Eventually-Serializable Data Services

Alan Fekete* David Gupta† Victor Luchangco† Nancy Lynch† Alex Shvartsman†

Abstract

We present a new specification for distributed data services that trade-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 operations requested 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. To demonstrate its utility, we present an algorithm, based on one of Ladin, Liskov, Shrira, and Ghemawat [12], that implements this specification. Our algorithm provides the interface of the abstract service, and generalizes their algorithm by allowing general operations and greater flexibility in specifying consistency requirements. We also describe how to use this specification as a building block for applications such as directory services.

1 Introduction

Providing distributed and concurrent access to data objects is a fundamental concern of distributed systems. The simplest implementations 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. To address this problem, many systems replicate the data object, and allow 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.

*Department of Computer Science, University of Sydney 2006, Australia

†Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. This work was supported by ARPA contract F19628-95-C-0118, AFOSR-ONR contract F49620-94-1-0199, and NSF contract 9225124-CCR.

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, 21, 18], majority consensus [22], and quorum consensus [8, 9]. 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 different notions of consistency. *Sequential consistency* [13], guaranteed by systems such as Orca [3], allows operations to be reordered as long as they remain consistent with the view of individual 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 [6, 5, 7] in order to get better performance.

Improving performance by providing weaker 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.

1.1 Background of our Work

As it is important that our specification be applicable for real systems, we build heavily on the work of Ladin, Liskov, Shrira, and Ghemawat [12] 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, so that the new one will not be applied at any replica until after the previous ones have been applied. Other than this, the system may respond with any value that is consistent with an arbitrary subset of previous operations. This allows any causality constraints to be expressed. They also provide two other types of operations, which 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. As long as most of the operations are of the weakest variety, their algorithm is very efficient.

The specification in [12] is tuned for their algorithm, and exposes some of the implementation details to the clients. This makes it difficult to decompose the algorithm into modules with clear benefits and costs that can be easily understood. For example, their specification exposes the client to multipart timestamps, which are used internally to order operations, and it is not clear which properties of their algorithm depend their use of multipart timestamps, and which depend only on the lazy replication strategy, nor how to compare their algorithm with others that do not use multipart timestamps. Also, their algorithm requires that all operations be either read-only *queries* or write-only *updates*, and, to guarantee consistency, that the ordering type of each update be specified by the application programmer rather than the user. Their algorithm requires that for any pair of operations with effects on the state of the data that are not commutative, one must depend on the other. If this is not valid, the algorithm can leave replicas inconsistent forever. That is, the apparent order on operations may not converge to a limiting total order.

1.2 Our Contributions

We present a formal specification for a data service that admits efficient implementations by permitting some transient inconsistencies in the state of the replicas, while providing unambiguous guarantees about system responses to clients' requests, and ensuring the *eventual serialization* of all operations requested. We also present an algorithm that implements the abstract specification, which we prove using invariants and a forward simulation [15]. By making simple assumptions about the timing of message-based communication, we also provide time bounds for the data service.

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: (a) *strict* operations, that 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 [12], 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. Our specification makes no assumptions about the semantics of the data object, and thus can be used as the basis for a wide variety of applications. Particular system implementations, of course, may exploit the semantics of the specific data objects to improve performance.

The algorithm we present is based on the lazy replication algorithm from [12]. 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 amongst 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 [12], 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 to incorporate this, and other optimizations, may be incorporated into the framework of this paper, and we demonstrate this with some examples.

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 implementation of distributed system building blocks layered on general-purpose distributed platforms (middleware) such as DCE [19], and the specification of the middleware components themselves. We provide an example of this, using a distributed directory service.

2 Specification of an Eventually-Serializable Data Service

The memory system manages data whose serial behavior is defined by some data object, and a collection of operators on that object. Formally, a data object is defined by a set Σ of states, with a distinguished initial state σ_0, a set V of reportable values, a set O of operators, and a transition function $f\colon \Sigma \times O \to \Sigma \times V$. We use *state* and *val* selectors to extract the appropriate components. We define $f^+\colon \Sigma \times O^+ \to \Sigma \times V$ by repeated application of f, i.e., $f^+(\sigma, \langle op \rangle) = f(\sigma, op)$ and $f^+(\sigma, \langle op_1, op_2, \ldots \rangle) = f^+(f(\sigma, op_1).state, \langle op_2, \ldots \rangle)$.

In the serial data specification, the resulting state and value for each of a sequence of operations are uniquely determined. To allow more efficient and fault-tolerant distributed implementations, our specification admits reordering of operations. However, it specifies that, in the limit, a total order, the *eventual total order*, be established on all operations. An operation is said to be

State

$wait_f$ for each frontend f: a subset of $\mathcal{O}$, initially empty; the operations requested but not yet responded to

$rept_f$ for each frontend f: a subset of $\mathcal{O} \times V$, initially empty; operations and responses to 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 $\mathcal{O}.id$, 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 $request_c(x)$
Eff: $wait_f \leftarrow wait_f \cup \{x\}$

Internal $enter(x, new\text{-}po)$
Pre: $x \in wait_f$ for some f
$x.prev \subseteq ops.id$
$new\text{-}po$ is a partial order on $\mathcal{O}.id$
$po \subseteq new\text{-}po$
$CSC(\{x\}) \subseteq new\text{-}po$
$\{(y.id, x.id) : y \in stabilized\} \subseteq new\text{-}po$
Eff: $ops \leftarrow ops \cup \{x\}$
$po \leftarrow new\text{-}po$

Internal $add\_constraints(new\text{-}po)$
Pre: $new\text{-}po$ is a partial order on $\mathcal{O}.id$
$po \subseteq new\text{-}po$
Eff: $po \leftarrow new\text{-}po$

Internal $stabilize(x)$
Pre: $x \in ops$
$\forall y \in ops, (y.id, x.id) \in po \vee (x.id, y.id) \in po \vee y = x$
$\forall y \in ops, (y.id, x.id) \in po \implies y \in stabilized$
Eff: $stabilized \leftarrow stabilized \cup \{x\}$

Internal $calculate(x, v)$
Pre: $x \in ops$
$x.strict \implies x \in stabilized$
$v \in valset(x, ops, po)$
Eff: if $x \in wait_f$ then $rept_f \leftarrow rept_f \cup \{(x, v)\}$
where $f = frontend(client(x.id))$.

Output $response_c(x, v)$
Pre: $(x, v) \in rept_f$
$x \in wait_f$
Eff: $wait_f \leftarrow wait_f - \{x\}$
$rept_f \leftarrow rept_f - \{(x, v') : (x, v') \in rept_f\}$

Figure 1: *Spec*: An Eventually-Serializable Data Service

stable when the prefix of the eventual total order up to that operation is known. Clients submit requests with operation descriptors that may restrict the eventual total order and the set of possible return values. An operation descriptor is a record consisting of a data object operator *op*, a unique identifier *id*, a set *prev* of identifiers of operations that must precede the requested operation, and a boolean flag, *strict*, that indicates whether the operation must be stable at the time of the response. The identifier *id* also specifies the client issuing the request; each client is responsible for ensuring that it issues unique identifiers to operations. The clients must also ensure that *prev* sets contain only identifiers of operations that have already been requested by some client.

To formally describe the system and its requirements, we use the I/O automaton model [16]. The system is defined by *Spec* in Figure 1, where $\mathcal{O}$ is the set of all operation descriptors. We denote the set of identifiers of the operations in $X \subseteq \mathcal{O}$ by $X.id$. The inputs are of the form $request_c(x)$, and the outputs are of the form $response_c(x, v)$, where $x \in \mathcal{O}$, $v \in V$, and $c = client(x.id)$ is the name of the client submitting the request. To model the clients, we use the automaton *Users* in Figure 2 to capture the well-formedness assumptions. It represents all clients, and uses shared state to encode the restrictions on the clients as generally and abstractly as possible; in a real implementation, there need not be any shared state.

State

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

$responded$, a subset of $requested$, initially empty

Actions

Output $request_c(x)$
Pre: $c = client(x.id)$
$x.id \notin requested.id$
$x.prev \subseteq requested.id$
Eff: $requested \leftarrow requested \cup \{x\}$

Input $response_c(op, v)$
Eff: $responded \leftarrow responded \cup \{x\}$

Figure 2: *Users*: Well-formed Clients

Each client c has a frontend $frontend(c)$, and there are state variables $wait_f$ and $rept_f$ for each frontend f.[1] The set $wait_f$ contains an operation descriptor for each outstanding request, and the set $rept_f$ contains operations with values that still need to be returned to the client.

Spec maintains a set *ops* of operations that have been *entered*, and a partial order *po* of constraints on the order of the entered operations, which evolves toward the eventual total order. This partial order must be consis-

[1] In this paper, we assume each client has its own frontend, and we equate the two, i.e., $frontend(c) = c$. We maintain the name distinction for clarity and extensibility, and use c for the external interface, and f internally.

tent with the *client-specified constraints* given by the *prev* sets. We denote the client-specified constraints of a set X of operation descriptors by $CSC(X) = \{(y.id, x.id) : x \in X \wedge y.id \in x.prev\}$. *Spec* also maintains a set *stabilized* of operations whose prefix in the eventual total order is fixed by *po*.

For each operation x, there are three internal actions of the form $enter(x, new\text{-}po)$, $stabilize(x)$ and $calculate(x, v)$. The $enter(x, new\text{-}po)$ action adds enough constraints to *po* to ensure that the new operation will follow everything specified by the client, and will maintain the stability of any operations in *stabilized*.[2] The $calculate(x, v)$ action chooses an arbitrary return value consistent with the constraints specified in *po*; it requires strict operations to be in *stabilized*. The $stabilize(x)$ action can occur only if the operation is totally ordered with respect to other operations in *ops*, and all preceding operations are already stabilized. These actions may be done repeatedly for each operation, though the response, of course, is only done once. Repeated *calculate* actions may produce different return values from which the *response* action may select nondeterministically. There is also an internal action $add\_constraints(new\text{-}po)$ which extends the partial order of constraints.

To formally specify the transition relation, we use an auxiliary function *valset*. Given a totally-ordered finite set $(X, to) = (\{x_1, \ldots, x_n\}, \{(x_i.id, x_j.id) : i < j\})$, we define the outcome of an operation $x \in X$ to be $outcome(x_k, X, to) = f^+(\sigma_0, \langle x_1.op, \ldots, x_k.op\rangle)$. If *po* is a partial order on the identifiers of X, we define the set of reportable values $valset(x, X, po)$ so that $outcome(x, X, to).val \in valset(x, X, po)$ iff *to* is a total order consistent with *po*.

Finally, we impose two liveness requirements on the system, that every request should receive a response, and that every operation must stabilize. This ensures that the limit of *po* defines a sequence, that is, a total order in which each operation is preceded by a finite number of operations. This total order must respect the client-specified order, and all requests entered after an operation has stabilized must follow that operation in this order. Thus, if all requests are strict, the data service becomes atomic.

3 An Abstract Algorithm with Replicated Data

In this section, we present an abstract algorithm that implements the eventually-serializable data service specification in the previous section. Again, we assume that each client has a frontend,[3] and that processes communicate using point-to-point channels. For now, we assume the system is entirely reliable (i.e., there are no process or communication faults), though it is easy to modify the algorithm to tolerate processor crashes and message losses. We make no assumptions, however, about the order of delivery. 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.

State

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

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

Actions

Input $request_c(x)$
 Eff: $wait_f \leftarrow wait_f \cup \{x\}$

Output $send_{f,r}(\langle\text{"request"}, x\rangle)$
 Pre: $x \in wait_f$

Input $receive_{r,f}(\langle\text{"response"}, x, v\rangle)$
 Eff: if $x \in wait_f$ then $rept_f \leftarrow rept_f \cup \{(x, v)\}$

Output $response_c(op, v)$
 Pre: $(x, v) \in rept_f$
 $x \in wait_f$
 Eff: $wait_f \leftarrow wait_f - \{x\}$
 $rept_f \leftarrow rept_f - \{(x, v') : (x, v') \in rept_f\}$

Figure 3: Automaton for frontend f

When a client submits a request, its frontend simply relays the request to one or more *replicas* that maintain a copy of the data object, and, when it receives a response, relays that back to the client. The frontend automaton is shown in Figure 3. A channel from process i to process j is modelled trivially with $send_{i,j}$ and $receive_{i,j}$ actions and a single state variable $channel_{i,j}$; the automaton is omitted.

The automaton in Figure 4 for replica r has a number of state components. The component $rcvd_r$ is the set of operation descriptors of all requests that this replica has received, either directly from a frontend, or else through gossip from other replicas. The component $done_r$ is an array of sets of operation descriptors, one for each replica. Each set represents the operations known to be "done" at the corresponding replica, that is, the operations for which the replica can compute a value. The component $solid_r$ is also an array of sets of operation descriptors, again one for each replica. The interpretation of $x \in solid_r[i]$ is that replica r knows that replica i knows that every replica has x in its *done* set.

Replicas assign *labels* uniquely[4] to operations from a well-ordered set $\mathcal{L}$. Each replica keeps a function $minlabel\colon \mathcal{O} \rightarrow \mathcal{L} \cup \{\infty\}$, which encodes the mini-

[2] Further constraints may be specified, but it must place at least this many.

[3] We use c for the external interface and f internally.

[4] Process identifiers can be used to break ties.

State

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

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

$done_r[i]$ for each replica i, a subset of $\mathcal{O}$; the operations r knows that i has "done"

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

$minlabel_r \colon \mathcal{O} \to \mathcal{L} \cup \{\infty\}$; the smallest label r has seen for $x \in \mathcal{O}$

Derived from $done_r[r]$ and $minlabel_r$: $val_r \colon done_r[r] \to V$; the value for $x \in done_r[r]$ using the $minlabel_r$ order

Actions

Input $receive_{f,r}(\langle\text{"request"}, x\rangle)$
  Eff: $pending_r \leftarrow pending_r \cup \{x\}$
  $rcvd_r \leftarrow rcvd_r \cup \{x\}$

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

Output $send_{r,f}(\langle\text{"response"}, x, v\rangle)$
  Pre: $x \in pending_i \cap done_r[r]$
  $x.strict \implies x \in \bigcap_i solid_r[i]$
  $v = val_r(x)$
  $f = frontend(client(x.id))$
  Eff: $pending_r \leftarrow pending_r - \{x\}$

Output $send_{r,r'}(\langle\text{"gossip"}, R, D, L, S\rangle)$
  Pre: $R = rcvd_r$; $D = done_r[r]$;
  $L = minlabel_r$; $S = solid_r[r]$

Input $receive_{r',r}(\langle\text{"gossip"}, R, D, L, S\rangle)$
  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 $i \neq r, r'$
  $minlabel_r = \min(minlabel_r, L)$
  $solid_r[r'] \leftarrow solid_r[r'] \cup S$
  $solid_r[r] \leftarrow solid_r[r] \cup S \cup (\bigcap_i done_r[i])$

Figure 4: Automaton for replica r

mum label that the replica knows has been assigned to an operation (by any replica), where $l < \infty$ for all $l \in \mathcal{L}$. As information is gossiped between replicas, the value of $minlabel_r(x)$ may be reduced when r learns of a lower label for x; however, an invariant shows that once $x \in solid_r[r]$, no further reduction is possible. The function $minlabel_r$ defines a partial order $local\_cons_r$ (on operation identifiers), where $local\_cons_r = \{(y.id, x.id) : minlabel_r(y) < minlabel_r(x)\}$. Because labels are assigned uniquely, $local\_cons_r$ defines a total order on $done_r[r]$. A replica uses this order to compute the value of an operation, $val_r(x) = outcome(x, done_r[r], local\_cons_r).val$ for $x \in done_r[r]$.

Replicas use gossip messages to keep each other informed about operations issued to other replicas, sending around the operations received and processed, as well as the labels associated with each. Hence a gossip message essentially contains the state of a replica at a given point in time, which will be "merged" with the state of the receiving replica. For each message $m = \langle\text{"gossip"}, R, D, L, S\rangle \in channel_{r',r}$, we define the partial order $msg\_cons(m) = \{(y.id, x.id) : \min(minlabel_r(y), L(y)) < \min(minlabel_r(x), L(x))\}$ on operations. As information about the operations is gossiped, the system converges on certain constraints. We denote this by $system\_cons = \bigcap_r local\_cons_r \cap \bigcap_{r,r'} \bigcap_{m \in channel_{r',r}} msg\_cons(m)$.

To show that this system meets the specification, we establish a simulation [15] from the algorithm automaton to the specification. Let $AbsAlg$ be the composition of all the frontend, channel, and replica automata, with the $send$ and $receive$ actions hidden. We have the following theorem:

Theorem 3.1 The relation F in Figure 5 defines a simulation from $AbsAlg \times Users$ to $Spec \times Users$.

To prove this theorem, we first establish several invariants about $AbsAlg$. The following are the key ones:

- $done_r[r] = \bigcup_i done_r[i]$ and $solid_r[r] = \bigcup_i solid_r[i] = \bigcap_i done_r[i]$
- $\bigcup_r rcvd_r - \bigcup_f wait_f \subseteq \bigcup_r done_r[r]$
- $done_i[r] \subseteq done_r[r]$ and $solid_i[r] \subseteq solid_r[r]$.
- $done_r[r] = \{x : minlabel_r(x) < \infty\}$
- If $x \in solid_r[r]$ then $minlabel_r(x) \leq minlabel_i(x)$ for all i. (Corollary: If $x \in \bigcap_r solid_r[r]$ then $minlabel_i(x) = minlabel_j(x)$ for all i, j.)
- $x \in \bigcap_r solid_r[r] \implies val_i(x) = val_j(x)$ for all i, j.
- $\mathrm{TC}(CSC(\bigcup_r done_r[r]) \cup system\_cons)$ is a partial order, where $\mathrm{TC}(R)$ is the transitive closure of R.
- If $x \in done_r[r]$ then $valset(x, \bigcup_i done_i[i], local\_cons_r) = \{val_r(x)\}$.

We define the relation F between states in *AbsAlg* and states in *Spec* such that $u \in F[s]$ if and only if:

- $u.wait_f = s.wait_f$ for all frontends/clients f
- $u.rept_f = s.rept_f \cup s.potential\_rept_f$
  where $potential\_rept_f = \{(x, v) : x \in wait_f \wedge \exists r \langle\text{“response”}, x, v\rangle \in channel_{r,f}\}$.
- $u.ops = \bigcup_r s.done_r[r]$
- $u.po = \mathrm{TC}(CSC(\bigcup_r s.done_r[r]) \cup s.system\_cons)$
- $u.stabilized = \bigcap_r s.solid_r[r]$

Figure 5: Forward Simulation from Algorithm to Specification

The table in Figure 6 gives, for each action of *AbsAlg*, the corresponding action or sequence of actions of *Spec*. Notice that some actions do not have any corresponding action in the specification, and the receipt of a gossip message corresponds possibly to several actions in the specification.

Performance

If we assume there is no congestion and ignore local computation time, then the delay for strict operations is at most $2d_{fr} + 3(d_{rr} + g)$ where d_{fr} is the maximum message delivery delay from frontend to replica, d_{rr} is the maximum message delivery delay between two replicas, and g is the “gossiping interval”, the maximum interval between two gossip messages from one replica to another. Similarly the delay for a nonstrict operation is at most $2d_{fr}+d_{rr}+g$. In the common case where each frontend always communicates with the same replica, and where each client-specified dependency mentions only operations which occured previously at the same frontend (or whose name was communicated through shared data), the delay for nonstrict operations is reduced to at most $2d_{rf}$.

4 Optimizations of the Abstract Algorithm

While the algorithm we present deals with the fundamental problems of maintaining consistency in a distributed, replicated data service, it is still written at a rather 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, that address these issues better.

4.1 Memoizing Stable State

In the *AbsAlg* automaton, since we were not concerned with modelling local computation, the val_r function at each replica is derived by computing all the preceding operations in the $minlabel_r$ 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 type 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 be necessary.

We modify the replica automata to model this more explicitly by augmenting the state with two new state variables, $stable\text{-}state_r$ and $stable\text{-}value_r$. The $stable\text{-}value_r$ function stores the values for all the stable operations, i.e., those in $solid_r[r]$, while $stable\text{-}state_r$ reflects the state of the data after applying all those operations. The return value of later operations can then be computed from $stable\text{-}state_r$, rather than the initial state of the data. Formally, we parameterize the *outcome* function with an initial state.[5] Then $val_r(x)$ is $stable\text{-}value_r(x)$ when $x \in solid_r[r]$, and $outcome_{stable\text{-}state_r}(x, done_r[r] - solid_r[r], local\_cons_r).val$ otherwise. With this new definition of val_r, the only change to the automaton is in processing gossip, where the $stable\text{-}state_r$ and $stable\text{-}value_r$ values have to be computed (see Figure 7).

If $AbsAlg'$ is the composition of these new replica automata and the original frontend a channel automata, then we can prove that *AbsAlg* and $AbsAlg'$ are equivalent. The key lemma is the following invariant:

Lemma 4.1 For all reachable states of $AbsAlg'$, $stable\text{-}state_r = outcome(y, solid_r[r], local\_cons_r).state$, where $y = \max(solid_r[r])$ (by the $minlabel_r$ order), and $stable\text{-}value_r(y) = outcome(y, solid_r[r], local\_cons_r).val$ for all $y \in solid_r[r]$.

[5] That is, $outcome_\sigma(x_k, X, to) = f^+(\sigma, \langle x_1.op, \ldots, x_k.op\rangle)$, where $(X, to) = (\{x_1, \ldots, x_n\}, \{(x_i, x_j) : i < j\})$.

Implementation	Specification
$request_c(x)$	$request_c(x)$
$send_{f,r}(\langle \text{“request”}, x\rangle)$	no-op
$receive_{f,r}(\langle \text{“request”}, x\rangle)$	no-op
$do\_it_r(x, l)$	$enter(x, new\text{-}po)$ if $x \in \cup_f wait_f$
$send_{r,f}(\langle \text{“response”}, x, v\rangle)$	$calculate(x, v)$
$receive_{r,f}(\langle \text{“response”}, x, v\rangle)$	no-op
$response_c(x, v)$	$response_c(x, v)$
$send_{r,r'}(\langle \text{“gossip”}, R, D, L, S\rangle)$	no-op
$receive_{r',r}(\langle \text{“gossip”}, R, D, L, S\rangle)$	$add\_constraints(new\text{-}po), stabilize^*(\cap_i s'.solid_i[i])$

Figure 6: Action Correspondence

Input $receive_{r',r}(\langle \text{“gossip”}, R, D, L, S\rangle)$
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 $i \neq r, r'$
$minlabel_r = \min(minlabel_r, L)$
$solid_r[r'] \leftarrow solid_r[r'] \cup S$
for $y \in \bigcap_i done_r[i] - solid_r[r]$ in $minlabel_r$ order:
$solid_r[r] \leftarrow solid_r[r] \cup \{y\}$
$(stable\text{-}state_r, stable\text{-}value_r(y)) \leftarrow f(stable\text{-}state_r, y.op)$

Figure 7: Computing stable information from gossip

4.2 Reducing Memory Requirements

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

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 the stable return values. This means that once an operation is stable, 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 a stable operation again, it can purge even that from its memory.[6] Thus, while $AbsAlg'$ seems to require more memory than *AbsAlg*, a reasonable implementation of $AbsAlg'$ may in practice be more memory efficient as well.

[6] For example, if communication is perfectly reliable, then once a response is sent to a frontend, 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.

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. The simplest method for this would be a time-based strategy. For example, if the identifiers included the date of request, and all operations are guaranteed to be stable within one day, then all identifiers more than a day old may be expunged from the memory. A more sophisticated approach might involve logical timestamps, such as the multipart timestamps of [12].

4.3 Exploiting Commutativity Assumptions

The algorithm of [12] 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 are determined by the application developer, not the client, based on “permissible concurrency”. This is important because otherwise it may be possible for clients to cause, even inadvertently, the data at different replicas to diverge irreconcilably.

In this section, we describe how to further reduce the need for recomputing operations, when all operations have sufficient “permissible concurrency.” We begin with a careful statement of the conditions under which this optimization can be made.[7]

We say that two operators, op_1 and op_2, (of the data type) *commute* if, for all $\sigma \in \Sigma$:

$$f^+(\sigma, \langle op_1, op_2\rangle).state = f^+(\sigma, \langle op_2, op_1\rangle).state$$
$$f^+(\sigma, \langle op_2, op_1\rangle).val = f(\sigma, op_1).val$$
$$f^+(\sigma, \langle op_1, op_2\rangle).val = f(\sigma, op_2).val$$

[7] This condition is very strong. However, a weaker variation may be sufficient for the algorithm of [12] since *updates* and *queries* are handled differently, and operations may not simultaneously read and write the data.

State

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

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

$done_r[i]$ for each replica i, a subset of $\mathcal{O}$; the operations r knows that i has "done"

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

$minlabel_r \colon \mathcal{O} \to \mathcal{L} \cup \{\infty\}$; the smallest label r has seen for $x \in \mathcal{O}$

$stable\text{-}state_r \in \Sigma$, initially σ_0; the state resulting from doing all the operations in $solid_r[r]$

$stable\text{-}value_r \colon solid_r[r] \to V$, initially empty; the values of the stable operations in the eventual total order

$current\text{-}state_r \in \Sigma$, initially σ_0; the state resulting from doing all the operations in $done_r[r]$

$val_r \colon done_r[r] - solid_r[r] \to V$, initially empty; the value for $x \in done_r[r] - solid_r[r]$

Actions

Input $receive_{f,r}(\langle\text{"request"}, x\rangle)$
  Eff: $pending_r \leftarrow pending_r \cup \{x\}$
    $rcvd_r \leftarrow rcvd_r \cup \{x\}$

Internal $do\_it_r(x, l)$
  Pre: $x \in rcvd_r - done_r[r]$
    $x.prev \subseteq done_r[r].id$
    $l > minlabel_r(y)$ for all $y \in done_r[r]$
  Eff: $done_r[r] \leftarrow done_r[r] \cup \{x\}$
    $(current\text{-}state_r, val_r(x)) \leftarrow f(current\text{-}state_r, x.op)$
    $minlabel_r(x) \leftarrow l$

Output $send_{r,f}(\langle\text{"response"}, x, v\rangle)$
  Pre: $x \in pending_r \cap done_r[r]$
    $x.strict \implies x \in \bigcap_i solid_r[i]$
    $v = \begin{cases} stable\text{-}value_r(x) & \text{if } x \in solid_r[r] \\ val_r(x) & \text{otherwise} \end{cases}$
    $f = frontend(client(x.id))$
  Eff: $pending_r \leftarrow pending_r - \{x\}$

Output $send_{r,r'}(\langle\text{"gossip"}, R, D, L, S\rangle)$
  Pre: $R = rcvd_r$; $D = done_r[r]$;
    $L = minlabel_r$; $S = solid_r[r]$

Input $receive_{r',r}(\langle\text{"gossip"}, R, D, L, S\rangle)$
  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 $y \in \bigcap_i D - done_r[i]$ in any order consistent with $CSC(D)$:
      $done_r[r] \leftarrow done_r[r] \cup \{y\}$
      $(current\text{-}state_r, val_r(y)) \leftarrow f(current\text{-}state_r, y.op)$
    $minlabel_r = \min(minlabel_r, L)$
    $solid_r[r'] \leftarrow solid_r[r'] \cup S$
    for $y \in \bigcap_i done_r[i] - solid_r[r]$ in $minlabel_r$ order:
      $solid_r[r] \leftarrow solid_r[r] \cup \{y\}$
      $(stable\text{-}state_r, stable\text{-}value_r(y)) \leftarrow f(stable\text{-}state_r, y.op)$

Figure 8: Automaton for replica r with current state

We require that clients explicitly order every pair of operations that do not commute; that is, if x is being requested, then for all $y \in requested$, either $x.op$ and $y.op$ commute, or $y.id \in x.prev$. Actually, it is sufficient to have $(y.id, x.id) \in \mathrm{TC}(CSC(requested \cup \{x\}))$. Formally, we define *SafeUsers* as the automaton resulting from adding this clause to the precondition of the $request_c(x)$ action of *Users*.

We now again modify the automaton of replica r, by augmenting that state with two additional state variables, $current\text{-}state_r$ and val_r. The latter is no longer a derived state variable, as in the earlier replica automata, hence, the operations do not need to be recomputed implicitly as before. Instead, val_r is computed as each operation is added to $done_r[r]$, whether by a do_it_r action, or by processing gossip received from another replica, and $current\text{-}state_r$ reflects all the operations in $done_r[r]$. The complete code for this new replica is given in Figure 8.

If *Commute* is the composition of these replica automata, and the original channel and frontend automata, then we want to show that *Commute* implements $AbsAlg'$. Notice that the computation for the responses to stable operations has not changed at all, so we only need to verify that the two computations of val_r are equivalent. This follows because under the commutativity assumption above, the return value for any operation is determined by the client-specified constraints.

4.4 Considerations for Communication Optimizations

In the abstract algorithm, replicas send gossip messages that include information previously gossiped. It is possible to reduce the gossip message sizes by sending only the incremental information. To accomplish this, a sequence number is incorporated in the gossip messages to impose a FIFO discipline on each point-to-point channel. This eliminates the need to send redundant information and allows replicas to only send incremental information.

It is also possible to reduce the overall number of messages sent during each round of gossip when such rounds are scheduled periodically according to some "gossiping interval". Instead of sending a quadratic number of replica-to-replica messages in each round, an intelligent implementation in terms of a broadcast/convergecast protocol that combines gossip messages can reduce the number of messages.

5 Uses of the Eventually-Serializable Data Services

We have stated earlier that an important consideration in our work is that our specification be reasonable for real systems. We are planning a prototype implementation and below we give examples of the uses of the data service specifications that we are considering.

5.1 Naming and Directory Services

Our data service framework is well-suited for specifying and implementing directory services. In a distributed computing enterprise, naming and directory are important and basic services used to make distributed resources accessible transparently to the locations or the physical addresses of users and resources. Such services include Grapevine [5], DECdns [14], DCE GDS (Global Directory Service) and CDS (Cell Directory Service) [19], ISO/OSI X.500 [11], and the Internet's DNS (Domain Name System) [10]. 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.

We have specified a simple distributed directory service (TDS, a Tiny Directory Service) layered on our data service. Fast queries and lazy updates can be implemented in terms of normal (non-strict) operations, the forced update can be implemented as a strict operation. Non-strict updates of course can be reordered by the service and this is consistent with what might happen in practice. Our specification includes periodic gossip operations that, when implemented, will be used to synchronize replicas. This is similar to, e.g., DCE CDS, where time schedules are used to initiate the convergence of replicas.

Directory services often use an object-based definition of names in which a name has a set of attributes determined by the type of the name. When a new name object is created it is necessary to guarantee that the attributes of the created object can be initialized and queried subsequently. In our implementation this is accomplished by including the identifier of the name creation operation in the *prev* sets of the attribute creation and initialization operations.

5.2 Distributed Information Repository

Another application of the data service is in implementing distributed information repositories for coarse-grained distributed object frameworks such as CORBA [17]. Important components of a framework is its distributed type system used to define object types, and its module implementation repository used for dynamic object dispatching [20]. In this setting the access patterns are again dominated by queries, while infrequent update requests can be propagated lazily with the guarantee of eventual consistency. We plan to specify such service using our framework.

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] 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.

[7] 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.

[8] 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.

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

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

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

[12] 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.

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

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

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

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

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

[18] 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.

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

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

[21] 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.

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