Implementing an Eventually-Serializable Data Service as a Distributed System Building Block*

Oleg M. Cheiner† Alex A. Shvartsman‡

1999

Abstract

This work presents an implementation of a distributed system building block that is formally specified as the Eventually-Serializable Data Service (ESDS) [7] proposed by Fekete *et al.* ESDS deals with replicated objects that allow the users of the service to relax consistency requirements in return for improved responsiveness, while providing guarantees of eventual consistency of the replicated data. The ESDS paper [7] includes a formal service specification and an abstract algorithm implementing the service. The algorithm is given in terms of I/O automata of Lynch and Tuttle [15]. An important consideration in formulating ESDS was that it could be employed in building real systems.

The work described here makes the following contributions. We develop an optimized implementation of ESDS and explore its behavior. We combine the implementation with different data types and clients, thus demonstrating the utility of the service as a *building block* suitable for serving as a distributed operating system component. The implementation has been experimentally evaluated on a network of workstations. The results confirm that the designed trade-off between *consistency* and *performance* is present in the implemented service. To make the implementation process less error prone, we develop and use a framework for mapping algorithms formally specified using I/O automata to distributed programs. The framework includes a set of conversion rules and a core set of object common to all target implementations.

1 Introduction

Extant network technology enables the creation of very large distributed platforms. Developing sophisticated distributed applications for such environments still

*Appeared in M. Mavronicolas, M. Merritt, and N. Shavit, editors, Networks in Distributed Computing, volume 45 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 43–72. American Mathematical Society, 1999.

†This author was supported by an NSF Graduate Research Fellowship.

‡This work was supported by the contracts ARPA F19628-95-C-0118, NSF 922124-CCR, ONR-AFOSR F49620-94-1-01997, and by the GTE Laboratories.

presents a challenge, despite the availability of distributed middleware packages, such as DCE [23]. In many cases, middleware provides functions at a lower-level of abstraction than those required by complex applications. Even when higher-level functions are available, they often have informal specifications, thus providing only a modest benefit. In particular, it is often difficult to reason about the properties of distributed applications that are built using middleware, and about the compositional semantics of the resulting applications.

Specification of building blocks for sophisticated distributed systems and development of supporting algorithms is an area of active research. However, even when specifications and algorithms are formally stated, deriving a distributed implementation from a specification is an error-prone process. It is often difficult to forecast the behavior patterns of distributed implementations in realistic deployment scenarios, especially when the performance bottleneck cannot be readily assessed and when the scalability of the underlying platforms becomes a limiting factor.

In this paper we present an implementation of a formally specified building block for a flexible eventually-serializable data service, or ESDS for short, proposed by Fekete *et al.* in [7]. The implementation is derived with the help of a framework that is designed to make the derivation process less error-prone, and to reduce the need to reimplement common functions when working with I/O automata specifications [16]. We also present experimental results of exploring the scalability of the service and its consistency/performance tradeoff.

1.1 Background

Replication is used in distributed systems to improve availability and to increase throughput. The disadvantage of replication is the additional effort required to maintain consistency among replicas when serializing client operations. Several notions of consistency have been defined. The strongest notion is *atomicity*, requiring that replicas emulate a single centralized object. Methods to achieve atomicity include write-all/read-one [3], primary copy [1, 18, 17], majority consensus [19], and quorum consensus [11]. Achieving atomicity often incurs a high cost, while some applications, such as directory services [20, 21], are willing to tolerate some transient inconsistencies. This gives rise to different notions of consistency. *Sequential consistency* [14], guaranteed by systems such as Orca [2], allows operations to be reordered as long as they remain consistent with the view of isolated clients.

Improving performance by providing weaker consistency may lead to more complicated semantics. While in practice, replicated systems are often incompletely or ambiguously specified, it remains important to provide formal consistency guarantees. Ladin, Liskov, Shrira, and Ghemawat [13] define a replicated data service. 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. Building on the work of Ladin *et al.*, Fekete *et al.* [7] developed a flexible eventually-serializable data service.

The definition of ESDS includes a specification of the data service and an abstract algorithm that implements it. ESDS relaxes consistency guarantees provided by serializable data services to improve system efficiency and availability. It also provides provable guarantees of long-term consistency of the data. An important consideration in the design of ESDS has been to make it suitable for employment in building real systems. In this work we implement and explore an optimized version of ESDS.

1.2 Our Contributions

We implement an optimized version of ESDS and explore its behavior in a distributed setting. We combine the implementation of ESDS with different data types and clients, thus demonstrating the suitability of the service as a general building block in a distributed computing environment. The implementation has been experimentally evaluated on a network of workstations. In this setting, the implementation scaled well with the number of processors, which reflects a designed trade-off between consistency and performance.

The ESDS algorithm is specified as a composition of I/O Automata [16]. The I/O automata notation is a declarative description language used to specify state machines. To implement ESDS, we need to convert the abstract algorithm to a design specification for a distributed program. To our knowledge, no general method for converting I/O automata specifications to distributed programs has been proposed before.

To assist us in the implementation process, we formulate and utilize a framework for mapping algorithms specified using I/O automata to distributed programs. The goal of the framework is to streamline the mapping of a specification to a message-passing implementation, thus reducing the number of errors that might be accidentally introduced. The framework also implements several common functions, thus eliminating the need to reimplement such functions when working with I/O automata specifications. We believe that the techniques in the framework are general and that they can be used to implement many other algorithms specified as I/O automata.

One of our main design goals was to confirm that the ESDS algorithm is suitable for implementation as a building block from which concrete applications can be built with minimal effort. The algorithm is specified to be independent of the serial data type of the replicated data object to facilitate its use as a building block. Our implementation of ESDS uses object-oriented techniques to ensure that this independence is preserved. We build four distinct applications on top of ESDS to demonstrate the viability of our design as a generic building block for real distributed data services.

The ESDS service has been developed and tested on a network of Sun workstations running SunOS 4.1.4. Four clients for sample ESDS service applications were developed. One client was developed for Win32 and tested under Windows 95 on an Intel Pentium machine. Three other clients ran under SunOS 4.1.4. We use MPI (Message Passing Interface) Standard [6] to implement communication between distributed components of the ESDS implementation. In selecting MPI, we took into account its suitability for implementing I/O automata, the simplicity of communication semantics, the availability of development tools, and portability.

We have instrumented an optimized implementation of ESDS with tools for monitoring interesting parts of the state of the data service and collecting information about performance characteristics of the system. Characteristics of interest include response time to user requests, system throughput, and deviation from strict consistency in system responses.

The empirical tests have provided data on the behavior of the implementation with varying number of participating replicas and with varying system load. In addition, the tests have confirmed that ESDS offers a tradeoff between consistency and performance, and that it is possible to shift the tradeoff balance in either direction according to the user's needs.

The rest of this paper is organized as follows. In Section 2 we overview the I/O automata model and introduce names for variants of the ESDS algorithm and its implementations that we use thereafter. Section 3 describes the framework for converting I/O automata to distributed implementations. Section 4 describes the ESDS algorithm. The ESDS implementation is dealt with in Section 5. Section 6 discusses the empirical results. Our conclusions are presented in Section 7, while Section 8 contains suggestions for future work.

2 Models and Definitions

The *Input/Output automaton*, or I/O automaton for short [15, 16] is a general model used for formal descriptions of asynchronous and distributed algorithms. The model provides a precise way of describing and reasoning about asynchronous interacting components. We provide a consise description of the model and refer the reader to [15, 16] for more details.

An I/O automaton is a *state machine* in which the *transitions* are associated with named *actions*. The actions are classified as either *input*, *output* or *internal*. Roughly speaking, the input and output actions are used for interaction with the automaton's environment. The internal actions work on the automaton's local state.

I/O automata code is given in a *precondition-effect* style. For each action, the code specifies the *preconditions* under which the action is permitted to occur, as a predicate on the automaton state, and the *effects* that occur as the result. The input actions are always enabled (i.e. the precondition clause of an input action is always *true*). The code in the effects clause gets executed atomically.

The *composition* operation allows an automaton representing a complex system to be constructed by composing automata representing individual system components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving an action π, so do all component automata that include the action π. The states and start states of the composition automaton are vectors of states and start states, respectively, of the component automata.

When we compose a collection of automata, output actions of the components become output actions of the composition, internal actions of the components become internal actions of the composition, and actions that are inputs to some components but outputs of none become input actions of the composition.

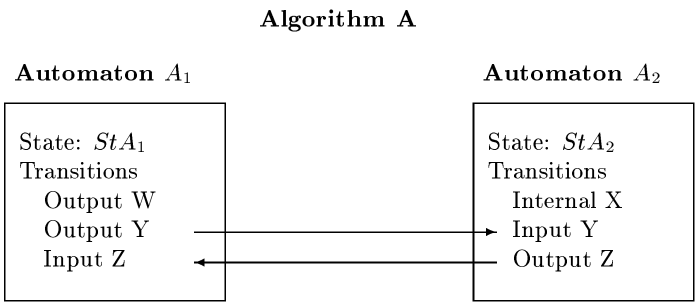

Figure 1: I/O automata composition and input/output combinations

We now name and briefly describe the different variations of the ESDS algorithm and its implementations; we also introduce additional terminology that will be used in describing interactions of automata in a composition.

The component automata in a composition communicate by means of input and output actions with the same name. We distinguish between two types of actions in an I/O automata composition.

Let an I/O automaton A be a composition of I/O automata A_1, A_2, ..., A_m. If there is an output action $X \in A$ that occurs as an output action in some A_i and as an input action in some other automaton A_j $(i \neq j)$, we call X an *input/output combination*, or *I/O combination* for short. We call A_i the *output end* with respect to X, and we call A_j the *input end* with respect to X. Any action $Y \in A$ that appears in one and only one automaton A_k is called a *regular* action.

Figure 1 gives an example of an automaton A composed of two component automata, A_1 and A_2. In the composition, W and X are regular actions, while Y and Z are I/O combinations.

I/O automata are typically used to specify distributed algorithms involving a collection of nodes by encapsulating the behavior of each node I as a separate automaton A_i. The entire algorithm is then represented by the composition A of component automata A_1, A_2, ..., A_m. The internal actions of A_i represent local processing at the corresponding node. The I/O combinations represent communication between the nodes. The input and output actions of each automaton that do not participate in an I/O combination represent the interaction of the corresponding node with its external environment. We assume this representation of distributed algorithms as I/O automata when discussing conversion of such algorithms to distributed programs in Section 3.

In the rest of the paper, we discuss several variants of the ESDS algorithm ans its implementations:

ESDSAlg: this refers to the unoptimized abstract algorithm for ESDS (described in Section 4).

SimpleESDSAlg: this is a simplified version of *ESDSAlg* that replaces channel automata with I/O combinations. This version restricts concurrency, but leads to a simpler implementation.

ESDSOptAlg: this is an optimized version of *ESDSAlg*. The optimizations pursued in *ESDSOptAlg* and a corresponding I/O automata description of them are presented in Section 4.2.

ESDSImpl, SimpleESDSImpl, and ESDSOptImpl are distributed programs that implement *ESDSAlg*, *SimpleESDSAlg*, and *ESDSOptAlg* respectively.

3 Converting I/O Automata to Programs

I/O automata have been effectively used for describing message-passing distributed algorithms and in proving their correctness properties. We develop a framework for converting commonly occurring algorithms that are specified using I/O automata compositions into distributed message-passing implementations using an imperative language (chosen in this work to be C++).

We call the I/O automata composition being converted the *source composition*. We call the algorithm represented by the source composition the *source algorithm*. The result of the conversion is the *target program*.

A key goal for our framework is to facilitate a faithful implementation of the source algorithm. Also, one must be able to directly relate each part of the target program to the corresponding parts of the source algorithm. Due to these requirements, the techniques discussed here are rather conservative, and they will usually lead to an overspecification of the algorithm specified using I/O automata. Nevertheless, this would not preclude one from representing a large and interesting subset of behaviors of the source algorithm in the target implementation.

The conversion rules presented in this sections are intended to help the programmer with the implementation task and to eliminate redundant software development work. Formalizing the rules is a task for future work.

3.1 Overall Approach

In the target program produced from the source composition A, each of A's component automata A_i is represented by a sequential process P_i[1]. Each action of the source composition has a corresponding fragment of code in the target program that implements the action. The conversion techniques ensure that these code fragments are atomic with respect to each other.

When dealing with a composition A of I/O automata, we assume that A consists of component automata $A_1, \ldots, A_m$. We further assume that an action named X belonging to automaton A_i has preconditions clause PXA_i and effects clause EXA_i. A component automaton A_i from composition A will correspond one-to-one to an implementation process P_i. The local state of a component automaton A_i is represented by the local state variables of the corresponding process P_i. We do not make provisions for representing the global state of A[2]. If A utilizes global state, it

[1]It is also possible to combine several automata to run as a single process if there is a reason to do so. For example, the algorithm may be split into automata to make the description modular. In this case, the component automata may not be meant to run in parallel.

[2]By global state of A, we mean state that is accessible to more than one component automaton.

Automaton A_i

> **Output** X
> Precondition: PXA_i
> Effects: EXA_i

Process P_i

> **if** Enabled(PXA_i)
> Send(P_j, "Initiate X");
> Transition(EXA_i);
> Receive(P_j, "Done X");
> **endif**

Automaton A_j

> **Input** X
> Precondition: None
> Effects: EXA_j

Process P_j

> **if** NBReceive(P_i, "Initiate X")
> Transition(EXA_j);
> Send(P_i, "Done X");
> **endif**

Figure 2: Converting an Input/Output Combination to Code

may not be easily implementable as a message-passing distributed program. Global state must be removed from such algorithms if one wishes to apply these techniques to them.

3.2 Converting Preconditions Clauses to Procedures

The purpose of the preconditions clause PXA_i in an action X is to determine whether the state transition EXA_i is enabled in the current automaton state. The preconditions clause is converted to a predicate procedure *Enabled* that checks the current state of the automaton and returns *true* if the action is enabled and *false* otherwise.

3.3 Converting Effects Clauses to Procedures

The effects clause EXA_i describes the state transition represented by action X. The effects clause is converted to a procedure named *Transition*. *Transition* requires X to be enabled and the desired state transition to be chosen among all enabled transitions represented by X. *Transition*'s effect on the state of P_i must correspond to the effects of EXA_i on the state of A_i.

3.4 Converting Regular Actions to Procedures

Conversion of a regular action or an input action to code is straightforward. All that needs to be developed are the *Enabled* and *Transition* procedures that implement the preconditions and effects clauses of the action, respectively (for an input action, *Enabled* is the constant *true*). The implementation of *Transition* needs to be atomic to preserve I/O automata semantics.

3.5 Converting Input/Output Combination Actions to Code

Implementation of an I/O combination is trickier because it must rely on asynchronous messages to implement the combination atomically. We give a technique for implementing an I/O combination in the special case when only two automata participate in the combination. This is sufficient for most existing algorithms that have been specified as I/O automata.

The rule for converting an I/O combination to code is illustrated in Figure 2. Here, automata A_i and A_j correspond to processes (or nodes) P_i and P_j. The Send() and Receive() calls in the pseudocode for processes P_i and P_j stand for sending and receiving asynchronous messages using MPI. The NBReceive() in the process P_j is a non-blocking receive of a message. If a message of the type *"Initiate X"* has not arrived at P_j, then the **if** block is skipped.

An I/O combination is always initiated at the process that represents the output end of the combination (P_i in Figure 2). When the call to *Enabled(PXA$_i$)* returns *true*, P_i sends a message to P_j initiating the combination. Any argument that X has is passed to P_j in the same message. Next P_i performs the local state transition associated with X by invoking the *Transition(EXA$_i$)* procedure. Then, P_i waits for an acknowledgment message *"Done X"* from P_j. This step "synchronizes" the execution of X at the two participating processes.

At the input end of the I/O combination, P_j watches for requests from P_i to initiate X. While the NBReceive call returns *false*, P_j can continue executing other actions. When P_j receives an *"Initiate X"* message, it executes its local state transition for X and then sends the acknowledgment message to P_i.

The *Transition* procedures representing the effects regular actions are atomic with respect to other actions, since they may affect local variables only and may not communicate with other processes. Informally, it is easy to see that the code implementing an input/output combination, as presented in Figure 2, is also atomic. Atomicity would be violated only if it were possible for some process to observe that process P_i had executed *Transition(EXA$_i$)* but process P_j had not executed *Transition(EXA$_j$)*, or vice versa. The synchronizing message *"Done X"* rules out both possibilities.

In our ESDS implementation it is sufficient to consider I/O combinations that involve actions of two automata. In the future we plan to extend the the mechanism of negotiation presented here to the more general case when multiple automata participate in the I/O combination at the input end. The approach may be as follows. The initiating process broadcasts *"Initiate X"* messages to participating automata, gathers *"Done X"* messages from all of them, and then broadcasts a *"Proceed X"* message to indicate that all of the effects clauses in the combination have been executed. Upon receiveing the *"Proceed X"* message, the participants continue with other actions.

3.5.1 Deadlock Avoidance

As presented, the translation from I/O automata to programs is safe, but suffers from deadlock. If two automata running concurrently enter the output part of two different input/output combinations and simultaneously attempt to initiate a combination with each other, it is possible for them to block at the Receive(A_i, *"Done X"*) and Receive(A_j, *"Done X"*) lines, repectively, and wait for each other indefinitely.

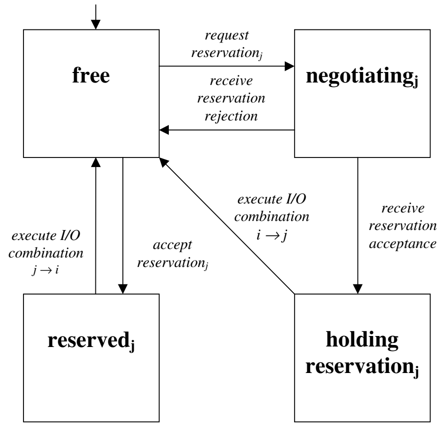

Figure 3: Reservation Status Finite State Automaton for Process P_i

We resolve this deadlock problem by setting up a *reservation system* for performing input/output combinations. Each process P_i maintains its *reservation status* in a state variable. The states of the reservation status at P_i are as follows:

free This is the initial state of reservation status. In this state P_i is free to initiate or accept reservation requests.

reserved$_j$, $j \neq i$ In the *reserved*$_j$ state P_i is waiting for process P_j to initiate an I/O combination.

holding-reservation$_j$, $j \neq i$ In the *holding-reservation*$_j$ state P_i may initiate an I/O combination with P_j.

negotiating$_j$, $j \neq i$ In the *negotiating*$_j$ state P_i is waiting for P_j to respond to a reservation request.

The complete finite state automaton for the reservation status of one process is depicted in Figure 3.

The reservation system imposes restrictions on processes' ability to engage in input/output combinations. Process P_i can initiate an input/output combination with process P_j only after P_i has obtained a reservation with P_j. In order to obtain the reservation, P_i must set its reservation status to *holding-reservation*$_j$, and P_j must set its reservation status to *reserved*$_j$. Figure 3 specifies the rules for requesting and granting reservations.

When a process P_i wants to initiate an input/output combination with process P_j, its first step is to send a message to the receiving process P_j requesting a reservation. P_i is allowed to do this only when its reservation status is *free*. After the request for a reservation is sent to P_j, P_i enters the *negotiating*$_j$ state and waits for a response to the request. If the reservation were granted by P_j, P_i enters the *holding-reservation*$_j$ state, while P_j enters the *reserved*$_j$ state. In this case, P_i is free to initiate an input/output combination as described in Section 3.5. If the reservation request were rejected, P_i "bounces back" to the *free* state.

State

$channel_{i,j}$, a multiset of messages

Actions

Input $send_{i,j}(m)$
Eff: $channel_{i,j} \leftarrow channel_{i,j} \cup \{m\}$

Output $receive_{i,j}(m)$
Pre: $m \in channel_{i,j}$
Eff: $channel_{i,j} \leftarrow channel_{i,j} - \{m\}$

Figure 4: Automaton for channel from process i to process j

Whenever the reservation status of a process is *free* and there is an incoming reservation request, the process may grant the request. Although we do not require that the process grant the reservation every time, it is necessary to grant them for the system to make progress. Furthermore, reservations should be granted fairly to prevent starvation.

We informally argue that, under the reservation system, deadlock cannot occur. The key is Invariant 3.1, which specifies the relationship between the reservation status values of two processes.

Invariant 3.1 *Let $\mathcal{P}$ be the set of process identifiers in the target program. Then for all i, j in $\mathcal{P}$ s.t. $i \neq j$, $P_i = holding\text{-}reservation_j$ implies $P_j = reserved_i$.*

We provide a sketch of the proof of Invariant 3.1. Process P_i must receive a reservation acceptance message from P_j before it can enter the *holding-reservation*$_j$ state. P_j must enter the *reserved*$_i$ state to send a reservation acceptance message to P_i. P_j remains in *reserved*$_i$ until it executes an I/O combination with P_i. When executing this combination, P_i must leave the *holding-reservation*$_j$ state. It follows that while P_i is in the *holding-reservation*$_j$ state, P_j must be in *reserved*$_i$ state.

By Invariant 3.1, process P_j cannot initiate an I/O combination when process P_i is in the *holding-reservation*$_j$ state. So when P_i initiates an I/O combination with P_j, P_j is not blocked but is able to participate. Therefore, the I/O combination executes successfully.

3.5.2 Optimizing Abstract I/O Channels Away

The reservation system for avoiding deadlock can be costly because it reduces the potential for concurrency. For algorithms specified using I/O automata that use channels with asynchronous message delivery for communication between its distributed components, an implementation that can provide more concurrency is possible. The I/O automata specification must obey the following restrictions. For any pair of component automata A_i and A_j that communicate with each other, the specification must have explicit asynchronous half-duplex channel automata $C_{i,j}$ and $C_{j,i}$ between them. $C_{i,j}$ is used for sending messages from A_i to A_j, and $C_{j,i}$ for sending messages from A_j to A_i.

A channel from automaton A_i to automaton A_j is modeled with $send_{i,j}$ and $receive_{i,j}$ actions and a single state variable $channel_{i,j}$ representing messages in transit. The channel has a simple specification (cf. [15]), which is given in Figure 4.

The $send_{i,j}$ and $receive_{i,j}$ actions form I/O combinations with the automata that wish to communicate via the channel. For instance, $C_{i,j}$ can share a $send_{i,j}$ I/O combination with A_i and a $receive_{i,j}$ I/O combination with A_j. In an execution, a $send_{i,j}$ event and a subsequent $receive_{i,j}$ event model asynchronous messages from A_i to A_j.

The message passing model used by MPI already contains the asynchronous channel discipline. An implementation of an algorithm that is specified using channels can use asynchronous *MPI-Send* at A_i and *MPI-Receive* at A_j instead of implementing explicit channel automata $C_{i,j}$. This optimization omits a significant portion of the code generated by the basic conversion framework. Specifically, it removes two I/O combinations (one at each of the sending and the receiving ends of the channel $C_{i,j}$) and a separate process for the channel automaton. Thus, removal of the I/O combinations provides more concurrency.

3.5.3 Abstract Algorithm Relaxation Through Introduction of I/O Channels

When appropriate, the algorithm designer can take advantage of the optimization in Section 3.5.2 by relaxing the abstract algorithm. Whenever algorithm correctness does not require the synchrony imposed on the system by an I/O combination, the combination can be replaced with an explicit asynchronous channel. The channel is then optimized away during conversion of the algorithm to a distributed program. This approach can lead to a significant performance improvement in the target program due to added concurrency.

3.6 Object-Oriented Implementation of the I/O Automata Framework

We developed a set of C++ objects that encapsulate the functions of the framework common to all I/O automata. These objects have been designed in accordance with the conversion techniques we defined, and are intended to be used as a foundation in converting specific algorithms to programs. The implementation takes advantage of polymorphism to provide generic service to any converted I/O automata algorithm; it includes an integral scheduling mechanism using either random or round-robin discipline, and it incorporates an exponential backoff scheme for avoiding livelock in the reservation system. The implementation details are given in [5]. We mention the two most important components here.

IOAutomaton class. This class encapsulates components needed in all implementations of I/O Automata. It handles scheduling of locally-controlled actions for execution (subject to them being enabled) and the reservation system for I/O combinations. *IOAutomaton* is the superclass of all classes that represent I/O automata.

IOAction class. *IOAction* is the superclass of all classes that represent locally-controlled actions. These classes must implement the *Enabled* and *Transition* methods, used by the *IOAutomaton* class to schedule and execute actions.

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 5: *ESDSAlg*: Automaton for front end f

4 The ESDS Algorithm

For reference, we provide a description of *ESDSAlg* component automata. The description follows that of Fekete *et al.* [7]. In the algorithm, each process that maintains a copy of the data objects is modeled as a *replica*. The replicas maintain an *order* on the operations they know about. A known prefix of the order at each replica is such that it is consistent with the eventual total order on operations. Replicas periodically exchange their knowledge in *gossip* messages. Operations can be *strict*, which means that the response to such operations must be consistent with the eventual order, or *non-strict*, which means that responses may not reflect the eventual order. Clients may also specify the *prev* set of operations that must be executed before the new operation.

4.1 The Frontends and Channels

Clients are represented in the algorithm by *frontends*. The frontend automaton is shown in Figure 5. When a client c submits a request (via the $request_c$ input action), 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 via the $response_c$ action. Frontends use state variables $wait_f$ and $rept_f$ for this purpose.

Channels (of the type specified in Figure 4) are used for request/response messages between frontends and replicas and for "gossip" between replicas.

4.2 The Replicas

The replica automaton is given in Figure 6. The automaton is specified for replica r, and it has a number of state components. The component $rcvd_r$ is the set of

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"
$minlabel_r : \mathcal{O} \to \mathcal{L} \cup \{\infty\}$; the smallest label r has seen for $x \in \mathcal{O}$
Derived from $done_r[r]$ and $minlabel_r$: $val_r : done_r[r] \to V$; 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: $x \in rcvd_r$
$x \notin done_r[r]$
$x.prev \subseteq done_r[r].id$
$l > minlabel_r(y)$ for all $y \in done_r[r]$
($l \in \mathcal{L}$, equivalently $l \neq \infty$)
Eff: $done_r[r] \leftarrow done_r[r] \cup \{x\}$
$minlabel_r(x) \leftarrow l$
$solid_r[r] \leftarrow solid_r[r] \cup \bigcap_i done_r[i]$

Output $send_{r,f}(\langle\text{"response"}, x, v\rangle)$
Pre: $x \in pending_r$
$x \in done_r[r]$
$x.strict \Rightarrow x \in \bigcap_i solid_r[i]$
$v = val_r(x)$
$f = frontend(client(x.id))$
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 $i \neq r, r'$
$minlabel_r \leftarrow \min(minlabel_r, L)$
$solid_r[r'] \leftarrow solid_r[r'] \cup S$
$solid_r[r] \leftarrow solid_r[r] \cup S \cup (\bigcap_i done_r[i])$

Figure 6: *ESDSAlg*: Automaton for replica r

operation descriptors of all requests that this replica has received, either directly from a frontend, or else through gossip from other replicas. The component $pending_r$ is the set of operation descriptors of all requests to which a reply by the replica is pending. The component $done_r$ is an array of sets of operation descriptors, one for each replica. Each set represents the operations known to have been "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[3] to operations from a well-ordered set $\mathcal{L}$. Each replica keeps a function $minlabel_r : \mathcal{O} \to \mathcal{L} \cup \{\infty\}$ that encodes the minimum 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, [7] gives an invariant stating 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 $\mathcal{O}.id$), where $local\_cons_r = \{(y.id, x.id) : minlabel_r(y) < minlabel_r(x)\}$.

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

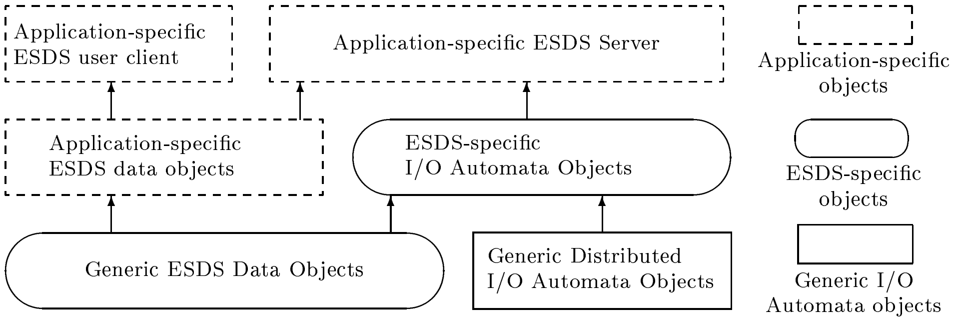

Figure 7: *ESDSImpl* Structure

Because labels are assigned uniquely, $local\_cons_r$ defines a total order for $done_r[r]$. A replica uses this order to compute the value of an operation, $val_r(x) = val(x, done_r[r], local\_cons_r)$ for $x \in done_r[r]$.

Replicas use gossip messages to keep each other informed about the operations issued to other replicas, about the operations received and processed, and the labels associated with each operation. 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 a complete treatment of the algorithm the reader is referred to [7].

5 Implementation of Experimental ESDS Systems

Using the framework and the classes implementing the I/O automata foundation, we have created distributed implementations of two versions of the unoptimized abstract ESDS algorithm, *ESDSAlg* and *SimpleESDSAlg*. We have also created and implemented a more practical optimized version *ESDSOptAlg* of the abstract algorithm.

Our implementation of the algorithm is independent of the data object that implements the serial datatype. A designer can use it as a building block for any type of data service. It is only necessary to implement the data object and add to it the few features needed to make it work with ESDS; no modification to ESDS itself is required. We have implemented several such applications. They are described in Section 5.5.

Figure 7 depicts the hierarchy of the objects that comprise the system. Arrows in Figure 7 represent the relationship "is used by." The objects are divided into three groups. The generic I/O automata objects are the base *IOAutomaton* and *IOAction* classes. They encapsulate functions shared by all I/O automata. The ESDS-specific objects implement *ESDSAlg* (see section 4.2 in [5] for details). These objects are independent of the particular data service application and do not require modification when one wishes to implement a new data service. Finally, the application-specific objects implement a particular data service. Application-specific objects have to be written for each such service.

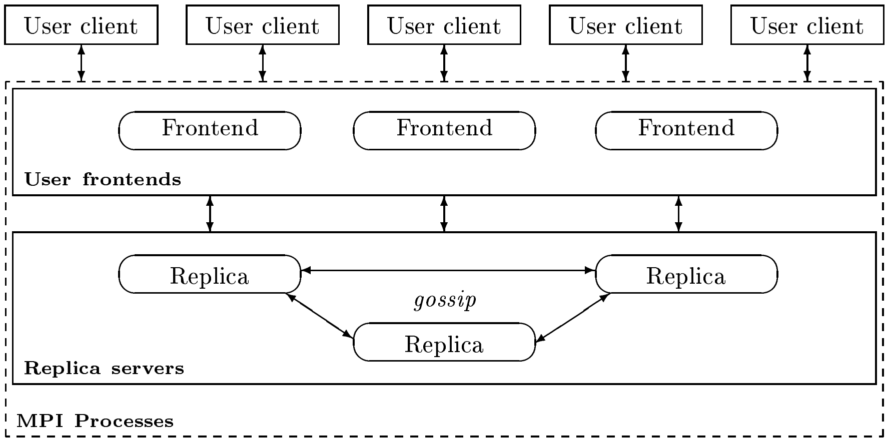

Figure 8: *ESDSImpl* Processes

5.1 Mapping Component Automata to MPI Nodes

Version 1.1 of the MPI standard does not allow dynamic management of nodes. The number of available processes is determined statically at invocation and cannot change during execution. Thus, the application clients in our data service implementation could not be integrated in the MPI framework because they need to be created and destroyed dynamically. We used sockets instead of MPI mechanisms for communication between application clients and ESDS frontends.

The mapping of ESDS components to system processes is depicted in Figure 8. In the figure ESDS replicas and frontends run inside the MPI environment, and the application clients connect to the system from outside the MPI environment. At the invocation of the program the ESDS system administrator specifies the number of MPI nodes that will participate in the execution. Three MPI nodes are reserved for system use (they are not depicted in Figure 8). The rest are divided between ESDS replicas and frontends. The administrator specifies how many nodes to allocate for each purpose.

After the invocation the number of replicas and frontends remains static throughout the execution. Replicas use MPI messages to receive requests from frontends, send gossip message to each other, and send responses back to the frontends. Client processes are dynamically created and destroyed by users. Clients use sockets to connect to one of the frontends. Once the connection is established, the client submits an operation to the frontend and waits for the response.

5.2 Communication Between Clients, FrontEnds, and Replicas

As we have already stated, *ESDSAlg* uses asynchronous channel automata for communication between replicas and frontends and for gossip among replicas. We implemented two different systems of communication among frontends and replicas.

The first version (*SimpleESDSImpl*) was produced using our scheme for converting I/O combinations to distributed programs (see Section 3.5).

The second version (*ESDSImpl*) takes advantage of the fact that *ESDSAlg* relies on asynchronous channels for communication among frontends and replicas. It uses reliable FIFO channels implemented by MPI, as discussed in Section 3.5.3. *ESDSImpl* is a more efficient implementation of ESDS than *SimpleESDSImpl* because it avoids the overhead of synchronizing communications among frontends and replicas.

Integration of this approach into *ESDSImpl* implementation is straightforward. Instead of negotiating with the receiving automaton for synchronized execution, each automaton is free to send an asynchronous request, gossip, or response message and continue executing normally. The pending messages accumulate in the MPI subsystem, which implements reliable FIFO channels and thereby relieves the programmer of that responsibility. In this implementation, the system takes advantage of the distributed nature of the application.

Different methods of interprocess communication constitute the only difference between *ESDSImpl* and *SimpleESDSImpl*. For convenience, both implementations are combined into a single program. The desired method of communication can be set with a switch in the program's configuration file.

5.3 Optimizing ESDS: ESDSOptImpl

In addition to implementing *ESDSAlg*, we implemented some of the optimizations suggested in [7]. In this section we describe the optimizations. We also present an I/O automaton for the optimized ESDS replica.

We make several algorithmic contributions. The optimized algorithm *ESDSOptAlg* contains a new gossiping scheme. It also caches current stable state at each replica. These modifications reduce unnecessary exchange of information between replicas and the amount of work needed to compute values for new requests. These optimizations make the ESDS algorithm more practical.

The complete *ESDSOptAlg* replica automaton is shown in Figure 9.

5.3.1 Incremental Gossip

ESDSAlg is composed of identical replicas. Each replica r periodically sends information about all operations it has seen to other replicas in gossip messages (Fig. 6). Thus, a typical gossip message contains a lot of information that has been gossiped previously between the same two replicas. Furthermore, the amount of redundant gossip increases linearly with the number of new operations. *ESDSImpl*, as a faithful implementation of *ESDSAlg*, requires gossip messages of unbounded size, and thus cannot be used continuously for long time periods without exhausting system resources or leading to unacceptable deterioration of system performance.

If we assume that replicas do not fail and that replicas communicate via reliable FIFO channels (as is the case with *ESDSImpl*), we can modify the replica automaton to send only the incremental gossip updates. Each replica keeps track of changes in its state and gossips only new information. This change improves system performance, but reduces the system's ability to tolerate lost gossip messages.

Data types

$P = \{1, \ldots, n\}$, the set of replica IDs

State

$\mathit{pending}_r$, a subset of $\mathcal{O}$; the messages which require a response
rcvd_r, a subset of $\mathcal{O}$; all operations that have been received
$\mathit{done}_r[i]$ for each replica i, a subset of $\mathcal{O}$; the operations r knows that i has "done"
$\mathit{solid}_r[i]$ for each replica i, a subset of $\mathcal{O}$; the operations that r knows are "stable at i"
$\mathit{gossip}_r[i]$ for each replica i, a subset of $\mathcal{O}$; the operations that r needs to gossip to i
$\mathit{minlabel}_r\colon \mathcal{O} \to \mathcal{L} \cup \{\infty\}$; the smallest label r has seen for $x \in \mathcal{O}$
Derived from $\mathit{solid}_r[r]$ and $\mathit{minlabel}_r$: $\mathit{max\text{-}stable}_r \in \mathit{solid}_r[r]$ s.t. $\forall y \in \mathit{solid}_r[r]$, $\mathit{minlabel}_r(\mathit{max\text{-}stable}_r) \geq \mathit{minlabel}_r(y)$
$\mathit{stable\text{-}state}_r \in \Sigma$, initially σ_0; the state resulting from doing all the operations up to and including $\mathit{max\text{-}stable}_r$
$\mathit{stable\text{-}value}_r\colon \mathit{solid}_r[r] \to V$, initially empty; the values of the stable operations in the eventual total order
Derived from $\mathit{done}_r[r]$ and $\mathit{minlabel}_r$: $\mathit{val}_r\colon \mathit{done}_r[r] \to V$; the value for $x \in \mathit{done}_r[r]$ using the $\mathit{minlabel}_r$ order

Actions

Input $\mathit{receive}_{f,r}(\langle\text{"request"}, x\rangle)$
Eff: $\mathit{pending}_r \leftarrow \mathit{pending}_r \cup \{x\}$
$\mathit{rcvd}_r \leftarrow \mathit{rcvd}_r \cup \{x\}$
for all i :
$\mathit{gossip}_r[i] \leftarrow \mathit{gossip}_r[i] \cup \{x\}$

Internal $\mathit{do\_it}_r(x, l)$
Pre: $x \in \mathit{rcvd}_r - \mathit{done}_r[r]$
$x.\mathit{prev} \subseteq \mathit{done}_r[r].\mathit{id}$
for all $y \in \mathit{done}_r[r]$:
$l > \mathit{minlabel}_r(y)$
Eff: $\mathit{done}_r[r] \leftarrow \mathit{done}_r[r] \cup \{x\}$
$\mathit{minlabel}_r(x) \leftarrow l$
for all i :
$\mathit{gossip}_r[i] \leftarrow \mathit{gossip}_r[i] \cup \{x\}$

Output $\mathit{send}_{r,f}(\langle\text{"response"}, x, v\rangle)$
Pre: $x \in \mathit{pending}_i \cap \mathit{done}_r[r]$
$x.\mathit{strict} \Rightarrow x \in \bigcap_i \mathit{solid}_r[i]$
$v = \begin{cases} \mathit{stable\text{-}value}_r(x) & \text{if } x \in \mathit{solid}_r[r] \\ \mathit{val}_r(x) & \text{otherwise} \end{cases}$
$f = \mathit{frontend}(\mathit{client}(x.\mathit{id}))$
Eff: $\mathit{pending}_r \leftarrow \mathit{pending}_r - \{x\}$

Output $\mathit{send}_{r,r'}(\langle\text{"gossip"}, R, D, L, S\rangle)$
Pre: $R = \mathit{rcvd}_r \cap \mathit{gossip}_r[r]$;
$D = \mathit{done}_r[r] \cap \mathit{gossip}_r[r]$;
$S = \mathit{solid}_r[r] \cap \mathit{gossip}_r[r]$;
$L = \mathit{minlabel}_r$; $r \neq r'$
Eff: $\mathit{gossip}_r[r] \leftarrow \{\}$

Internal $\mathit{solidify}_r$
Pre: $|P| = 1$
Eff: $\mathit{solid}_r[r] \leftarrow \mathit{solid}_r[r] \cup (\bigcap_i \mathit{done}_r[i])$
for y s.t. $\mathit{minlabel}_r(y)$
$\leq \mathit{minlabel}_r(\mathit{max\text{-}stable}_r)$
and $\mathit{stable\text{-}value}_r(y)$ is undefined,
in $\mathit{minlabel}_r$ order:
$(\mathit{stable\text{-}state}_r, \mathit{stable\text{-}value}_r(y)) \leftarrow$
$f(\mathit{stable\text{-}state}_r, y.\mathit{op})$

Input $\mathit{receive}_{r',r}(\langle\text{"gossip"}, R, D, L, S\rangle)$
Eff: for all i : $\mathit{gossip}_r[i] \leftarrow \mathit{gossip}_r[i]$
$\cup(R - \mathit{rcvd}_r)\cup$
$\cup(S - (\bigcap_j \mathit{done}_r[j]))\cup$
$\cup(S - (\mathit{solid}_r[r] \cap \mathit{solid}_r[r]))\cup$
$\cup(D - (\mathit{done}_r[r] \cap \mathit{done}_r[r]))\cup$
$\cup\{x : \mathit{minlabel}_r(x) > L(x)\}$
$\mathit{rcvd}_r \leftarrow \mathit{rcvd}_r \cup R$
$\mathit{done}_r[r'] \leftarrow \mathit{done}_r[r'] \cup D \cup S$
$\mathit{done}_r[r] \leftarrow \mathit{done}_r[r] \cup D \cup S$
$\mathit{done}_r[i] \leftarrow \mathit{done}_r[i] \cup S$ for all $i \neq r, r'$
$\mathit{minlabel}_r \leftarrow \min(\mathit{minlabel}_r, L)$
$\mathit{solid}_r[r'] \leftarrow \mathit{solid}_r[r'] \cup S$
for all i : $\mathit{gossip}_r[i] \leftarrow \mathit{gossip}_r[i]\cup$
$\cup((\bigcap_j \mathit{done}_r[j]) - \mathit{solid}_r[r])$
$\mathit{solid}_r[r] \leftarrow \mathit{solid}_r[r] \cup S \cup (\bigcap_i \mathit{done}_r[i])$
for y s.t. $\mathit{minlabel}_r(y)$
$\leq \mathit{minlabel}_r(\mathit{max\text{-}stable}_r)$
and $\mathit{stable\text{-}value}_r(y)$ is undefined,
in $\mathit{minlabel}_r$ order:
$(\mathit{stable\text{-}state}_r, \mathit{stable\text{-}value}_r(y)) \leftarrow$
$f(\mathit{stable\text{-}state}_r, y.\mathit{op})$

Figure 9: Automaton for optimized replica r

The optimized replica automaton has a new state component $gossip_r$, which is an array of sets of operation descriptors, one set for each replica. Each $gossip_r[i]$ set contains operations that replica r needs to send to replica i in the next gossip message. An operation enters all $gossip_r[i]$ sets whenever the replica learns new information about the operation. Set $gossip_r[i]$ is emptied every time replica r sends a gossip message to replica i.

Remark: Explicit sequencing of gossip messages combined with retransmission and removal of duplicates is sufficient to make the optimization work with *unreliable* channels that allow message losses, duplicate messages, and out of order delivery. We may consider this enhancement in the future.

5.3.2 Removal of Self-Gossip

ESDSAlg assumes that each replica sends gossip messages to itself as well as to other replicas. This behavior is inefficient in a practical implementation, but if we removed it from the *ESDSAlg* replica automaton, its behavior would be incorrect when there is only one replica in the system. The reason is that *ESDSAlg* updates a replica's set of operations that it knows to be stable only during receipt of gossip messages. In a one-replica system execution without self-gossip messages the operations would never stabilize, violating the requirement of eventual serializability. This optimization adds another action $solidify_r$ to the replica automaton to preserve correctness. The new action detects one-replica executions and updates the set of stable operations independently from gossip actions.

5.3.3 Memoizing Stable State

ESDSAlg ignores the cost of local computation at the replicas. A replica r gets the current value the value for operation op_n from the initial state σ_0 by re-computing it as $f^+(\sigma_0, \langle op_1, op_2, \ldots, op_n \rangle)$ for $op_1, op_2, \ldots, op_n$ in $minlabel_r$ order (the function f^+ applies $op_1, op_2, \ldots, op_n$, in that order, to σ_0 [7]). *ESDSImpl* faithfully implements the same inefficient behavior. Testing *ESDSImpl* under heavy operation load confirmed that the time consumed by recomputation can be significant. In addition, the algorithm requires all operations to stay in memory indefinitely to enable recomputation. These problems make the naïve implementation of the algorithm unsuitable for practical applications.

In [7], an optimization is suggested that involves *memoizing* stable state at each replica. Here we specify a more aggressive variation of this optimization. We apply operations to the stable state of a replica as soon as they have stabilized at that replica, whereas in [7], it is suggested to wait until the operation stabilizes globally before applying it. Our version of the optimization results in faster stabilization of operations and a corresponding increase in performance.

We add a state component $stable\text{-}state_r$ to each replica that keeps track of the *stable state*, which is the result of applying all operations, whose total order has been fixed at all replicas, to the initial state. To compute the current state, replica r needs only to apply all operations in $done_r[r]$ that have not yet stabilized to the stable state.

The computation of the new stable state takes place every time replicas receive gossip messages (see Fig. 9). Among all operations that have not yet entered the stable state, replica r finds one with the highest minlabel that has entered the $solid_r[r]$ set. Call this operation $max\text{-}stable_r$. All operations with minlabels lower than that of $max\text{-}stable_r$ are guaranteed to never change minlabels again, and no operation with a lower minlabel can be received later. This means that the order of operations up to and including $max\text{-}stable_r$ can never be altered again at replica r. Thus, the replica applies all operations with minlabels lower than that of $max\text{-}stable_r$ to the old stable state to compute the new stable state.

Remark: Combined with the multipart timestamp optimization (see next section), this optimization makes it possible to discard almost all information about operations as soon as they enter the stable state. We have not implemented this.

5.3.4 Multipart Timestamps

In *ESDSAlg*, an operation may be *causally dependent* on other operations previously processed by the system. To represent this dependence, the operation's state contains a *prev* component, which is a set of operation *id*s that must be executed before it. We substituted a more efficient method for tracking causal dependencies between operations in place of *prev* sets. Our approach is based on a technique called *multipart timestamps*.

A multipart timestamp t is a n-tuple $(t_1, \dots, t_n)$ of nonnegative integer counters. In the context of ESDS, n is the number of replicas. A partial order is defined on multipart timestamps: $t \leq s$ iff $t_j \leq s_j$ for $j \in [1..n]$. Two multipart timestamps are *merged* by taking their component-wise maximum.

In this optimization we remove *prev* sets from operation state and redefine the protocol for keeping track of dependencies between operations with multipart timestamps.

In the new protocol the state of operation j includes two multipart timestamps, *prev-ts* and *op-ts* (*op-ts* is initialized to all zeros). Replica state also gets two multipart timestamps, *val-ts* and *rep-ts*, both initially all zeros. The meanings of these new state components are as follows:

- *op-ts* is assigned to each new operation by the receiving replica in the manner described below. *Op-ts* is guaranteed to be unique for each operation.
- *prev-ts* plays the same role for an operation j that the *prev* set played in the unoptimized version. It specifies that any other operation with an *op-ts* smaller than j's *prev-ts* must be done before j. In other words, for each pair of operations i and j, $j.op\text{-}ts < i.prev\text{-}ts$ implies that j is in i's *prev* set.
- *val-ts* is the merge of *op-ts* timestamps of all operations done at the replica.
- *rep-ts* is the current replica timestamp, used to assign values to *op-ts* of newly submitted operations in the protocol below.

The protocol works as follows:

1. When replica r receives a new operation i from a front end, it increments r*.rep-ts[r]*, assigns r*.rep-ts* to i*.op-ts*, and sends the value of i*.op-ts* to the front end. The front end then forwards i*.op-ts* to the client.

2. When a client wants to specify that operations $i_1, \ldots, i_k$ must precede operation j, it merges i_1*.op-ts*$, \ldots, i_k$*.op-ts* and assigns the result to j*.prev-ts*. This method of specifying causal constraints restricts the kinds of constraints that the client can specify. In particular, it requires that *all* operations with *op-ts* $\leq j$*.prev-ts* must be done before j, not just $i_1, \ldots, i_k$. Thus, multipart timestamps trade off flexibility for efficiency.

3. When replica r does operation i (i.e., moves it into $done_r[r]$), it merges i*.op-ts* into r*.val-ts*.

4. Gossip messages from replica r to replica r' contain r*.rep-ts*. Upon receipt of the gossip message, replica r' merges r*.rep-ts* into r'*.rep-ts*. For all operations $i_k \in done_r[r]$ included in the gossip message, r' merges i_k*.op-ts* into r'*.val-ts*.

5. When replica r wants to do operation i and needs to check that i's dependencies have been satisfied, it checks that i*.prev-ts* $\leq r$*.val-ts*.

The multipart timestamp optimization does not introduce any changes to the abstract description of the optimized replica automaton in Figure 9. The optimization only changes the way the precondition $x.prev \subseteq done_r[r].id$ of the do_it_r action is implemented in *ESDSOptImpl*.

Remark: To complete the multipart timestamp implementation, it is necessary to take care of the case when a client submits an operation to more than one replica simultaneously and gets different *op-ts* values for the operation. This has been relegated to future work, as it is not essential to our goal of implementing a working timestamp prototype.

5.4 Fault Tolerance in ESDS

We have already remarked on how to make our implementation able to cope with unreliable channels by introducing gossip message sequencing, retransmission and removal of duplicates.

Our implementation also includes some *ad-hoc* fault tolerance mechanisms for handling fail-stop faults and restarts of replicas. These mechanisms make strong timing assumptions about the environment. In particular, they rely on time bounds on communication latency. We are in the process of relaxing these assumtions and specifying these fault tolerance mechanisms formally.

5.5 Applications Using ESDS as a Generic Building Block

In order to test our implementations and to provide a proof-of-concept of the suitability of our implementation as a generic building block, we have implemented four data service applications that use ESDS.

5.5.1 String Concatenation Service

The *String Concatenation Service* is a simple data service application. The data object is a single string that supports two operations, *Read* and *Concatenate*. The *Read* operation gives the current value of the string. The *Concatenate* operation appends its argument to the string and gives back the new value. Because of its simplicity, the String Concatenation Service was used for testing *ESDSImpl* and *ESDSOptImpl* during development and for running empirical measurements of *ESDSOptImpl* performance.

5.5.2 Counter Service

The *Counter Service* is another simple data service application, similar to String Concatenation in its level of sophistication. The data object is a integer counter variable that supports two operations, *Read* and *Add*. The *Read* operation gives the current value of the variable. The *Add* operation adds an integer argument to the current counter value and gives back the new counter value.

The Counter Service differs from the String Concatenation Service in one potentially important respect. Its update operation *Add* commutes with other *Add*s, whereas the *Concatenate* operation of the String Concatenation Service does not commute with other *Concatenate* operations (unless one of them has the empty string as an argument). The Counter Service was created with the purpose of testing whether commutative update operations like *Add* lead to a smaller percentage of inconsistent responses than non-commutative update operations like *Concatenate* (as we will see in Section 6, it does not).

5.5.3 Distributed Spreadsheets

The purpose of creating a third, more sophisticated client has been to demonstrate the viability of ESDS as a platform for creating diverse and capable data service applications. The *Distributed Spreadsheets* client creates an environment where several people can simultaneously enter spreadsheet data into the same Microsoft Excel workbook. Their additions are sent to ESDS replicas, which maintain the current state of the workbook and can refresh each user's copy on demand. A possible use of this combination of Excel and ESDS is to allow multiple users to enter disjoint data into a single Excel file concurrently, see the updates of others automatically, and not worry about overwriting other user's additions with their own.

5.5.4 A More Extensive ESDS Application: Bank Accounts

The original work on lazy replication and ESDS [7] suggests that directory and information services (and similar applications) are the most suitable candidates for ESDS-based implementation because immediate consistency is not important to users of such systems. To show that ESDS could potentially be used for a wider variety of applications, we have implemented an ESDS application that keeps track of bank accounts. The bank application demonstrates how ESDS-based services can utilize strict and non-strict operations and multipart timestamp-based dependencies.

The service maintains a database of customer accounts in a bank, implemented as an application layer on top of ESDS. Branches of the bank are assumed to be located at physically different sites. At least one replica node resides at each branch. Operations submitted at a particular branch are forwarded to the local front end. During normal operation, the front end submits these operations to the local replica. However, if that replica happens to be down, the branch can continue to function by having the front end submit the operations to replicas at remote branches.

The database maintains a set of data tuples in the form of (*name*, *amount*). In addition to opening a new account and closing an established one, there are three basic operation which can be carried out on an account. The operations with their corresponding ESDS specifications are as follows: (1) *Withdrawal*: strict = *true*, *prev-ts* = $full_r$, (2) *Deposit*: strict = *false*, *prev-ts* = *empty*, (3) *Balance*:

- Local, Hurried: strict = *false*, *prev-ts* = *empty*
- Local, Quick: strict = *false*, *prev-ts* = *local-*$full_r$
- Global, Prompt: strict = *false*, *prev-ts* = $full_r$

By *empty*, *local-*$full_r$, and $full_r$ values of *prev-ts* we mean the following. Assume that the last operation submitted to replica r had been assigned timestamp $(t_1, \ldots, t_n)$. Then *empty* $= (0, \ldots, 0)$, *local-*$full_r = (0, \ldots, 0, t_r, 0, ..., 0)$, and $full_r = (t_1, \ldots, t_n)$. The interpretations of these values of *prev-ts* as *prev* sets are as follows. An *empty* value means that the *prev* set contains no operations. A *local-*$full_r$ value means that the *prev* set includes all operations previously submitted at replica r (but no others), and a $full_r$ value means that the *prev* set includes all previously submitted operations at all replicas that replica r knows about.

A *Deposit* operation always succeeds, and it is independent of its ordering relative to other operations on the same account. Thus, *Deposit* is specified without any dependency constraints. On the other hand, a *Withdrawal* of amount m can result in different answers to the client, depending on whether the account has sufficient funds. If the case when it does not, the *Withdrawal* operation does not change the amount in the account and returns an error message. Otherwise, it decreases the amount in the account by m. Permitting two *Withdrawal* operations on the same account to occur concurrently at different replicas would allow the client to withdraw money she does not have. Therefore, we have implemented *Withdrawal* as a strict operation.

It is up to the customer to determine what level of inconsistency she can tolerate in a *Balance* operation in exchange for lower latency. Using the Hurried option, there is no guarantee that previously submitted operations for the account will be visible by the *Balance* lookup. With Quick *Balance* lookup, all previously submitted operations at the local branch will be visible, but there is no guarantee with respect to operations submitted at other branches. Using Prompt *Balance* lookup, all operations on the account known at the local replica will be visible, but there is no guarantee that deposit operations carried out at other branches and not yet gossiped to the local branch will be visible.

6 Empirical Testing and Analysis

We have evaluated our implementation on a network of workstations. All tests were done on a 10 Mbps Ethernet LAN of 12 Sun workstations running SunOS 4.1.4 using the *ESDSOptImpl* implementation. The workstations were not dedicated to this project, and their loads fluctuated. To account for this variance, we performed each test 10 times and averaged the results. We ran some tests with over 20 replicas; however, for performance testing we used 10 replicas only, so that each replica ran on a separate processor. Each test run consisted of three hundred operations. We measured two performance characteristics of the prototype, average response time, and average throughput, which we now define.

The *response time* for an operation is the elapsed time between submission of the operation to a replica and response from the replica in a given execution of the implementation.

The average *throughput* is the number of operations the system processes per unit time in a given execution of the implementation.

We would also like to know how the percentage of strict operations among all operations submitted to the system affects performance and the degree of "inconsistency" in responses. For a given execution of the implementation, we say that a response to an operation x is *inconsistent* if the value returned to the user for x differs from the value of x in the eventual total order of operations. More formally, let $response_r(x, v_x)$ be a response sent by replica r to a front end. Let $val_{to}(x)$ be the value of x in the eventual total order of all operations. Then $response_r(x, v_x)$ is *inconsistent* if $v_x \neq val_{to}(x)$. In a finite execution of an implementation, the *degree of inconsistency* is the percentage of inconsistent responses among all responses to user operations returned by the system during the execution.

We have conducted three series of tests using *ESDSOptImpl*. The first series was designed to test the flexibility of the software, regardless of the underlying hardware. The second series was used to determine how system performance, characterized by response time and throughput, depends on the number of replicas. For the first two series, we used the String Concatenation Service with empty strings; the datatype and content of the application data were irrelevant for these experiments. All operations in these tests were non-strict.

The third series measured the changes in system performance and degree of inconsistency in response to varying the percentage of the submitted strict operations. We used the Counter Service and the *Add* operation for our test setup to find out whether commutative update operations like *Add* affect the percentage of inconsistent responses when compared to non-commutative update operations.

Three quantities were measured for each run: (1) the average time T_{fe} from the submission of an operation by front end f to one of the replicas until the receipt of replica response by f, (2) the average time T_r from the receipt of operation x by replica r till the response from r with a value for x, and (3) the total time τ for the system to process and respond to all three hundred operations.

From these data, we obtained two different measures of the response time AT_{fe} and AT_r, and one measure of system throughput AP, as follows. For each number of replicas from $N = 1$ to $N = 10$, we averaged T_{fe} over 10 runs to get the average

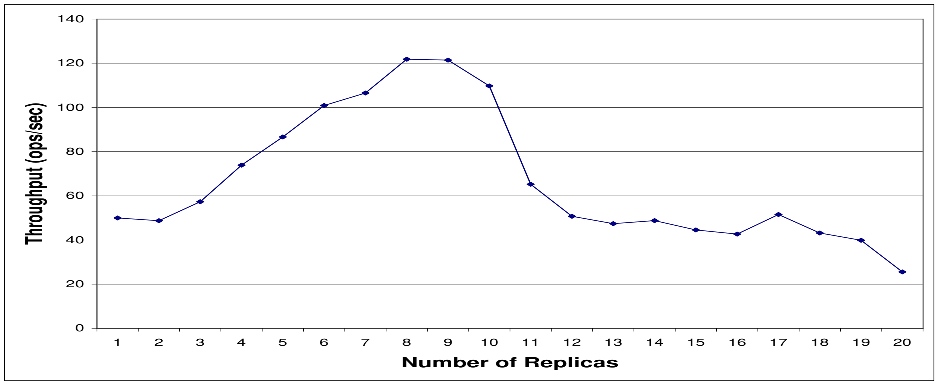

Figure 10: System Throughput AP (submission rate is 330 operations/sec)

time AT_{fe} it took a front end to receive a response from a replica after it sent the request message. The average of T_r over 10 runs produced the average time AT_r it took a replica to process a request and send back a response. Finally, the average value of $300/\tau$ over 10 runs gives the average system throughput AP.

We determined experimentally that a single replica can keep up with requests if each comes approximately once in 30 milliseconds. If this frequency increases, a single replica cannot keep up and messages accumulate in the MPI message queues. Therefore, in our testing, we ranged the frequency of requests from one every 30 milliseconds to approximately one every 30 times N milliseconds, where N is the number of replicas.

The complete results are presented in [5]. Some key observations follow.

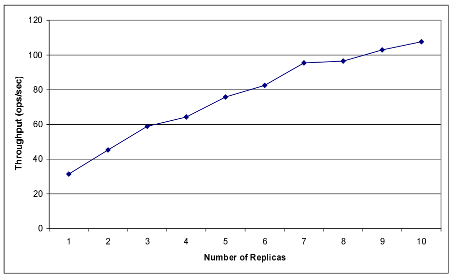

Figure 11: System Throughput AP (submission rate is 33*N operations/sec, N = Number of replicas)

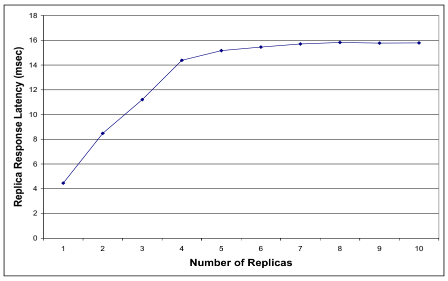

Figure 12: Response Time at the Replicas AT_r (submission rate is 33 operations/sec)

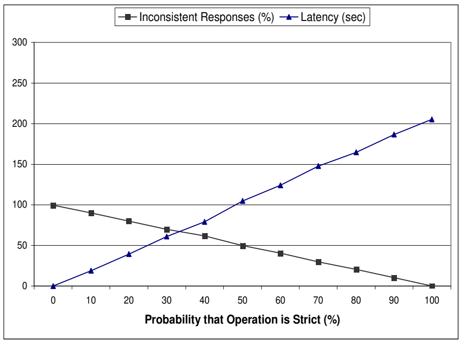

Figure 13: Tradeoff Between Response Time and Consistency (4 Replicas)

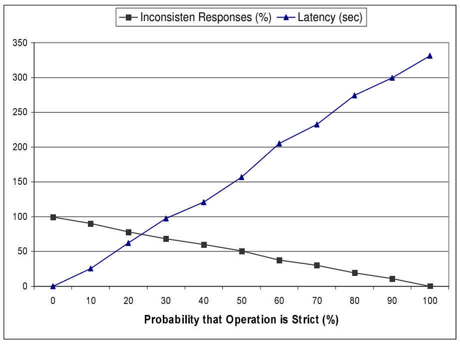

Figure 14: Tradeoff Between Response Time and Consistency (6 Replicas)

We ran tests in which the number of replicas exceeded the number of available processors. This demonstrated the flexibility of our implementation to run in a variety of settings. The test increased the number of replicas from $N = 1$ to $N = 20$, while operations were submitted at the constant rate of 330 operations/sec. The collected data are plotted in Figure 10. The system throughput increases with the number of replicas participating in the system and performing submitted operations. However, the throughput drops off when the system runs out of physical processes (at $N = 10$). The overhead of context switches and the forced serialization of communications between replicas sharing a processor adversely impact system performance. In all other examples, we have limited the number of replicas to the number of available processors.

We now describe how system throughput AP and response time at replicas AT_r depend on the number of replicas and the rate of operation submission. Figure 11 shows that throughput rises (nearly linearly) by adding new replicas when the frequency of requests starts at thirty three operations/sec for one replica and is increased proportionally to the number of replicas. However, throughput does not come close to the estimated limit of three hundred and thirty operations per second. This might be justified by the increasing gossip overhead.

Figure 12 shows response time for a constant, low frequency of requests and with increasing number of replicas. This frequency is just short of saturating one replica. Not surprisingly, the best response time shows up in a one-replica execution: the replica can keep up with the submissions, there is no gossip, and all operations stabilize immediately. As the number of replicas grows, operations take longer to stabilize (recall that all replicas must perform the operation before it stabilizes); moreover, the time spent on gossip processing is also increased. Each replica is busy re-applying non-stable operations and gossiping some of the time. As a consequence, we observe an increase in AT_r for $N = 1$ through $N = 4$. However, the system reaches a steady state for $N \geq 4$. After that point, the load on individual replicas is sufficiently low, and they can keep up with both gossip and new operations simultaneously.

Finally, we performed a test to observe the trade-off between the response time and consistency when the number of required strictly consistent responses increases. This test was conducted using the Counter Service application, using *Add* operations. The results for four and six replicas are summarized in Figures 13 and 14, respectively.

The percentage of inconsistent responses goes down linearly as the percentage of strict operations climbs. On the other hand, since strict operations require the system to stabilize the operation's value at all replicas before responding, the latency of responses to strict operations is significantly higher than to non-strict operations. This difference between strict and non-strict operation latencies is reflected in the linear increases of average latency with percentage of strict operations in Figures 13 and 14. The coefficient of the linear increase is higher for the larger number of replicas (6), possibly because the time required to synchronize all replicas with respect to a particular operation increases with the number of replicas.

The trade-off between consistency and performance is clearly demonstrated by these results. Users willing to tolerate transient inconsistencies in system responses can submit primarily non-strict operations. For these users the system will function in the region on the left side of Figures 13 and 14, where transient inconsistency is high but response latency is low. Conversely, users who require strict consistency in responses to some or all of their operations will pay the cost of higher response latency.

We have run the same experiment using the String Concatenation service in place of the Counter service and found that the results did not differ from those shown in Figures 13 and 14. Thus, we found no evidence that commuting operations like Counter service's *Add* substantially change the percentage of inconsistent responses compared to non-commuting operations like String Concatenation service's *Concatenate*.

7 Conclusions

We have defined a set of techniques for converting source algorithms specified as I/O automata compositions into target distributed programs written in an imperative language. We demonstrated that the techniques support object-oriented design for target programs by implementing a set of C++ objects that encapsulate common properties of I/O automata and can be used in designing the target program. Our techniques are applicable to commonly occurring algorithms that use asynchronous channels or input/output combinations that involve two automata for communications between distributed components.

Using our conversion techniques, we implemented the abstract ESDS algorithm *ESDSAlg* as a distributed program *ESDSImpl*. The modular design of *ESDSImpl* allowed us to create modules specific for ESDS only once and then build several distinct data services without any need to further modify these modules. In this way, we show how *ESDSAlg* can be used as a building block for distributed data service implementations. A data service built on top of ESDS inherits its characteristics. Strict consistency in such a service may be relaxed by specifying operations as non-strict and requiring only the explicitly stated causal relations between individual operations (given by *prev* sets) to be preserved. The benefit of such relaxation is improved system performance. At the same time, the system is assured of reaching a globally consistent state, as specified and proven in [7]. This is a property of *ESDSImpl* that differentiates it from the implementation in [13], in which additional measures must be taken by an application designer to ensure that replica states do not diverge irrevocably.

We have added several optimizations to *ESDSAlg*, producing an optimized abstract algorithm *ESDSOptAlg* and a corresponding *ESDSOptImpl* implementation. *ESDSOptImpl* fixes some inefficiencies of *ESDSImpl* and makes ESDS a more practical system. Some current and future work on other optimizations to ESDS is discussed in the next section.

The empirical tests on *ESDSOptImpl* show how its performance, characterized by response time and throughput, is affected by changing the number of replicas participating in the execution and by changing the system load. The data also confirm that ESDS performance reflects an inherent tradeoff between performance and consistency.

8 Ongoing and Future Work

Several theoretical and practical aspects of our work are open for further exploration. On the theoretical side, it remains to be shown that *ESDSImpl* (as well as *ESDSOptImpl* and *ESDSOptAlg*) implement *ESDSAlg* (in the mathematical sense). This requires formalization of our framework for converting I/O automata to distributed programs. More ambitiously, it would be interesting to develop a framework for showing that a practical implementation of an algorithm, derived with our techniques and treated as a mathematical object, correctly implements the I/O automata specification of the original algorithm.

As we have remarked, the mechanism for converting I/O combinations to code (Section 3.5) needs to be extended to the more general case when multiple automata participate in the combination. Section 5.3.4 contains suggestions for further work on the multipart timestamp optimization.

A possible application of ESDS is a wide-area network data service. To be useful in practice as a WAN service, ESDS must accomodate dynamic changes in the number of replicas and tolerate server and communication failures. We are currently experimenting with a version of ESDS that tolerates simulated fail-stop crashes of replicas and allows crashed replicas to rejoin the system after recovery.

Finally, our implementation can be viewed as a proof-of-concept of ESDS as a generic component of distributed operating systems. Serious applications that can benefit from such service include directory services and distributed type repositories useful for distributed object systems. It would be interesting to formalize ESDS as a distributed operating system service and use it in implementing a real application.

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*, pp. 627-644, Oct. 1976.

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

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

[4] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):260-274, 1982.

[5] O. Cheiner, *Implementation and Evaluation of an Eventually-Serializable Data Service*, M. Eng. Thesis, Electrical Engineering and Computer Science Massachusetts Institute of Technology, 1997.

[6] J. Dongarra, S. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, Vol.39, No. 7 (July 1996), pp. 84-90

[7] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-Serializable Data Services. *Principles of Distributed Systems* 1996, pp. 300-310.

[8] 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*, pp. 70-75, March 1982.

[9] H. Garcia-Molina, N. Lynch, B. Blaustein, C. Kaufman, and O. Schmueli. Notes on a Reliable Broadcast Protocol. *Technical Memorandum*, Computer Corporation America, October 1985.

[10] W. Gropp and E. Lusk. User's guide for MPICH, a portable implementation of MPI. *Technical Report ANL-96/6*, Argonne National Laboratory, 1994.

[11] D. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th ACM Symposium on Principles of Operating Systems Principles*, pp. 150-162, December 1979.

[12] M. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computer Systems*. 4(1):32-53, February 1986.

[13] 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.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multi-process programs. *IEEE Transactions on Computers*, 28(9):690- 691, September 1979.

[15] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[16] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[17] 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*, August 1988.

[18] 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.

[19] R. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180-209, June 1979.

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

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

[22] The Message Passing Interface Forum. *The MPI message- passing interface standard.* http://www.mcs.anl.gov/mpi/standard.html, May 1995.

[23] Open Software Foundation. *OSF DCE Application Development Guide.* Prentice Hall, Englewood Cliffs, NJ, 1993

Acknowledgements

The authors are grateful to Nancy Lynch, Victor Luchangco, and Istvan Derenyi for valuable technical suggestions and to Istvan Derenyi for his contributions to the implementation of the Bank Accounts service.

Author Contact Information

Oleg M. Cheiner
Department of Computer Science,
Carnegie Mellon University, 5000 Forbes Ave.
Pittsburgh, PA 15213-3890, USA,
Email: `oleg@cmu.edu`.
The work of the first author was substantially done
at the Massachusetts Institute of Technology.

Alex A. Shvartsman
Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square NE43-368,
Cambridge, MA 02139, USA,
Email:`alex@theory.lcs.mit.edu`
and
Department of Computer Science and Engineering,
University of Connecticut,
191 Auditorium Road, U-155, Storrs, CT 06269, USA,
Email: `aas@cse.uconn.edu`.