, replication to provide fast response to queries, and lazy propagation of information for updates. A service can also provide a special update feature that ensures that the update is applied to all replicas expediently.

Directory services often use an object-based definition of names, in which a name has a set of attributes determined by its type. When a new name object is created, it must be possible to initialize, and subsequently query, the attributes of the created object. With an eventually-serializable data service, this can be accomplished by including the identifier of the name creation operation in the *prev* sets of the attribute creation and initialization operations.

Another application of the eventually-serializable data service is in implementing distributed information repositories for coarse-grained distributed object frameworks such as CORBA [22]. Important components of such a framework include the distributed type system used to define object types, and the module implementation repository used for dynamic object dispatching [25]. In this setting, the access patterns are again dominated by queries, and infrequent update requests can be propagated lazily with the guarantee of eventual consistency.

Acknowledgments: The authors thank Oleg Cheiner for his technical insight, and for pointing out errors in a preliminary version of this paper, and Paul Attie for his careful reading of a much later version. We are also grateful to Roberto Segala, whose detailed comments greatly improved the presentation of this paper, and led us to significantly extend the discussion on performance and fault-tolerance.

References

[1] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 627–644, Oct. 1976.

[2] H. Attiya and J. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2), 1994.

[3] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, Mar. 1992.

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

[5] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, 1982.

[6] O. Cheiner. Implementation and evaluation of an eventually-serializable data service. Master of Engineering thesis, Massachusetts Institute of Technology, Aug. 1997.

[7] O. Cheiner and A. Shvartsman. Implementing and evaluating an eventually-serializable data service. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, page 317, June/July 1997. Full paper will appear in a DIMACS volume, *Networks and Distributed Computing*, eds. M. Mavronicolas, M. Merritt and N. Shavit, 1998.

[8] J. Dongarra, S. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, July 1996.

[9] M. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM Symposium on Database Systems*, pages 70–75, Mar. 1982.

[10] H. Garcia-Molina, N. Lynch, B. Blaustein, C. Kaufman, and O. Schmueli. Notes on a reliable broadcast protocol. Technical memorandum, Computer Corporation America, Oct. 1985.

[11] D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Principles of Operating Systems Principles*, pages 150–162, Dec. 1979.

[12] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, Feb. 1986.

[13] IETF. *RFC 1034* and *RFC 1035* Domain Name System, 1990.

[14] International Standard 9594-1, Information Processing Systems—Open Systems Interconnection—The Directory, ISO and IEC, 1988.

[15] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy replication: Exploiting the semantics of distributed services. *ACM Transactions on Computer Systems*, 10(4):360–391, Nov. 1992.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Sept. 1979.

[17] B. Lampson. Desiging a global name service. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 1–10, Aug. 1986.

[18] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[19] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989.

[20] N. Lynch and F. Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, Sept. 1995.

[21] N. Lynch and F. Vaandrager. Forward and backward simulations — Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[22] Object Management Group, Framingham, MA. *Common Object Request Broker Architecture*, 1992.

[23] B. Oki and B. Liskov. Viewstamp replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, Aug. 1988.

[24] Open Software Foundation, Cambridge, MA. *Introduction to OSF DCE*, 1992.

[25] A. Shvartsman and C. Strutt. Distributed object management and generic applications. Computer Science TR 94-176, Brandeis University, 1994.

[26] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transaction on Software Engineering*, 5(3):188–194, May 1979.

[27] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.