Implementation and Evaluation of an Eventually-Serializable Data Service

by

Oleg M. Cheiner

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

August 21, 1997

Author ____

Department of Electrical Engineering and Computer Science
August 21, 1997

Certified by ____

Alex Allister Shvartsman
Thesis Supervisor

Accepted by ____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Implementation and Evaluation of
an Eventually-Serializable Data Service
by
Oleg M. Cheiner

Submitted to the
Department of Electrical Engineering and Computer Science

August 21, 1997

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis builds on the work of Fekete et al. [1], who defined an eventually-serializable data service (ESDS) and an abstract algorithm for it. ESDS allows its users to relax consistency requirements in return for improved responsiveness, while providing guarantees of eventual consistency of the replicated data. An important consideration in formulating ESDS was that it could be employed in building real systems.

We formulated a framework that assists a programmer in mapping algorithms specified using I/O Automata notation [3] to distributed implementations. Using the framework, we developed a distributed implementation of ESDS and explored its behavior in a distributed setting. We combined the implementation of ESDS with different data types and clients, thus demonstrating the suitability of the service as a general building block. The implementation was experimentally evaluated on a network of workstations. In this setting the implementation scaled in the number of processors and reflected a designed trade-off between consistency and performance.

Thesis Supervisor: Alex Allister Shvartsman
Title: Research Associate, MIT Laboratory for Computer Science

Contents

Chapter 1

Introduction

Specification of distributed systems building blocks and development of supporting algorithms is one of the main research areas at the Theory of Distributed Systems group (TDS) at the MIT Laboratory for Computer Science. TDS recently defined a flexible eventually-serializable data service (ESDS) [1]. The definition includes a formal specification of the data service and an abstract distributed algorithm that implements the service. ESDS relaxes consistency guarantees provided by serializable distributed 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 was that it could be employed in building real systems. In this work we develop a distributed experimental implementation of the ESDS algorithm. Using this implementation, we explore the practical issues associated with using the ESDS specification and abstract algorithm in real systems.

1.1 Background

As outlined in [1], 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 operations submitted by clients. Several notions of consistency have been defined. The strongest notion of consistency is *atomicity*, in which replicas emulate a single centralized object. Methods to achieve atomicity include write-all/read-one [4], primary copy [5, 6, 7], majority consensus [8], and quorum consensus [9, 10]. Achieving atomicity often has a high cost, some applications, such as directory services, are willing to tolerate some transient inconsistencies. This gives rise to different notions of consistency. *Sequential consistency* [11], guaranteed by systems such as Orca [12], allows operations to be reordered as long as they remain consistent with the view of isolated clients. Other systems provide even weaker guarantees to the clients [13, 14, 15] to get better performance.

Improving performance by providing weaker consistency guarantees may lead to more complicated semantics. While in practice, replicated systems are often incompletely or ambiguously specified, it remains very important to provide formal consistency guarantees. Ladin, Liskov, Shrira, and Ghemawat [18] define one highly available 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. Responses to operations may be out-of-date, not reflecting the effects of operations that have not yet been received by a given replica. Building on the work of [18], Fekete et al. [1] specify a flexible eventially-serializable data service that we use in this work.

1.2 Experimental ESDS Implementation

The ESDS algorithm is specified as a composition of I/O Automata. I/O Automata [2] are specified as state machines using a declarative description language. 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 published. We develop a framework for converting I/O Automata-based algorithms to distributed implementations that use message passing. We use the framework in designing the ESDS system. We believe that the techniques in the framework are general and that they can be used to implement other I/O Automata-specified algorithms.

The design of a distributed ESDS system is an important part of our work. The abstract ESDS algorithm is specified to be independent of the serial data type of the replicated data object. Our implementation of ESDS is built using object-oriented techniques to ensure that this independence is preserved in the implementation. Our design provides a layer of abstraction between the objects that implement the ESDS algorithm and the objects that vary with each specific data service application built on top of the ESDS system.

We implement a functioning ESDS system and build three simple applications on top of it to demonstrate the viability of our design as a generic building block for real distributed data services. The implementation relies on a standard message-passing subsystem to ensure portability. Implementation and testing were done on a network of Sun workstations running the SunOS 4.1.4 operating system.

1.3 Empirical Testing and Analysis

We 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 provided data on the behavior of the implementation with varying number of participating replicas and with varying system load. The tests also confirmed that ESDS represents 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.

1.4 Roadmap

The rest of this thesis is organized as follows. Chapter 2 gives the models and definitions used in other chapters and describes the hardware and software environment in which the ESDS prototype was implemented. Chapter 3 describes a framework for converting I/O Automata-specified algorithms to distributed implementations. Chapter 4 describes how an experimental ESDS service was implemented using that framework. Chapter 5 discusses the empirical results obtained using the experimental service. Our conclusions and suggestions for future work are in Chapter 6.

Chapter 2

Models, Definitions, and Platforms

This chapter gives an overview of the models and terminology that we use throughout the rest of the thesis. Section 2.1 gives a brief introduction to the I/O Automata model, which was used to specify the ESDS service [1]. Section 2.2 describes the hardware and software environment and tools used in implementing and testing the ESDS service. Section 2.3 defines a nomenclature used to identify and distinguish different versions of the ESDS algorithm specification and corresponding implementations. It also introduces terminology for use in later chapters.

2.1 Models: An Introduction to I/O Automata

We now overview a formal model for asynchronous computation, the Input/Output Automaton (I/O Automaton) model. This is a general model, suitable for describing distributed algorithms. The model provides a precise way of describing and reasoning about asynchronous interacting components. For a complete description of the I/O Automaton model, the reader is referred to [2] and [3], from which this section is abstracted.

An I/O Automaton models a distributed system component that can interact with other system components. It is a state machine in which the transitions are associated with named actions. The actions are classified as either input, output or internal actions. The inputs and outputs are used for communication with the automaton's environment, while the internal actions are visible only to the automaton itself. The input actions are not under the automaton's control, while the automaton itself specifies what output and internal actions should be performed.

An Input/Output automaton's "signature" is a description of its input, output and internal actions. A *signature* S is a triple consisting of three disjoint sets of actions: the *input actions* $in(S)$, the *output actions* $out(S)$ and the *internal actions* $int(S)$. The *external actions*, $ext(S)$, are $in(S) \cup out(S)$, the *locally controlled actions*, $local(S)$, are $out(S) \cup int(S)$, and $acts(S)$ are all the actions of S. The *external signature*, or *external interface*, $extsig(S)$, is defined to be the signature $(in(S), out(S), \emptyset)$.

An I/O automaton consists of five components:

- $sig(A)$, a signature,
- $states(A)$, a set of states,
- $start(A)$, a nonempty subset of $states(A)$ known as the *initial states*,
- $trans(A)$, a *state transition relation*, and
- $tasks(A)$, a *task partition*, an abstract description of "threads of control" within the automaton (not used in the ESDS specification).

We call an element (s, π, s') of $trans(A)$ a transition or step of A. The transition (s, π, s') is called an *input transition*, *output transition*, etc., based on whether the action π is an input action, output action, etc.

If for a particular state s and action π, A has some transition of the form (s, π, s'), then we say that π is enabled in s. Since every input action is required to be enabled in every state, automata are said to be input-enabled.

I/O Automata are often described in a *precondition-effect style*. This style groups together all the transitions (s, π_n, s') that involve each particular type of action into a single piece of code. The code specifies the *preconditions* under which the action is permitted to occur, as a predicate on s. Then it specifies the *effects* that occur as a result of applying π_n to s. The code in the effects clause gets executed atomically.

Next, we define (informally) the operation of *composition* for I/O Automata.

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 π, so do all component automata that have π in their signatures.

We impose certain restrictions on the automata that may be composed. First, since internal actions of an automaton A are intended to be unobservable by any other automaton B, we do not allow A to be composed with B unless the internal actions of A are disjoint from the actions of B. At most one component automaton "controls" the performance of any given action; that is, we do not allow A and B to be composed unless the sets of output actions of A and B are disjoint.

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.

The states and start states of the composition automaton are vectors of states and start states, respectively, of the component automata. The transitions of the composition are obtained by allowing all the component automata that have a particular action π in their signature to participate simultaneously in steps involving π, while all the other component automata do nothing. Since individual component automata are input-enabled, so is their composition. It follows that a composition of several automata is an I/O Automaton.

2.2 Platforms

In this section we describe the hardware and software environment in which the ESDS system was developed and tested.

2.2.1 Hardware and Operating Systems

The prototype ESDS service was developed and tested on a network of Sun workstations running SunOS 4.1.4. The MPI (see Section 2.2.2) implementation used with the prototype was MPICH version 1.0.12 [17]. Three clients for sample ESDS service applications were developed. One client was developed for Win32 and tested under Windows 95 on an Intel Pentium machine. Two other clients ran under SunOS 4.1.4.

2.2.2 Interprocess Communications

This section describes MPI, our choice of the method of communication between distributed components of the ESDS implementation. It explains its advantages and disadvantages.

In selecting a method for communication, we took into account

- Suitability for implementing I/O Automata,
- Simplicity of communication semantics,
- Availability of development tools, and
- Portability

We chose to use the Message Passing Interface (MPI) Standard [16] in implementing ESDS. MPI is a practical and portable message passing system. It contains a large set of communication primitives, and it makes it possible to write message passing applications using only a few primitives. This has the advantage of simplifying programs and making it easier to reason about their behavior.

Main reasons for choosing MPI are as follows:

- Simplified mapping of I/O Automata to message-passing code,
- MPI message-passing primitives have simple semantics,
- MPI is implemented on many distributed platforms, together with development tools.

An MPI program is a collection of *MPI nodes*. Each node is a sequential thread of control with a private memory space. MPI nodes are specified to execute concurrently and asynchronously.

We used a small set of MPI features in our implementation. The message passing primitives we used are:

MPI-Send Sends a point-to-point message from one MPI node to another. *MPI-Send* operates in three different modes of communication. In *standard* mode, MPI is free to decide whether to buffer the message and return from *MPI-Send* immediately or wait until a matching *MPI-Recv* has been posted. A *buffered* mode send operation

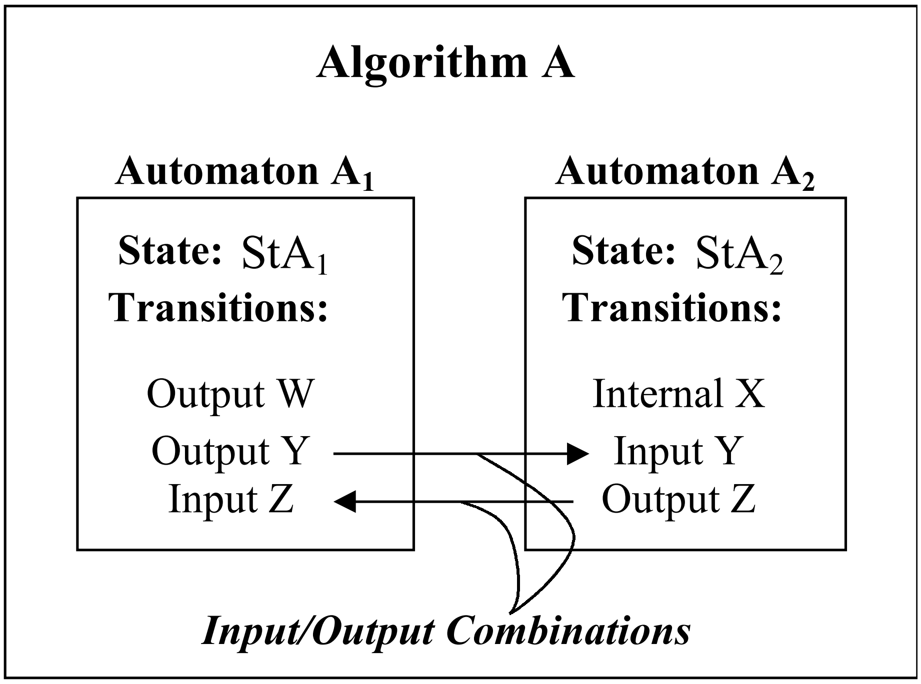

Figure 2.1: I/O Automata Composition

forces MPI to buffer the message and return as soon as that is done. A buffered send is *local* - its completion does not depend on an occurrence of a matching receive. In *synchronous* mode a send will block the caller node until a matching receive has been posted. In the ESDS prototype we used the standard mode and left memory and performance management to the MPI implementation.

MPI-Recv Receives a point-to-point message from another MPI node. Messages sent between any two MPI nodes are guaranteed to arrive exactly once in FIFO order. *MPI-Recv* blocks the caller node until a matching *MPI-Send* has been posted.

MPI-Iprobe Returns a boolean value that indicates whether the calling MPI node has any pending messages from another MPI node, or from any one of a group of other nodes.

2.3 Definitions and Terminology

In this section we assign names to different variations of the ESDS algorithm [1] and its implementations. We also introduce additional terminology relating to I/O Automata that we use later.

Two definitions needed in the description of a framework for converting I/O Automata to distributed implementations relate to composition of I/O Automata. As described in Section 2.1, individual automata in a composition communicate among each other by means of input and output actions with the same name. We distinguish between two types of actions in an I/O Automata composition.

Definition 2.3.1 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 A_j ($i \neq j$), we call such action in A 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 A_k is called a *regular* action.

Figure 2.1 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 and Y and Z are I/O combinations.

The following names identify abstract ESDS algorithms and their implementations:

ESDSAlg refers to the unoptimized abstract algorithm for ESDS [1].

SimpleESDSAlg is a simplified version of *ESDSAlg* that replaces channel automata with I/O combinations. This is done in two steps. The first step is removing channel automata from the composition. The second is substituting an I/O combination for each pair of channel connection points of the form **Output** $send_{i,j}$(<"msgtype", $arg_1, arg_2, \ldots$ >) and **Input** $receive_{i,j}$(<"msgtype", $arg_1, arg_2, \ldots$ >). The substituted I/O combination is named $msgtype_{i,j}(< arg_1, arg_2, \ldots >)$ at both ends.

ESDSOptAlg is an optimized version of *ESDSAlg*. The optimizations included in *ESDSOptAlg* and an I/O Automata description of the optimizations are presented in Section 4.3.1.

ESDSImpl, *SimpleESDSImpl*, and *ESDSOptImpl* are distributed programs (written in C++ using MPI) that implement *ESDSAlg*, *SimpleESDSAlg*, and *ESDSOptAlg* respectively.

Chapter 3

A Framework for Converting I/O Automata to Distributed Programs

I/O Automata have been effectively used for describing message-passing distributed algorithms and in proving correctness properties of the algorithms. In this chapter we present a framework for converting such commonly occurring algorithms specified with I/O Automata compositions into distributed implementations using an imperative language (we used C++ in our work).

The source I/O Automata composition being converted is called the *source composition.* We also call the algorithm represented by the source composition the *source algorithm.* The result of the conversion is a program. We call it the *target program.*

It is important to be able to reason that the target program is an accurate implementation of the source algorithm specification. Because of this, the techniques discussed here are conservative and will usually lead to an overspecification of the I/O Automata-specified algorithm, but they still allow for a large and interesting subset of behaviors to be reflected in the target implementation.

3.1 Overall Approach

I/O Automata notation can be used to specify distributed algorithms involving a collection of communicating nodes. This is normally done by encapsulating the behavior of each node I as a separate automaton A_i. The entire algorithm is represented by the composition A of component automata A_1, A_2, ..., A_m. The internal actions of each component automaton A_i represent local processing at the corresponding node. The Input/Output combinations represent communication between the nodes. The input and output actions of each automaton that do not participate in an Input/Output combination represent the interaction of the corresponding node with its external environment.

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. (It is also possible to combine several automata to run as a single process if there is a reason to do so.) Note that if the composition does not model a distributed system, the techniques presented in this chapter can still be applied to convert it to an imperative language program, but of course this will not yield a distributed implementation of A.

Each action of the source composition will have a corresponding fragment of code in the target program that implements the action. The conversion techniques ensure that each such fragment of code appears to be atomic.

The rest of this chapter describes the procedures to be followed for converting a composition A of I/O Automata to a distributed program. A is assumed to consist of component automata $A_1, \ldots, A_m$. An action with the name X belonging to automaton A_i is defined to have preconditions clause PXA_i and effects clause EXA_i. A component automaton A_i will correspond one-to-one with an implementation process P_i.

Section 3.2 describes how to represent the state of A's component automata in the target program's processes. Section 3.3 describes how to convert precondition-effect style actions to code. Section 3.4 presents our implementation of these techniques in C++.

3.2 Representing Component I/O Automaton State

The local state of a component automaton A_i is represented by the state variables local to the corresponding process P_i. We do not make provisions for representing global state of A. If A utilizes global state, it may not be easily implementable as a distributed program. Global state must be removed from such algorithms if one wishes to apply these techniques to them.

3.3 Converting Individual Actions to Code

In this section we describe the procedures to be used to convert individual actions to sequential code.

3.3.1 Converting Preconditions Clauses into 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 should be 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.

An action X may represent infinitely many state transitions of the automaton containing it, one per each instantiation of its arguments. More than one of these transitions may be enabled simultaneously. If that is the case, we require the predicate *Enabled* to return *true* for action X, but we leave it to the programmer to specify means for selecting the state transition. In our framework, the selection must be made at the time of execution of *Enabled*. This is done by choosing values for X's local variables such that the preconditions clause is satisfied. The chosen values are then used in the execution of the *Transition* procedure that implements the effects clause EXA_i (see Section 3.3.2). It is up to the programmer to ensure that the algorithm used in selecting the state transition gives all enabled state transitions a chance to execute.

3.3.2 Converting Effects Clauses into Procedures

The effects clause EXA_i describes the state transition(s) 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 (see Section 3.3.1). *Transition*'s effect on the state of P_i must correspond to the effects of EXA_i on the state of A_i.

3.3.3 Converting Regular Actions to Code

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

Automaton A_i
Output X
Preconditions: 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
Preconditions: None
Effects: EXA_j

Process P_j
IF NBReceive(P_i, "Initiate X");
Transition(EXA_j);
Send(P_j, "Done X");
ENDIF

Figure 3.1: Converting an Input/Output Combination to Code

and effects clauses of the action (in the case of an input action, the *Enabled* procedure will always return *true* and have no side effects). (We have already described the techniques for creating *Enabled* and *Transition* procedures in Sections 3.3.1 and 3.3.2.)

3.3.4 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 I/O Automata algorithms. The general I/O Automata model allows multiple automata to participate in one such combination. The mechanism of negotiation presented here should be extensible to the more general case, but we do not address this here.

The rule for converting an I/O combination to code is illustrated in Figure 3.1. 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. They are implemented by *MPI-Send* and *MPI-Recv*, respectively. The NBRecieve() in the process P_j is a non-blocking receive of a message, implemented by calling MPI-Iprobe and then calling *MPI-Recv* if there is a pending message of type *"Initiate X"* from P_i. 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 3.1). When the call to *Enabled(PXA$_i$)* returns *true*, P_i sends a message to P_j initiating the combination. Any argument that X may have 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. P_i then 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.

In a distributed implementation that follows this design, the effects of multiple actions and Input/Output combinations can be executed concurrently. For a regular action, the effects will be local to the automaton executing it. For an I/O combination (like the one in Figure 3.1), both effects clauses will finish executing before either participating automaton is able to continue with other actions. Therefore, only the state local to the participating automata can be changed by the effects clauses. It follows that the procedures representing regular actions and Input/Output combinations are atomic.

3.3.5 Deadlock Avoidance

As presented, the design is safe, but it 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_j, *"Done X"*) line and wait for each other indefinitely.

The deadlock problem can be resolved by setting up a *reservation system* for performing Input/Output combinations. Each process P_i maintains its *reservation status* in a state

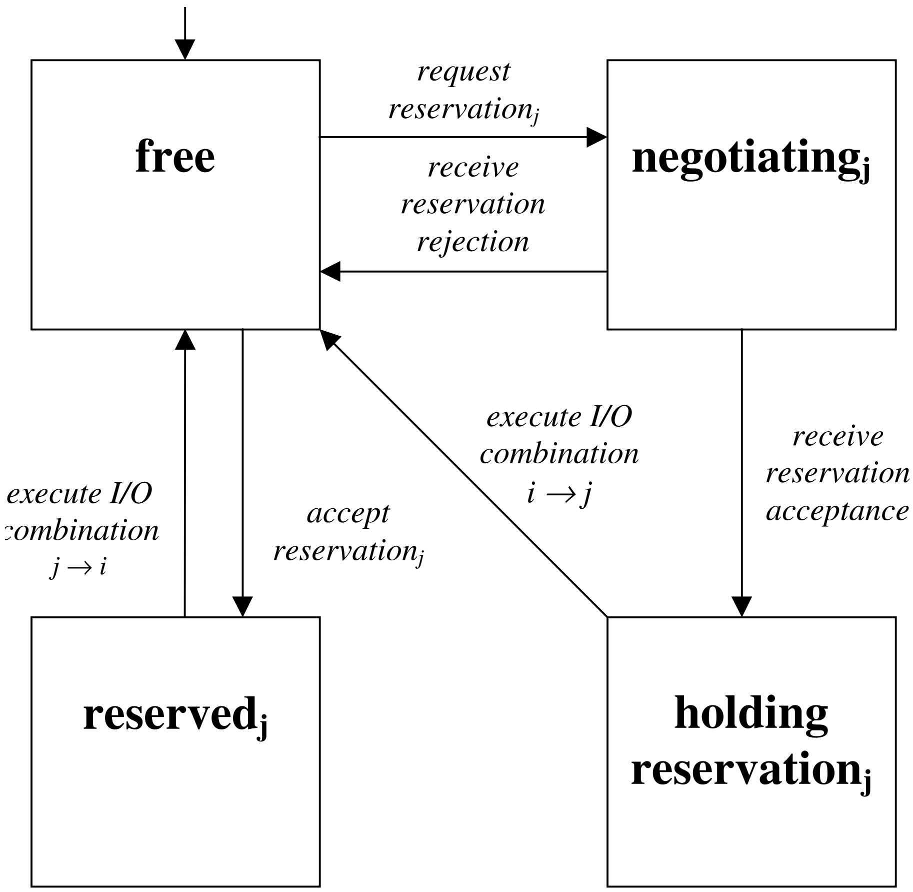

Figure 3.2: Reservation Status Finite State Automaton for Process P_i

variable. The states of the reservation status are: *free*, *reserved*$_j$, *holding reservation*$_j$ and *negotiating*$_j$, where j is the process number of another process ($i \neq j$). Reservation status's initial state is *free*. In this state the process is free to initiate or accept reservation requests. In the *reserved*$_j$ state P_i is waiting for process P_j to initiate an I/O combination. In the *holding-reservation*$_j$ state P_i may initiate an I/O combination with P_j. 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 3.2.

The reservation system imposes the following restrictions on the execution of P_i. P_i can initiate an Input/Output combination with P_j only if its reservation status is *holding-reservation*$_j$. P_i can participate in an Input/Output combination initiated by P_j only if its reservation status is *reserved*$_j$. The rules for obtaining and granting of reservations are specified in Figure 3.2.

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 *negotiating*$_j$ reservation status and waits for a response to the request. If the reservation was granted by P_j, P_i enters *holding-reservation*$_j$ and P_j enters *reserved*$_j$. P_i is then free to initiate an Input/Output combination as described in Section 3.3.4. If the reservation request was rejected, P_i bounces back to *free* reservation status.

Whenever a process is in *free* reservation status 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 accept them for the system to make progress.

It should be possible to prove that under the reservation system deadlock cannot occur. We informally argue why that is so.

The key is Invariant 3.3.1.

Invariant 3.3.1 Let $\mathcal{P}$ be the set of process identifiers in the target program. Then $\forall i, j \in \mathcal{P}$ s.t. $i \neq j$, P_i is in *holding-reservation*$_j$ $\Rightarrow$ P_j is in *reserved*$_i$.

Invariant 3.3.1 holds because 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.

Because of Invariant 3.3.1, 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 and is able to participate. Therefore, the I/O combination executes successfully.

3.3.6 Optimizing Abstract I/O Channels Away

The proposed mechanism for avoiding deadlock is costly, as it reduces potential concurrency in the system. For better performance it is desirable to avoid such a mechanism. For I/O Automata-specified algorithms that use channels with asynchronous message delivery for communication between its distributed components, an implementation that can preserve more concurrency is possible.

This is done by taking advantage of the fact that the message passing model used by MPI already implements the asynchronous channel discipline. The implementation of an algorithm that is specified using channels can use the message passing library instead of explicit channel automata. This removes a significant portion of the code that otherwise would have to be executed every time the algorithm interacts with a channel. Specifically, this optimization removes two I/O combinations (one at each the sending and the receiving end of the channel) and a separate process for the channel automaton. Since a channel-based composition of I/O automata uses I/O combinations only at the points where the channel connects to the sender and the receiver, the optimized implementation would not need to execute any (expensive) I/O combinations.

ESDSAlg uses asynchronous channels for communication among frontend and replica automata and thus can benefit from this optimization.

3.3.7 Abstract Algorithm Relaxation Through Introduction of I/O Channels

Some abstract algorithms that need to be converted to a distributed programs do use I/O combinations (e.g. when the atomic property of I/O combinations is needed to prove algorithm properties). While the atomic property is useful in proving correctness, it may have a severe performance penalty. When the algorithm is converted using the framework presented in this chapter, the performance of the target program may be adversely affected due to the costs imposed by the synchronization of communications needed to preserve atomicity (Section 3.3.4) and the overhead of the reservation system (Section 3.3.5).

When appropriate, we can relax the abstract algorithm by replacing I/O combinations with asynchronous channels. The channels are then optimized away during conversion of the algorithm to a distributed program (as in Section 3.3.6). This approach can lead to a significant performance improvement in the target program due to more concurrency. At the same time, the program would no longer be an implementation of the original algorithm, but instead will implement the relaxed channel-based version that may not have the same provable properties. In applications that remain correct under such relaxation, this optimization can be beneficial.

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

In the previous sections of this chapter we presented a framework that is useful in converting abstract algorithms to distributed implementations. We designed a set of C++ objects to complement the framework. The objects encapsulate the common functions of I/O Automata. They have been designed in accordance with the conversion techniques described in Sections 3.2 and 3.3 and are intended to be used as a foundation in converting specific algorithms to programs. This section briefly describes their design.

3.4.1 Components of the Framework Implementation

The overall design goal was to minimize redundant work in converting different I/O Automata-specified algorithms to distributed programs. The design includes four categories of objects:

The base *IOAutomaton* class This class encapsulates components needed in all implementations of I/O Automata. In our implementation this class handles scheduling actions for execution (subject to them being enabled) and the reservation system for I/O combinations.

The base *IOAction* class *IOAction* encapsulates components needed in implementing any locally-controlled I/O Automaton action. We decided to create a separate class to represent such actions rather than encapsulate them in the class that represents an entire automaton. The need for having separate objects representing locally-controlled actions arises from the fact that locally-controlled action scheduling is handled in the *IOAutomaton* class. To schedule actions for execution, *IOAutomaton* needs to be able to test the *Enabled* predicate of the action and call *Transition* to execute the effects clause, knowing nothing about specifics of the I/O Automaton that is built on top of it. *IOAutomaton* works exclusively with the base class representing locally-controlled actions, *IOAction*. In an implementation of a specific I/O Automata-based algorithm the classes for all locally-controlled actions are derived from the base *IOAction* class. C++ polymorphic capabilities (virtual functions) ensure that the *IOAutomaton* class calls the correct *Enabled* and *Transition* code at runtime.

To perform its scheduling task, *IOAutomaton* class requires that *IOAction* class and all specific action classes derived from *IOAction* support the two procedures already familiar to us from Section 3.3.1:

Enabled This method returns a boolean value that indicates whether the action is currently enabled, i.e. if its preconditions are satisfied. If the method returns *true*, it is required to provide its *IOAutomaton* class with local variable values that unambiguously identify the state transition to be performed.

Transition This method executes the effects clause of the local action, using local variable values provided by an earlier call to *Enabled*. Note that *Transition* should never be called without a call to *Enabled* immediately preceding it.

The argument to both methods is the object representing the automaton containing the action. This argument is needed so that the action object has access to its automaton's state variables, plus (in the case of *Transition*) the values for the local variables that have been selected by *Enabled*. The scheduler contained in the base *IOAutomaton* class checks the preconditions clause by calling *Enabled* and, if all preconditions are satisfied, executes the effects clause by calling *Transition*.

The derived I/O Automaton class A specific I/O Automaton is represented by a class derived from the base *IOAutomaton* class. The state of the derived class consists of a representation of the automaton state and an instance of each locally-controlled action. The derived class handles initialization of the automaton state, processes input actions, and calls the base class's scheduler to invoke locally-controlled actions.

Recall that input actions are not under control of the I/O Automaton containing them. As such, they are not controlled by the base *IOAutomaton* class's scheduler. It follows that the base *IOAutomaton* class does not need to know anything about the input actions. So in our design the derived I/O Automaton object representing a specific automaton takes care of processing its input actions directly by calling a method that implements the input action's effects clause.

The derived I/O Action class This class overrides *IOAction*'s base versions of *Enabled* and *Transition* for every action with new versions that do processing specific to a particular action. As we mentioned before, C++ polymorphic features ensure that the correct version of the method gets called by the base *IOAutomaton* class at runtime.

3.4.2 Execution Scheduling

The target program process running a component automaton handles locally-controlled actions by calling the base class scheduler to execute them. It also looks for incoming messages from other automata and from the external environment. When such a message arrives, the process dispatches it to the appropriate input action procedure.

We implemented a random action scheduler and a round-robin action scheduler for the *IOAutomaton* class. If the source algorithm requires more sophisticated scheduling semantics, the scheduler can be re-implemented in the derived I/O Automaton class. Receipt of messages initiating input actions is always scheduled in the derived I/O Automaton class.

3.4.3 Notes on Implementing Deadlock Avoidance

In the deadlock avoidance scheme in Section 3.3.5 processes cannot grant reservations when they are waiting for a response to their own reservation request. This can lead to contention among system processes and result in livelock: processes would repeatedly request reservations and get rejected by other processes, who are also waiting for responses to reservation requests.

To deal with livelock resulting from contention, we used an exponential backoff scheme [19]. Process P_i maintains a variable $q_{i,j}$, an interval of time that P_i waits between sending reservation requests to process P_j $(i \neq j)$. P_i doubles the value of $q_{i,j}$ after each rejected reservation request to P_j, and adds a random term from a fixed range to $q_{i,j}$. Exponential backoff reduces contention by reducing the time P_i spends trying to get a reservation from busy processes. This makes P_i available to grant more reservation requests itself.

Chapter 4

Design and Implementation of Experimental ESDS Systems

Using the framework and the classes implementing the I/O Automata foundation from Chapter 3, we created distributed implementations of two versions of the unoptimized abstract ESDS algorithm, *ESDSAlg* [1] and *SimpleESDSAlg*. We also created and implemented an optimized version *ESDSOptAlg* of the abstract algorithm that incorporates some of the optimizations necessary to produce a more practical implementation of ESDS.

For reference, we provide a description of *ESDSAlg* component automata from the ESDS paper [1] in Figures 4.1 and 4.2. To avoid a complete restatement, we refer the reader to the paper for a detailed description of the algorithm.

This chapter describes the design of our implementations *ESDSImpl*, *SimpleESDSImpl*, and *ESDSOptImpl*. Section 4.1 gives an overview of the major design goals and methods used to achieve them. *ESDSImpl* and *SimpleESDSImpl* are described in Section 4.2 (their designs are similar). Section 4.3 deals with the optimized implementation *ESDSOptAlg*. Finally, three specific data service applications that were built on top of the ESDS implementation are described in Section 4.4.

4.1 Overview

The main design goal was to make *ESDSImpl* completely independent of the data object that implements the serial datatype. The design carries through the idea presented in [1]

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 4.1: *ESDSAlg*: Automaton for frontend f

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"
$minlabel_r : \mathcal{O} \rightarrow \mathcal{L} \cup \{\infty\}$; the smallest label r has seen for $x \in \mathcal{O}$
Derived from $done_r[r]$ and $minlabel_r$: $val_r : done_r[r] \rightarrow V$; 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: $x \in pending_r$
$x \in done_r[r]$
$x.strict \implies x \in \bigcap_i solid_r[i]$
$v = val_r(x)$
$f = frontend(client(x.id))$
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 4.2: *ESDSAlg*: Automaton for replica r

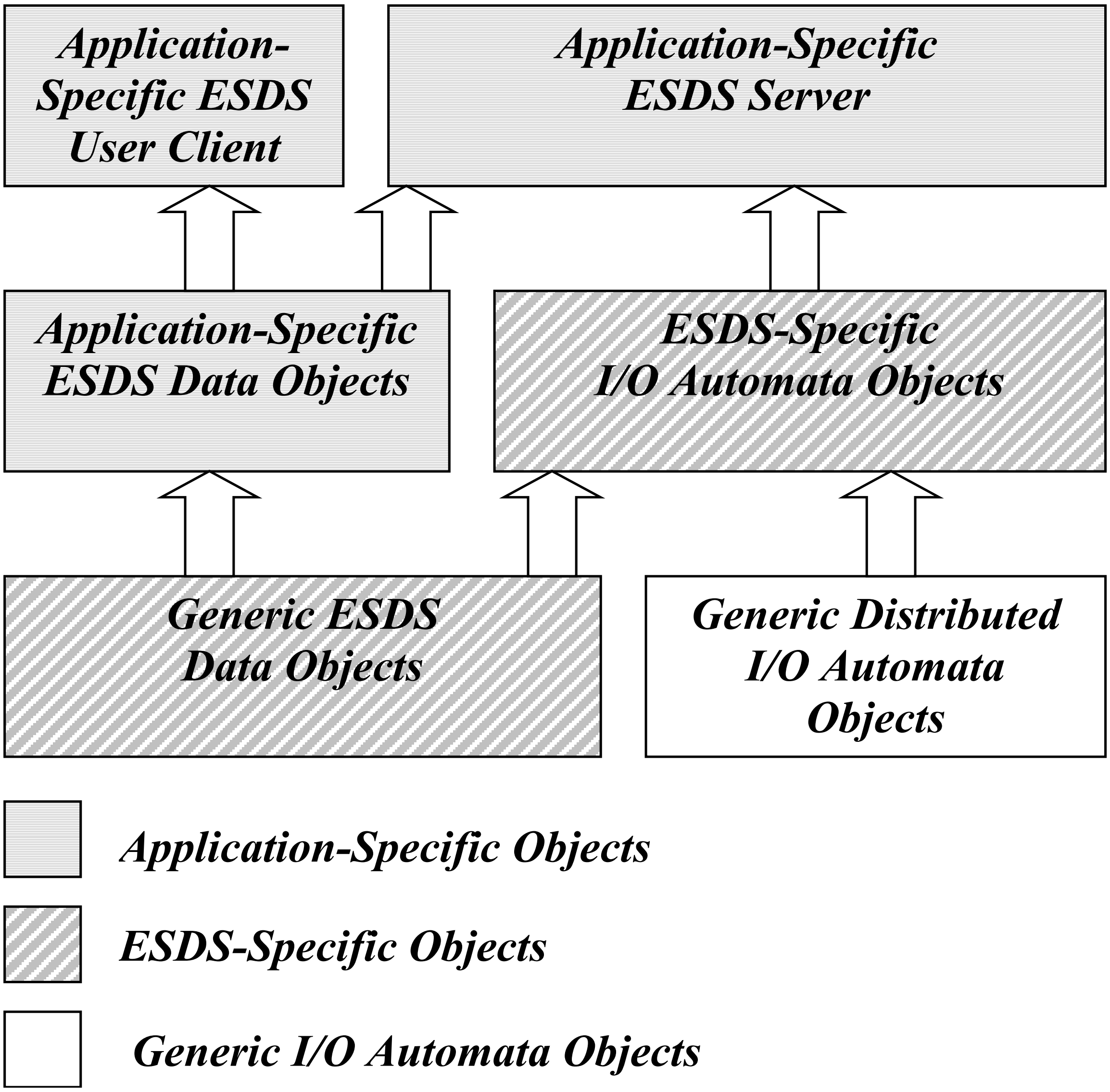

Figure 4.3: *ESDSImpl* Structure

that the ESDS components need to be designed and implemented only once. *ESDSImpl*'s ESDS components can be used by an application programmer as building blocks for any type of data service. All that is required of the programmer is to implement the data object and add to it the few methods needed to make it work with ESDS.

Figure 4.3 depicts the hierarchy of the objects that comprise the system. Arrows in Figure 4.3 represents 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, as described in Section 3.4. The ESDS-specific objects implement *ESDSAlg* [1]. 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 4.4 illustrates the mapping of distributed components of ESDS to system processes. In the figure MPI nodes are represented by circles. Each MPI node runs as a single system process. Non-MPI processes are represented by rectangles. A more detailed discussion of the mapping is found in Section 4.2.6.

4.2 Implementing ESDS: ESDSImpl and SimpleESDSImpl

In this section we present the high-level design considerations and key low-level details of *ESDSImpl* and *SimpleESDSImpl*. The main design goal was to demonstrate that the ESDS algorithm is suitable for implementation as a building block from which a variety of concrete applications can be build with minimal effort. We now give low-level details about the C++ structure of the *ESDSImpl* program, ESDS-specific and application-specific objects data representation, *ESDSImpl*-specific scheduler, and the *ESDSImpl* runtime environment.

An abstraction layer is required to separate ESDS-specific code from application code. The application object supports a standard interface that forms this abstraction boundary. We used the round-robin algorithm to schedule locally-controlled and input actions in *ESDSImpl*.

Top level design of *ESDSImpl* has four major components: application object, ESDS operation, replica automaton, and the frontend automaton.

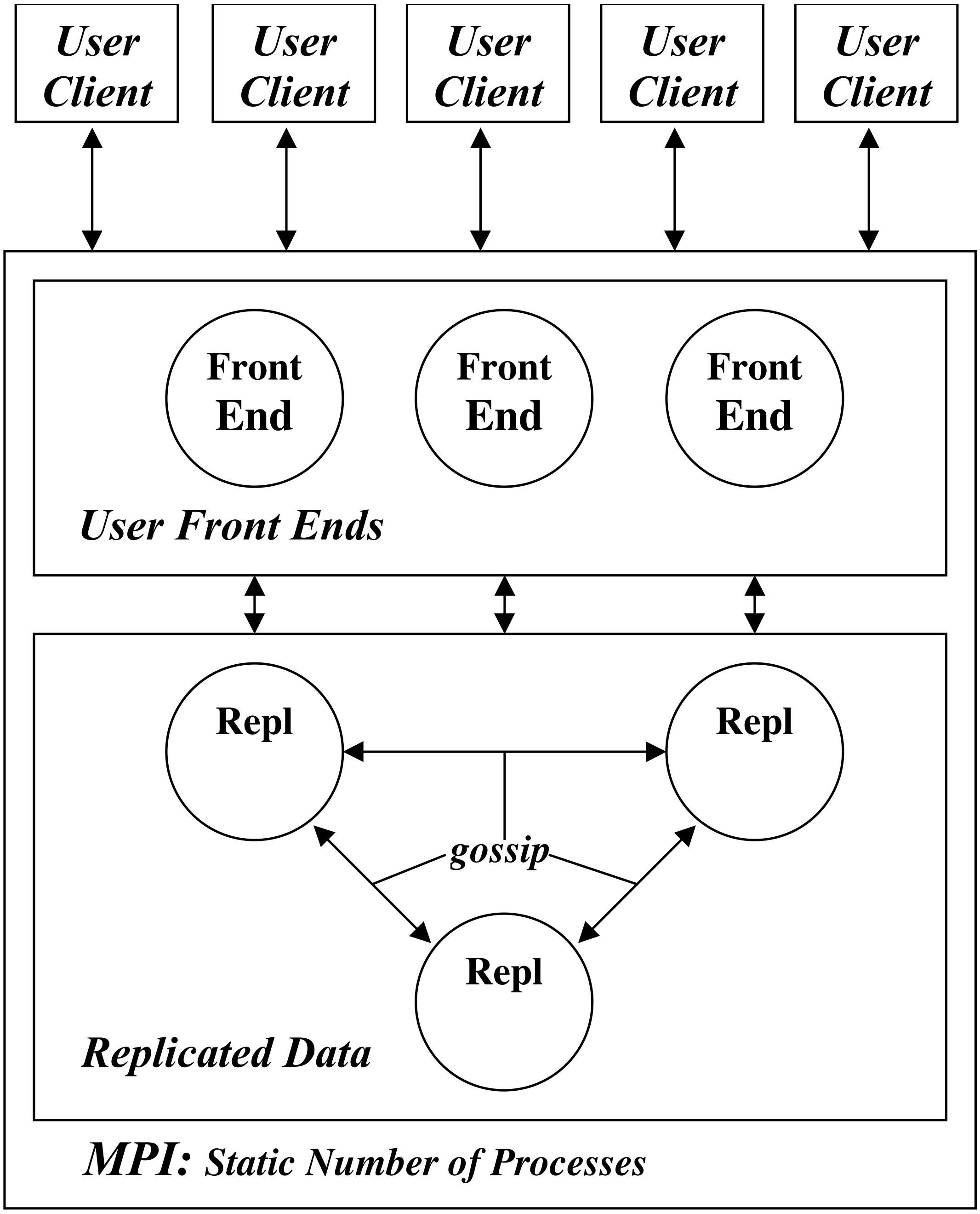

Figure 4.4: *ESDSImpl* Processes

4.2.1 Application Object

ESDSAlg does not place any restrictions on the serial data type of the application object. An implementation of the application data type object needs to support a special interface to be compatible with *ESDSImpl*. We do not publish the details of the interface in this work, but describe it informally below.

The prototype provides three base classes from which the application classes are derived: *ESDSApplicationState*, *ESDSApplicationOp*, and *ESDSApplicationValue*. An application-specific class derived from *ESDSApplicationState* represents the application state maintained by the data service. *ESDSApplicationState* provides routines for instance construction and destruction, and for packing and unpacking the object into binary representations. The packing and unpacking procedures are called when the application state needs to be communicated to other processes in the distributed environment.

An application-specific class derived from *ESDSApplicationOp* represents all operations that the application datatype supports. It must support the same methods as the *ESDS-ApplicationState*-derived class, plus an *Apply* method that takes the application state object as an argument and changes its state according to the semantics of the operation. The *Apply* method must generate and return a value for the operation, in the form of an *ESDSApplicationValue*-derived object.

An application-specific class derived from *ESDSApplicationValue* represents the range (or the set of return values) of the application operations. It must support the same methods as the *ESDSApplicationState*-derived class. *ESDSImpl* returns a value for a submitted operation as an instance of *ESDSApplicationValue*.

4.2.2 ESDS Operation Object

An ESDS operation object represents a single request submitted by the user to the system. The object encapsulates all the information about the user request and all the bookkeeping information about the operation's status in the system. Below is a description of the ESDS operation object's state components.

Operation Descriptor The operation descriptor corresponds to the operation descriptor specified in [1]. The descriptor has the following components:

- an operation identifier *id* that is unique for the current invocation of the system. In *ESDSAlg* the identifier contains a reference to the frontend that originated the operation. We implement this by including a separate component *sender* in the descriptor. *Sender* identifies the originating frontend for the operation.
- a set *prev* of operation identifiers that indicates which operations must precede the owner of the descriptor in the order of application to data object state.
- a boolean flag *strict* that specifies whether the responses to the operation must be consistent with the eventually established serialization.
- an object *appl* that contains application-specific information about the operation. The information includes the operator that must be applied to the current data object, along with supporting parameters and data for the operator. The application programmer derives the class of *appl* from *ESDSApplicationOp*.

The *id*, *prev*, and *strict* descriptor components represent identically named descriptor components in *ESDSAlg*. The *appl* descriptor component represents the *op* descriptor component in *ESDSAlg*.

When deciding on the choice of representation for operation identifiers, we looked at several considerations. Since *ESDSAlg* devolves the responsibility of assigning operation identifiers and ensuring their uniqueness on the clients of the system, there needed to be a way to do this without consulting the rest of the system, and therefore without any *a priori* knowledge of the identifiers that have already been used for other operations. Another consideration is the size of the identifier, which affects the size of gossip messages and memory requirements in *ESDSImpl*. We settled on the 128-bit universally unique identifier (UUID) scheme defined by OSF/DCE [20]. This scheme allows the client application to pick an identifier using only the resources available on the local machine. The identifier was implemented in an object-oriented manner to allow other representations to be substituted. In particular, for the purposes of running empirical tests we implemented identifiers as integers issued in sequence.

Operation Minlabel *ESDSAlg* represents the order in which user operations are applied by a function *minlabel* from the set of all operations $\mathcal{O}$ to some well-ordered set $\mathcal{L}$. The minlabel function is implemented by assigning a *minlabel* state component to each operation.

Minlabels are represented as pairs of integers (*counter-value*, *rid*). The *rid* component is the unique numerical identifier of the replica that created the minlabel. Each replica keeps a counter value which it assigns to the first component of newly created minlabels. The counter is incremented each time the replica creates a new minlabel, ensuring uniqueness of minlabels. The order on minlabels is the same as the order of their *counter-value* component, with ties broken by the order of the *rid* component.

Operation Value This is an application-specific object used to represent the value that *ESDSImpl* returns to the user after the operation is completed. The type of the object is a class derived from *ESDSApplicationValue*. When a replica performs the user operation, it computes this value and passes it along with the operation identifier to the frontend. The frontend stores the value until it is ready to give the response to the user who submitted the operation.

Sets of Operations The frontend automaton and the replica automaton in *ESDSAlg* group user operations into sets as specified by the algorithm. The majority of *ESDSAlg*'s actions deal with a single operation. We represent the sets of operations in *ESDSAlg* as doubly-linked lists. The links reside inside the operation objects themselves. This arrangement has advantages over explicit set representation with respect to the operations most frequently performed by the algorithm.

Remark: When the program obtains a reference to an operation, inserting, deleting, and testing for membership in a set requires $O(1)$ time with respect to all sets that the operation might belong to. The initial search for an operation in a set still requires $O(n)$ time in the doubly-linked list set representation, so the design is open to the possibility of replacing linked lists with more efficient data structures. However, we did not attempt to optimize this data structure, since simplicity of the implementation was an important factor in our work.

In addition to representing algorithm state components, *ESDSImpl* maintains bookkeeping information.

Front End bookkeeping Front ends maintain counters for each operation that indicate how many times the operation was sent to each replica. In *ESDSAlg* frontend is allowed to send an operation to (nondeterministically chosen) targets arbitrarily many times until it receives a response. In a practical system it is desirable to limit the number of such requests to reduce the amount of unnecessary communication. This limit depends on a number of factors. If the system is under light load and replicas can provide fast responses to an operation, it may be sufficient for a front end to submit each operation only once. If the system is under heavier loads, or if some replicas are slow to respond, the frontend can benefit by submitting the operation to several replicas in hope of a faster response. Another factor affecting the submission pattern is communication reliability. In a reliable network that guarantees delivery of messages, it is unnecessary to submit the operation more than once to any single replica. However, in an unreliable system more than one submission may be necessary before a replica receives the operation.

The prototype design allows us to experiment with all of these behaviors. (see Section 6.3 for future work suggestions).

4.2.3 Replica Automaton Design

The class representing a replica automaton is built on the *IOAutomaton* class discussed in Section 3.4.1. In *ESDSImpl* the replica implementation corresponds to the automaton presented in [1]. The replicas are numbered from 0 to $N-1$, with N replicas participating in the system. The state of each replica includes this number as the replica identifier.

Each of the $pending_r$, $rcvd_r$, $done_r(i)$, and $solid_r(i)$ sets in *ESDSAlg* is implemented by linking all operations belonging to one set into a circular doubly-linked list.

Replicas assign unique minlabels to operations as follows. Each replica keeps a counter variable $lbl\text{-}counter_r$. When replica r does an operation, it assigns minlabel $(lbl\text{-}counter_r, r)$ to the operation. Using replica identifiers guarantees system-wide uniqueness of minlabels.

In *ESDSAlg* a gossip message from replica r consists of the $minlabel_r$ function and the entire sets $rcvd_r$, $done_r[r]$, and $solid_r[r]$. In *ESDSImpl* the corresponding gossip message consists of all the operations in the $rcvd_r$ set. A gossip message includes boolean flags that indicate which sets each operation belongs to.

Otherwise, the basic implementation of a replica corresponds to the *ESDSAlg* replica automaton code [1].

4.2.4 FrontEnd Automaton Design

The implementation of the frontend automaton in *ESDSImpl* follows the frontend automaton code in *ESDSAlg* [1].

4.2.5 Application Clients

In *ESDSAlg* system users interacting with the data service frontends are represented as application clients. The application programmer is free to choose how the client should be implemented. Our design of *ESDSImpl* specifies only the mechanism for communication between clients and frontends and the protocol that the clients use to submit operations and receive responses from frontends. Section 4.2.7 discusses the choice of communication mechanism for *ESDSImpl* clients.

4.2.6 Mapping Component Automata to System Processes

ESDSImpl and the other ESDS implementations run on a network of Sun workstations using the MPICH implementation of MPI [17]. The prototype is based on MPI Standard 1.1 [21].

There is a deficiency in the MPI standard version 1.1 and the MPICH library that limited implementation choices. This version of the standard does not allow dynamic management of processing nodes. The number of available processes is determined statically at invocation and cannot change during execution. For the purposes of a distributed data service, this means that application clients, which need to be created and destroyed dynamically, cannot be integrated in the MPI framework. *ESDSImpl* sidesteps this issue by using Berkeley Sockets instead of MPI mechanisms for communication between application clients and ESDS frontends. At the time of this writing, the work on the next version of the MPI standard and the MPICH implementation includes the dynamic process management capability. It is not known whether this capability in the next version of MPI can be used with *ESDSImpl*.

The limitations of the current MPI standard dictated the mapping of ESDS components to system processes depicted in Figure 4.4. 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 4.4). The rest are divided between ESDS replicas and frontends. The administrator specifies how many nodes to allocate for each use.

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 system users. Clients use sockets to connect to one of the frontends. When the connection is established, the client can submit an operation to the frontend and receive a response when it is available.

4.2.7 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. Application clients and frontends communicate via I/O combinations.

We implemented two different systems of communication among frontends and replicas. The first version implements communications in *SimpleESDSImpl*. It is produced using the techniques for converting I/O combinations to distributed programs (see Section 3.3.4).

The second version implements communications in *ESDSImpl*. This version 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 3.3.7. *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 is free to execute asynchronously, thus taking 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 program's configuration file.

4.3 Implementing ESDS: ESDSOptImpl

In addition to implementing *ESDSAlg*, we implemented some of the optimizations suggested in the ESDS paper [1]. In this section we describe the implementation of the optimizations. We also present an I/O Automaton for the optimized ESDS replica.

Section 4.3.1 presents abstract descriptions of the optimizations that have been applied to *ESDSAlg* to produce *ESDSOptAlg*. At the end of the section we present the updated replica I/O Automaton. Most *ESDSOptImpl*'s design is identical to *ESDSImpl*, and the implementation of the differences is straightforward and lacks interesting features. Therefore, we do not present design details for *ESDSOptImpl*, as we did for *ESDSImpl*.

4.3.1 Abstract Description of Optimizations

We describe the optimizations to *ESDSAlg* included in *ESDSOptAlg* and present the revised replica automaton.

Incremental Gossip

ESDSAlg is specified in terms of identical servers, each of which contains an object replica. Replica r periodically sends entire $done_r[r]$ and $solid_r[r]$ sets to other replicas in gossip messages (Fig. 4.2). Thus, a typical gossip message contains a lot of information that has been gossiped previously between the same two replicas. Furthermore, the amount of such redundant information increases linearly with the number of new operations. *ESDSImpl*, as an 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 gossip only new information. This change improves system performance, but reduces the system's ability to tolerate lost gossip messages.

Remark: Explicit sequencing of gossip messages combined with retransmission and removal of duplicates is needed to make the optimization work with *unreliable* channels that allow message losses, duplicate messages, and out of order delivery.

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, thus violating the requirement of eventual serializability. This optimization adds another action 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.

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 [1]). *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.

Our optimized implementation uses a variation of the stable-state optimization suggested in the ESDS paper [1]. It adds a state component to each replica that keeps track of the stable state, which is the result of applying all completely ordered operations 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. This optimization is a part of *ESDSOptImpl*.

The computation of the new stable state takes place every time replicas receive gossip messages (see Fig. 4.5). 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 $max\text{-}stable_r$'s minlabel 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 $max\text{-}stable_r$'s minlabel to the old stable state to compute the new stable state.

Note that our version of the stable state optimization differs from the scheme presented in the ESDS paper [1]. We apply operations to the stable state of a replica as soon as they have stabilized at that replica, whereas the ESDS paper version of the optimization waits until the operation stabilizes globally before applying it. We conjecture that our version of the optimization results in faster stabilization of operations and a corresponding increase in performance.

Remark: This optimization makes it possible to discard almost all information about the operations as soon as they enter the stable state. In *ESDSOptImpl* operation identifiers are kept around forever because they may enter the *prev* sets of future operations. However, if this optimization is combined with the multipart timestamp optimization (see Section 6.1), even the operation identifiers may be discarded. We have not implemented this.

4.3.2 Optimized Replica I/O Automaton

This section formalizes the optimizations discussed in the previous section. It presents a modified version of the *ESDSAlg* replica automaton [1], reflecting the optimizations in Section 4.3.1. The optimized replica automaton is presented in Figure 4.5.

The modified replica automaton r maintains a $gossip_r[i]$ state variable in addition to other state variables from the *ESDSAlg* replica automaton. The $send_{r,r'}(\langle\text{“gossip”}, R, D, L, S\rangle)$ action that sends a gossip message from replica r to replica r' is enabled only if the $gossip_r[r']$ set is non-empty. When it is enabled, only the operations in the $gossip_r[r']$ set are gossiped. Thus, an operation x needs to be added to the $gossip_r[i]$ set for all i whenever

Data types

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

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"

$gossip_r[i]$ for each replica i, a subset of $\mathcal{O}$; the operations that r needs to gossip to i

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

Derived from $solid_r[r]$ and $minlabel_r$: $max\text{-}stable_r \in solid_r[r]$ s.t. $\forall y \in solid_r[r]$, $minlabel_r(max\text{-}stable_r) \geq minlabel_r(y)$

$stable\text{-}state_r \in \Sigma$, initially σ_0; the state resulting from doing all the operations up to and including $max\text{-}stable_r$

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

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: $pending_r \leftarrow pending_r \cup \{x\}$
  - $rcvd_r \leftarrow rcvd_r \cup \{x\}$
  - $gossip_r[i] \leftarrow gossip_r[i] \cup \{x\}$ for all i

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: $done_r[r] \leftarrow done_r[r] \cup \{x\}$
  - $minlabel_r(x) \leftarrow l$
  - $gossip_r[i] \leftarrow gossip_r[i] \cup \{x\}$ for all i

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

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

Internal $solidify_r$
- Pre: $|P| = 1$
- Eff: $solid_r[r] \leftarrow solid_r[r] \cup (\bigcap_i done_r[i])$
  - for y s.t. $minlabel_r(y) \leq minlabel_r(max\text{-}stable_r)$ and $stable\text{-}value_r(y)$ is undefined, in $minlabel_r$ order:
    $(stable\text{-}state_r, stable\text{-}value_r(y)) \leftarrow f(stable\text{-}state_r, y.op)$

Input $receive_{r',r}(\langle \text{"gossip"}, R, D, L, S\rangle)$
- Eff: $gossip_r[i] \leftarrow gossip_r[i] \cup (R - rcvd_r) \cup$
  $\cup (S - (\bigcap_j done_r[j])) \cup$
  $\cup (S - (solid_r[r] \cap solid_r[r])) \cup$
  $\cup (D - (done_r[r] \cap done_r[r])) \cup$
  $\cup \{x : minlabel_r(x) > L(x)\}$
  for all i
  - $rcvd_r \leftarrow rcvd_r \cup R$
  - $done_r[r'] \leftarrow done_r[r'] \cup D \cup S$
  - $done_r[r] \leftarrow done_r[r] \cup D \cup S$
  - $done_r[i] \leftarrow done_r[i] \cup S$ for all $i \neq r, r'$
  - $minlabel_r = \min(minlabel_r, L)$
  - $solid_r[r'] \leftarrow solid_r[r'] \cup S$
  - $gossip_r[i] \leftarrow gossip_r[i] \cup$
    $\cup ((\bigcap_j done_r[j]) - solid_r[r])$ for all i
  - $solid_r[r] \leftarrow solid_r[r] \cup S \cup (\bigcap_i done_r[i])$
  - for y s.t. $minlabel_r(y) \leq minlabel_r(max\text{-}stable_r)$ and $stable\text{-}value_r(y)$ is undefined, in $minlabel_r$ order:
    $(stable\text{-}state_r, stable\text{-}value_r(y)) \leftarrow f(stable\text{-}state_r, y.op)$

Figure 4.5: *ESDSOptAlg*: Automaton for optimized replica r

replica r has new information about x. The $gossip_r[i]$ state variables get updated inside $receive_{f,r}(\langle\text{"request"}, x\rangle)$, $do\_it_r(x, l)$, and $receive_{r,r}(\langle\text{"gossip"}, R, D, L, S\rangle)$ actions.

Since a replica r can update its $solid_r[r]$ set only when receiving a gossip message, the algorithm behaves incorrectly in the case when there is only one replica if we remove self-gossip messages. As we want to compare single-replica performance with multiple-replica performance in the empirical tests, the single-replica execution of *ESDSOptImpl* must be correct. We add a new internal action $solidify_r$, which corrects the problem by making updates to the $solid_r[r]$ set independently from gossip actions (see Figure 4.5). We omit performing the $solidify_r$ action whenever there are two or more replicas. This optimization is worthwhile since it restricts the need for potentially costly set operations required by the $solidify_r$ action.

The optimized automaton contains memoization of stable state. Three new state components are added to the replica automaton. The $stable\text{-}state_r$ and $stable\text{-}value_r$ components represent, respectively, the current stable state of the automaton and the stable values of the operations that enter $stable\text{-}state_r$. These state components are identical in function to the same components in the stable state memoization code presented in the ESDS paper. The significance of the third new state component, $max\text{-}stable_r$, and the procedures for maintaining $stable\text{-}state_r$ and $stable\text{-}value_r$ were described in Section 4.3.1.

As explained in Section 4.3.1, our version of the stable state memoization optimization is different from the one presented in the ESDS paper. The complete code given in Figure 4.5.

4.4 Applications

This section describes the three data service applications that were implemented.

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.

The advantage of the String Concatenation Service is the simplicity of its implementation. It was used for testing *ESDSImpl* and *ESDSOptImpl* during development and for running empirical measurements of *ESDSOptImpl* performance.

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 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 the next chapter, it does not).

Distributed Spreadsheets

The purpose of creating a third, more sophisticated client was to demonstrate the viability of ESDS as a platform for creating diverse and capable data service applications. This application was constructed as a proof-of-concept. The Distributed Spreadsheets client makes use of ESDS capabilities to create an environment where several people can simultaneously enter spreadsheet data into the same Microsoft Excel workbook. Their additions get sent to ESDS replicas, which maintain the current state of the workbook and can refresh each user's copy on demand. One 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 people's additions with your own.

Chapter 5

Empirical Testing and Analysis

5.1 Test System Configuration

All performance tests were done on a 10 Mbps Ethernet LAN of 12 Sun workstations running SunOS 4.1.4. The tests were performed using the *ESDSOptImpl* implementation.

The workstations we used were not dedicated to this project, and their loads fluctuated with time. To account for the variance in test results due to this factor, we performed each test 10 times and averaged the results to minimize the variance due to other tasks running concurrently with the tests. In testing our implementation, we ran it with over 20 replicas. However, for performance testing we used up to 10 replicas only. This allowed us to run performance tests in a setting where a replica corresponded to a networked processor. Due to limited time available for testing, we limited the number of operations submitted to the system to 300 for each test run.

5.2 Definitions

We measured two performance characteristics of the prototype: (1) average response time, and (2) average throughput.

Definition 1: The *response time* for an operation is the elapsed time between submission of the operation to a replica and response from the replica.

Definition 2: The system *throughput* is the number of operations the system processes per unit time in a given execution of the implementation

We also wanted 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, a response to a user operation is *inconsistent* if its value differs from the value of the same operation in the eventual total order of operations. Formally,

Definition 3: 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 *to* of all operations. Then $response_r(x, v_x)$ is *inconsistent* iff $v_x \neq val_{to}(x)$.

Definition 4: In a finite execution of the implementation, the *degree of inconsistency* is the percentage of inconsistent responses among all responses to user operations returned by the system during the execution.

5.3 Test Setup

We conducted three series of tests. The purpose of the first series was simply to ensure that *ESDSOptImpl* can, if necessary, run more than one replica on a single processor and still show decent performance. The purpose of the second series was to determine how system performance, characterized by response time and throughput, depends on the number of replicas participating in the computation. Our third series of tests measured the changes in system performance and degree of inconsistency in response to changing percentage of strict operations among the operations submitted to the system.

The first and second series were set up as follows. A total of 12 workstations were available for the testing. One workstation ran the master process. This process was responsible for initializing the system, setting up the test parameters, and submitting a fixed number of non-strict operations to the system. Another workstation ran a front end that distributed the operations to available replicas in a balanced fashion. Ten other workstations ran replicas, with more than one replica per machine if the number of replicas in the execution exceeded 10.

The test software measured three quantities for each run:

1. Average time T_{fe} from the submission of an operation by the frontend to one of the replicas to the receipt of replica response by the frontend

2. Average time T_r from the receipt of an operation by a replica to the replica sending back a response for that operation

3. Total time τ it took the system to process and respond to all 300 operations.

From these measurements we obtained two different measures of response time and one measure of system throughput as follows. For each number of replicas from $N = 1$ to $N = 10$, we averaged the results of 10 runs and computed the average time AT_{fe} it took a frontend to receive a response from a replica after it sent the request message, the average time AT_r it took a replica to process a request after the replica received it, and the average system throughput $AP = 300/\tau$.

In preliminary testing we determined that a single replica can keep up with the user requests if they come approximately once in 30 milliseconds. If the rate of request submission is faster, a single replica gets overwhelmed and cannot keep up. Incoming requests pile up in the MPI message queue, waiting to be processed by the replica. In this case, the average response time to an operation at frontends, T_{fe}, suffers dramatically because the response time depends on how long the operation has to wait in the MPI queue before being received by a replica.

Our third series of tests measured the changes in system performance and degree of inconsistency in response to changing percentage of strict operations. The tests used the same workstation configuration as in the first two series of tests. One workstation ran a front end distributing 300 operations to a constant number of replicas as the percentage of strict operations varied from 0 to 100 in 10% increments.

In the next three sections we present the results of the tests. In analyzing the test results, we are primarily concerned with the performance trends exhibited by *ESDSOptImpl*. We did not seek to minimize the absolute performance numbers. In particular, we made no attempt to run the tests on faster processors, or to use faster networks.

5.4 Test Series 1: Virtual Replicas

This purpose of this test is to demonstrate that it is possible to run *ESDSOptImpl* on a system where the number of available physical processors was smaller than the number of distributed system components. The test increased the number of replicas from $N = 1$

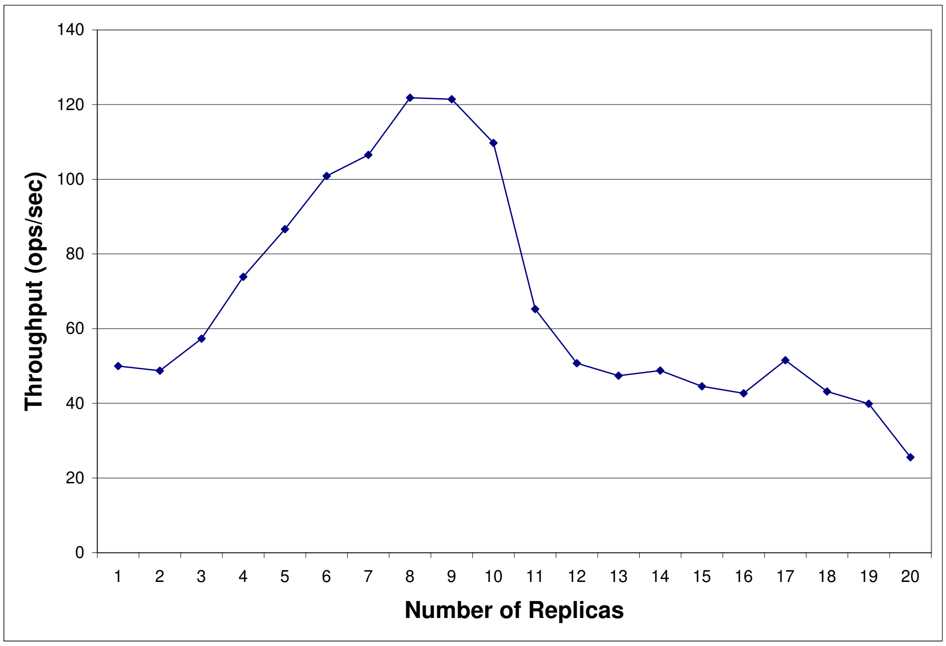

Figure 5.1: System Throughput (submission rate is 330 operations/second)

to $N = 20$ and submitted operations to the system at the constant rate of 330 operations/second. It measured system throughput for each number of replicas. The collected data is plotted in Figure 5.1.

The system's throughput rises as the number of replicas participating in the system and performing submitted operations increases. However, the throughput drops off again at the point where the system runs out of physical processes for replicas (this happens at $N = 10$) and puts additional replicas on processors that already run other replicas. The overhead of context switches and the forced serialization of communications between replicas that share a single processor has an adverse impact on system performance. Therefore, in all other tests we limit the number of replicas to the number of available processors.

5.5 Test Series 2: System Performance

In this section we examine how average system throughput and average response time at replicas (AT_r) and frontends (AT_{fe}) are affected by varying number of replicas and varying rate of submission of new operations.

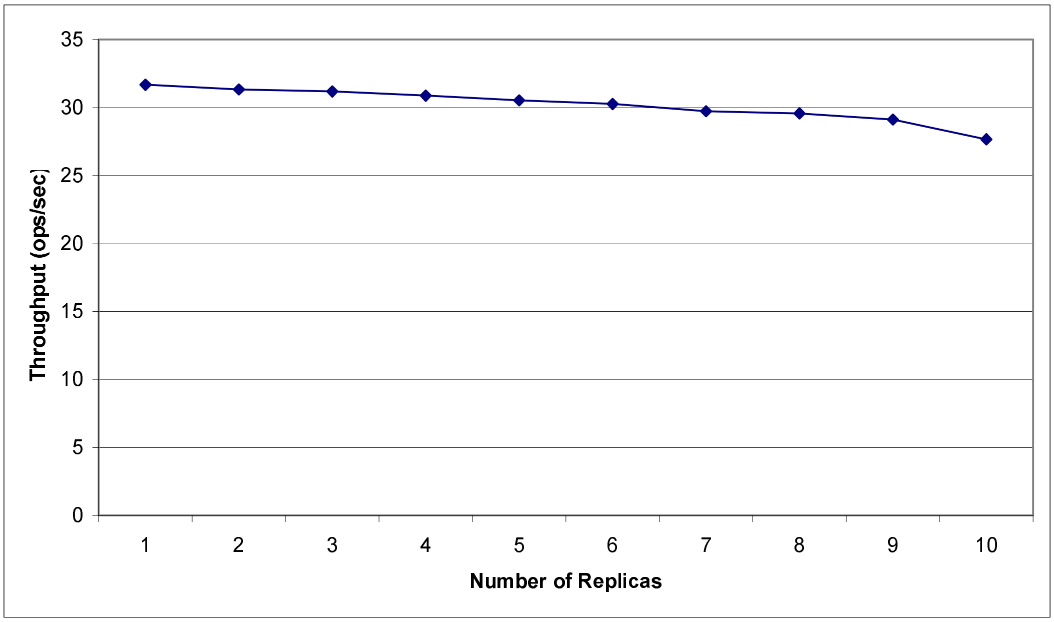

Figure 5.2: System Throughput (submission rate is 33 operations/second)

5.5.1 System Throughput

We expect the following factors to affect system throughput:

- The rate of submission of new operations. This rate is the upper bound on system throughput.
- The number of participating replicas. Each additional replica should increase the total throughput by adding its own capacity to the total capacity. However, the magnitude of the increase in throughput is expected to be adversely affected by the amount of gossiping that replicas need to do.

To verify our hypotheses, we ran the throughput test with three different rates of submission of new requests, each time varying the number of replicas from 1 to 10.

At first the rate of submission was set to one operation every 30 milliseconds, or approximately 33 operations per second. As explained in Section 5.3, at this rate one replica is able to keep up and process all incoming requests without adverse queuing effects. Therefore, we expected that additional replicas would not increase the throughput. The empirical results shown in Figure 5.2 confirm the expectation. The system throughput is close to its

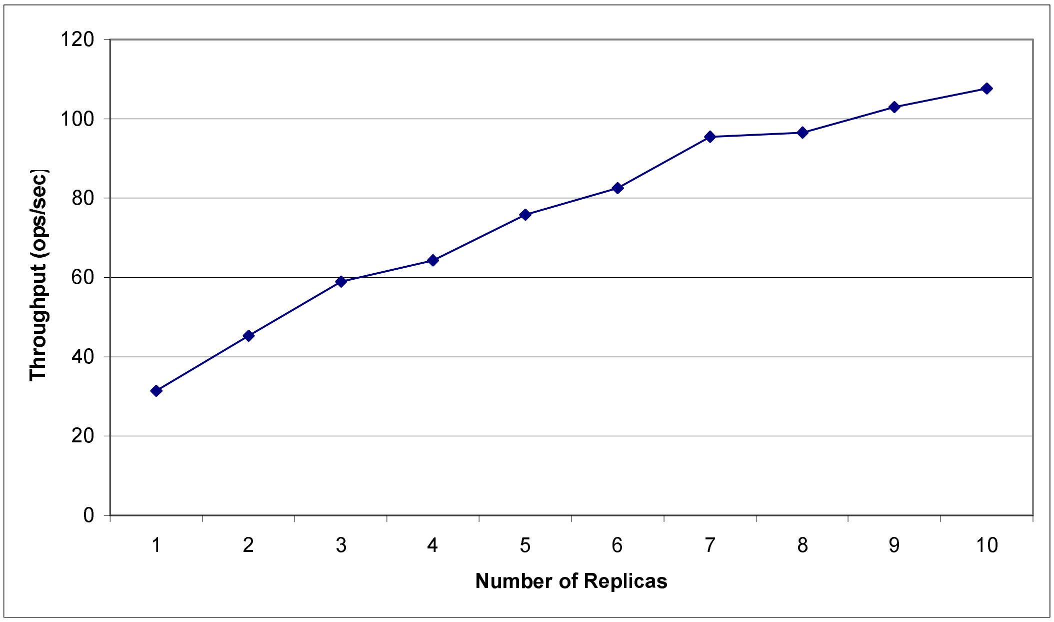

Figure 5.3: System Throughput (submission rate is $33 * N$ operations/second, N is the number of replicas)

theoretical limit of 33 operations per second, and it actually declines slightly as the number of replicas is increased. The decline may be attributable to the increasing gossip overhead.

In the next test setup the rate of submission was set to start at 33 operations/second again for one replica and increase proportionally with the number of replicas, topping out at 330 operations per second for 10 replicas. In this setup we expected the system throughput to rise with the number of replicas. The empirical results are shown in Figure 5.3. The throughput rises nearly linearly with the number of replicas, although it does not come close to reaching its theoretical limit of 330 operations per second. This result suggests that all replicas are working at full capacity and are still unable to keep up with the rate of submission. This might be explained by the increasing gossip overhead.

In the final test of system throughput we set the rate of submission constant again, this time at 330 operations/second. At this point we already know that throughout this test all replicas are working at full capacity. Therefore, we expect an increase in throughput as more replicas join the effort. The empirical results in Figure 5.4 confirm the expectation. Unlike the previous graphs, this time there are secondary effects in the trend. We do not have an explanation, but this could be due to the fluctuating load on the test workstations.

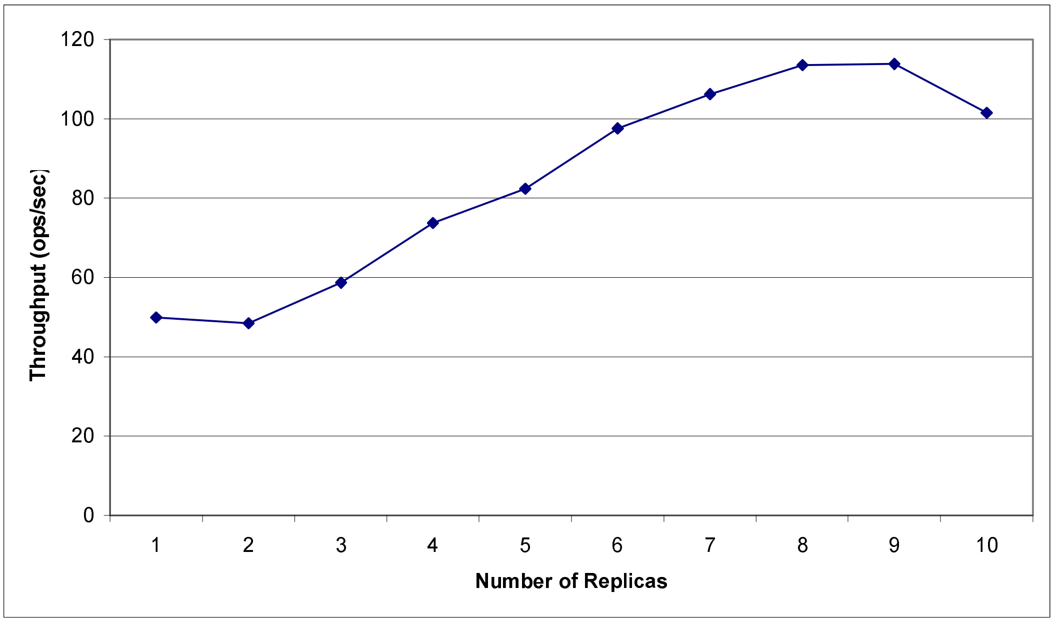

Figure 5.4: System Throughput (submission rate is 330 operations/second)

5.5.2 Response Time

Response Time at Replicas (AT_r)

In *ESDSOptImpl* the scheduling algorithm for replica actions is such that after receiving a non-strict operation from a frontend, a replica immediately applies the operation and sends back a response. The replica does not send or receive gossip messages in the meantime.

Based on this fact, we expect the following factors to affect average response time AT_r at replicas:

- The number of non-stable operations that the replica needs to re-apply to get the value for the new operation (re-application takes a non-negligible amount of time).

- The number of participating replicas. When a small number of replicas are running, operations stabilize faster, decreasing the number of operations that need to be re-applied to get the value of a new operation. With a large number of replicas, actions take a long time to stabilize, since a replica needs every other replica to tell it that the operations is done there before stabilization can occur. Therefore, we expect higher response times as the number of replicas increases.

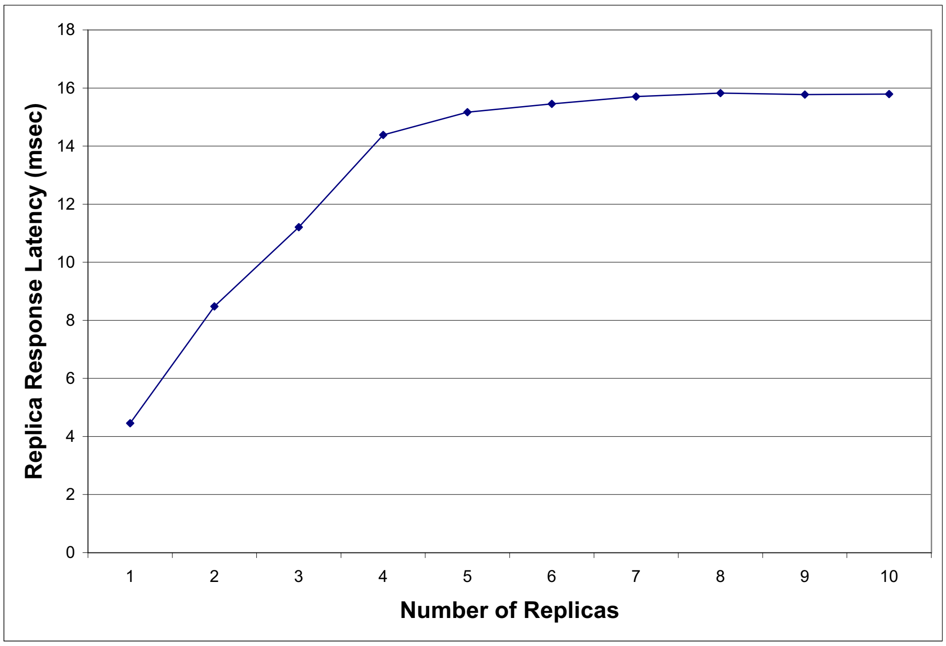

Figure 5.5: Response Time at the Replicas (submission rate is 33 operations/second)

To verify our hypotheses, we tested the average response time at replicas in *ESDS-OptImpl* with three different setups, each time varying the number of replicas from 1 to 10.

In the first test setup the rate of submission of new requests was set at 33 operations/second. Our hypotheses explain the experimental results in Figure 5.5. The response time is small when only one replica is running, since in *ESDSOptmpl* all new operations immediately stabilize through the $solidify_r$ action, meaning that no re-application of old operations takes place when the replica computes the value for the new operation. As the number of replicas increases, operations take longer to stabilize. This increases the number of re-applications of old operations and drives the response time up. The response time levels off at $N = 4$, which means that for $N \geq 4$ virtually no operations manage to stabilize before the end of the test run, meaning that almost all of them need to be re-applied when computing the value of new operations for $N \geq 4$.

For the second test, the setup was identical to the second test in Section 5.5.1. The rate of submission started at 33 operations/second and increased proportionally with the number of replicas. Since we do not identify the rate of submission as having a significant

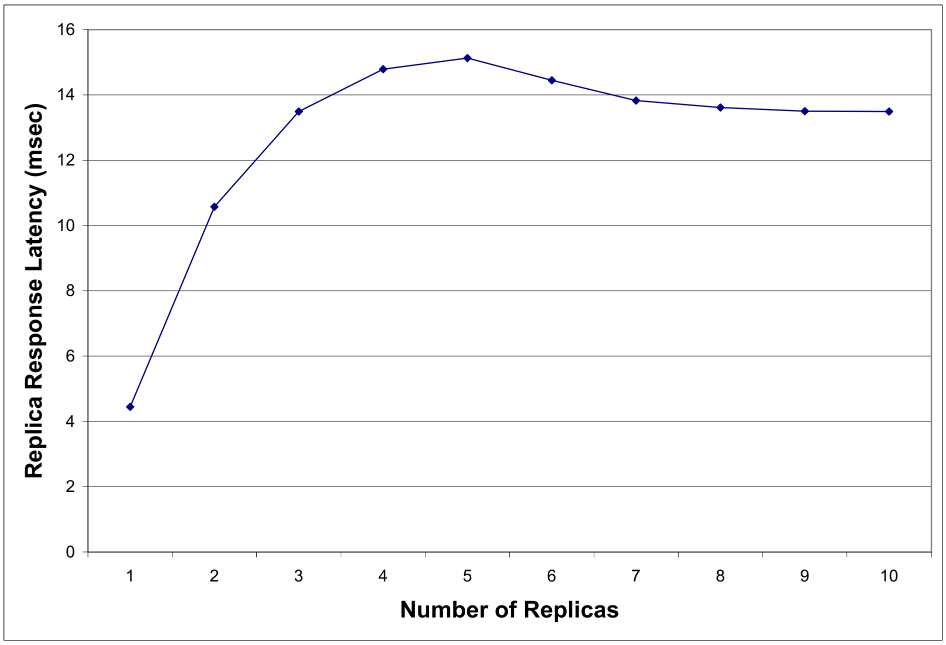

Figure 5.6: Response Time at the Replicas (submission rate is $33 * N$ operations/second, N is the number of replicas)

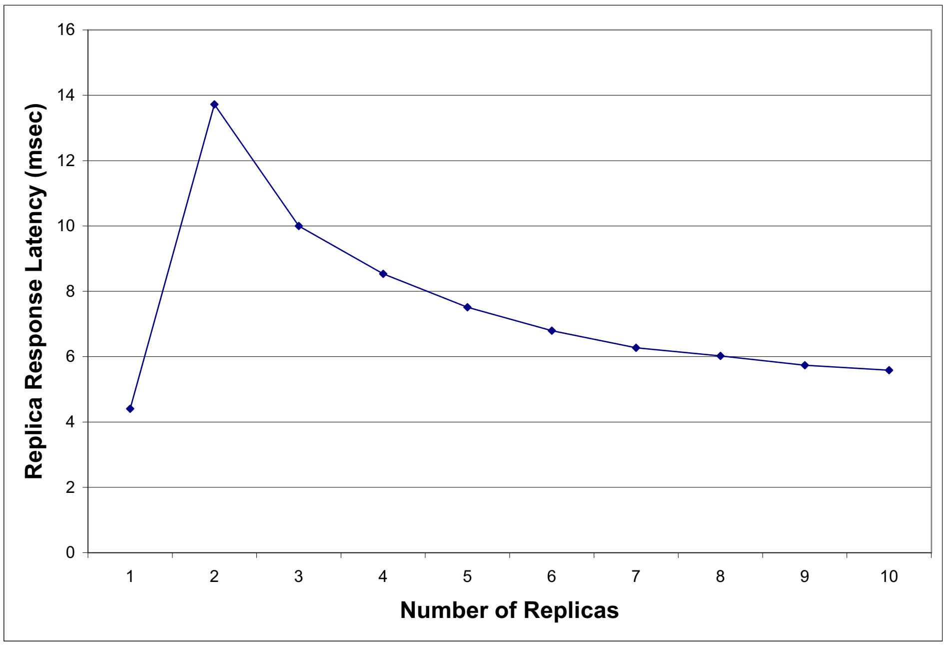

Figure 5.7: Response Time at the Replicas (submission rate is $33 * N$ operations/second, N is the number of replicas, gossip is disabled)

impact on the response time at replicas, the trend for this test was expected to remain the same as it was for the first test in this section. The empirical results in Figure 5.6 confirm that this is so. Together with the second test in Section 5.5.1, this test demonstrates that for larger numbers of replicas it is possible to increase the rate of submission for new operations and achieve better throughput without incurring a penalty in the form of a higher response time at replicas.

The final test in this section duplicates the setup of the second test, except the gossip messages have been disabled. Without gossip, replicas only know about the operations that were sent to them directly by the front end. In this setup, we expect the response time to go up at first as in the previous two tests, but then drop as the number of operations that individual replicas know about and have to re-apply goes down. The empirical results in 5.9 bear out this hypothesis.

Response Time at FrontEnds $\mathbf{AT}_{fe}$

The response time for an operation at a frontend is the sum of the response time for the operation at a replica, the time the request message spends in the MPI channel from the frontend to the replica, and the time the response message spends in the MPI channel from the replica to the frontend.

We therefore expect the factors that were shown to affect the response time at replicas in Section 5.5.2 to also affect the response time at frontends. In addition, the following factors may affect average response time AT_{fe} at frontends:

- The load of replicas and frontends. If the replicas or frontends cannot keep up with incoming messages, the messages lose time waiting in the MPI queue to be received by the process. This increases AT_{fe}.
- The roundtrip time between frontends and replicas. This factor should be negligible in our testing because the network connections between test workstations are fast.

To verify our hypotheses, we tested the average response time at frontends AT_{fe} in *ESDSOptImpl* with the same setups that were used in Section 5.5.2 to test the response time at replicas.

For the first test with the rate of submission of new requests at 33 operations/second we expect AT_r to grow with the number of replicas because of the results in Section 5.5.2 and the average time spent by new request in the MPI channel from frontends to replicas to grow because replicas become busier with a growing number of gossip messages. As a consequence, AT_{fe} should grow with the number of replicas as well. The results in Figure 5.8 confirm these expectations.

The same considerations apply to the second test, where the rate of submission grows proportionally with the number of replicas. In this case the replicas should be even busier than in the first test, and we would expect even longer response times at frontends. The empirical evidence in Figure 5.9 confirms this, but show that AT_{fe} does not exhibit the expected steady upward trend in the response time. The best we can say is that the evidence warrants further exploration to determine additional factors that influence AT_{fe} in this setting. However, the next test indicates that gossip plays a large role in determining AT_{fe}.

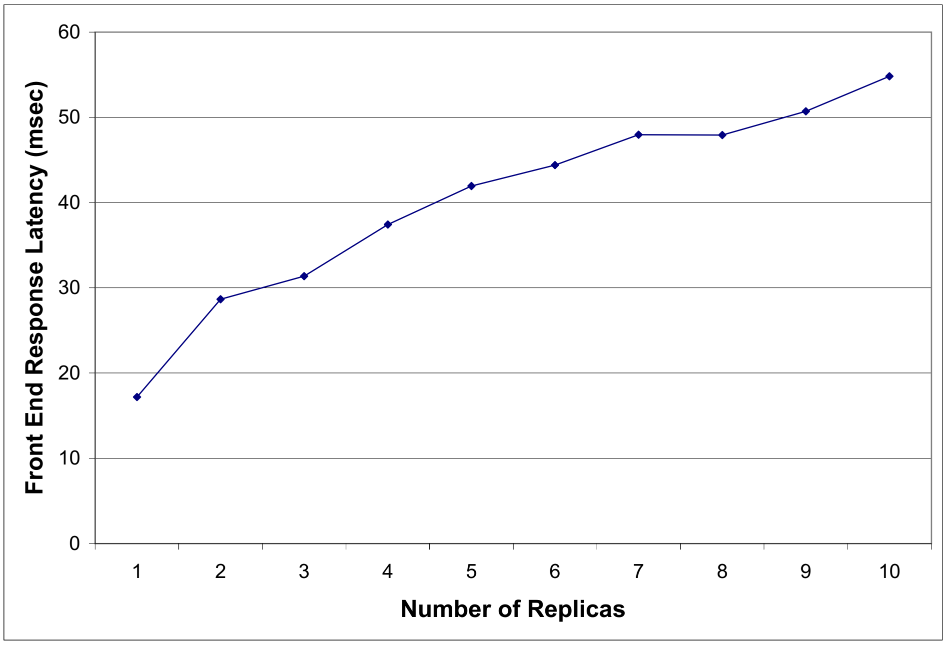

Figure 5.8: Response Time at the FrontEnd (submission rate is 33 operations/second)

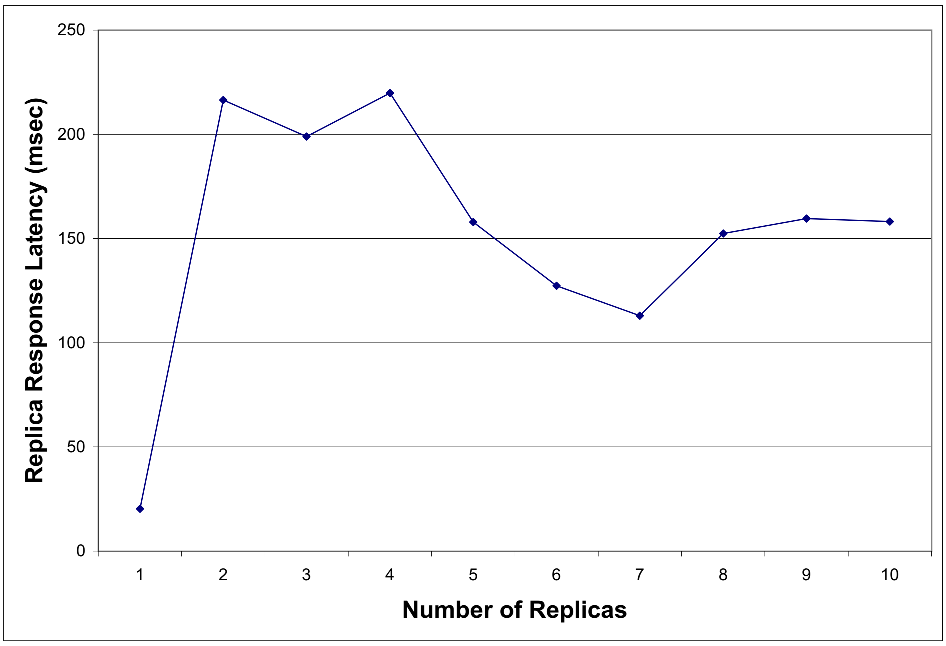

Figure 5.9: Response Time at the FrontEnd (submission rate is $33 * N$ operations/second, N is the number of replicas)

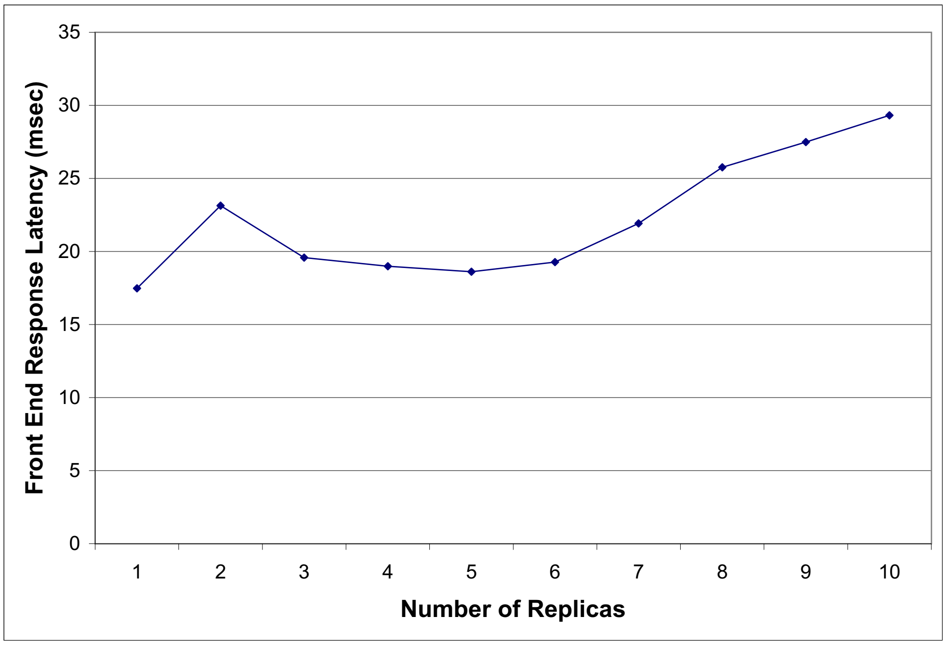

Figure 5.10: Response Time at the FrontEnd (submission rate is $33 * N$ operations/second, N is the number of replicas, gossip is disabled)

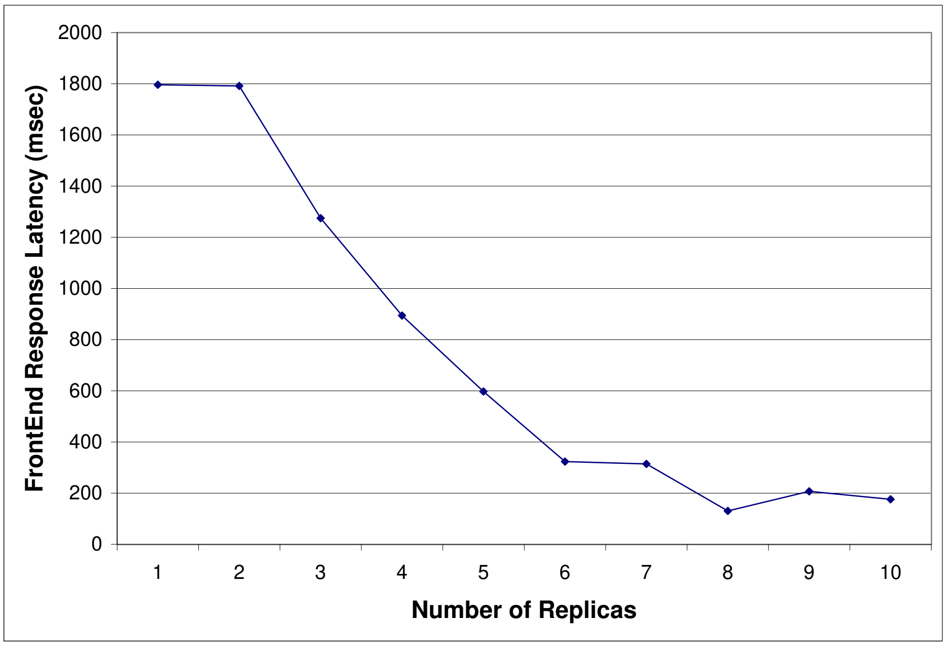

Figure 5.11: Response Time at the FrontEnd (submission rate is 330 operations/second)

The third test has the same setting as the second, except gossip messages are disabled. The results are presented in Figure 5.10. It is evident that without gossip T_{fe} is much smaller than it was when gossip was enabled. In the first part of this graph T_{fe}'s trend is the same as T_r's trend in Figure 5.9: a jump in the beginning, followed by a steady decline. For larger numbers of replicas an upward trend takes over. This trend is due to the increasing rate of submission of new requests, which leads to busy replicas and long queue waits for new requests.

Finally, we run the system with the rate of submission of new requests at 330 operations/second. The results are presented in Figure 5.11. At this high rate replicas are overwhelmed when there are only a few of them. Messages with new requests wait a very long time in the MPI queue before the replicas receive them. Consequently, the response time is very high when the number of replicas is low, but it drops down as more replicas join the system and assume some of the load.

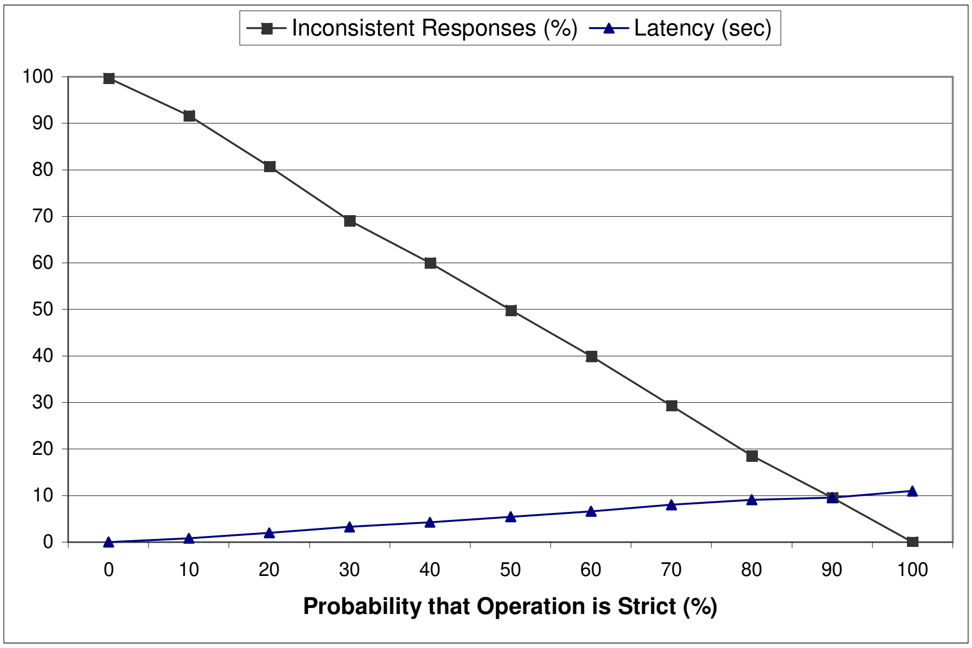

Figure 5.12: Tradeoff Between Response Time and Consistency (2 Replicas)

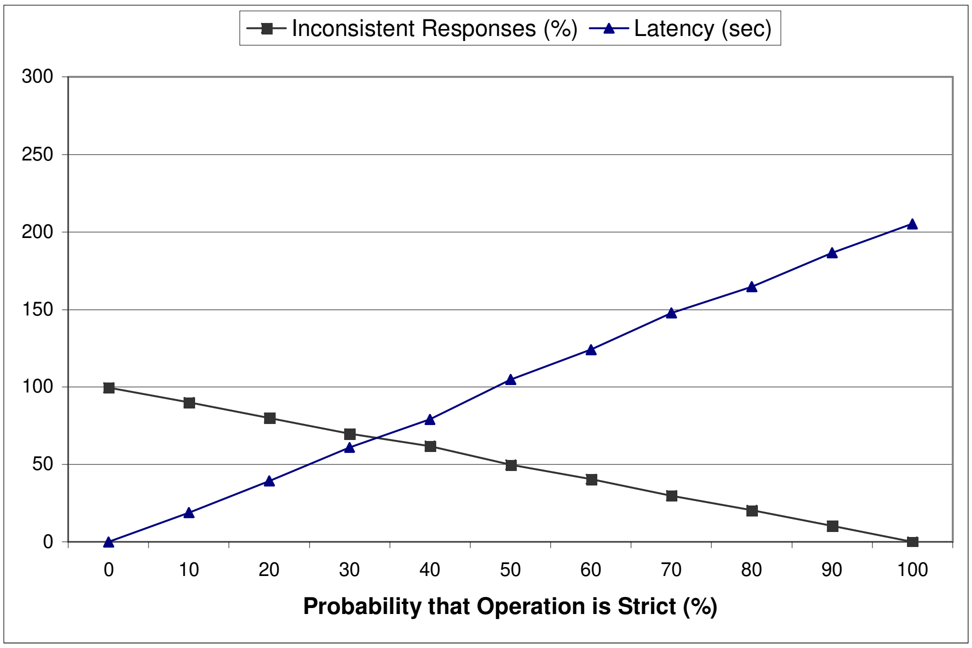

Figure 5.13: Tradeoff Between Response Time and Consistency (4 Replicas)

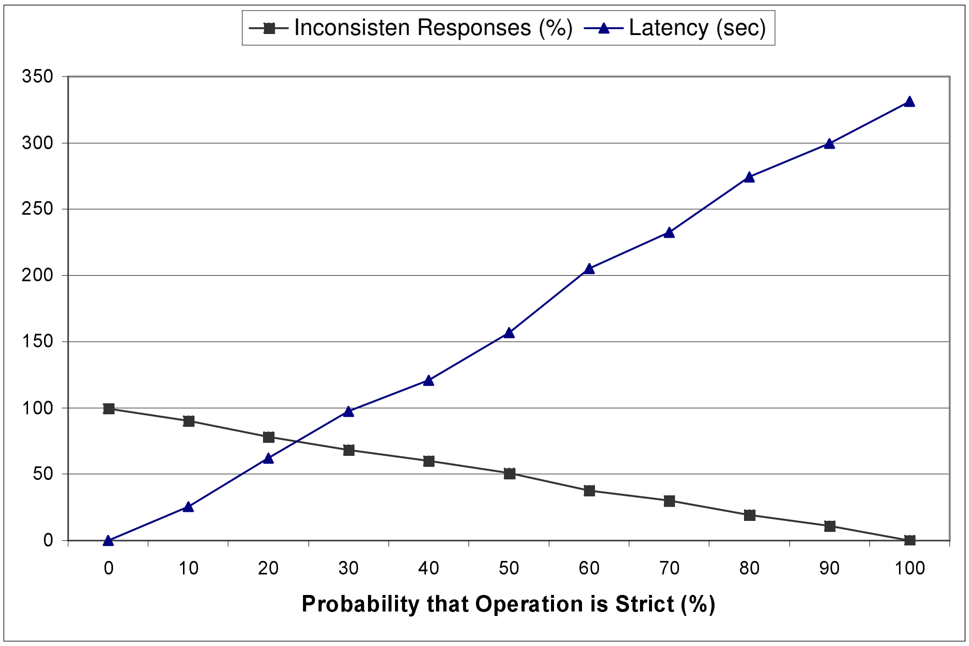

Figure 5.14: Tradeoff Between Response Time and Consistency (6 Replicas)

5.6 Test Series 3: Performance/Consistency Tradeoff

This test was conducted using the Counter Service application, using *Add* operations. The results for 2, 4, and 6 replicas are summarized in Figures 5.12, 5.13, and 5.14.

Predictably, the percentage of inconsistent responses goes down linearly as the percentage of strict operations climbs. However, 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 dramatically higher than the latency of responses to non-strict operations. This is reflected in the linear increases of average latency with percentage of strict operations in Figures 5.12, 5.13, and 5.14. The coefficient of the linear increase is higher for a larger number of replicas, since the time required to synchronize all replicas with respect to a particular operation increases with the number of replicas participating in the system. The trade-off between consistency and performance is clearly demonstrated by these results.

We also conducted this test using the String Concatenation Service application, using *Concatenate* operations. We did not observe substantial differences in the results. This suggests that the percentage of inconsistent responses to non-strict commutative operations such as *Add* is not substantially lower than the percentage of inconsistent responses to non-commutative operations such as *Concatenate*.

Chapter 6

Conclusions and Future Work

We 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 involving two automata for communications between distributed components. An interesting topic for future work is to generalize these techniques to cover all types of I/O Automata compositions.

Using our techniques, we implemented the abstract ESDS algorithm *ESDSAlg* [1] as a distributed program *ESDSImpl*. The modular design of *ESDSImpl* allowed us to write code specific to the ESDS service once and then create several distinct data services without modifying this code. In this way we showed that *ESDSAlg* can be effectively used as a building block for distributed systems.

We strove to create a faithful implementation of *ESDSAlg* and its derivatives, but it remains to be shown that *ESDSImpl* does in fact implement *ESDSAlg*. More ambitiously, it would be interesting to develop a framework for showing that a practical implementation, treated as a mathematical object, correctly implements a formal specification of an abstract algorithm.

After implementing *ESDSAlg*, we implemented several optimizations suggested in the ESDS paper [1] and produced an optimized abstract algorithm *ESDSOptAlg*. We then introduced the optimizations to *ESDSImpl* to produce *ESDSOptImpl*, which implements *ESDSOptAlg*. By producing *ESDSOptImpl* we fixed some inefficiencies of *ESDSImpl* and moved our implementation of ESDS closer to being a practical system. Much work remains to be done in this area. One important optimization that could be applied to *ESDSOptImpl* is discussed in Section 6.1.

We conducted empirical tests on *ESDSOptImpl* and learned how its performance, characterized by response time and throughput, is affected by changing the number of replicas participating in the execution and by the system load. We also obtained empirical evidence confirming that ESDS performance reflects a tradeoff between performance and consistency. The balance can be shifted toward consistency and away from performance by increasing the number of strict operations submitted to the system, and vice versa. Future work in empirical evaluation of ESDS is discussed in Section 6.4.

6.1 Future Optimizations: Multipart Timestamps

Although the *prev* sets used by ESDS to identify dependencies between operations are very intuitive from the point of view of the ESDS developer, they do not give the user of a data service based on ESDS a manageable way of specifying those dependencies. Users of a practical ESDS-based system are not aware of operation identifiers and could not specify long dependency arrays. Furthermore, *prev* sets are memory-inefficient. A *prev* set may include any operations that have been previously submitted to the system, and therefore the upper bound on the size of *prev* sets grows linearly with the number of operations submitted to the system. As discussed in Section 4.3.1, the system is not able to take advantage of stabilization of old operations and discard their identifiers because the identifiers may later appear in a new operation's *prev* set.

The goal of the multipart timestamp optimization is to remove the inefficiencies that result from using *prev* sets. This optimization utilizes the multipart timestamp technique in place of *prev* sets to keep track of system dependencies. The approach is similar to the timestamp-based implementation in [18].

6.2 Dealing with Unreliable Channels

Explicit sequencing of gossip messages combined with retransmission and removal of duplicates is needed to make the incremental gossip optimization work with *unreliable* channels that allow message losses, duplicate messages, and out of order delivery.

6.3 Formally Defining ESDSImpl Behaviors

The goal of this project was to create a faithful implementation of *ESDSAlg* and its derivatives, but it remains to be shown that *ESDSImpl* does in fact implement *ESDSAlg*. More ambitiously, it would be interesting to develop a framework for showing that a practical implementation, treated as a mathematical object, correctly implements a formal specification of an abstract algorithm.

6.4 Future Empirical Investigation

The empirical results presented in Chapter 5 cannot answer whether it is possible to create an implementation of ESDS the can be effectively used as a practical data service. The next step toward answering this question is to create a complete and useful distributed data service based on the ESDS algorithm and run it with real-world users.

Acknowledgments: I thank Alex Shvartsman for overseeing all phases of this project and giving his support and encouragement and Nancy Lynch for suggesting the topic. I also thank Victor Luchangco for valuable suggestions.

I presented parts of this work at a TDS seminar. I am grateful to the members of the TDS group for their feedback that helped me improve the quality of this thesis.

Bibliography

[1] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-Serializable Data Services. *PODC* 1996, pp. 300-310.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

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

[12] 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

[13] 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.

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

[15] 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.

[16] 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

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

[18] 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.

[19] Institute of Electrical and Electronic Engineers. *IEEE Standard 802.3*, 1985.

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

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